GRAPHICS PROCESSING UNIT-BASED COMPUTER-AIDED DESIGN ALGORITHMS FOR ELECTRONIC DESIGN AUTOMATION

by

Yiding Han

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Dr. Koushik Chakraborty Major Professor	Dr. Sanghamitra Roy Committee Member
Dr. Chris Winstead Committee Member	Dr. YangQuan Chen Committee Member
Dr. Dan Watson Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2014

Abstract

Graphics Processing Unit-Based Computer-Aided Design Algorithms for Electronic Design Automation

by

Yiding Han, Doctor of Philosophy

Utah State University, 2014

Major Professor: Dr. Koushik Chakraborty
Department: Electrical and Computer Engineering

This dissertation presents research focusing on reshaping the design paradigm of electronic design automation (EDA) applications to embrace the computational throughput of a massively parallel computing architecture. The EDA industry has gone through major evolution in algorithm designs over the past several decades, delivering improved and more sophisticated design tools. Today, these tools provide a critical platform for modern integrated circuit (IC) designs composed of multi-billion transistors. However, most of these algorithms, although showcasing tremendous improvements in their capabilities, are based on a sequential Von Neumann machine, with limited or no ability to exploit concurrency. While such limitation did not pose any significant end effect in the past, the advent of commodity multicores during the beginning of this decade created a need to embrace concurrency in many fields, including EDA algorithms. This need is now fast gaining urgency with the recent trends in the emergence of the general purpose computation on graphics processor units (GPU).

Through algorithmic overhaul, and novel solution space exploration strategies, this research has shown a concrete path in which inherently sequential problems can benefit from the massively parallel hardware, and gain higher computation throughput. Broadly,

two important EDA topics are discussed in this dissertation: (1) A floorplanner using a GPU-based simulated annealing algorithm, and (2) a global router framework using GPU architecture and a fast congestion analysis framework. Both topics aim to use GPU as a testbed for high throughput computation. Optimization strategies are studied for the GPU implementations. The GPU-based floorplanning algorithm is able to render 4-166X speedup, while achieving similar or improved solutions compared with the sequential algorithm. The GPU-based global routing algorithm is shown to achieve significant speedup against existing state-of-the-art global routers, while delivering competitive solution quality. The proposed methodology of a design paradigm shift for sequential EDA algorithms has a profound impact on the efficiency and design quality of future IC design flow.

(144 pages)

Public Abstract

Graphics Processing Unit-Based Computer-Aided Design Algorithms for Electronic Design Automation

by

Yiding Han, Doctor of Philosophy

Utah State University, 2014

Major Professor: Dr. Koushik Chakraborty
Department: Electrical and Computer Engineering

The electronic design automation (EDA) tools are a specific set of software that play important roles in modern integrated circuit (IC) design. These software automate the design processes of IC with various stages. Among these stages, two important EDA design tools are the focus of this research: floorplanning and global routing. Specifically, the goal of this study is to parallelize these two tools such that their execution time can be significantly shortened on modern multi-core and graphics processing unit (GPU) architectures. The GPU hardware is a massively parallel architecture, enabling thousands of independent threads to execute concurrently. Although a small set of EDA tools can benefit from using GPU to accelerate their speed, most algorithms in this field are designed with the single-core paradigm in mind. The floorplanning and global routing algorithms are among the latter, and difficult to render any speedup on the GPU due to their inherent sequential nature.

This work parallelizes the floorplanning and global routing algorithm through a novel approach and results in significant speedups for both tools implemented on the GPU hardware. Specifically, with a complete overhaul of solution space and design space exploration, a GPU-based floorplanning algorithm is able to render 4-166X speedup, while achieving similar or improved solutions compared with the sequential algorithm. The GPU-based global

routing algorithm is shown to achieve significant speedup against existing state-of-the-art routers, while delivering competitive solution quality. Importantly, this parallel model for global routing renders a stable solution that is independent from the level of parallelism. In summary, this research has shown that through a design paradigm overhaul, sequential algorithms can also benefit from the massively parallel architecture. The findings of this study have a positive impact on the efficiency and design quality of modern EDA design flow.

This dissertation is lovingly dedicated in memory of my father, Fei Han, and of my grandfather, Zhenyi Chen, who each inspired my life through their strength, faith, and love.

Acknowledgments

I would like to express my very great appreciation to my adviser, Dr. Chakraborty, for his insightful advice, generous financial support, and patient guidance throughout my entire PhD research. Without his motivation and insights this work would have never been complete. Also, I would like to thank Dr. Roy for assisting me in many publications, her useful critiques of this research, as well as her teachings which helped me tremendously in understanding the EDA algorithms. Also, I would like to express my deep gratitude to all of my committee members, Dr. Chen, Dr. Winstead, and Dr. Waston, for their valuable comments on this research, and the patience to work with me even with the barrier of long distances.

I would like to thank various student members of the Bridge Lab for their constant support, encouragement, as well as making Bridge Lab such a pleasant and rewarding place to work. Specifically, I would like to thank Vilasita for her assistance in the initial work of GPU-based floorplanning; Dean for his valuable contribution and insights for the GPU-based global routing work; and Saurabh for the opportunity to work on the project involving timing analysis of NBTI aging effects. I wish also to acknowledge the help provided by Kshitij Bhardwaj, Jason Allred, Hu Chen, Manzi Dieudonne, Rajesh JS., Harshitha Pulla, Brian Cluff, and Shayan Taheri.

I would like to express my great appreciation to the ECE department and all of the staff members, for offering me this opportunity of PhD research, as well as the financial assistance towards my tuition. I would like to offer my special thanks to Dr. Moon for helping me out in many tough situations. I am particularly grateful for the assistance given by Mary Lee Anderson and Tricia Brandenburg, who have helped me through numerous drafts and formatting of the dissertation. I would also like to extend my thanks to Trent Johnson and Scott Kimber for providing technical support and maintaining the computer laboratory.

Last, but not least, special thanks should be given to my lovely wife, Susanna, and everyone in our families, for their support and encouragement throughout my study.

Yiding Han

Contents

List of Tables

List of Figures

Acronyms

EDA	electronic design automation
IC	integrated circuit
ASICs	application-specific integrated circuits
FPGA	field-programmable gate array
GPU	graphic processing unit
API	application programming interface
ILP	integer linear programming
IP	integer programming
GPGPU	general purpose GPU
SIMD	single instruction multiple data
SIMT	single instruction multiple thread
SP	streaming processor
CAD	computer aided design
VLSI	very-large-scale integration
CUDA	compute unified device architecture
GRP	global routing problem
RRR	rip-up and re-route
IP	integer programming
ILP	integer linear programming
NLC	net-level concurrency
FGC	fine-grain concurrency
RSMT	rectilinear Steiner minimal tree

Chapter 1

Introduction

As the design complexity of modern integrated circuit (IC) aggregates exponentially, the electronic design automation (EDA) tools, which provide a critical platform for computer-aided design, have become increasingly important. Modern EDA tools designs dedicate considerable efforts to improve the capability of handling complex design constraints and very large size circuits within limited design period. Most of these algorithms, although showcasing tremendous improvements over the past decades, are based on a sequential Von Neumann machine, which have very little ability to exploit concurrency.

Due to the advent of commodity multi-cores during the beginning of this decade, parallel computing has gradually become the major computational paradigm to replace sequential computing. In many data and arithmetic intense fields, including EDA, this shift of paradigm calls for an embrace of concurrency in algorithm designs. Broadly, there are two aspects to this shift: (1) parallelize the application to take advantage of the concurrency provided in modern hardware; (2) understand the performance characteristics of the parallel architectures for algorithmic and implementational optimization. Interestingly, this shift is now fast gaining urgency with the recent trends in the emergence of the general purpose computation on graphics processor units (GPU) [1].

1.1 EDA and Emerging GPU Computing Paradigm

Modern GPUs are inherently concurrent designs, with several thousands of processing units within a single GPU. They not only demonstrate tremendous computation bandwidth (orders of magnitude improvement from commodity multicores), but also the ability to allow non-graphics applications to harness their computing prowess. It is the latter development that will make significant impact in EDA algorithms, as algorithms designed for GPUs in

the next decade are poised to bear little resemblance to the existing body of EDA tools.

This section briefly outlines the design flow of modern EDA tools (Section 1.1.1), draws the emphasis of this research in the physical design category of EDA tools (Section 1.1.2), and presents the current trend of GPU computing in EDA tools (Section 1.1.3).

1.1.1 Overview of Modern EDA Design Flow

The EDA design flow is a combination of computer-aided design tools, which are used to accomplish the design of an integrated circuit (IC). EDA tools are specialized for different IC design methodologies, such as full custom design, application-specific integrated circuits (ASICs) standard cell design, and field-programmable gate array (FPGA) designs, etc.

This work addresses EDA tools that are specific for the ASICs standard cell design flow. During the ASICs flow, several stages of EDA design are used to bring an IC from register-transfer level (RTL) to graphic data system stream format (GDSII). RTL models a synchronous digital circuit with an abstraction of the hardware, while defining the signals, registers, and logical operations performed by the circuit. The EDA tool chain realizes the RTL circuit using components from ASICs standard cell libraries, and generates the output of the IC design in a GDSII format, which plots the physical layout of the circuit ready for chip fabrication.

An ASICs design flow can be further categorized as the following three classes: logic synthesis, physical design, and verification. Figure 1.1 illustrates this classification and the relations among these categories of tools. Starting with the logic synthesis tools, the goal of this stage is to translate the RTL specification into netlist, which consists of IP blocks, gates and interconnects, etc. These components are subsequently arranged and assembled by the physical design tools. The final design output is printed in GDSII format for fabrication. The verification tools are typically interleaved throughout the entire design process to ensure the IC meeting various design specifications. The verification includes many standalone processes, including logic simulation, timing analysis, formal verification, signal integrity check, and design rule checking, etc.

This work will mainly focus on design tools from the physical design family. Specifically,

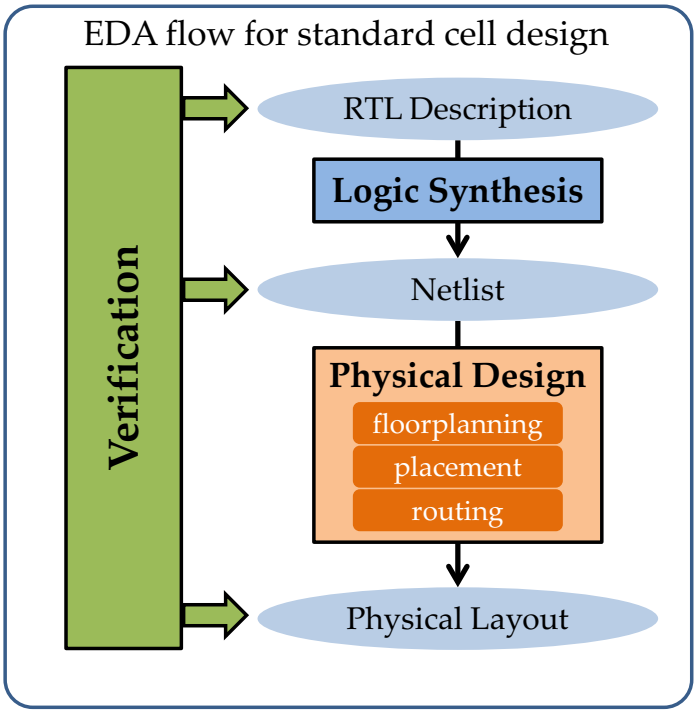


Fig. 1.1: The EDA design flow for standard cell consists of many stages, typically including logic synthesis, physical design, and verification tools.

it studies tools for floorplanning and global routing of an IC. The following sections will further describe the roles of these physical design tools and address their design challenges.

1.1.2 Physical Design of EDA Flow

The body of this research focuses on the physical design tools of EDA family. A typical physical design flow can be broadly divided into the placement and routing phase. The difference between these two phases is that the placement focuses on finding the physical locations IC components, such as transistors and IP blocks, while the routing phase focuses on the wiring among these components. The design flow starts with placement of the IC components and ends with successful routing of wires that connect the placed components.

The placement phase itself has multiple stages, typically consisting of floorplanning, global placement, and detailed placement. Each stage works with an abstraction of the physical layout in a top-down manner. Floorplanning divides the chip components into

relatively large size blocks and tries to find a close-to-optimal configuration to lay down these blocks within the chip area. Global placement finds general locations for all gates and modules, and detailed placement places them at exact locations on the IC chip.

Similarly, the routing phase is also multi-staged, including global routing and detailed routing. The routing phase is the most complex one throughout the EDA design flow, since it must obey all design and electrical rules of an IC. A multi-staged routing flow alleviates the complexity of the problems. First, global routing phase connects circuits on a coarse-grain level, which identifies and corrects large-scale congestion and routing issues. Then, detail routing takes the coarse-grain solution, and lays down the fine-grain physical interconnect on IC circuits. Typically, the interconnects are built as metal tracks and vias, hence their electrical characteristics must be taken into consideration to avoid problems such as crosstalk, antenna effects, etc. As the complexity of IC design grows, new design constraints are introduced to facilitate features such as multi-layer interconnects, multi-pattern lithography, and make routing extremely difficult.

In modern designs where routability becomes the primary obstacle, the usage of global routing is increasingly critical. To mitigate the routability issue, it is common to utilize congestion analysis tools to reveal routability issues in the early design stages, such as floor-planning and placement. With the early congestion information, designers can rearrange floorplan and massage placement of IC components to avoid a potentially difficult-to-route design. Interestingly, modified global routing engine can be used as a congestion analyzer, leading to a trend of integrating and interleaving global router with other tools in modern physical design flow.

1.1.3 GPU in EDA Design Flow

The advent of GPU computing has successfully rendered improvement in runtime efficiency of many EDA design tools. Compute intensive algorithms like fault simulation, power grid simulation and event-driven logic simulation have been successfully mapped to GPU platforms to obtain significant speedups [2–5]. Recently, Liu and Hu proposed a gate sizing and threshold voltage algorithm optimized for GPUs [6]. Cong and Zou [7] use GPU com-

puting on a force-direct algorithm for global placement, which renders impressive speedup over the CPU sequential implementation. One key similarity in all these previous works is the presence of a fixed common topology/data structure across parallel threads that are fed with separate attributes for concurrent evaluation (e.g. distinct gate sizes and threshold voltages for a single circuit topology, distinct input patterns for a single circuit, distinct localization of components placement). The unmodified topology is highly amenable to the single instruction multiple thread (SIMT) style in GPUs, as it does not require frequent data modification and reuse.

The adaptation of GPU platform for EDA algorithms is, however, at an early stage. The focus of this work is to remodel floorplanning and global routing algorithms for GPU computing, which pioneers the GPU computation research. Specifically, this is the first work in open literature to apply GPU computing for floorplanning and global routing algorithms. Importantly, neither of these algorithms exhibit distinct topology for parallel computing as in the existing paradigm of GPU computing research. This challenge dictates novel context sensitive design space and exploration space explorations to overhaul the paradigm of existing sequential algorithms, formulating the main body of this research.

1.2 Motivation

In contrast to several GPU adapted applications, a great number of EDA algorithms still struggle to keep pace with the performance improvement of GPU hardware platform, as illustrated in Figure 1.2. Typically, unlike applications with abundant data-parallel operations, these EDA applications have limited ability to explore concurrency with their irregular data structures and data dependencies. Even implemented on the GPU platform, the performance boost achieved in such category is considerably lower [7, 8].

Among the above mentioned category are some of most computationally intensive EDA applications in the very large scale integrated (VLSI) circuit design flow. Today's aggressive technology scaling introduces many additional constraints in physical designs of VLSI circuits. Such explosion in design rules fundamentally increases the complexity of the problems such as placement and routing, which are two of the most time consuming stages of

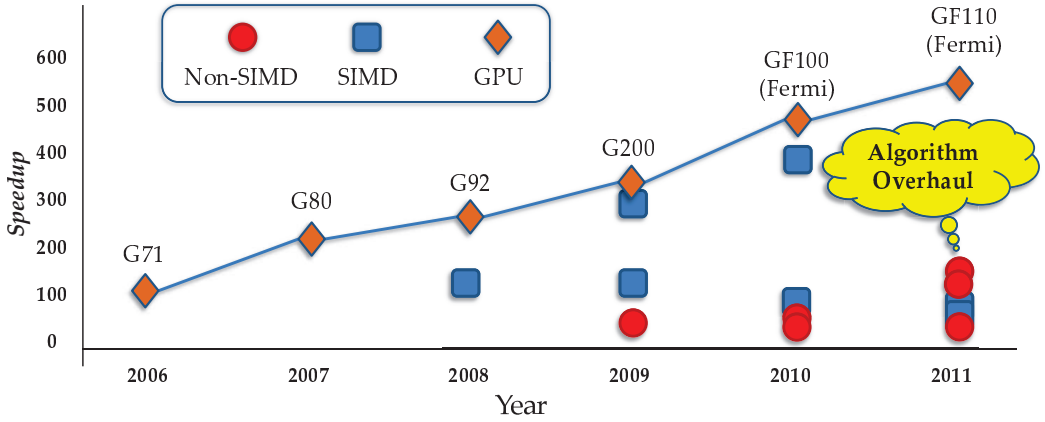


Fig. 1.2: Evolution of EDA algorithms in GPU platforms. The figure compares the speedup achieved in NVIDIA GPU architectures relative to a sequential processor (measured in giga-floating point operations), with the reported speedups from the first generation GPU-enabled EDA applications.

VLSI physical design. However, these application typically adopt intrinsically sequential algorithms that are originally designed for a single-core architecture. Without altering the algorithms themselves, these applications can hardly benefit from the extra concurrency created by modern parallel architectures.

These EDA applications face a two-fold challenge. On the one hand, the drastically increased problem size and complexity dictates more complicated algorithms to ensure design quality. But these needs also prolong the design cycle and ultimately lead to delay of time-to-market; on the other hand, with the single-core performance coming at a halt, the performance of the single threaded EDA tools can hardly benefit from the hardware performance progression. As a result, without the ability to scale in parallel environment, these EDA applications can become bottle-neck of the VLSI design flow.

The key research question is **whether it is possible to completely overhaul the aforementioned EDA algorithms, and reshape their design space exploration to better utilize the throughput of massively parallel computing platforms**. This design philosophy elevates this research from existing general purpose GPU computing works, which mainly focus on inherently parallel applications. Through a design paradigm shift,

this research has shown that intrinsically sequential algorithms and applications can gain higher computational throughput using the massively parallel hardware. Importantly, although GPU is used as a testbed for high throughput computation, the proposed methodologies are developed in a generic manner widely applicable on parallel architectures at large.

1.3 Contributions of this Research

This research focuses on reshaping the design paradigm of EDA applications to embrace the computational throughput of a massively parallel computing architecture. Through algorithmic overhaul, and novel solution space exploration strategies, this research has shown a concrete path in which inherently sequential problems can benefit from the massively parallel hardware, and gain higher throughput. The designs of the GPU algorithms are generic for massively parallel architectures, making them applicable to other throughput computing platforms for future EDA applications.

Publications made during the period of this research are listed as follows:

Journal Papers

- Exploring High Throughput Computing Paradigm for Global Routing, Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy, *IEEE Transactions on Very Large Scale Integration Systems*, *Accepted*

- Design and Implementation of a Throughput Optimized GPU Floorplanning Algorithm, Yiding Han, Koushik Chakraborty, Sanghamitra Roy and Vilasita Kuntamukkala, *ACM Transactions on Design Automation of Electronic Systems*, Volume 16, Issue 3, No. 23, June 2011, Pages 23:1–23:21

Conference Papers

- A Global Router on GPU Architecture, Yiding Han, Koushik Chakraborty, and Sanghamitra Roy, *IEEE International Conference on Computer Design*, 2013, Pages 74–80
- DOC: Fast and accurate congestion analysis for global routing, Sanghamitra Roy, Yiding Han, Koushik Chakraborty, *IEEE 30th International Conference on Computer Design*, 2012, pages 508–509
- Exploring High Throughput Computing Paradigm for Global Routing, Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, Sanghamitra Roy, *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, November 2011, San Jose, Pages 298–305
- Optimizing Simulated Annealing on GPU: A Case Study with IC Floorplanning, Yiding Han, Sanghamitra Roy and Koushik Chakraborty, *12th IEEE International Symposium on Quality Electronic Design (ISQED)*, March 2011, Pages 1–7
- A GPU Algorithm for IC Floorplanning: Specification, Analysis and Optimization, Yiding Han, Koushik Chakraborty, Sanghamitra Roy and Vilasita Kuntamukkala, *24th IEEE/ACM International VLSI Design Conference*, 2011, Pages 159–164

1.3.1 A Floorplanner Using GPU-Based Simulated Annealing Algorithm

This work proposes a fundamentally different approach of exploring the floorplan solution space. Several performance optimization techniques are demonstrated for this algorithm in GPUs. This research is published in the proceedings of VLSID-11 [9], ISQED-11 conferences [10]. A journal version is published in the TODAES-11 journal [11].

A novel floorplanning algorithm for GPUs is proposed. Floorplanning is an inherently sequential algorithm, far from the typical programs suitable for SIMT style concurrency in a GPU. In this work, a fundamentally different approach of exploring the floorplan solution space is proposed. It illustrate several performance optimization techniques for this algorithm in GPUs. To improve the solution quality, a comprehensive exploration of

the design space is presented, including various techniques to adapt the annealing approach in a GPU. Compared to the sequential algorithm, the proposed techniques achieve 6-188X speedup for a range of MCNC and GSRC benchmarks, while delivering comparable or better solution quality. The contributions of this work are:

- An algorithm to perform floorplanning on a GPU machine. This is the first work on GPU-based floorplanning.
- An in-depth analysis of the profiling in different GPU components, and illustrate several optimizations for the GPU floorplanning algorithm.
- A modified algorithm based on the speedup vs quality analysis to improve the solution quality, and show detailed results of the modified algorithm.
- Based on algorithmic foundation, a comprehensive design space exploration of the solution space is presented. The analysis spans from detailed breadth and depth analysis, adapting the annealing technique for GPU, and dynamic selection and exploration of the solution space.

1.3.2 A GPU-CPU Hybrid Global Router

This is the first work on utilizing GPUs for global routing. It explores a hybrid GPU-CPU high-throughput computing environment as a scalable alternative to the traditional CPU-based router. A novel parallel model is proposed for router algorithms that aims to exploit concurrency at the level of individual nets. This work is published in the proceedings of ICCAD 2011 conference [12]. Its journal version is accepted by the TVLSI-12 journal.

With aggressive technology scaling, the complexity of the global routing problem is poised to rapidly grow. Solving such a large computational problem demands a high throughput hardware platform such as modern GPUs. This work explores a hybrid GPU-CPU high-throughput computing environment as a scalable alternative to the traditional

CPU-based router. A *net level concurrency (NLC)* model is introduced. NLC is a novel parallel model for router algorithms that aims to exploit concurrency at the level of individual nets.

To efficiently uncover NLC, a *Scheduler* is designed to create groups of nets that can be routed in parallel. At its core, the Scheduler employs a novel algorithm to dynamically analyze data dependencies between multiple nets. Such an algorithm can lay the foundation for uncovering data-level parallelism in routing: a necessary requirement for employing average of 4X speedup over NTHU-Route 2.0 with negligible loss in solution quality. The contributions of this works are:

- An execution model that allows cooperation of the GPU and the CPU to route multiple nets simultaneously through NLC. The GPU global router uses a breadth first search (BFS) heuristic while the CPU router uses A* search routing. Together, they provide two distinct classes in the routing spectrum. The high-latency low-bandwidth problems are tackled by the CPU, whereas the low-latency high-bandwidth problems are solved by the GPU. This classification is the key to efficiently tackle the complexity increase of the global routing problem on massively parallel hardware.

- A scheduler algorithm to explore NLC in the global routing problem. The scheduler produces concurrent routing tasks for the parallel global routers based on net dependencies. The produced concurrent tasks are distributed to the parallel environments provided by the GPU and multi-core CPU platforms. The scheduler is designed to dynamically and iteratively analyze the net dependencies, hence limiting it computational overhead.

- A Lee algorithm based on breadth-first search path finding on a GPU. This algorithm utilizes the massively parallel architecture for routing and back tracing. The approach is able to find the shortest weighted path, and achieves high computational throughput by simultaneously routing multiple nets.

1.3.3 A Fine Grain Concurrency Model for Global Router on GPU

This is the first work that utilizes GPU alone for global routing, without offloading any routing workload to the CPU. Based on the net-level concurrency model proposed in the previous GPU-CPU hybrid concept, this work extended the concurrency model to a fine-grain level, which significantly improves the exploitable parallel workloads. A multi-agent GPU routing engine is developed based on A* search algorithm. This work is published in the proceedings of ICCD 2013 conference [13].

In the modern VLSI design flow, global router is often utilized to provide fast and accurate congestion analysis for upstream processes to improve the design routability. Global routing parallelization is a good candidate to speedup its runtime performance while delivering very competitive solution quality. This work first study the cause of insufficient exploitable concurrency of the existing NLC model, which has become a major bottleneck for parallelizing the emerging design problems. It mitigates this limitation with a novel fine grain parallel model, with which a GPU-based multi-thread global router is designed. Experimental results indicate that the parallel model can effectively support the GPU-based global router, and deliver stable solutions. The contribution of this work includes:

- A concurrency model that exploits parallelism on the Steiner edge level. The main reason of the limited exploitable concurrency is identified to be a false data dependency in the NLC parallel model. The Steiner edge-based net decomposition scheme effectively mitigates this issue, and increases the exploitable concurrency.

- A routing engine on GPU architecture to allow multi thread global routing based on an A* search multi-source multi-sink maze routing algorithm. This A* search is designed broadly based on an existing A* search GPU implementation, but specialized for global routing optimizations.

- The router is designed with the ability to work with the recent routability-driven benchmarks. Experimental results indicate successful parallelization with the GPU-based router, which renders a deterministic solution. Moreover, the run-time of the proposed global router is up to 3.0X faster than that of the NCTUgr2 [14].

1.3.4 Congestion Analysis

This work presents a fast and accurate congestion analysis tool at the global routing stage. It focuses on capturing the difficult-to-solve congestion in global routing designs. The proposed framework identifies the routing congestion using a novel orthogonal congestion correlation (OCC) factor, which identifies the hard-to-route hot-spots. A key contribution of this work is a fast global router to minimize congestion caused by long nets and accurately reveal the distribution of hard-to-route spots due to high density short nets. The global router uses a dynamic representation of net to allow fast topology transformation. The proposed framework can evaluate the routability of a placement solution, and be utilized to aid the placer for a congestion-aware design.

The focuses of this research is in introducing new techniques to improve the accuracy of congested region detection, as well as reducing the runtime overhead of a congestion analyzer. The major contributions are as following:

- An accurate overflow identification approach based on a novel orthogonal congestion analysis. The analysis uses a new metric to model the difficult-to-route region by studying the nature of difficult-to-route problems. The proposed metric is shown to be able to pinpoint problematic regions with high accuracy.

- A set of dynamic edge moves that can efficiently resolve congestion. As the efficiency of routing engine plays a key role for congestion analysis, complex routing algorithm is avoided to allow extremely fast analysis. The dynamic edge moves provides a fast yet accurate routing methods for congestion analysis.

Chapter 2

Literature Survey and Related Work

A comprehensive literature survey is undertaken to build the foundation of this research. The relevant areas of this study includes the current adaptation of GPU for EDA tools, the existing body of research for floorplanning and its parallelization, previous works on global routing and its parallel schemes, and the advent of modern congestion analysis tools. To serve the purpose of a literature survey, the following sections are organized as follows: Section 2.1 outlines the previous works that utilize GPU for EDA applications to render speedup; Section 2.2 reviews the current floorplanning technologies and previous parallelization schemes; Section 2.3 lists a collection of existing global routing techniques; Section 2.4 visits the current frameworks used for global routing; Section 2.5 gives a survey on the congestion analysis research.

2.1 EDA on GPU

Some EDA algorithms are able to enjoy the adaption to the GPU platform. Compute intensive algorithms like fault simulation, power grid simulation and event-driven logic simulation have been successfully mapped to GPU platforms to obtain significant speedups [2–5]. Recently, Liu and Hu proposed a gate sizing and threshold voltage algorithm optimized for GPUs [6]. One key advantage in many of these previous works is the *unmodified* topology/data structure across parallel threads, which make them highly amenable to the SIMT style in GPUs. More recently, GPU optimized algorithms for irregular computing patterns like sparse matrix vector product and breadth first graph traversal have been proposed [8]. These algorithms can be applied to certain EDA problems like static timing analysis and force directed placement that use sparse matrix operations. Cong and Zou also optimized the force-directed placement algorithm on GPU platforms [7]. Frishman and Tal used GPUs

to optimize force directed graph layout problem, where certain steps required simulated annealing [15, 16].

In contrast, the floorplanning and global routing algorithms pose severe challenges as they both involve a chain of dependent modifications to the data structures. There is little similarity between existing GPU work and the focus of this research. For example, the layout problem allowed partitioning the graphs, and calculating the desired objective in each partition. In IC floorplanning, no such partitioning is possible during objective calculation as a given move can impact nearly all possible blocks in the floorplan. Due to such fundamental differences in problem structure and resulting restrictions, their proposed algorithm with optimization techniques are not applicable for the GPU-based floorplanning research.

2.2 Floorplanning and Parallelization

The modern fixed-outline floorplanners are typically based on simulated annealing algorithms [17–19]. The simulated annealing is a heuristic that applies random moves to a floorplan to approach for an optimal solution. It is shown recently by Zeng and Chen that searching agents such as random walk for optimal search is fundamentally linked with fractional dynamics [20], which can provide insight for future design and optimization of floorplanning algorithms.

Typically the existing fixed-outline floorplanning problems target to minimize overall area and half-parameter wire length. Cong et al. present a floorplanner to also consider thermal distribution of 3D ICs [21]. To improve the efficiency of modern floorplanner, a fast floorplanner for multilevel design is presented by Chen et al. [22], Chen and Chang present a fast simulated annealing based on B*-tree representation [19]. However, all of the above floorplanning algorithms are based on sequential architectures.

There has also been some works on parallel simulated annealing, albeit not on floorplanning. Kravitz and Rutenbar [23] presents strategies for implementing simulated annealing-based standard cell placement on shared memory multiprocessors. They propose two orthogonal approaches to exploit parallelism: decomposing single move and compute multiple

moves simultaneously. Done by Rose et al. [24], a SA based standard cell placement algorithm generates and investigates different coarse placements to exploit parallelism. Another parallel simulated annealing is done by Ram et al. [25]. The authors propose to exploit parallelism by expanding the search space. Each processor individually performs SA on a solution space. However, algorithmic exposition for IC floorplanning, with optimization challenges in GPU.

2.3 Global Router: Current Methodology

The global routing problem is NP-hard [26]. Therefore, heuristics are applied to reach the approximate optima of the global routing solution. In general, the algorithms used to solve global routing can be categorized into two classes: (1) sequential; (2) concurrent. The sequential algorithm is typically based on a rip-up and re-route (RRR) scheme, which rips-up the interconnects that cause overuse of routing resources and re-route their paths individually. The routing resources claimed by the previously routed paths have deterministic affects on the decisions made to route the later paths. For this reason, nets in general must be routed in an explicit sequential order.

The concurrent algorithm often utilizes the integer programming approaches, which attempt to route all nets concurrently. The global routing problem is modeled with {0, 1}-integer linear programming (ILP) problem that selects solutions out of candidate paths that optimize the desired global objectives. Such approaches typically possess better global information to the problem, hence can reach a much closer approximation to the optima than the sequential approach. However, in a run time comparison, the concurrent algorithms are generally significantly slower.

Routing techniques solve the fundamental problem in global routing of finding the least costly path to connect two vertices on a rectilinear grid map. Current routing techniques can be roughly categorized into two classes: (1) maze routing; (2) pattern routing.

2.3.1 Maze Routing

Maze routing was first introduced by Moore [27]. The core procedure of maze routing

has two stages: propagation and back-tracing. The propagation phase creates a wave from the source node, and spreads out the wave on a grid map to search for sink node. Once found, the back-tracing stage identifies the least costly path in a reverse direction connecting sink to source node.

Several algorithms have been proposed to realize maze routing for VLSI global routing problem, among these are Lee algorithm [28], Hadlock's algorithm [29], Dijkstra's algorithm [30], and extensions of the Dijkstra's algorithm, such as the A* search algorithm [31]. These algorithms have significantly enhanced maze routing in memory allocation and run time, although maze routing's main procedure remains intact.

A key advantage of maze routing is its flexibility in accommodating zigzag shaped solutions, which enables the router to avoid complex obstacles in a routing grid. However, the maze routing technique is generally costly in run time with large scale modern designs. Techniques such as bounding-box [32] and wire length bounded routing [14] are common enhancements to alleviate long run time stress.

2.3.2 Pattern Routing

Pattern routing highly restricts the shapes of the searching paths to obtain speedup over maze routing technique. The patterns in which the search paths are specified are typically L-shape, Z-shape, and C-shape routes, all limiting the number of bends within the search region. In general, pattern routing can identify a path with exceptional speed, but it also results in inferior solution quality.

2.4 Global Routing: Routing Framework

A routing framework typically defines the methodology with which a global router iteratively "improves" the global solution. The goal of the routing framework is to remove physical constrains violations, such as congestion and cross-talks, etc., while maintaining an approximation to the optimal global cost, such as the total wire length and via counts, etc. As mentioned earlier, current routing frameworks can be roughly classified to the RRR based sequential approach and the IP-based concurrent approach.

2.4.1 Rip-up and Re-route (RRR)

The most commonly applied framework is the Rip-up and Re-Route scheme, for it has a much better trade-off between solution quality and run time. In each iteration of RRR, all nets are visited in a specific order. Nets with violations in their paths are ripped-up and re-routed, altering the routing resources available for the subsequent nets. Such sequential procedure creates a long chain of dependencies among the nets. As explained before, the RRR scheme heavily relies on the order in which the nets are visited. Several existing approaches have focused on improvement of the net ordering to achieve better solution quality [33–40].

McMurchie and Ebeling have proposed a negotiation-based RRR routing to mitigate the congestion issue [36]. Although their work focuses on field-programmable gate array (FPGA) routing, a similar negotiation based technique has been applied to several modern academic global routers to effectively reduce congestion [32, 34, 41, 42]. Typically, the negotiation-based RRR scheme keeps track of history-based edge weights. Once an edge is found with consistent congestion throughout several iterations, it is assigned with a significantly higher cost. Such strategy encourages the global router to generate detours around heavily congested regions.

2.4.2 Integer Programming

The integer programming (IP) framework solves the routing problem at a global scale. Instead of sequentially improving individual net, the IP-based framework solves multiple nets concurrently. Typically, each net is initially routed with multiple instances of solution candidates. The integer linear model then selects the best combination from all candidates to optimize the global design objective. In each iteration of the IP solver, more accurate net topology candidates are generated, gradually leading the overall routing solution to the global optima.

The IP-based approach has several advantages over the RRR. Thanks to the vision of a global solution, the integer programming approach is inherently superior to the sequential approach in routing solution quality. Also, its concurrent framework frees the algorithm

from the constraints of net ordering, allowing great potential for parallelization. However, these advantages also come at a great cost. The IP-based approaches are often extremely computationally intensive. For a time-driven design, the prolonged run time nearly renders the IP-based approach prohibitive [43]. As a compromise to solution quality, the IP model can be integrated into a RRR framework and solve problems with much smaller scale [44,45], thereby occupying much shorter run time.

2.5 Congestion Analysis

Congestion analysis tools have undergone considerable improvements to predict the routability of a design. Probabilistic congestion predictions are first introduced to provide feedback of routability information to the upstream processes [46–48]. However, the probabilistic analysis can be inaccurate in modern designs [49]. Newly proposed congestion analysis tools, such as CGRIP [45], typically use a fast global router engine under strict timing or wire length enclosure to highlight the routability problems in a placement design. Specifically, this enclosure is enforced by providing a fixed bounding-box to each net and prohibits long detours. Consequently, the congestion analysis reveals the *true congestion* that will stresses the router in later design stages. In addition, several recent works have shown that an integration between the placer and a fast global router can improve solution routability [50–52]. Within this integration, placers use techniques such as cell bloating to mitigate congestion discovered by the fast global router. Therefore, the tool chain can effectively address routability issues from the upstream processes, and significantly accelerate the overall design convergence.

Chapter 3

General Purpose GPU Computing

3.1 Evolution of GPU General Purpose Computing

GPU architecture has several revisions. It started out as a pure graphic rendering hardware. The early GPU architecture were exclusively designed to exploit parallelism of graphics applications. Although limited configurability was enabled to allow various graphics rendering effects, the GPU architecture was still designed as a fixed-function processor. The graphics pipeline excelled at three-dimensional (3D) graphics but little else. Even so, designers with extensive knowledge of graphics API and GPU pipeline explored the possibility of utilizing the GPU hardware for general purpose computing tasks. A few scientific applications realized their implementation on the GPU [53], which rendered unprecedented speedup (more than 100X) over their original sequential implementation on CPUs. These events were the advent of the movement called general purpose GPU (GPGPU) computing.

While achieving significant performance with the GPU hardware, the challenge of programming the GPU for real-world application was immense. Historically, such practice required the use of graphics programming APIs like OpenGL and Cg to program the GPU. Data involved during the computation must be carefully crafted as vertices and textures to fit onto the GPU pipeline. Little data structure support was provided, and no random address access was allowed. Even the commonly used control logic in a CPU-based application were absent in the GPU architecture. These factors limited the accessibility to the tremendous capability of GPUs for general purpose computing.

Fortunately, nowadays the GPU architecture has evolved into a much more programmable processor, which not only excels in the 3D image rendering tasks, but is also accessible by a large number of general purpose tasks. The architecture of the GPU transformed from a

fix-function graphics pipeline to a unified multi-processor with enormous arithmetic capability. The first ever of such GPU architecture was designed by AMD for the Xenos GPU in the XBox 360 game consoles. The first GPU in a PC system using unified processor was named G80. Developed by NVIDIA, the unified device architecture of G80 uses a large number of identical computational elements to accomplish various tasks in a pipeline. The benefit for 3D graphics rendering is better load-balancing. Because the programmable units now divide their time among vertex, fragment, and geometry computation, both coarse and fine grain parallelism are exploited. The model is easily scalable to tailor for diverse tasks.

For GPGPU users, the benefits of the unified shader architecture is much clearer. Rather than the previous approach in which computational tasks had to be divided across multiple fixed-function hardware, the unified shader architecture allows programmers to directly target the many core processor. Providing support for random address access, integer arithmetic, and control logic, etc., the unified shader architecture renders an important paradigm shift in the way GPU hardware is used for general purpose application.

3.2 Programming Model: CUDA

With the GPU hardware evolution facilitating general purpose computing, it is also critical to present a programming model for GPU. The programmable units of the GPU follow a single instruction multiple data (SIMD) model. Each instruction has effect on multiple data elements in parallel. As the shader and general purpose programs become more complex, it is sometimes preferable to allow different threads to take different processing paths through the same instruction. This notion leads to a more general single instruction multiple threads (SIMT) model, which is used by the programming model proposed by NVIDIA, namely CUDA.

CUDA defines the thread model and memory model of a GPU based application. Threads are managed by CUDA in a hierarchical and scalable manner. The CUDA thread hierarchy is best explained with Figure 3.1, where threads are grouped into thread blocks. The size of each thread block is user specified, and typically limited to 512 threads. Threads inside the same block may synchronize with barriers, but cannot do so with threads in

another block. The thread blocks are grouped into a grid. The execution of the thread blocks is independent from each other, and can take any order or run simultaneously. There is no hardware synchronization mechanism among the thread blocks. The only way to synchronize the thread blocks is through re-launching the kernel, which behaves as an implicit barrier. Each kernel launch corresponds to one grid. The GPU hardware allows multiple grids to concurrently occupy a single device.

The thread model of CUDA also provides transparent scalability. Thanks to the independence among all thread blocks, a grid can scale across any number of parallel cores. Figure 3.2 illustrates this concept. A grid with eight thread blocks can be mapped to a dual-core or quad-core GPU, resulting run time that is reversely proportional to the size of the

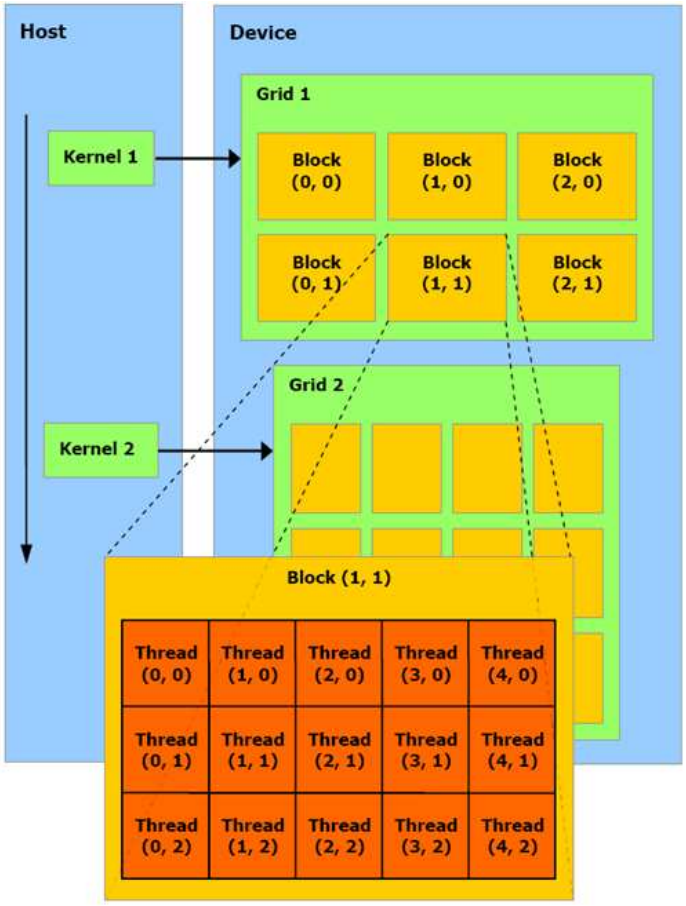


Fig. 3.1: CUDA thread hierarchy is configured in a Grid-Block-Thread model to allow programmer managing large quantity of parallel threads.

device. The GPU family produced by NVIDIA often uses the same core design throughout the entire generation, but each product differs in the number of cores and memory bandwidth to cover the high-end to low-end spectrum. The scalability of CUDA programming model ensures the performance of CUDA program on any product without the need for software reconfiguration.

The memory model of CUDA is an abstraction of the memory subsystem of GPU architecture. As evidence in Figure 3.3, similar to the thread model, the GPU memory subsystem is also arranged in a hierarchical manner. The most abundant memory is the off-chip device memory, which is slow in access latency, but wide in bandwidth. The on-chip memory is much more scarce, but is addressable and has significantly lower access latency. User controllable cache structures are also present in GPU memory system, including texture cache and constant cache. These above memory structures are built-in with the CUDA abstraction layer to allow user intervention with ease. However, the utilization of different GPU memory subsystems has a profound impact on the performance characteristic of a

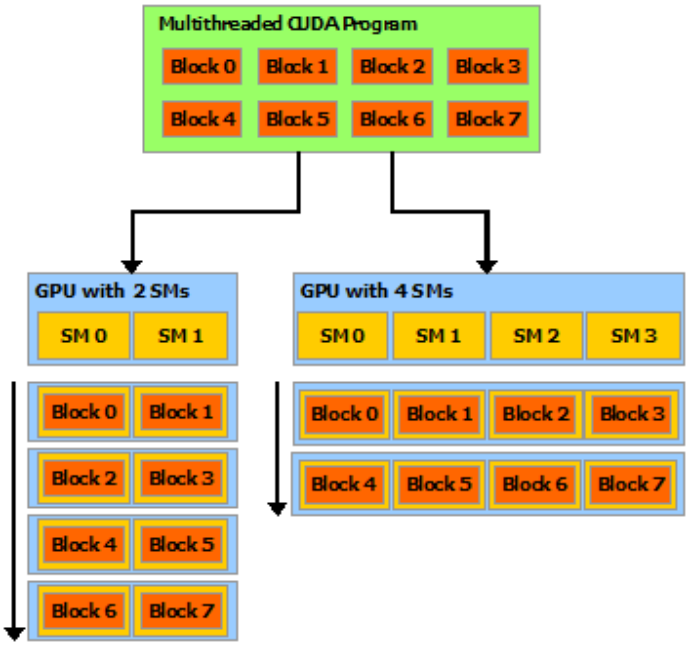


Fig. 3.2: CUDA grid can scale across different configurations of GPU hardware, and achieve the best possible workload balancing during runtime.

GPU application, and should be treated with the utmost care.

CUDA's model also provide linkages between the GPU and system memory. Typically, with the exception of embedded systems and specialized game consoles, the memory subsystem of a GPU is separated from the system memory. CUDA programs typically do not directly access the system memory, although such access is possible through mapping a block of page-locked system memory to the address space of the GPU device memory. More commonly, data communication between GPU and system memory is done with explicit copy through the PCI-E interface.

From a programmer's perspective, CUDA encompasses a C-like language for GPU programming, and a set of APIs to establish communication between the CPU (the host)

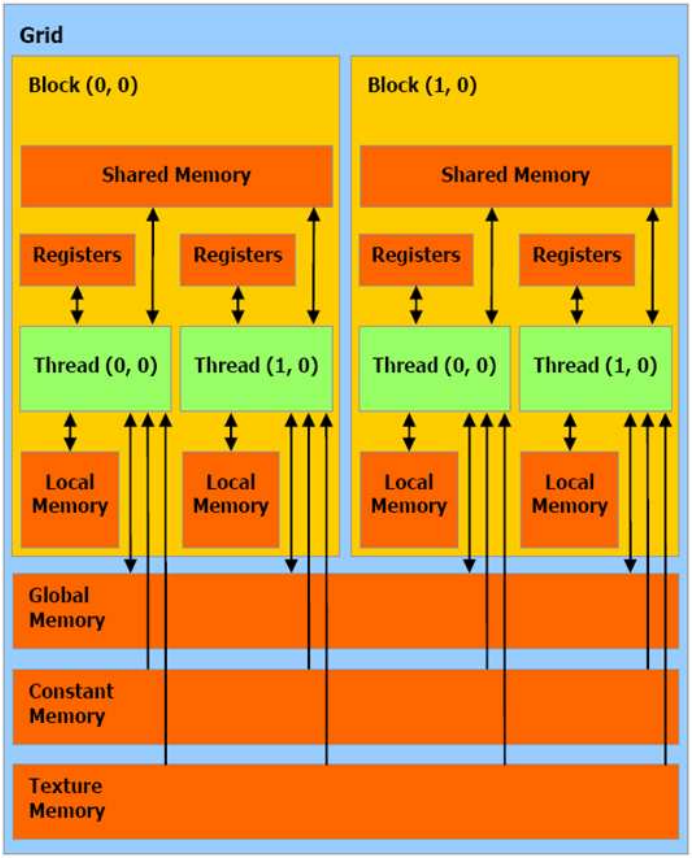


Fig. 3.3: CUDA abstracts the GPU memory subsystem in a hierarchical manner. Its different memory components exhibit diverse characteristics, which should be taken into extensive consideration for performance optimization of a GPU application.

and the GPU (the device). As a programming model, CUDA adopts a language that essentially is a super set of C, adding some features from C++. An experienced programmer can quickly adapt to the CUDA language environment. CUDA's additional semantics and symbols provide facilities to assist programmers, who do not have the knowledge of GPU internal architecture, to utilize the GPU specific features, such as thread synchronization, local and global memory addressing, etc. CUDA host side API's provide utilities such as data copy between system and GPU memory, GPU kernel configuration, and kernel launching, etc. These APIs are called by the CPU applications, and interact directly with the GPU device driver.

Chapter 4

GPU-Based Floorplanning

4.1 Floorplanning

With VLSI designs becoming increasingly complex, the floorplanning stage of physical design is critical in determining the quality of design [19, 54]. Typically, a large number of candidate floorplans are evaluated before converging to a good solution. Therefore, one of the critical design issues is choosing an efficient data structure for floorplan representation, which can ease the operations and management of floorplans. Among several alternatives, one of the most widely adopted representations is a B* tree.

4.1.1 B* Tree

B* tree is an ordered binary tree data structure that inherits all desirable properties from the O tree and comes with additional advantages [54]. While handling non-slicing floorplans, the B* tree overcomes the irregularity in tree structure, typical in O trees. B* tree has a 1-1 correspondence with its admissible placement—a compacted placement, where individual modules cannot move down or left [55]. This attributes certainty to the operational complexity involved with B* trees and enables flexible realization allowing usage of either static or dynamic memory structure. On choosing a static structure, operations like insertion, deletion, and search can be done in linear time.

Typically, simulated annealing, a generic probabilistic optimization algorithm, is applied on the B* tree for floorplanning. The different moves that can explore the entire solution space in a simulated annealing based floorplanning algorithm using B* trees are: (1) rotating a module; (2) moving a module to a new location; and (3) swapping two modules. Each of the three moves are primarily based on the insertion and deletion operations of

the B* tree data structure, which automatically preserves the validity of the new floorplan.

4.1.2 Simulated Annealing

Figure 4.1 illustrates the simulated annealing algorithm using a B* tree. The algorithm starts with an initial B* tree representation of the floorplan and randomly chooses and applies one of the three moves to the tree. Next, the objective of the newly generated B* tree (area and/or wire length) is evaluated and the move is accepted or rejected based on the simulated annealing schedule. These steps are repeated several times until a satisfactory solution is obtained.

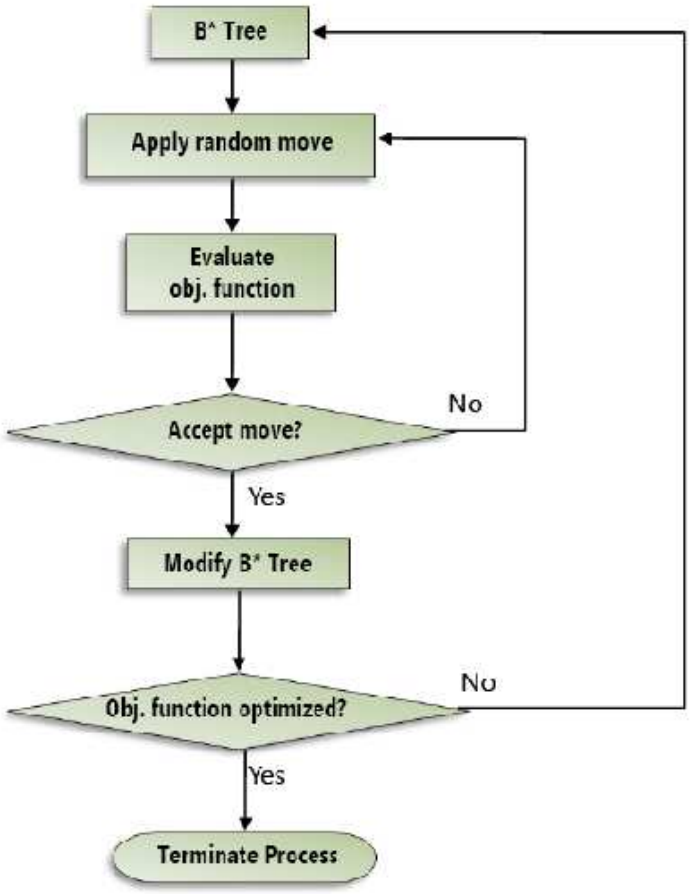


Fig. 4.1: Floorplanning with CPU.

4.1.3 GPU Design Issues

A GPU is a massively multi-threaded multiprocessor system that excels in single-instruction multiple-thread (SIMT) style concurrency. Threads in a GPU are organized in a two-level hierarchy. The lower level consists of a group of programmer specified threads forming a block. The higher level consists of a collection of blocks forming a grid. The GPU hardware facilitates extremely fast context-switching between threads, thereby allowing efficient overlap of long latency operations of one thread with computations from another. For NVIDIA GPUs, the CUDA programming model provides a platform for non-graphics applications to exploit the computation bandwidth of the GPU.

The key to effectively using the CUDA interface is understanding the memory hierarchy of a GPU. The GPU memory hierarchy consists of: (1) device or global memory, (2) shared memory, and (3) texture memory. Unlike in commodity multi-core systems, programmers can explicitly manage data in these memories. Carefully managing the program data structures in these memory modules is paramount for speeding up applications using the GPU.

The *device memory* is the global memory for the GPU, which is accessible from all the GPU threads, as well as the CPU. It has high access bandwidth but its access latency is high. *Shared memory* is the local memory that is shared by all the threads in a block. Therefore, changes made in this memory are visible to all the threads within a block, but invisible to other blocks. Its access latency is low, but its storage space is limited, usually 16KB per block. *Texture memory* is for explicitly declared read-only data, and is accessible to all threads. Since it is also automatically cached, the latency of accessing such memory is low.

At a high-level, the CPU uses the GPU as an accelerator for a specific function, known as the *kernel function*. The CPU typically will send a kernel function to the GPU, along with data for manipulation. All threads execute the same kernel function. However, depending on the kernel specification, different threads may perform different data manipulation as desired by the programmer.

4.2 GPU Floorplanning: Algorithm Overview and Specification

This section presents an overview of GPU floorplanning algorithm with the detailed specifications. As mentioned earlier, to parallelize the floorplanning algorithm, it is essential to break the dependency chain. Analyzing the general sequential framework from Figure 4.1, it is observed that multiple concurrent moves can be applied on a given floorplan. Fundamentally, this strategy breaks the dependency chains in the sequential algorithm, and explores the solution space in a completely different manner.

Based on the above insight, Figure 4.2 presents a high-level overview of a parallel floorplanning algorithm. The initial floorplan represented as a B* tree is selected in the CPU. Several concurrent GPU threads are then launched, after copying the necessary state from the CPU to the GPU. B is the number of parallel threads launched to perform concurrent moves, and the later sections (Sections 4.4 and 4.4.5) will further explore its significance towards performance later . Each parallel GPU thread applies a separate move, and evaluates the objective function for the resulting floorplan. Consequently, several moves are concurrently evaluated in the GPU. The CPU then inspects the objective evaluations, and accepts one of the moves evaluated during the concurrent phase. The algorithm repeats these steps unless a stopping criteria is met. While conceptually simple, there are several key design issues that needs to be addresses throughout the rest of this chapter.

4.2.1 Algorithm Specification

Table 4.1 presents a detailed algorithmic specification for the proposed GPU floorplanning algorithm. The initial phase (steps 1-5) is executed on the CPU. The CPU creates an initial B* tree implementation and copies the B* tree and its attributes to the GPU device memory. The CPU subsequently launches B parallel thread blocks in the GPU.

The next phase (steps 6-10) is the concurrent phase in the GPU. Each block copies the B* tree to its own shared memory. Since the tree is modified in place (shared memory) when a move is applied, only one thread can perform a move within a thread block. After computing the new floorplan, this thread evaluates the objective of the updated floorplan and stores the objective in the device memory.

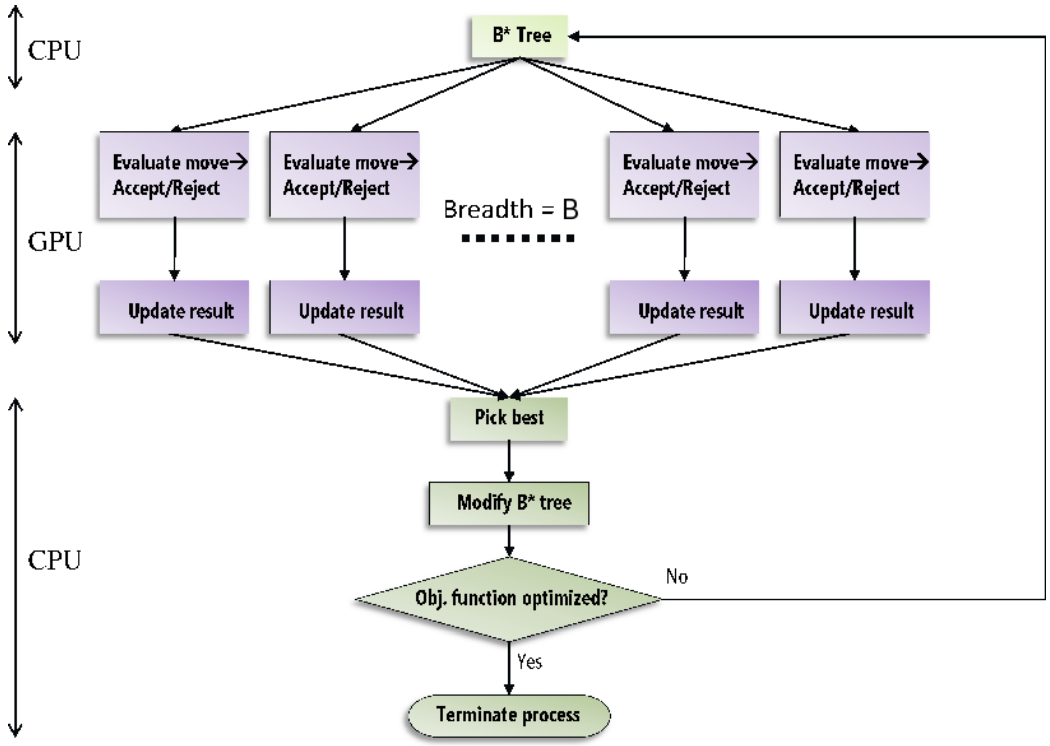


Fig. 4.2: Floorplanning in CPU-GPU.

Finally, in the last phase (steps 11-14), the B objectives are copied back from the device memory to the host memory. The CPU now picks the best move from the candidate objectives, and makes a single update on its B* tree applying the best move. Steps 3-14 are repeated until the stopping criteria is satisfied. The stopping criteria used is identical to the sequential version (exploring a fixed number of moves that depend on the number of modules).

4.2.2 Implementation: CPU-GPU Dataflow

Figure 4.3 shows the dataflow between the CPU (host) and the GPU (device). In step 1, the B* tree structure of a floorplan along with a few circuit related constants are copied to the device memory. The constants consist of details of a module (e.g. width and height) that do not change during runtime. Subsequently, multiple thread blocks, each consisting of a single thread, copy the tree to their own shared memories concurrently (step 2). In

Table 4.1: Throughput optimized floorplanning algorithm.

ALGORITHM GPU_Floorplanning:
Input:
1. A given circuit with n modules
Output:
1. A floorplan that optimizes objective (area, wire length)
Begin
1. Read input circuit
2. Construct initial floorplan in a B* tree form
3. **while** stopping criteria not met **do**
4. Copy tree and attributes to GPU device memory
5. Launch B parallel thread blocks
6. | Copy tree and attributes to shared memory
7. | Select and perform move
8. | Modify tree /*local copy in shared memory*/
9. | Evaluate objective
10. | Write objective in GPU device memory
11. Copy B objectives from GPU device memory to host memory
12. Pick best move
13. Modify tree with best move
14. **end** /*end while loop*/
End

step 3, different moves are explored in different thread blocks, and the objective function is evaluated and stored in the device memory (step 4). Finally, the objective results are copied back to the host memory (step 5).

4.3 Preliminary Results

This section describes the methodology and initial results obtained through the proposed algorithm.

4.3.1 Methodology

The hardware platforms that are used to evaluate the algorithm are listed in Table 4.2. There are two systems: (1) a 2.40GHz Intel Core 2 Quad Q6600 Kentsfield processor with a NVIDIA GeForce GTS 250 G92 GPU; and (2) a 2.27GHz Intel Xeon E5520 Nehalem-EP processor with four NVIDIA Tesla C1060 GPUs. CUDA 2.3 SDK is used for the GPU

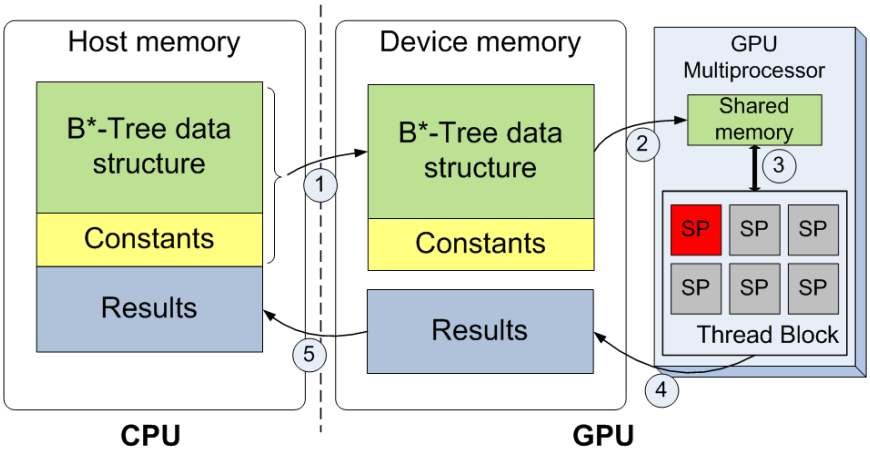


Fig. 4.3: Dataflow between CPU and GPU. Active thread is shown with a darker shading within a thread block.

implementations. The NVIDIA GPU driver version is 190.18 beta.

The Parquet Floorplanner from the UMpack suite of VLSI-CAD tools is used as the sequential version of floorplanning [18]. The GPU version is implemented by suitably modifying this code to incorporate both algorithmic transformations as well as GPU specific CUDA code. The run time of the floorplanning program is used as a benchmark to evaluate the efficiency of the algorithm. A CPU timer is used to measure the distribution of the total time spent in the CPU and the GPU. Within the GPU, the GPU timers are used to measure time spent in different execution stages (described later).

Table 4.2: CPU and GPU specs.

	NVIDIA Geforce GTS 250 G92	NVIDIA Tesla C1060	Intel Core 2 Q6600	Intel Xeon E5520 Nehalem-EP
No. of multiprocessors	16	30	1	2
No. of cores	128	240	4	8
Max. bandwidth	58 GB/s	73 GB/s		

4.3.2 Results

Table 4.3 presents the preliminary results on MCNC benchmark circuits. Each benchmark is run several times, and the best run for both the algorithms is reported in the table. The number of blocks in each floorplan, shown in parentheses, indicates the size of the floorplan. For each benchmark, the execution time of the sequential CPU version is presented, as well as the speedup achieved using parallel GPU implementation on two different machines. Additionally, the results also show the relative quality of the solution achieved in the GPU, compared to the CPU. Without loss of generality, for the purpose of this paper, only the area of the floorplan is considered as the objective function. The algorithm can be trivially modified to account for other objectives such as wire length and temperature. Thus, the results measure the quality by comparing the area of the floorplan resulting from the two algorithm versions.

Across all benchmarks except *ami49*, the parallel GPU floorplanner shows promising speedup compared to the sequential version. For example, *hp* and apte show 31% and 27% speedups, respectively. However, for *ami49*, the parallel algorithm takes nearly double the time it takes to run in the sequential mode. Interestingly, *ami49* is also the largest benchmark among the set of circuits that are considered here.

It is noticeable that the solution quality varies significantly depending on the benchmark. While smaller benchmarks show comparable solution quality in the parallel version, larger benchmarks tend to suffer a modest loss of solution quality. For example, the solution achieved in *ami33* is about 7% worse. To understand these results better, it is important

Table 4.3: Preliminary results. Number of modules in a benchmark is indicated in parentheses.

Benchmark	CPU run time	Geforce GTS 250 Speedup	Tesla C1060 Speedup	Quality
xerox(10)	0.388s	1.29X	1.56X	−1.2%
ami49(49)	8.186s	0.53X	0.54X	−5.37%
ami33(33)	3.770s	1.02X	1.02X	−7.26%
apte(9)	0.322s	1.27X	1.53X	−1.28%
hp(11)	0.464s	1.31X	1.61X	−2.1%

to inspect the execution time in the parallel GPU floorplanner in more details and show where most of the run time is spent.

4.3.3 Where Does Time Go in a GPU?

Figure 4.4 shows the breakdown of the total execution time of the GPU floorplan implementation for *ami49* on G92. Results for other benchmarks, as well as those on Tesla C1060, are similar to this breakdown. First, it is clear that most of the computation is able to be offloaded to the GPU, as only 1.5% of the total time is spent in the CPU. Second, the bulk of the total time is spent in executing the kernel function in the GPU (98%), with negligible time spent in data communication between the CPU (host) and the GPU (device).

To take a closer look at different execution components within the GPU, we divide the kernel execution run time into another four parts:

- Device to shared memory copy,
- B* tree move,
- Objective evaluation,
- Write to device memory.

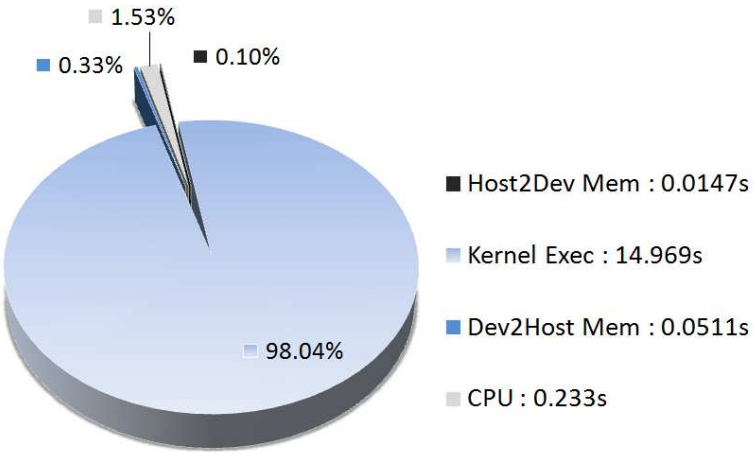


Fig. 4.4: Execution time breakdown for *ami49*.

Figure 4.5 shows the run time breakdown of the kernel execution. Interestingly, most of the time in the GPU kernel is spent in copying data from the device memory to the shared memory (97%). Time spent to perform the computation to evaluate the objective function for the newly generated floorplan constitute 3% of the total run time.

Together, Figures 4.4 and 4.5 indicate that optimizing data copy from the GPU device memory to the shared memory is likely to give substantial benefits. The next section describes the efforts in optimizing the GPU floorplan implementation.

4.4 Performance Optimization

This section describes several techniques that have been explored for optimizing the GPU floorplanning algorithm, all of them targeting the time to copy data from the device memory to the shared memory (step 2 in Figure 4.3).

4.4.1 Limiting Data Copy to Shared Memory (OPT1)

Shared memory allows very fast operations, and it is generally beneficial to copy all the accessory data structures that remain unchanged, to the shared memory. However, many of these data structures (e.g. block width and height) are accessed infrequently during evaluating a candidate move. In the context of floorplanning, the additional latency of

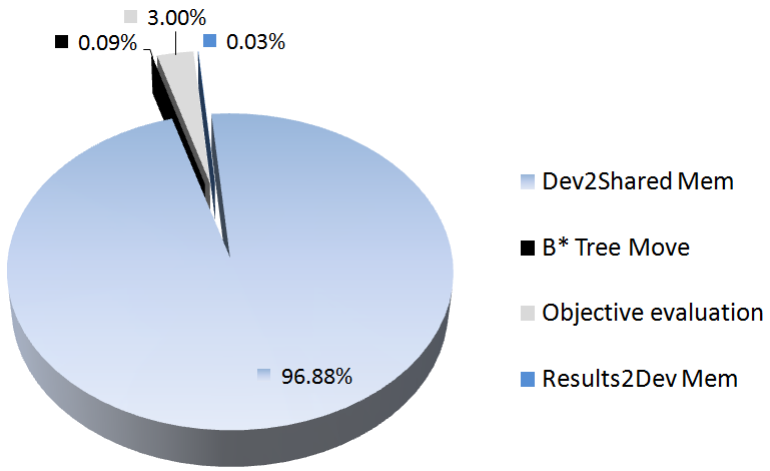


Fig. 4.5: GPU execution time breakdown for *ami49*.

infrequent access to the device memory for these data items is substantially lower than the cost incurred to copy them to the shared memory. Therefore, in the first optimization (*OPT1*), copying of these constants to the shared memory is avoided.

4.4.2 Parallelize Device to Shared Memory Copy (OPT2)

When applying a move on a floorplan, the B* tree representation of the floorplan is modified in the shared memory. As a result, only use one thread per block is utilized to perform the move and subsequent objective evaluation (Section 4.2.1). However, when copying data from the device memory to the shared memory, multiple threads *can* be employed in a block to concurrently copy data items. Although device memory access is slow, its bandwidth is very high in a GPU, thereby easily accommodating concurrent thread accesses.

Figure 4.6 illustrates this technique, where we concurrently copy data from the device memory to the shared memory. During the copy phase, each thread block employs several concurrent threads to copy the data. After the copy phase, only one thread performs the necessary computation for evaluating a move on the floorplan. Interestingly, the number of threads utilized to perform this copy has an impact on the speedup achieved through this technique.

Figure 4.7 shows this tradeoff between number of threads and speedup achieved in *ami49*. The data is normalized to the execution time where a single thread is used for copying. As the number of threads is increased, concurrent device memory access is increased, leading to a speedup. However, after a certain point, the device memory bandwidth is saturated, and adding more threads does not lead to additional speedup. G92 machine shows maximum relative speedup with 32 threads, while the Tesla C1060 shows maximum speedup with 256 threads. The latter supports higher memory bandwidth, thereby showing better speedup with higher threads than G92.

4.4.3 Memory Access Coalescing (OPT3)

One way to exploit the high bandwidth from the device memory is to coalesce memory

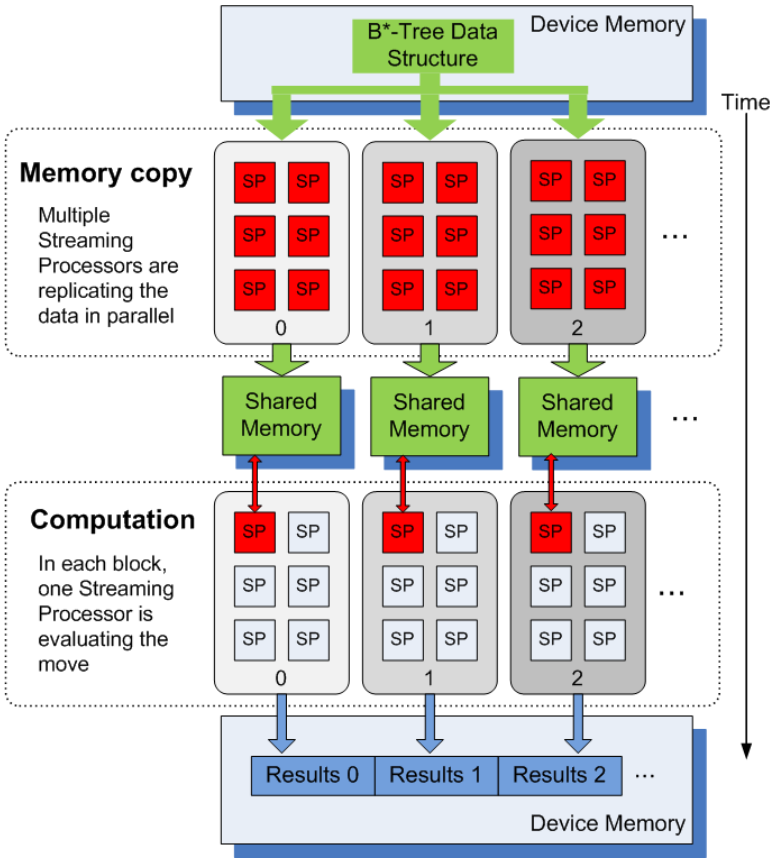


Fig. 4.6: Parallelizing data movement between shared memory and device. Active threads on streaming processors (SP) are shown with darker shading within a thread block.

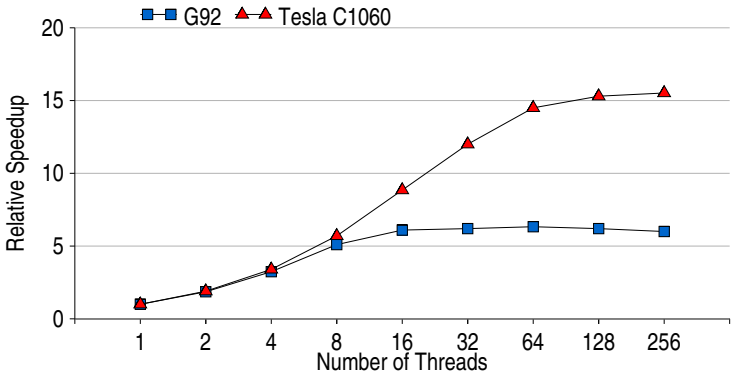


Fig. 4.7: Speedup comparison with concurrent threads to copy data.

accesses, i.e. let a half warp (16 threads) access a contiguous region of the device memory, in which case 16 data elements can be loaded in one transaction. Data structure like the B*

tree representation is frequently copied between the device and the shared memory. Hence, coalescing device memory access is essential to the overall performance.

Irrespective of how the B* tree floorplan representation is structured and interpreted, its data can be contained within a flat and dense memory space. Consequently, it is possible to copy the entire data as a memory chunk. This technique provides ample flexibility to handle other floorplan representations as well.

In the third optimization, the access to the floorplan representation is coalesced by treating it as a flat chunk of data. For example, copying sparse data elements from arrays of structures is avoided, which causes non-coalesced device memory access patterns as shown in Figure 4.8(A). The data copying scheme is configured such that a half warp can fetch a segment of the data chunk within one transaction. As shown in Figure 4.8(B), the segment contains 16 contiguous data elements, which are fetched simultaneously by 16 threads. Hence, the access to device memory benefits from a much higher bandwidth. Moreover, the size of each data element is required to be 4, 8, or 16 bytes to coalesce device memory access. The 4-byte configuration is chosen to ensure correct alignment for the B* tree representation of the floorplan.

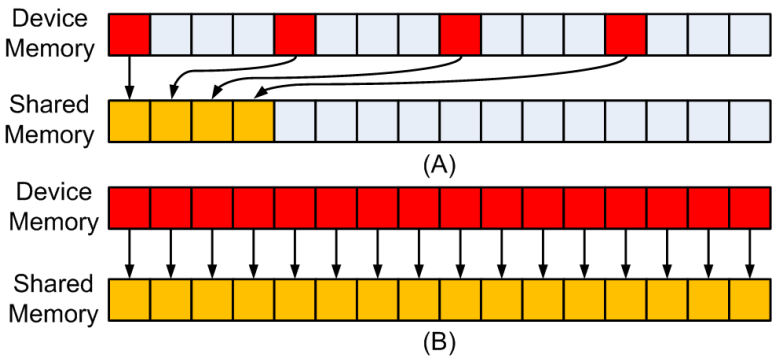


Fig. 4.8: (A) Non-coalesced device memory access. Only four threads are active in the transaction. (B) Coalesced device memory access. Sixteen data elements are copied in one transaction.

4.4.4 Results

Figure 4.9 shows the GPU execution time breakdown after applying all the optimizations. Comparing to Figure 4.5, which shows the GPU execution time before optimization, a dramatic reduction can be noticed in the time spent in data communication between the shared memory and the device memory. Only 22.3% of the overall time is spent in copying data into the shared memory. Once the major performance bottleneck is reduced, there is a relative increase in other computation components such as Evaluate, as expected.

Figures 4.10 and 4.11 show the speedup achieved after applying the performance optimization techniques on G92 and Tesla C1060, respectively. All the speedups are relative to the sequential version in the CPU. For each benchmark, four bars are presented. From left these represent results for the preliminary version, only *OPT1*, applying both *OPT1* and *OPT2*, combining all three optimizations (*OPT1, OPT2, OPT3*), respectively.

Each optimization gives significant speedup, but combining them together gives us dramatic speedup. For example, *ami49* achieves an overall speedup of nearly 17X on the Tesla machine, compared to the sequential version. Larger circuits yield greater speedup, as the data copy cost is better amortized by concurrently computing costlier move operations. Across the range of circuits, the GPU floorplan implementation achieves **9.4–14.6X** and **10–17X** speedups on G92 and Tesla C1060 machines, respectively. Given the long dependency chains and iterative nature of the original sequential algorithm, these improvements

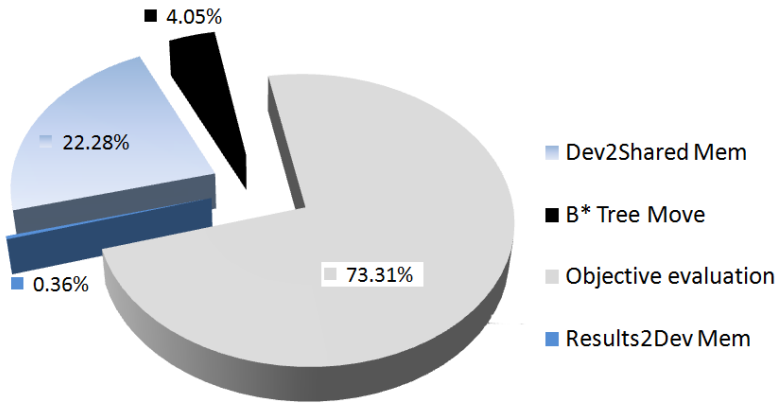


Fig. 4.9: GPU execution time breakdown for *ami49* after optimization.

are significant.

4.4.5 Quality vs. Speedup

The GPU floorplanning algorithm achieves dramatic performance improvement, while delivering solution quality comparable to the sequential version. However, it is interesting to explore techniques that can improve the solution quality even further, and investigate their impact on runtime performance.

So far, the GPU floorplanner evaluates about the same number of moves as its CPU counterpart. However, the inherent nature of exploring the solution space is different in the GPU floorplanner. In a GPU, the algorithm attempts to cover more breadth by applying many concurrent moves on a given intermediate floorplan. The sequential version, however, attempts to cover more depth by performing several successive moves. The slight loss in solution quality, especially for large circuits, in the GPU version stems from this fundamental difference in solution space exploration.

One way to mitigate this problem in the GPU floorplanner is to simply explore more moves than the sequential version to increase the number of intermediate floorplans on which concurrent moves are applied. However, evaluating more moves can increase the run time, and it is important to inspect the improvement in solution quality achieved through this technique. Figure 4.12 shows this tradeoff by plotting the speedup against the solution

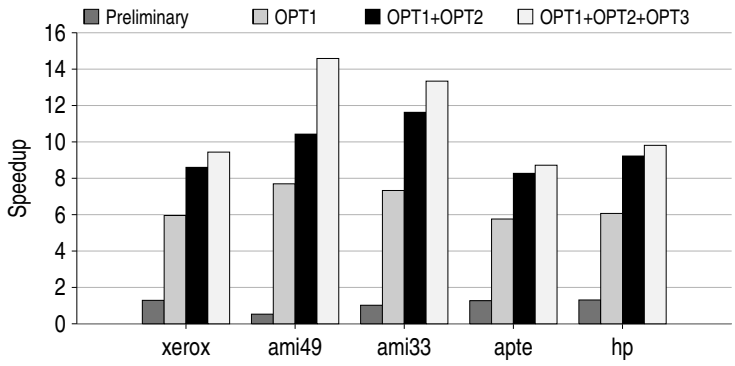


Fig. 4.10: Speedup achieved using G92.

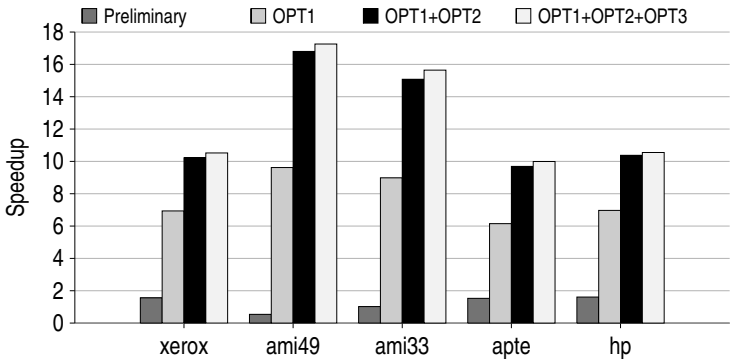


Fig. 4.11: Speedup achieved using Tesla C1060.

quality relative to the sequential version. Indeed, it is clear that sacrificing speed can yield better quality. For example, *ami33* can reach within 2% of the quality achieved in the sequential version, if less than 2X speedup is accepted.

As evident from Figure 4.12, improvement in solution quality comes at a significant run time cost. A key question then is how to restructure the GPU floorplanning algorithm that would allow us to improve the solution quality without sacrificing the speedups that have achieved.

4.5 Algorithm Restructuring to Improve Solution Quality

The slight loss in solution quality stems from exploring more breadth in the GPU

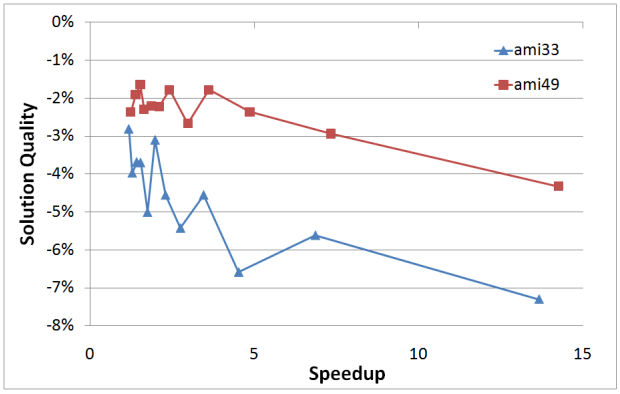


Fig. 4.12: Solution quality vs. speedup tradeoff.

algorithm. Therefore, it is instructive to investigate if it is possible to restructure the algorithm to strike a balance between exploring depth (sequential version) and breadth (GPU version). In the current version, as explained in Figure 4.2, each GPU thread applies a single move on the floorplan during every GPU function call from the CPU. Instead of applying a single move, each GPU thread is allowed to apply multiple successive moves. In this fashion, each GPU thread will be able to explore a larger part of the solution space.

4.5.1 Algorithm Overview

Figure 4.13 shows this new restructured algorithm. The algorithm is designed in a generic way such that the user can alter the depth (D) and breadth (B) of the algorithm in runtime. While applying D successive moves, each GPU thread can choose to either accept or reject a given move, similar to the sequential CPU algorithm. The acceptance of a move depends on whether the move was successful in achieving a better objective. In addition, a sub-optimal move (uphill) is accepted with a random probability that decreases as the algorithm progresses, modeling the simulated annealing behavior. Hence, there are D successive moves in each thread, and B such threads in parallel in the GPU. The solution quality of the resulting floorplan will depend on the choice of the variables B and D, respectively.

4.5.2 Implementation

The implementation of the new algorithm differs considerably from the implementation that was discussed so far. The entire floorplan data structure (B* tree) is now completely resident in the GPU, thereby considerably reducing the GPU-CPU data communication. The CPU guides successive GPU iterations by selecting the best candidate floorplan.

However, this implementation also results in an increase in data copy operation between the shared memory of a thread block to the device memory. In successive GPU iterations, a GPU thread may start from the intermediate floorplan computed in another GPU thread. Thus, to accomplish this data communication between multiple GPU threads (belonging to different thread blocks), each thread block now copies the intermediate floorplan represen-

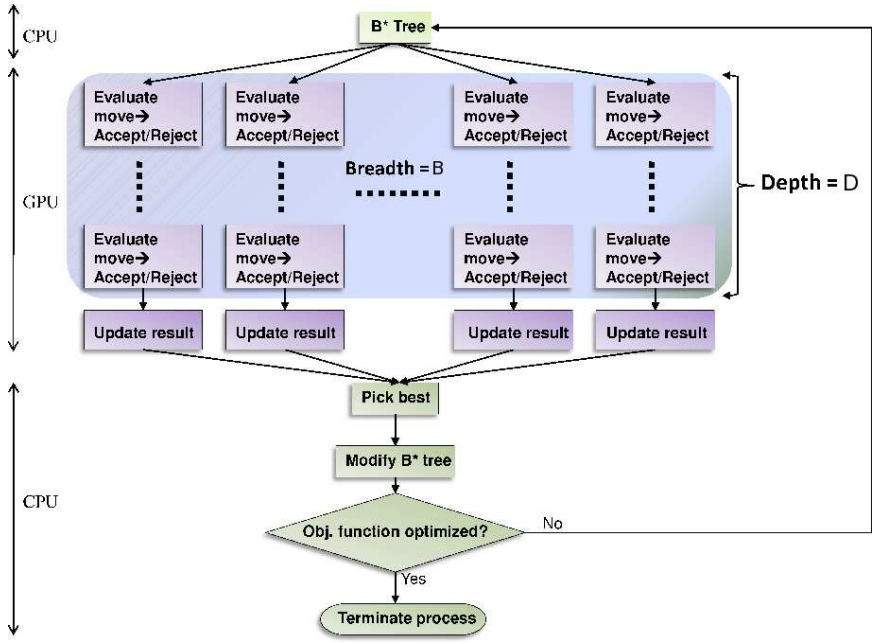


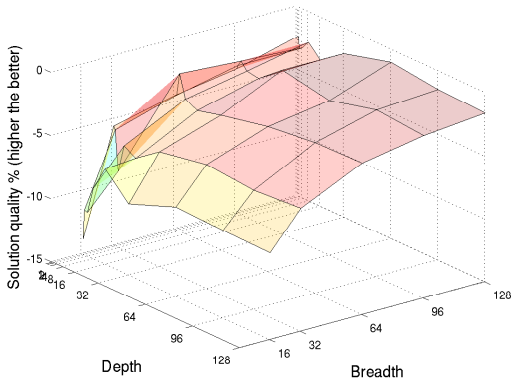
Fig. 4.13: New CPU-GPU floorplanning algorithm.

tation from its shared memory to the device memory. In summary, the modified algorithm implementation largely eliminates the traffic between CPU and GPU, but increases the data communication between the GPU device memory and shared memory.

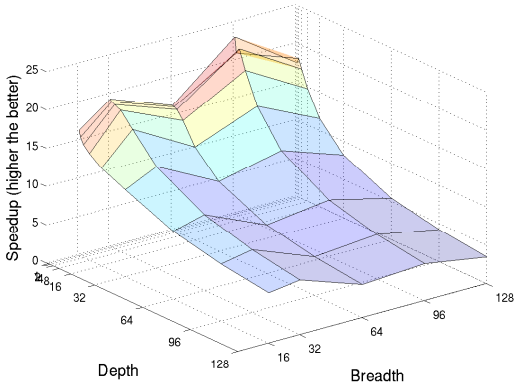
4.5.3 Results

The results from the implementation of the modified algorithm is presented in the section. Since the algorithm can now tune both the breadth (number of concurrent threads) and the depth of exploring the solution space, the results are shown in 3D contour plots. Figure 4.14 shows the solution quality and speedup for three benchmarks: *hp*, *ami33*, and *ami49*, respectively. Figure 4.14 shows the results achieved in the G92 machine, and the Tesla C1060 follows the same pattern seen in this figure.

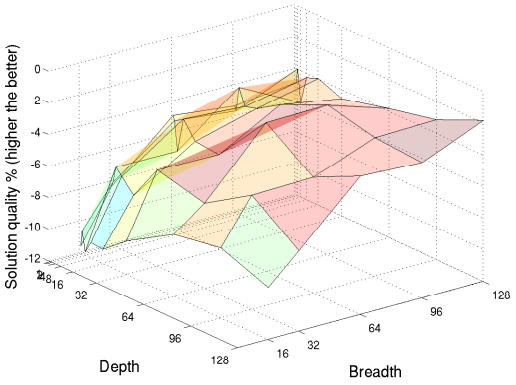
It is observable that the algorithmic modification is particularly effective in improving the solution quality across all benchmarks. In *hp*, there is an *improvement in solution quality over the CPU version.* Exploring successive moves can better exploit the aggregate computation bandwidth for the smaller overall solution space (due to the smaller number



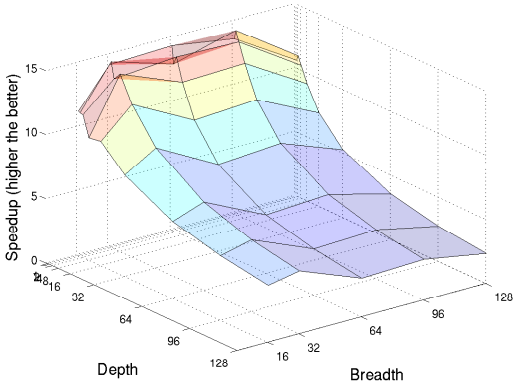
(A) Solution quality for *ami49*



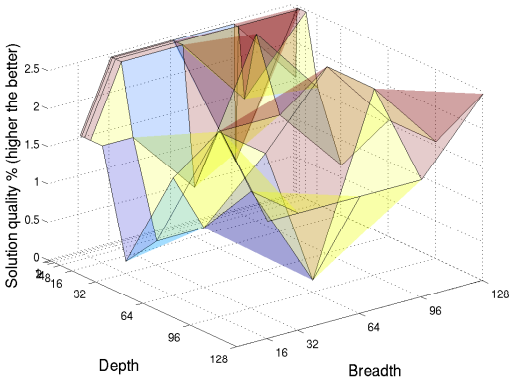
(B) Speedup for *ami49*



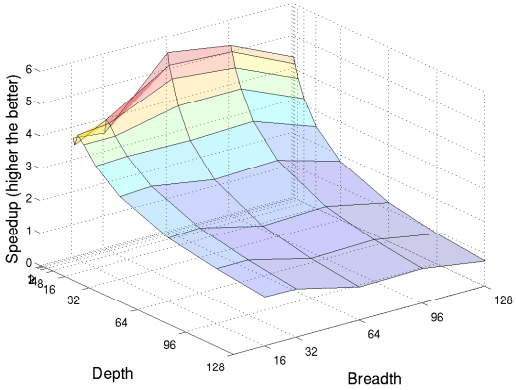
(C) Solution quality for *ami33*



(D) Speedup for *ami33*



(E) Solution quality for *hp*



(F) Speedup for *hp*

Fig. 4.14: Depth vs. breadth: Solution quality and speedup for different benchmarks.

of blocks), resulting in better quality. The modification achieves up to 2.5% better solution quality, especially for smaller depth. The speedup achieved in *hp* is also significant: nearly 5.6X for a depth of 1 and breadth of 64. Notice that this speedup is lower than that achieved previously (Figure 4.10). This result is due to increased shared memory to device memory copy where the B* tree representing the floorplan is copied from the shared memory of each thread block to the device memory. Unlike *hp*, both *ami33* and *ami49* show a distinct pattern in solution quality and speedup based on the breadth and the depth. There is a substantially better solution quality when increasing the depth. However, better speedups are observed with smaller depth and larger breadth. Unlike *hp*, there is an increase in maximum speedup achieved in *ami49*. Since *ami49* is the largest benchmark, elimination of floorplan data communication between GPU and CPU outweighs the increase in shared memory to device memory communication, resulting in an overall speedup.

There is a tradeoff involved in achieving speedup and solution quality. A high speedup usually degrades the quality of the solution and vice versa. The speedup and solution quality of the algorithm are two very different attributes, and hence it is necessary to normalize them to develop a combined metric. For this purpose, a metric $\mathcal{SQ}$ is defined to obtain a tradeoff between the speedup (SP) and the solution quality (Q), and is used to select the desirable depth and breadth of the proposed algorithm. $\mathcal{SQ}$ is obtained by adding the speedup and solution quality normalized with their corresponding maximum values as shown in Equation (4.1).

$$\mathcal{SQ} = \frac{Speedup}{|Max\ \ Speedup|} + \frac{Quality}{Max\ \ |Quality|} \tag{4.1}$$

Tables 4.4 and 4.5 present depth and breadth combinations that achieve best speedup, best quality and best combined metric ($\mathcal{SQ}$), respectively. Across all benchmarks, there is a **4–23X** speedup in G92, and **4–30X** speedup in Tesla C1060 machine. Using the combined metric, It has resulted in excellent solution quality while maintaining remarkable speedup. For example, *ami49* achieves **23X** and **30X** speedup, while delivering marginally worse

(-3.6%) and (-2.06%) solution qualities in G92 and Tesla C1060, respectively.

4.6 Adapting Annealing in GPU

This section discusses the adaptation of the conventional annealing technique in the GPU. Typically, simulated annealing algorithms employ a number of sequential moves for every temperature state. The number of moves employed is proportional to the floorplan size. The number of moves in each temperature state is bounded by integer Z. In the Parquet annealer, for example, Z is defined as $Z = 4 \times sizeof(floorplan)$. Once the number of sequential moves exceeds Z, the algorithm moves to the next temperature state.

In the GPU annealing algorithm, the number of moves applied at a given temperature state is determined by the depth and breadth used during execution. The GPU annealer completes $M = Depth \times Breadth$ number of moves for every kernel launch. Therefore, each temperature state drives $n \cdot M$ moves, where n is the number of kernel launches in this state. Since, $n \cdot M \geq Z$ is satisfied, the GPU typically performs more moves than the CPU. Moreover, the number of total moves can vary with the chosen breadth and depth, resulting in a somewhat unfair comparison.

Table 4.4: Tradeoff in solution quality and speedup (G92). D and B represent the depth and breadth, respectively.

Benchmark	Best speedup			Best quality			Best $\mathcal{SQ}$		
	{D,B}	SP	Q	{D,B}	SP	Q	{D,B}	SP	Q
xerox	{1,64}	4.79	0.42	{2,32}	3.4	1.38	{8,64}	3.59	1.37
apte	{1,64}	4.08	-0.83	{4,96}	3.42	0	{4,96}	3.42	0
hp	{1,64}	5.52	2.46	{1,64}	5.52	2.46	{1,64}	5.52	2.46
ami33	{2,32}	14.45	-8.01	{128,128}	2.32	-1.86	{4,96}	13.47	-4.09
ami49	{1,96}	22.94	-3.6	{64,128}	5.35	-1.08	{1,96}	22.94	-3.6

Table 4.5: Tradeoff in solution quality and speedup (Tesla C1060). D and B represent the depth and breadth, respectively.

Benchmark	Best speedup			Best quality			Best $\mathcal{SQ}$		
	{D,B}	SP	Q	{D,B}	SP	Q	{D,B}	SP	Q
xerox	{2,64}	4.99	0.45	{4,16}	2.77	1.37	{8,64}	3.33	1.37
apte	{1,64}	4.19	-1.23	{8,16}	2.19	0	{1,96}	3.33	0
hp	{1,64}	6.04	0.65	{1,32}	3.1	2.46	{1,96}	5.96	2.46
ami33	{1,64}	17.22	-6.31	{96,96}	2.66	-2.47	{1,64}	17.22	-6.31
ami49	{2,128}	30.68	-3.67	{128,128}	6.4	0.05	{1,128}	29.97	-2.06

To address this problem, a new temperature scheme is implemented for the GPU annealing algorithm. The objective is to maintain identical temperature spectrum in the GPU implementation. This goal is achieved by making sure that the temperature drop in successive kernel launches are based on number of moves made in each kernel launch. First, the function used by the CPU annealing algorithm to calculate the decreasing temperature is shown in Equation (4.2):

$$t' = t \times f(t), \tag{4.2}$$

where $f(t)$ is a function of the current temperature t, $f(t)$ has a range of $[0, 1]$, and t' is the new temperature. Second, the GPU annealing temperature scheme is modified, and calculate t' using Equation (4.3):

$$t' = t \times f(t)^{\frac{n \cdot M}{Z} \times \frac{1}{\Omega}}, \tag{4.3}$$

where Ω is a user parameter that tunes the total number of moves by the GPU annealing algorithm, in terms of the *ratio of the number of total GPU and total CPU moves*. Parameter Ω provides an additional degree of freedom in tuning the GPU annealing algorithm. Thus, the total number of moves employed remains fixed across all depth and breadth combinations.

4.7 Design Space Exploration

This section presents a design space exploration in multiple degrees of freedom afforded by the GPU floorplanning algorithm.

4.7.1 Solution Selection Through Binary Tree Reduction (BTR)

The breadth of the GPU floorplanning algorithm allows exploring multiple floorplans at the same time, or searching for better quality by applying different moves in different threads on a single floorplan. The former method horizontally expands the solution space, hence avoiding the algorithm being trapped at a local minima. The later method makes the algorithm more greedy, allowing it to rapidly approach a minimum solution. Under

different contexts and stages of annealing, either methods can increase the efficiency of the GPU floorplanning algorithm.

A binary tree reduction algorithm is implemented to combine both these techniques. Figure 4.15 illustrates the process of selecting solutions through binary tree reduction. The algorithm starts with 16 solutions. Given a reduction width of 2, the first row solutions are divided into eight groups, which are circled within colored background. The best solution in each group, illustrated as a white block, is selected and duplicated within the group. Hence, the solutions in the second row are generated using the best solutions from the first row (the process is shown as an arrow). The selection and duplication processes are then repeated for successive iterations with new values of reduction width.

The reduction width grows with power of 2 along with decreasing temperature. A smaller reduction width leads to a larger solution space exploration. As shown in Figure 4.15, the reduction width starts with 2, allowing eight solutions to be explored simultaneously. On the other hand, larger reduction width renders more concentrated exploration. With a reduction width of 16, only one solution is preserved, and explored subsequently. The reduction width is reset once it grows beyond the breadth of the GPU annealing. Therefore, the greediness of the GPU annealing is repeatedly altered by recursively applying the binary tree reduction along the direction of decreasing temperature.

4.7.2 Annealing Diversity in GPU Threads (ADT)

This technique introduces diversity to different GPU annealing threads by applying divergent temperatures in parallel threads. Variation in temperature causes each annealing thread to differ in the degree of greediness. The threads with scaled-up temperature bear more acceptance to uphill moves, while the scaled-down threads are predominantly downhill. The temperature of each thread is scaled based on the initial temperature t of the iteration, using the following Equation (4.4):

$$t' = t \times C_1 \times (C_2 \times \sin(\frac{blk_{Idx}}{blk_{Dim}} \cdot \frac{\pi}{2}) + 1), \tag{4.4}$$

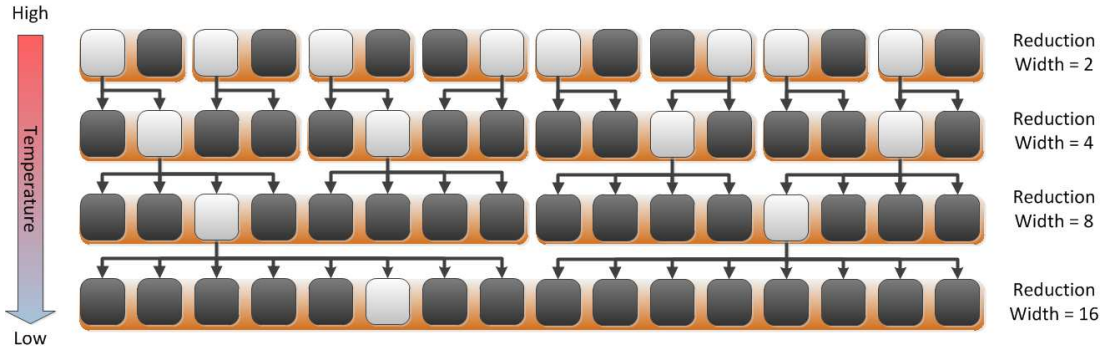


Fig. 4.15: One iteration of binary tree reduction.

where blk_{Dim} is the dimension of the CUDA blocks, which is the number of annealing threads of this case. blk_{Idx} is the index of the annealing thread, and $blk_{Idx} \in [1, blk_{Dim}]$. t is the initial temperature of the iteration. t' is the temperature of the annealing thread with block index of blk_{Idx}. C_1 and C_2 are constants, and given as $C_1 = 0.1$ and $C_2 = 20$.

4.7.3 Dynamic Depth (DD)

One of the major issues with the GPU annealing algorithm is the insufficient amount of moves during the uphill stage of the annealing. The GPU algorithm employs the same amount of moves as the sequential CPU implementation, but they are spread out into multiple threads. Therefore, the number of moves in each GPU thread is significantly lower. The brief uphill stage may cause later annealing to be trapped in a local minima, degrading the final solution quality, especially for large benchmarks.

To address the problem of insufficient uphill moves, the dynamic depth is implemented in the early uphill stage of annealing. Typically, before the temperature drops below half of the initial temperature, the GPU annealing algorithm forces 8X depth in each iteration. Interestingly, the uphill stage is very short in the whole annealing process. For GPU annealing, it typically takes between 1% to 2% of all iterations. Therefore, the additional moves at this stage only increase the overall number of moves by a small fraction. On the other hand, to avoid performance penalty, this technique aggressively avoids objective evaluation after each move (evaluate move in Figure 4.13) to allow 100% acceptance rate.

4.7.4 Results

This section presents results from the design space exploration. The impact of the design space exploration is shown in Table 4.6. Across all schemes, breadth and depth are set to 16 and 96, respectively, and Ω is set to 2 (2X as many moves as the CPU). Results using the earlier implementation (Section 4.5) are presented in the column named *Original*. In all cases, the speedup and solution quality are normalized to the CPU. In addition to the MCNC benchmarks, the results for three GSRC hard-block benchmarks are also listed. Their CPU run time using the Parquet floorplanner is listed in Table 4.7.

BTR technique is able to improve the solution quality, with negligible performance loss. This improvement is especially marked in larger floorplanning benchmarks. However, when combining the annealing diversity (ADT), the solution quality is degraded (column marked BTR+ADT). Thus, subsequent results do not employ ADT.

Maximum benefit is seen when combining BTR with DD (column marked BTR+DD). Adding more flexibility to perform uphill moves through DD substantially improves the solution quality. In fact, optimistically avoiding costly objective evaluation also further improves the speedup seen under the DD. For example, *ami49* is able to maintain the solution quality within 1% of the CPU, while also achieving nearly 20X performance improvement.

Table 4.6: Comparison of using BTR, BTR+ADT, and BTR+DD. All results are in G92 machine with B=16, D=96, and $\Omega = 2$.

	Original		BTR		BTR+ADT		BTR+DD	
	Speedup	Quality	Speedup	Quality	Speedup	Quality	Speedup	Quality
xerox	14.19X	0.96%	13.18X	0%	14.19X	-0.99%	18.45X	1.37%
apte	12.88X	-0.83%	12.39X	-0.83%	11.93X	0%	17.89X	0%
hp	14.97X	1.54%	14.97X	2.16%	17.7X	1.54%	24.42X	1.54%
ami33	18.22X	-4.01%	17.54X	-4.54%	17.7X	-7.56%	20.72X	-3.68%
ami49	16.81X	-1.39%	17.16X	-1.07%	17.46X	-3.72%	20.02X	-0.62%
n100	18.58X	-2.41%	18.17X	-1.07%	18.92X	-8.55%	21.14X	-0.32%
n200	18.91X	-3.95%	18.92X	-3.38%	18.65X	-11.37%	20.46X	-2.3%
n300	18.44X	-12.84	18.44X	-9.17	18.63X	-11.81	20.3X	-8.91%

Table 4.7: CPU run time of GSRC hard-block benchmarks.

Benchmark	n100	n200	n300
# Blocks	100	200	300
Run time (s)	34.6	136.6	305.6

In the light of this impact of various design space explorations, the comprehensive results are presented for combining BTR and DD in Table 4.8 and Table 4.9. By varying breadth, depth as well as Ω, the results present the best obtained ones (similar in spirit to results shown in Table 4.4).

Clearly, the GPU floorplanner has improved significantly in both speedup and solution quality. The algorithm renders exceedingly high speedup when the breadth and depth are both high. As expected, the solution quality degrades dramatically when encountering such high speedup. Only *apte* benchmark obtains a reasonable 0.83% worse solution quality, due to its small size.

When optimizing solely for solution quality, it is observable that the GPU algorithm can generate equal or better solution quality in most benchmarks, while still maintaining significant speedups. One can observe that two groups of benchmarks show distinct pattern in solution quality and speedup based on the breadth and the depth. The best solutions in the group of *ami33*, *n100*, *n200*, and *n300* are obtained with $\mathbf{\Omega} = 2$, and $B = 8$. These two parameters results in the maximum amount of moves per GPU thread, allowing more deeper explorations of the solution space. *ami33*, *n100*, and *n200* can generate marginally

Table 4.8: Tradeoff in solution quality and speedup. B and D represent the breadth and depth, Ω represents the time of CPU moves. Results are from G92 GPU.

Name	Best Speedup			Best Quality			Best $\mathcal{SQ}$		
	{B,D,Ω}	SP(X)	Q(%)	{B,D,Ω}	SP(X)	Q(%)	{B,D,Ω}	SP(X)	Q(%)
ami33	{96,128,1}	471.36	-65.22	{8,64,2}	9.98	-0.24	{96,64,1}	157.13	-5.65
ami49	{128,128,1}	292.39	-20.62	{32,32,2}	36.88	0.35	{96,64,1}	163.74	-3.48
xerox	{128,32,1}	123.02	-13.69	{32,64,2}	36.90	1.54	{32,96,1}	73.80	0.72
apte	{96,32,1}	160.98	-0.83	{96,64,1}	160.98	0.00	{64,96,1}	160.98	0.00
hp	{96,64,1}	232.08	-23.46	{128,16,1}	116.01	2.46	{32,64,1}	116.01	2.46
n100	{128,64,1}	88.70	-7.65	{8,96,2}	10.46	0.34	{128,32,1}	82.17	-5.48
n200	{128,128,1}	47.66	-20.69	{8,1,2}	6.64	-0.87	{96,16,1}	40.66	-5.77
n300	{128,128,1}	43.90	-25.33	{8,64,2}	10.07	-4.47	{32,32,1}	40.71	-11.38

Table 4.9: Tradeoff in solution quality and speedup. B and D represent the breadth and depth, Ω represents the time of CPU moves. Results are from Tesla C1060 GPU.

Name	Best Speedup			Best Quality			Best $\mathcal{SQ}$		
	{B,D,Ω}	SP(X)	Q(%)	{B,D,Ω}	SP(X)	Q(%)	{B,D,Ω}	SP(X)	Q(%)
ami33	{96,128,1}	498.40	-30.75	{8,32,2}	10.39	-0.24	{96,64,1}	161.13	-5.81
ami49	{128,128,1}	298.58	-19.92	{32,16,2}	37.49	0.97	{64,64,1}	169.87	-2.75
xerox	{128,32,1}	135.47	-13.17	{32,64,2}	41.03	1.37	{32,96,1}	80.06	0.86
apte	{96,32,1}	193.51	-2.73	{64,96,1}	188.75	0.00	{96,64,1}	188.75	0.00
hp	{96,64,1}	309.50	-24.54	{96,8,1}	155.87	2.46	{96,8,1}	155.87	2.46
n100	{128,64,1}	126.30	-10.53	{8,64,2}	14.89	0.61	{96,64,1}	116.97	-4.98
n200	{128,128,1}	67.93	-20.69	{8,8,2}	9.95	-0.24	{96,16,1}	58.89	-5.82
n300	{128,128,1}	62.95	-24.63	{8,64,2}	14.50	-3.65	{32,64,1}	60.23	-10.49

equal quality solutions. Testcase *n300*, which includes 300 blocks, can achieve only within 4% of the best quality seen in the CPU (on Tesla). The other group gets benefit from a broader solution space exploration, allowing them to obtain both better quality and greater speedup by employing more threads (higher breadth).

Interestingly, the results cannot discern any direct relationship between the depth parameter and the solution quality. Typically, the depth is inversely proportional to the number of kernel launches. Hence, large depth can reduce the kernel launch overhead, but also lead to the steeper temperature gradient. In the best quality and best $\mathcal{SQ}$ results, it is shown that the value of the depth is most commonly distributed near 64.

4.8 Conclusion

This work proposes a novel GPU floorplanning algorithm. The algorithm evaluates multiple concurrent moves, thereby breaking the long dependency chains of data and control in the sequential algorithm. Several optimization techniques have been demonstrated in the context of implementing this parallel floorplanning algorithm on a modern GPU. To improve the solution quality, this work presents a comprehensive exploration of the design space, including various techniques to adapt the annealing approach in a GPU. Speedups ranging **6–188X** are observed across all benchmarks. These speedups are achieved while also improving solution quality in several benchmarks, and maintaining nearly similar quality in the rest.

Chapter 5

GPU-Based Global Routing

5.1 Global Routing

VLSI physical design is a multi-phase process, typically falling into three categories: partitioning, placement, and routing. The partitioning phase splits the entire chip into smaller and more manageable pieces, such that each one can be independently designed. The placement phase fixes the locations of each pieces while minimizing an estimated wire length with which the blocks are connected. The routing phase finds a realization of the connections among the blocks, while avoiding possible physical constraints, such as congestion and crosstalk effect.

Typically, the routing phase is divided into global routing and detail routing processes. In global routing, the interconnects are approximately laid out on a grid map. Pins that fall into the same tile location are mapped to the center of the tile. The goal of global routing is to connect the tiles that are marked with common pins with rectilinear Steiner Trees on the grid map. In detailed routing, the pin connectivity within each tile is solved to produce the exact geometric layout of the interconnect. This work will focus on the global routing problem.

Global routing problem (GRP) is one of the most computationally intensive processes in VLSI design. Since the solution of the GRP is used to guide further optimizations before tape-out, it also becomes a critical step in the design cycle. Consequently, both the execution time and the solution quality of the GRP substantially affect the chip timing, power, manufacturability as well as the time-to-market.

Aggressive technology scaling introduces several additional constraints in the GRP, significantly increasing the complexity of this important VLSI design problem [56, 57]. Alpert

et al. predicts that at 32nm there will be 4-6 metal widths and 20 thicknesses across 12 metal layers [58]. Furthermore, IBM envisions an explosion in design rules beyond 22nm that will make GRP a multi-objective problem [59]. Unfortunately, current CPU-based routers will prove to be inefficient for the increasingly complex GRP as these routers only solve simple optimization problems [60, 61].

Tackling this huge computationally complex problem would require a platform that offers high-throughput computing such as a *graphics processor unit (GPU)*. Traditionally, a GPU's computing bandwidth is used to solve massively parallel problems. GPUs excel in applications that repeatedly apply a set of operations on a big data set, involving single instruction multiple data (SIMD) style parallel code. Several existing VLSI CAD problems have seen successful incarnation in GPUs, delivering more than 100× speedup [62–64]. However, the canonical GRP does not fit well into such an execution paradigm because routing algorithms repeatedly manipulate shared data structures such as routing resources. This sharing of resources disrupts the data-independence requirement of traditional GPU applications. Hence, existing task-based parallel routing algorithms must be completely revamped to make use of the GPU bandwidth.

In the light of these technology trends, this work propose a hybrid GPU-CPU routing platform that enables a collaborative algorithmic framework to combine data-level parallelism from GPUs with thread-level parallelism from multi-cores. This work specifically addresses the scalability challenges posed to current global routers. Till date, there has been very few works that parallelize the GRP by using multi-core processors [14, 65]. However, none of these are designed to exploit high throughput computing platforms such as the GPU.

Exploiting the computation bandwidth of GPUs for the GRP is a non-trivial problem as the overhead of sharing resources hurts the overall performance. In this work, a fundamentally new mode of parallelism is used to uncover the performance potential of the GPU. This work propose a novel *net level concurrency (NLC)* model to *efficiently* consider the data dependencies among all simultaneously routed nets. This model enables parallelism

to scale well with technology and computing complexity.

5.2 Problem Definition

The GRP is defined as follows. There is a grid graph G that is composed of a set of vertices V and edges E. Each vertex v_i in V corresponds to a rectangular cell, while each edge e_{ij} in E represents a connection between two adjacent vertices i and j (or a boundary between two cells). There is also a set of nets N, for which every n_i in N is made up of a set of pins P. Each pin in a net coincides with a vertex in V. The capacity c_{ij} of an edge between vertices i and j represents the number of nets that can pass through that edge. The demand d_{ij} represents the current number of nets passing through the edge. Overflow of an edge is then defined as the difference $d_{ij} - c_{ij}$. A net n_i is routed when a path is found connecting all the pins of the net utilizing edges of graph G. The *wire length* of a net is determined by the number of edges it crossed to route all of its pins. A *solution* to the global routing problem is achieved when all the nets n_i in N are routed.

5.3 Related Works on Global Routing

This section presents a comprehensive literature review over existing academic global routers. In 2007 and 2008, the International Symposium on Physical Design (ISPD) held two global router competitions [66, 67]. These contests promoted the development of many recent global routers. The following briefly introduce their key features:

- DpRouter [68] uses dynamic-programming based routing technique to perform segment movement on rectilinear minimum Steiner tree. To route 2-pin nets, they only use patter routing technique. The run time of their approach is very short, but they cannot generate competitive routing solutions, especially on todays large-scale designs.

- Archer [69] is based on an iterative rip-up and re-route (RRR) scheme. To explore the trade-off between run time and solution quality, Archer routes net outside of congestion regions with pattern routing, and use maze routing for the rest. It also

uses a Lagrangian relaxation based algorithm to dynamically modify the Steiner trees to reduce congestion.

- FastRoute [37–40] is very competitive for time-driven routing. It uses Hanan grid structure to construct the Steiner trees, then the Steiner tree construction is improved for simultaneous via computation. Their approach mainly uses monotonic routing and multi-source multi-sink maze routing. However, solution quality was not their strong suit until their later versions, in which FastRoute is able to route all benchmarks with competitive solution quality in addition to the fast routing speed.

- BoxRouter [35, 44] integrates IP formulation to a sequential framework. BoxRouter uses progressive IP that initially routes a small box, and iteratively expands the size of the box to include more unrouted nets. Their routing mechanism is mainly based on L-shaped pattern routing and maze routing. The later version of BoxRouter includes a post-routing phase that is based on negotiation-based RRR scheme to improve solution quality.

- NCTUgr [41] is based on iterative RRR scheme. Their key contributions are adaptive pseudo-random net-ordering and evolution-based two-stage cost function that is based on the negotiation-based RRR scheme.

- NTUgr [42] is based on an enhanced iterative RRR scheme that creates forbidden regions to allow detours away from congestion. The approach helps the global router to escape from local optima that traps the solution exploration as number of iterations increases.

- NTHU-Route [33, 34] is based on iterative RRR scheme. It combines several techniques such as the negotiation-based RRR scheme, and congestion region identification method to specify the net ordering. They use routing techniques such as the monotonic routing and an adaptive multi-source multi-sink maze routing method. NTHU-Route 2.0 won the ISPD 2008 global routing contest.

- FGR [32] is based on iterative negotiation-based RRR scheme. It has the ability to directly route on 3D layers, or use a fast layer assignment followed by a 3D clean-up phase to project 2D solution to the multi-layer design. FGR won the 2D category of the ISPD 2007 global routing contest. It also generated the best solution among most benchmarks at that time.

- MaizeRouter [70] is primarily based on extreme edge shifting and edge retraction techniques to restructure the Steiner tree topologies to obtain solutions that reduce routing congestion. The method also allows resources sharing among the wire segments of a tree to reduce the wire length. MaizeRouter won the 3D category of the ISPD 2007 global routing contest.

- GRIP [43] uses 0, 1-integer linear programming (ILP) to model global routing. It currently holds the best solution quality in open literature of all ISPD 2007 and 2008 benchmarks as of today. GRIP has reported an significant 11.3% average improvement in total wire length and via cost compared to the solutions from sequential approaches. However, with most benchmarks taking several days to reach a solution, the long run time renders their approach impractical.

The most recent works on parallelization of global routing framework are listed as following:

- PGRIP [71] is a parallel version of the GRIP router. It divides the entire routing problem into sub-problem by partitioning the routing map, and uses IP-based approach to solve each sub-problem individually. Once done, they use IP to connect partial routing solutions in a patching phase. Their methodology creates a flexible and highly scalable distributed algorithm for global routing. All benchmarks can finish within a target run time of 75 minutes, while the global router can scale freely on several hundred CPUs. More importantly, the solution generated by their parallel router has roughly the same quality as the sequential version.

- NCTUgr [14] uses traditional RRR based algorithm at its core. But it parallelize the sequential algorithm in a task-based multi-threaded framework on a quad-core platform. As a result, multiple nets can be ripped-up and re-routed simultaneously by concurrent threads. A collision-aware heuristic is included to address the solution quality issue when concurrent nets access common routing resources. As a result, the run time of their approach is significantly improved, and the solution quality is very competitive among RRR based approaches. However, their scalability on many-core systems is not as good as the PGRIP.

5.4 Tackling GRP with GPU-CPU Hybrid System

This section introduces the design motivation and spectrum of the GPU-CPU hybrid system for global routing.

5.4.1 Wire Length Distribution of GRP

Technology scaling continues to pack more transistors in a chip and circuits become increasingly complex. The routing benchmarks provided in ISPD 2007 and ISPD 2008 show that modern global routing problems typically come in considerably large scales. Each problem generally packs a number of sub-problems in the magnitude between $10^6 - 10^7$, while the difficulty of each sub-problem, defined as the length of the 2-pin net, varies in a wide range.

The distribution of sub-problem difficulties is illustrated in Figure 5.1. The diagram represents the histogram of the difficulties of all sub-problems from six benchmarks in the ISPD 2007 suite. The sub-problems are defined as the 2-pin nets acquired by decomposing the multi-pin nets. The difficulty of each sub-problem is characterized with the Manhattan distance between the two pins. The peaks on the left side of the diagram show that significant amounts of the sub-problems are easy to solve. Considerable numbers of difficult sub-problems still exist, forming a long-tail distribution on the right side of the diagram. Given such properties of global routing problem, a router should be designed to maximize the computational throughput with the available hardware.

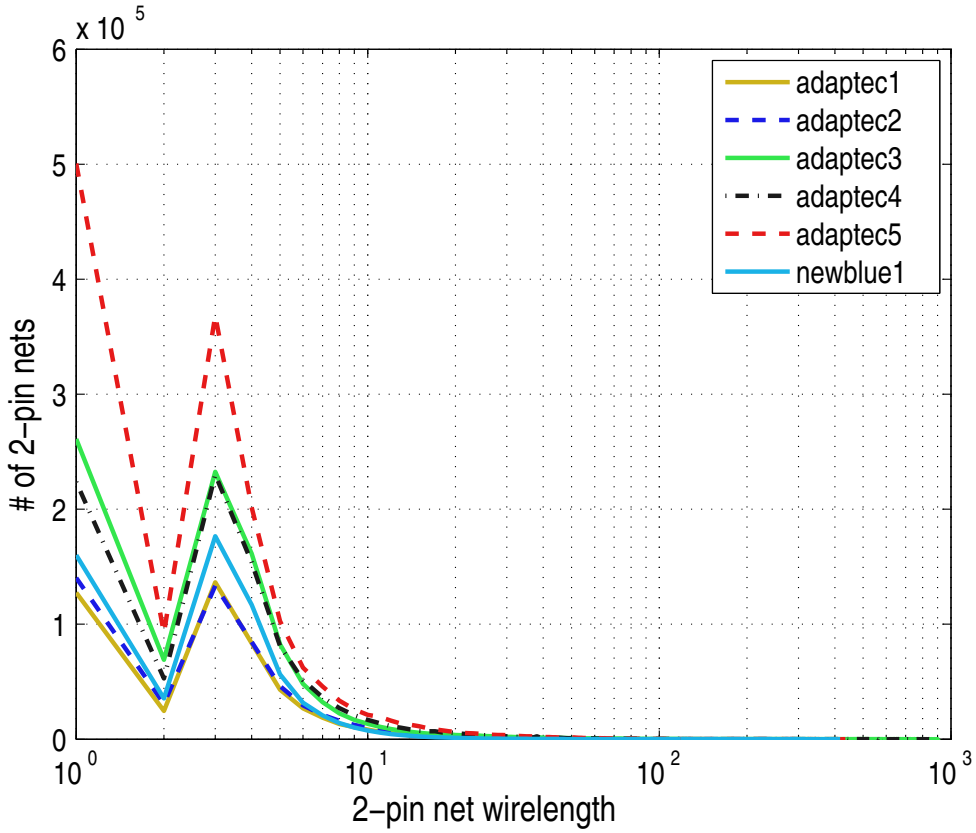


Fig. 5.1: Wire length distribution indicates co-existence of large number of long and short wires. Wire length is measured in Manhattan distance.

5.4.2 GPU-CPU Hybrid

A conceptual design spectrum is demonstrated in Figure 5.2, where the latency and bandwidth of the existing computing platforms are compared orthogonally. The single-core system can solve each routing problem with low latency, but falls short when the problems come in extremely large numbers. In comparison, the multi-core system provides larger computational bandwidth with similar latency, making it the most common choice in parallel global routing [14,65]. However, the GPU platform can easily prevail in a bandwidth contest by routing a large number of nets simultaneously. The latency for a GPU solution is likely to be longer due to the additional traffic between the GPU and the CPU.

Given the long-tail distribution of the GRP, a GPU-CPU hybrid solution appears to be attractive. The short nets, which are also the majority of the entire workload, are good

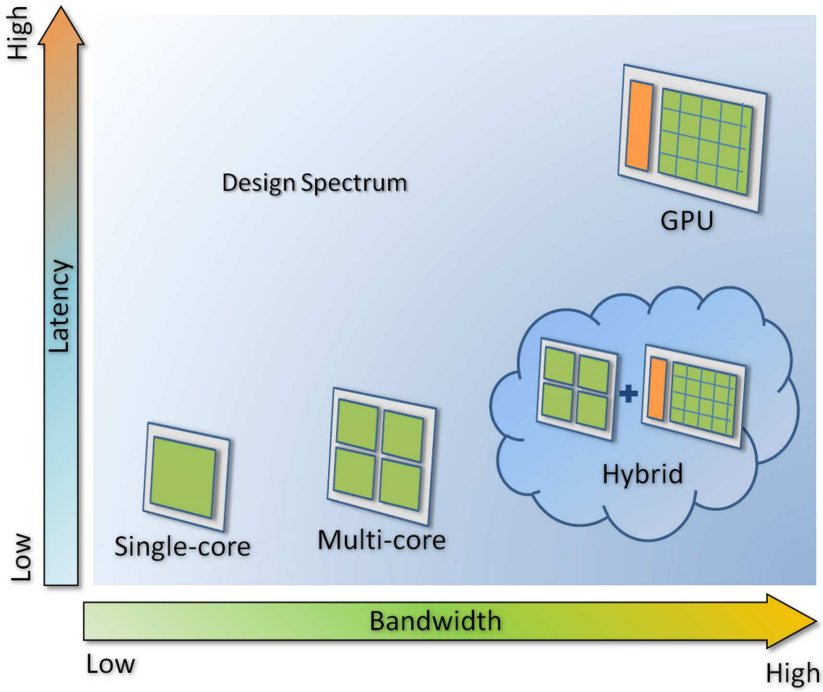


Fig. 5.2: Conceptual picture of computational bandwidth and latency of existing computing platforms.

fit for the GPU platform to utilize the broad computational bandwidth. The long nets are simultaneously assigned to the multi-core platform to exploit parallelism with lower latency. The goal is to design such heterogeneous parallel model to approach the GRP with a wide-bandwidth low-latency computing platform.

This work implements the proposed GPU-CPU hybrid model for parallel global routing. The experimental results show the heterogeneous parallel model can yield significant speedup across different benchmarks compared to the single-core implementation, while delivering similar routing quality.

5.5 Overview of GPU-CPU Global Routing

This section gives an overview of the GPU-CPU global router.

5.5.1 Objective

Like other global routers [14,32,34,35,40–42,65,69,70], the GPU-CPU global router has three major objectives. First, the minimization of the sum of overflows in all edges; second, the minimization of the total wire length of all routed nets; and third, the minimization of the total run time needed to obtain a *solution*.

5.5.2 Design Flow

The flow of the global router is shown in Figure 5.3. The initial global routing solution is generated as the following: first the multi-layer design is projected on a 2D plane and use FLUTE 3.0 [72] to decompose all multi-pin nets into sets of 2-pin subnets. Consequently, edge shifting [37] is applied on all the nets to modify their initial topologies. Then the initial routing is performed to create the congestion map.

During the main phase, the negotiation-based scheme is applied to iteratively improve the routing quality by ripping-up and rerouting the subnets that pass through overflowing edges. Each subnet is routed within a *bounding box*, whose size is relaxed if the subnet is unable to find a feasible path (Section 5.8.4). The order of the subnets to be ripped-up and rerouted is determined through *congested region identification*, an algorithm that collects subnets that are bounded within the congestion region (Section 5.8.5).

The main phase completes when no overflow is detected. Then layer assignment is applied to project the 2D routing plane back onto the original multi-layer design. The layer assignment technique is similar to that described by Roy and Markov [32].

5.5.3 Global Routing Parallelization

The parallel global router strives for high throughput by maximizing the number of simultaneously routing nets. However, the negotiation-based RRR scheme is strongly dependent on the routing order. Routing nets simultaneously regardless of their order might cause degradation in solution quality and performance.

This problem can be tackled by examining the dependencies among nets, and extracting concurrent nets from the ordered task queue. These nets can then be routed simultaneously

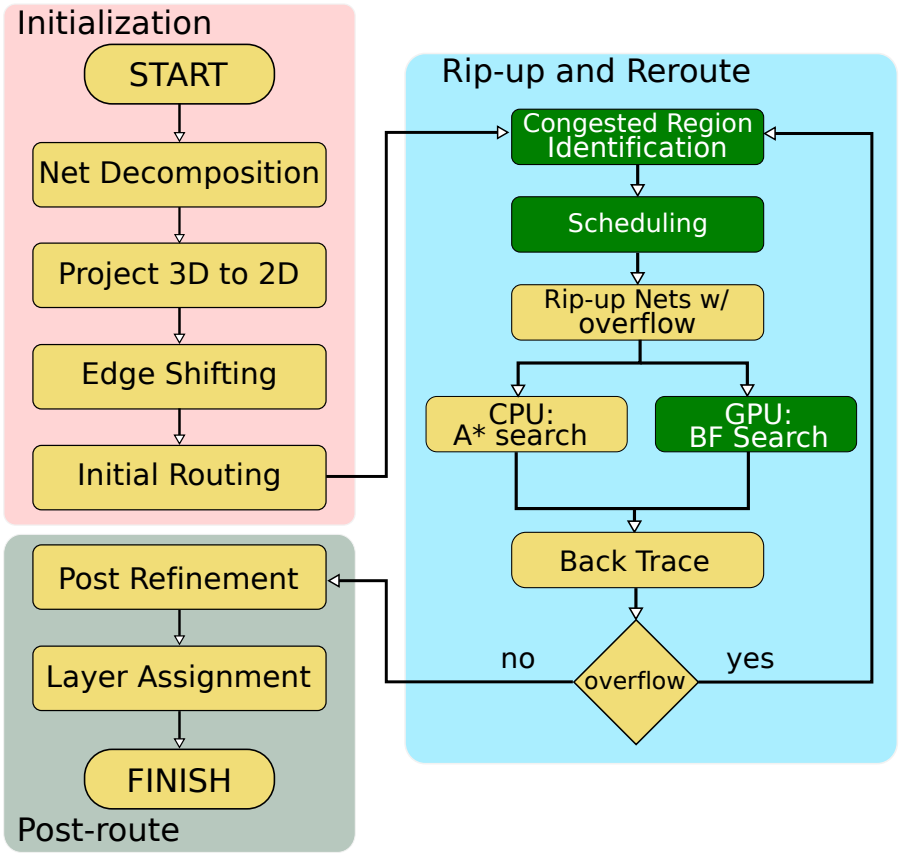


Fig. 5.3: Global router design flow: The top left section is initialization phase while bottom left is post-routing phase. Steps in these two sections are also present in other CPU-based routers. The right section is RRR. This section is enhanced with a scheduler. The contributions from this work are highlighted in dark shading background.

without jeopardizing the solution quality. The challenges of extracting concurrent nets are discussed (Section 5.6), and a novel *Scheduler* is demonstrated to tackle them (Section 5.7).

5.6 Enabling Net Level Parallelism in Global Routing

This section explains the requirements of enabling data-level parallelism in the GRP: a necessary step for exploiting high throughput platforms such as GPUs.

5.6.1 Challenge in Parallelization of Global Routing

There are two main approaches in parallelizing the global routing problem. First, the routing map can be partitioned into smaller regions and solved in a bottom-up approach

using parallel threads [65,73,74]. Second, individual nets can be divided across many threads and routed simultaneously. It is called *net level concurrency (NLC)*. NLC can substantially achieve better load-balancing and uncover more parallelism. However, it also comes at the cost of additional complexity.

Unlike partitioning, NLC allows sharing of routing resources between multiple threads. Consequently, to effectively exploit NLC, one must ensure that threads have current usage information of the shared resources. Without such information, concurrent threads may not be aware of impending resource collisions, leading to unintentional overflow and degradation in both performance and solution quality [14]. This phenomenon is demonstrated in Figure 5.4, where both the threads consume the lone routing resource resulting in an overflow.

5.6.2 Achieving NLC

Unfortunately, collision-awareness alone cannot guarantee reaping performance benefits by employing NLC on high throughput platforms. This is best explained through Figure 5.5. In this example, two assumptions are made: first there is a task queue and that threads continually get a net to route from this queue; second the nets in the queue are ordered based on the congestion of their path. Finally, there are two layers (horizontal and vertical) available for routing with demands indicated as $demand_H$ and $demand_V$. When multiple

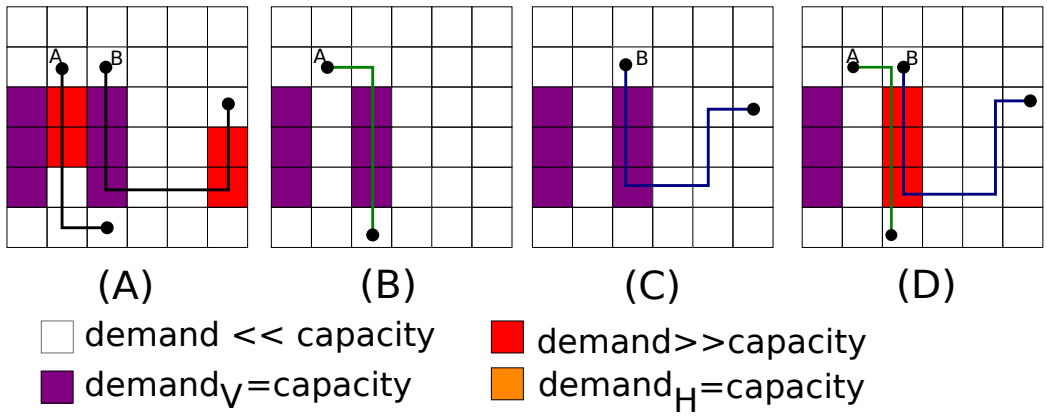


Fig. 5.4: Parallel router must have consistent view of resources. (A) Routings before RRR. (B) and (C) Viewpoint of each thread, which unknowingly allocates conflicted resources. (D) An overflow is realized at the end of RRR when both threads track back.

threads are processing a congested region, as in Figure 5.5(B), concurrent routing will cramp limited resources to specific threads, sacrificing performance and solution quality. This problem is exacerbated with increasing number of threads and circuit complexity. However, it implies that the router can recover substantial performance by allowing threads to concurrently route in different congested regions. In other words, a single thread can process the region shown in Figure 5.5(C).

Therefore, the **key to achieving high throughput in NLC is to identify congested regions and co-schedule nets that have minimal resource collisions**. The next Section will further discuss the proposed *Scheduler* design.

5.7 Scheduler

This section describes the Scheduler that dynamically examines the dependencies among nets in an ordered task queue, and extracts concurrent nets to be dispatched to the GPU and CPU routers for parallel routing.

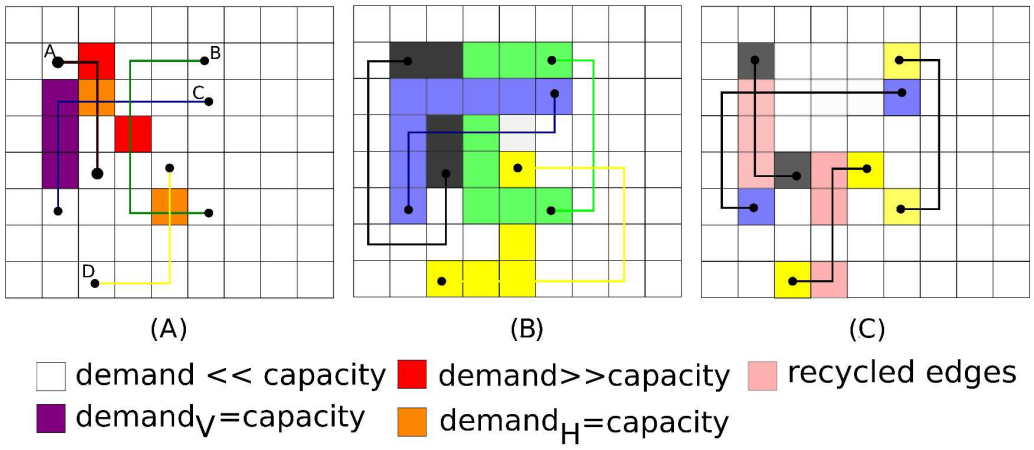


Fig. 5.5: Collision awareness alone can hurt routing solution: (A) Four-thread router processing a particular congested region, one net per thread. (B) Routing solution generated via collision-aware algorithm. Some resources are wasted due to overhead of collision awareness because threads are discouraged to route on cells (black, green, yellow, and blue cells) that were previously used by another thread. (C) With proper scheduling, only one thread is processing this particular region and some of the resources are recycled. Remaining threads are routing other congested areas on the chip (not shown).

5.7.1 Scheduler Overview

Figure 5.6 shows the preliminary design overview of a global router on a hybrid platform [12]. This design aims to exploit NLC within the traditional RRR global routing algorithms.

The heart of this approach lies in the *Scheduler* design, which identifies the concurrent nets to be routed in parallel. Since nets can share routing resources (e.g. tiles on the routing grid), concurrent nets are chosen in a manner that reduces resource conflicts where multiple routing threads attempt to use a given routing resource. Consequently, this approach restricts the level of parallelism that can be exploited during the global routing.

The *Scheduler* dynamically populates two task queues with sets of nets that can be routed simultaneously. These task queues separately serve CPU and GPU threads. The task queues are decoupled from each other, to ease the load balancing and synchronization between the CPU and the GPU. Nets for GPU threads are chosen in a manner so that the entire routing for those nets can be efficiently done in the available shared memory in the GPU architecture. On an NVIDIA Fermi architecture, which limits the shared memory to 48KB, nearly 99% of the nets from the ISPD benchmark suites can be routed on the GPU.

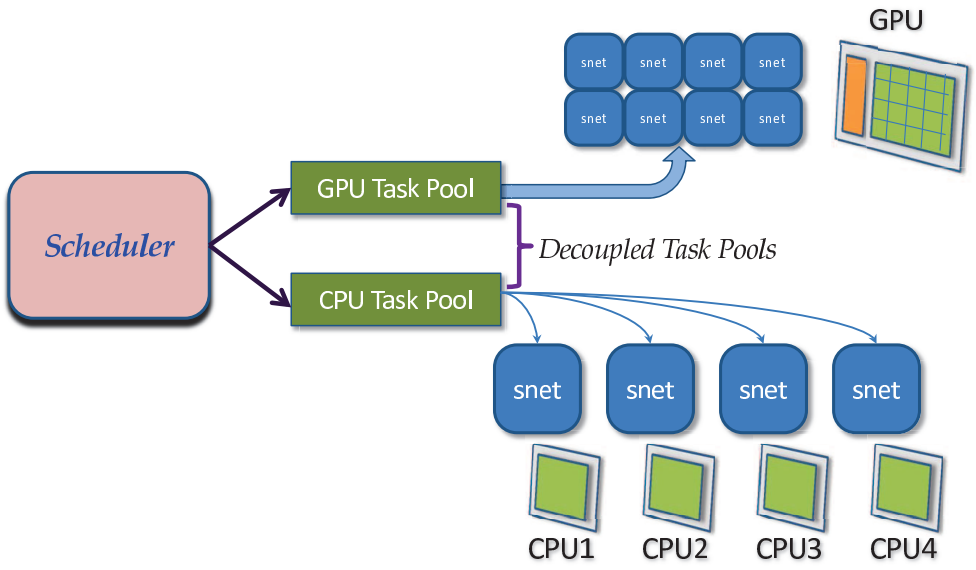


Fig. 5.6: Overview of GPU-CPU router concurrent subnets (snet) being distributed to GPU and CPU task pools.

As explained in Section 5.8.2, longer nets are assigned to the CPU, as routing these nets can uncover several other candidate nets for parallel routing.

5.7.2 Nets Data Dependency

This section explains the concept of nets data dependencies, and discuss the parallel model that examines this dependency to exploit parallelism.

In each RRR iteration, the router first create an explicit routing order for all the nets located within the congested regions. This order is derived by comparing the areas and aspect ratios of nets. Higher priorities are assigned to nets that cover larger area with smaller aspect ratios, and routed first. Because these nets are easier to find an overflow free route, reassigning their routing resources before the smaller nets has a clear advantage in minimizing congestion and the overall wire length.

By enforcing this explicit routing order, the routing process essentially creates *data dependencies* among the nets. Specifically, the routing order dictates different priorities for all nets when reassigning their routing resources. For example, when two nets both need the same resources to find paths, the one routed first has a higher priority to obtain those resources. If the order is changed, a degraded solution is likely to occur. Due to this reason, a conventional RRR process is typically implemented sequentially to ensure the nets data dependencies.

In this parallel model, it tries to exploit parallelism by routing nets that do not have a dependency violation among each other. Since net data dependencies only confines nets that have sharing routing resources, the key that allows us to parallelize the routing process while maintaining the data dependencies is to examine the shared routing resources among the nets. If no shared resource exists, then the router can safely exploit the parallelism by routing these independent nets simultaneously.

Honoring the net data dependencies is crucial for this parallel model. The existing task-based parallel global router does not examine the net dependency when exploiting concurrency [14]. As a result, more than 41% of the subnets are affected by collisions in shared routing resources. This model does not suit well in a GPU-based concurrency

framework. Due to the lack of synchronization mechanism for thread blocks in the GPU hardware, the parallel model needs to avoid resource collision on the GPU's device memory.

This work proposes a *Scheduler* to generate independent routing tasks for the parallel global routers. The data dependency is iteratively analyzed, thereby limiting its analysis overhead while providing precise dependency information. First, the data dependency among nets is constructed in a dependency graph. Then the router exploits parallelism by routing the independent nets. The parallelism created by the model can exploit massively parallel hardware without causing resource collision or jeopardizing the routing order. As a result, the GPU-CPU parallel global router achieves *deterministic* routing solutions.

5.7.3 Net Dependency Construction

This section presents the scheduler algorithm to construct the net dependencies. As an example, a selection of 2-pin nets (subnets) are illustrated in Figure 5.7. This example assumes that each net's bounding region is the same size as their covering area, and that the routing order derived is:

$$D > C > F > B > E > G > A.$$

The following section explains how to construct the dependency graph, and exploit the available concurrency without causing conflict in routing resource or order. This approach is mainly divided into the following three steps.

Tile Coloring: In this step, each tile identifies its occupancy by iterating through the ordered subnets. The first subnet region that covers the tile is considered as its occupant. Using the example from Figure 5.7, the results of the colored tiles are shown in Figure 5.8(A). One can observe that most of the map is colored by subnet D because it has the highest priority in routing order. Given this colormap, each subnet can visualize the other subnets that it is overlapping with, hence determining its dominant subnets.

Finding Available Concurrency: With the help of the colormap, the scheduler can

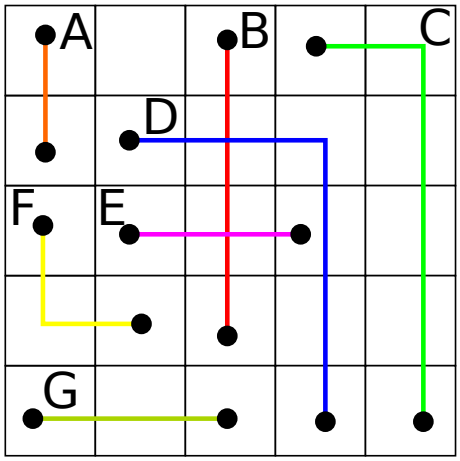


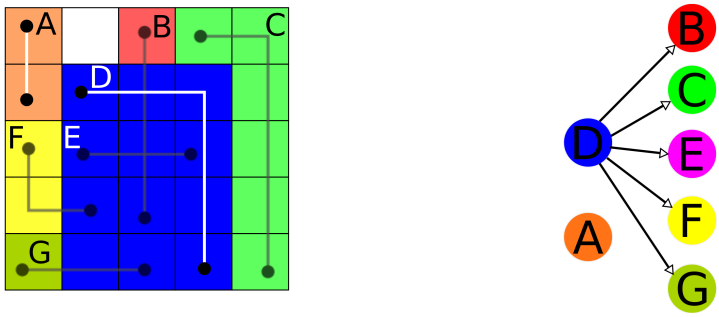
Fig. 5.7: Routing problem with nets overlapping each other.

easily find subnets that can be routed by checking if it occupies all of its routing regions. If not, then there is a dependency on other subnets and it must wait until the dependency is resolved. In Figure 5.8(A), subnets A and D occupy all of their routing regions. Hence, they can be scheduled together.

Here the concept of **dependency level** is introduced. This metric is used to determine the urgency of routing certain subnets. The dependency level is scored as the number of routing regions that one subnet invades. For instance, in Figure 5.8(A) subnet D scores 5 because it invades the area of five subnets: B, C, E, F, and G.

From Figure 5.8(B) one can make an interesting observation on the dependencies among all subnets. Subnets B, C, E, F, and G are dependent on subnet D but the scheduler cannot identify the dependencies among them. The algorithm intentionally leaves these detailed dependencies for future computation, so as to reduce complexity while extracting the available concurrency in a timely manner.

Tile Recoloring: In this step, the algorithm uncovers detailed dependencies by reconstructing the colormap. After the scheduled subnets are routed, the colormap must be reconstructed to resolve previous dependencies. Figure 5.9(A) shows the new colormap when subnets A and D are already routed. Recoloring only needs to consider subnets that were dominated by A and D. In this case, they are $\{C, F, B, E, G\}$ for D's region, and $\emptyset$



(A) Colored tiles (B) Dependency graph

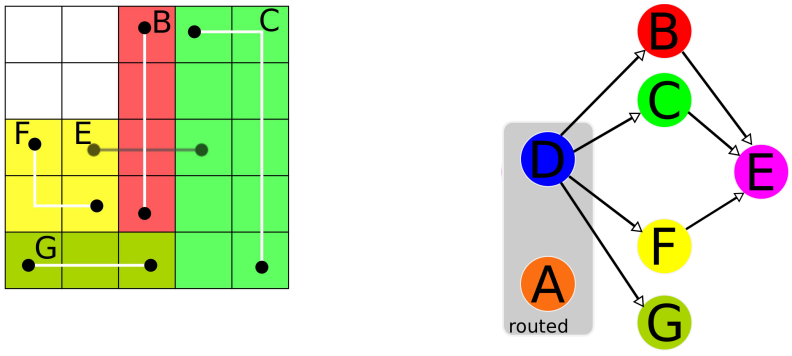
Fig. 5.8: Results after the 1st iteration: (A) Coloring of tiles: bigger nets dominate ownership over smaller ones. Only A and D can be routed together because other nets are dependent on D. (B) Net dependencies are derived from the colormap.

for A.

The dependency graph is updated using the new map. Figure 5.9(B) shows the new graph that reveals a new dependency between subnets B, E, C, and F. Similar to the previous iteration, the dependency graph indicates subnets B, C, F, and G can be scheduled once subnet D has finished, while E can only be scheduled after B, C, and F are completed.

5.7.4 Implementation and Optimization

The above three steps are recursively applied until all dependencies are resolved. *The*



(A) Colored tiles (B) Dependency graph

Fig. 5.9: Results after the 2nd iteration: (A) After D and A are routed, nets C, B, F and G can be routed together because they have no dependencies. (B) More detailed dependencies are revealed in the graph.

scheduler thread and the global router threads execute in parallel with a producer-consumer relationship. The scheduler keeps constructing the dependency graph with the given task queue, and producing concurrent nets to the router threads. The routers consume these nets simultaneously with different priorities indicated by the dependency levels, and return the completed nets to release more concurrent nets.

The efficiency of the scheduler algorithm affects the available computational throughput for routing. The complexity of this algorithm increases as the region size and net count increases. In practice, the problem size is reduced using the identified congestion region (Section 5.8.4). Instead of exploring concurrency on the entire routing map, the search area is restricted to only the congestion regions, hence easing the computation load of tile coloring.

In addition, a *task window* is used to restrict the number of nets being examined for concurrency in each iteration. In some cases, congestion regions contain a large number of nets. The task window can effectively limit the search space and speed-up the dependency tree construction (Section 5.8.3).

5.8 Implementation

This section discusses details of the GPU-CPU router implementation. It focuses on several key issues such as the maze routing implementation on the GPU (5.8.1), efficient GPU implementation of Scheduler (5.8.3), directional congested region identification algorithm (5.8.4), the bounding box expansion method (5.8.5), and distribution of nets among CPU and GPU threads (5.8.6).

5.8.1 Maze Routing Implementation on GPU

The GPU router uses the parallel Lee algorithm [75] to find the weighted shortest paths. This widely applied approach, although known for its high memory usage and slow search speed, is an attractive candidate for implementation on parallel systems.

Typically, the front wave expansion scheme of the Lee algorithm can be parallelized, allowing us to simultaneously explore the vertices at the same depth, which are defined as

frontiers. This concurrency model enables us to utilize the GPU's large number of threads in a single block to exploit fine-grain parallelism during the concurrent frontiers expansion. This concurrency model is shown in Figure 5.10.

In addition, the router can safely perform multiple wavefront expansions and back traces in parallel without causing collision. This level of parallelism is coarse-grain, hence is exploited by the GPU's grid blocks. It should be mentioned that no global synchronization mechanism is required among the GPU thread blocks when routing multiple nets in parallel, since the nets are mutually independent. This concurrency model is illustrated with Figure 5.11.

Another factor that makes the Lee algorithm appealing to GPU architecture is the use of simple grid arrays to represent the routing map. No complex data structure is required to store the pending route candidates. In this implementation, a fixed-sized flat memory is used to store the costs of expanding paths.

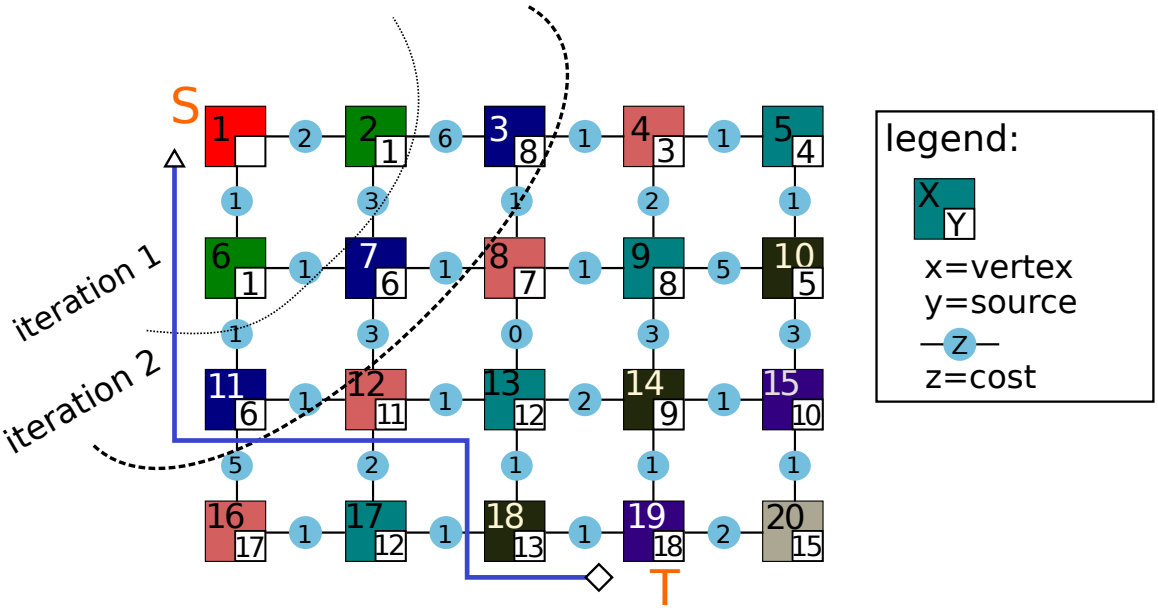


Fig. 5.10: Pathfinding in a GPU: The propagation starts from the source node. The breadth-first search fills up the entire search region, and continues until all frontiers are exhausted. Then the router back traces from the target node to find the shortest weighted path.

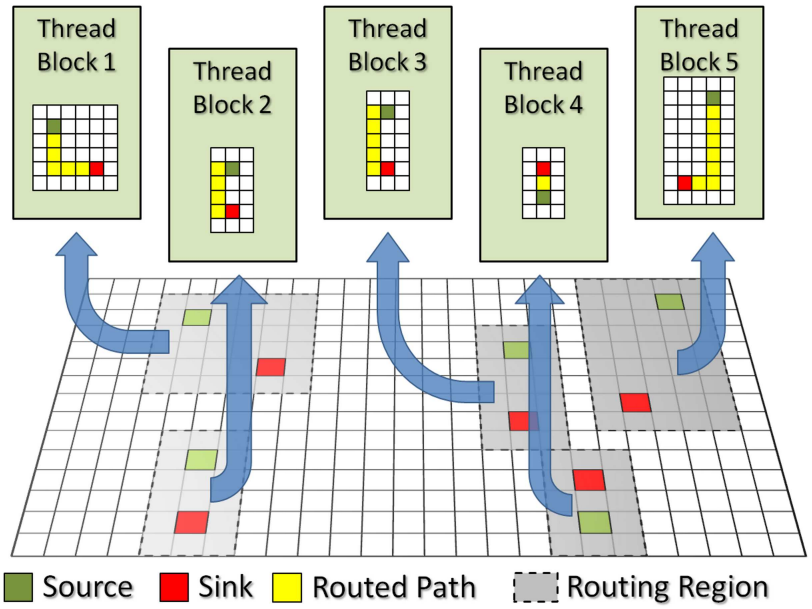


Fig. 5.11: GPU routing overview: Each thread block finds route for a single set of source and sink. The routing is done locally on the shared memory of each thread block.

The following section explains the GPU-based Lee algorithm in detail. Like the sequential version, the GPU-based Lee algorithm is divided into two phases:

- **Wave Propagation:** In this phase, propagation starts from the *Source* tile. The search stops when the frontiers are all exhausted, hence guarantees all possible paths within the search region are traversed. Details of the wave propagation approach is described in Algorithm 5.1, with the explanation of terms in Table 5.1. The algorithm describes the routing kernel executed on each GPU thread. Each GPU thread is mapped to a tile on the routing grid. Each tile is indexed using the corresponding GPU thread ID (*tid*). If the tile is identified as a frontier tile, then the GPU thread will try to propagate to the neighboring tiles, which are indexed as *nid* in the algorithm.

- **Back Tracing:** In this phase, the GPU kernel essentially reverses the propagation direction. Starting from the *Sink* tile, the route with the least cost is traced and the traversed edges are marked as the resulting path until the *Source* tile is reached. In the end, only one successful routing path is returned by the kernel.

Algorithm 5.1: GPU Lee algorithm kernel.

```
1: tid ←getThreadID
2: if Frontier[tid] then
3:    Frontier[tid] ←false
4:    for all neighbors nid of tid do
5:       Edge ← Vertex(tid, nid)
6:       if Edge exists then
7:          AddedCost ← TileCost[tid] + EdgeCost[Edge]
8:          if AddedCost < TempTileCost[nid] then
9:             TempTileCost[nid] ← AddedCost
10:         end if
11:      end if
12:   end for
13: end if
14: SYNCHRONIZE_THREADS()
15: if TileCost[tid] > tempTileCost[tid] then
16:   TileCost[tid] ← tempTileCost[tid]
17:   Frontier[tid] ← true
18:   DONE ← false
19: end if
20: tempTileCost[tid] ← TileCost[tid]
```

This design integrates the propagation and back tracing phases into one CUDA kernel function instead of two separate ones. Doing so reduces the overhead of loading intermediate data between the shared memory and device memory, and the overhead of additional kernel launch.

This algorithm is fundamentally different from the previously proposed BFS algorithms for GPUs [76, 77], since (1) the algorithm tackles the weighted shortest path problem; (2) an individual 2-pin net is routed within each thread block. Therefore, performance boost is attained by routing large amount of nets concurrently.

5.8.2 GPU Memory Arrangement

The performance of a GPU application is largely dependent on its memory arrangement. This section explains the GPU memory arrangement to enable high throughput maze routing on the GPU.

In the GPU Lee algorithm, the costs of traversing tiles are arranged in grid arrays,

Table 5.1: GPU Lee algorithm notations.

Term	**Description**
$Frontier$	Boolean list that marks the frontier tiles in the current iteration. Contains source vertex initially.
$EdgeCost$	Array that stores cost of all edges.
$Vertex$	Function that returns the $Edge$ between two Vertices.
$AddedCost$	Intermediate variable to store the new tile cost.
$TempTileCost$	Array that stores cost of traversing tiles. Initialized as infinite (Inf.).
$TileCost$	Array that stores minimum cost of traversed tiles. Initialized as Inf.
$DONE$	Boolean indicating all frontiers are explored.

which are stored in the shared memory. This arrangement is demonstrated in Figure 5.11. The size of the grid is determined by the bounding box of the routing net, hence it is partial to the complete routing grid. Individual blocks are allowed to update the local grid on the shared memory, without synchronization to the global device memory. This arrangement has a much higher efficiency, but comes at the cost of generality. Due to the shared memory size limitation, the number of tiles that can be traversed by each thread block is constrained to about 2500. Fortunately, this size is reasonable for the GRP in most cases. According

to experimental observation, more than 99% of all 2-pin nets can be fitted within this area.

The vertices topology and edge cost data are stored in the GPU texture memory. The texture memory is allocated on the device memory. But with texture binding, this memory is cached and optimized for read-only data. Fetching from texture memory provides high bandwidth on memory space with good spatial locality, i.e. if a cell is visited, then its neighbors are also traversed. Hence, the vertices and edge costs array are binded with `3D-Texture` and `1D-Texture` memories, respectively. Each cell in the vertices in `3D-Texture` points to six different adjacent cells: `-X, -Y, -Z, +X, +Y, +Z`. These directions are used to identify the edge index, which locates the cost of the edge from the edge cost texture.

5.8.3 Scheduler

The efficiency of the scheduler is critical to the overall throughput of the GPU-CPU hybrid global router. On the one hand, the scheduler needs to be able to produce enough concurrent nets for all parallel threads to consume. On the other hand, the scheduler needs to be light weight enough to deliver the workload in a timely manner. This section introduces an efficient scheduler design.

The scheduler algorithm is implemented on a GPU. The algorithm is parallelizable on a fine-grain level, with each GPU thread dedicated to color a single tile. This implementation also has low latency due to the small amount of data packages that are copied between the CPU and the GPU. In addition, transferring the computation to the GPU allows us to free significant CPU resources, which can be dedicated to maze routing.

The complexity of the scheduler algorithm increases as the region size and net count increases. In practice, the problem size is reduced using the identified congestion region (Section 5.8.4). Instead of exploring concurrency on the entire routing map, the search area is restricted to only the congestion regions, hence easing the computation load of tile coloring.

In addition, a *task window* is used to restrict the number of nets being examined for concurrency in each iteration. The task window has significant impact on the overall performance of the parallel router. A short window usually leads to less trade-off in performance,

but might yield insufficient workloads; whereas a long window might over examine the available concurrency, and introduce degradation in performance.

Figure 5.12 shows the distribution of workload verses the parallel window size on ISPD 2007 newblue2 benchmark. On the left side of the figure, insufficient workloads due to a small window size causes most nets being distributed onto a single thread. Due to this reason, the GPU thread has very low utilization rate. This problem is alleviated by increasing the size of the task window. However, it should be mentioned that when the window size is too wide, the overhead of the scheduler itself begins to dominate, hence degrading the overall router performance.

5.8.4 Congested Region Identification (CRI)

To discuss the CRI algorithm, the problem of finding congested regions is formulates by explaining several terminologies. The first step in the identification of congested regions is the generation of congestion map from the initial routing result. A congestion map is an $X \times Y$ matrix of congestion values. Each element in the map b_{ij} is the average congestion

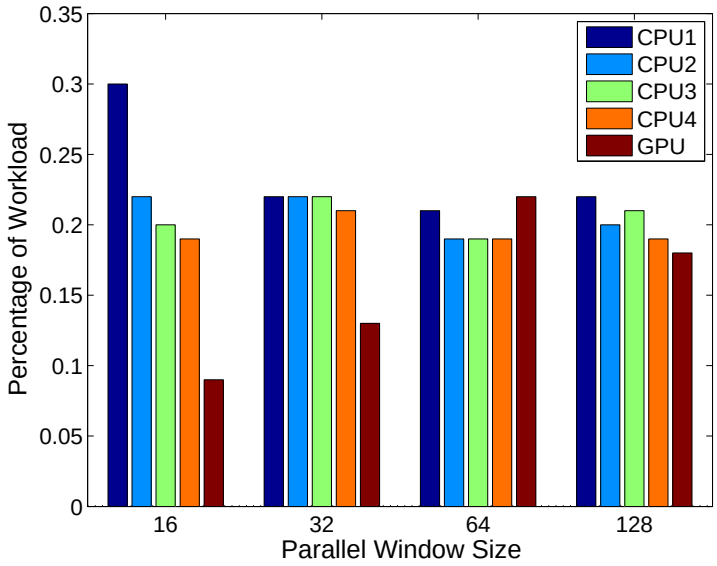


Fig. 5.12: Workload distribution with different task window. With the increasing size of parallel window, workload is easier to be balanced amongst CPUs and GPU, but it also comes at higher overhead.

of the top and right edges of cell (i, j). Congestion $m_{i,j;k,l}$ is the congestion of edge between cell (i, j) and (k, l). If let $d_{i,j;k,l}$ and $c_{i,j;k,l}$ be the corresponding demands and capacities of that edge then $m_{i,j;k,l}$ can be specified as

$$m_{i,j;k,l} = \frac{d_{i,j;k,l}}{c_{i,j;k,l}}. \tag{5.1}$$

Consequently,

$$b_{ij} = \frac{m_{i,j;i+1,j} + m_{i,j;i,j+1}}{2}. \tag{5.2}$$

All b_{ij}'s are scaled with respect to the maximum such that $0 \leq b_{ij}^{'} \leq 1$, where $b_{ij}^{'}$ is the scaled value.

In order to accurately identify the congested region, a *directional expansion algorithm* is used to adaptively expand to the region in the directions that result in the highest congestion. Figure 5.13 best explains this situation. The algorithm begins by taking the most congested tiles (red cells in the figure) and adaptively expanding until the average congestion for the region is below a certain threshold. This method divides the congestion value into several congestion levels, much like NTHU-Route [34]. The number of congestion levels to be modeled will dictate the size of each region. In this example, there are four congestion levels. After these regions are found, the nets inside them are routed in parallel based on Section 5.6. Pertinent details are presented in Algorithm 5.2, Table 5.2 lists the notations and their descriptions.

5.8.5 Bounding Box Expansion

Bounding box is widely applied in global routers. This technique constrains the searching region of 2-pin nets within a rectangle, reducing the space complexity of maze routing and decreasing the solution wire length.

In the GPU-CPU router, bounding box is a crucial component for the Scheduler to determine nets dependencies. However, in order to allow nets to explore a larger solution

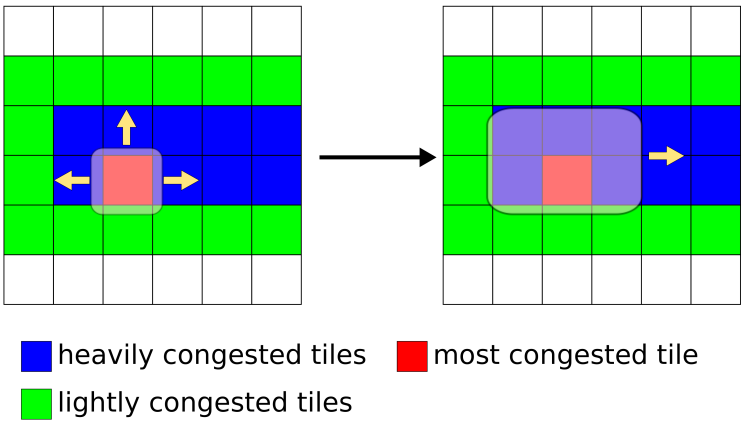


Fig. 5.13: Directional expansion algorithm: Bounding box adaptively expands in the directions with the highest congestion.

Algorithm 5.2: Directional expansion algorithm.

1 $r = max(b_{ij})$
2 $\forall\ 0 \leq i \leq X;\ 0 \leq j \leq Y$
1: **for** $level = 1$ to 4 **do**
2: **while** $Ave_4(r) > \mathcal{L}_{\mathcal{B}}(level)$ **do**
3: $expand4sides(r)$
4: **end while**
5: **while** $Ave_3(r) > \mathcal{L}_{\mathcal{B}}(level)$ **do**
6: $expand3sides(r)$
7: **end while**
8: **while** $Ave_2(r) > \mathcal{L}_{\mathcal{B}}(level)$ **do**
9: $expand2sides(r)$
10: **end while**
11: **while** $Ave_1(r) > \mathcal{L}_{\mathcal{B}}(level)$ **do**
12: $expand1sides(r)$
13: **end while**
14: **end for**

space, the size of the bounding box is expanded as the RRR iteration proceeds. As the remaining overflow decreases, the constraint of the bounding box can be relaxed to allow the maze router to obtain overflow free route at the cost of longer wire length. But as the constraint of bounding box relaxes to more than 10 times as large as the original, the expansion stops again to avoid excessively long routes.

An adaptive method is chosen, rather than a fixed parameter function, to expand

Table 5.2: Algorithm notations.

Term	Description
r	rectangle with bottom left coordinates (i, j) and top right (k, l)
$Ave_i(r)$	average congestion value inside the expanded rectangle r in the direction of i side(s)
$\mathcal{L}_{\mathcal{B}}(level)$	returns the lower bound value for a particular congestion level
$expand4sides(r)$	expands region r in all directions
$expand3sides(r)$	expands three sides of region r towards the maximum congestion
$expand2sides(r)$	expands two sides of region r towards the maximum congestion
$expand1sides(r)$	expands one side of region r towards the maximum congestion

the bounding box. The search area constraint is relaxed according to the percentage of remaining overflow. The size of the bounding box is kept unchanged until 99% of all overflow is resolved. This phase passes very fast since the searching areas are small. Then the router linearly increases the size of the bounding box as RRR iteration proceeds, until its size reaches the upper limit.

5.8.6 Workload Distribution Between GPU and CPU

This section introduces the heuristic used for workload balancing. The scheduler dispatches workloads among the CPUs and GPU for optimum computational throughput. Typically, the CPU routers achieve a single solution with lower latency than the GPU router, but the latter can achieve much higher throughput by routing multiple nets in a single kernel call. Urgent nets, which release several subsequent nets to be routed concurrently,

are more likely to be scheduled on the CPUs. In addition, nets with large bounding boxes are also routed in the CPU due to the shared memory limits on the GPU. The detailed scheduling heuristic is based on the following criterion:

- **Routing Region Size:** The GPU router has constraints on the size of the routing region for each 2-pin net. If a workload exceeds the area limitation, it will be scheduled on a CPU.
- **Net Size Preference:** The concurrent nets are sorted with respect to their problem size. The CPU maze-router consumes workload from the larger end of the queue, while the GPU consumes from smaller end.
- **Lower-Bounded Scheduling on the GPU:** Since the GPU router strives for high bandwidth, it only schedules nets to route if the number of available workloads meet a certain lower boundary. Typically, this number is set to be $\frac{1}{4}$ of the current parallel window size.

These criterion can dynamically consume the available workloads as quickly as possible. The heterogeneous structure of a GPU-CPU hybrid system makes it difficult to predict a perfectly balanced schedule. Hence certain router threads are allowed to wait in idle. This is especially the case when the number of concurrent nets is small.

5.9 Results

The GPU-CPU hybrid router is implemented with C/C++ on an Intel Quad-core 2.4GHz machine with 8GB of RAM. The GPU used in the experiment is an NVIDIA Geforce GTX 470 with the Fermi architecture. The C/C++ code is compiled with Intel C Compiler 11.1; CUDA code is compiled with CUDA Toolkit 4.0. The multi-threading framework is designed using pthread. ISPD 2007 and ISPD 2008 benchmark suites are used in the experiments shown below.

5.9.1 GPU and CPU Router

First the routing throughput of the GPU and CPU routers is compared. In this experiment, the same nets are scheduled to the GPU router and a single CPU router, then the average wall clock time is recorded for both routers. To make it a fair comparison, the process includes routing, back tracing, and data transfers between the GPU and the system memory.

The results are shown in Figure 5.14 and Figure 5.15. The run time comparison in Figure 5.14 shows a linear increase of CPU router run time with the growing number of routing nets. Interestingly, the GPU run time slope is much flatter than the CPU. Consequently, CPU router has a much shorter latency when routing individual nets, while the GPU can deliver a much higher bandwidth when scheduled with multiple nets.

The relative speedup between the GPU and CPU routers is illustrated in Figure 5.15. About 5X speedup can be observed when both the routers are scheduled with 30 nets. One should notice that the speedup keeps growing with more scheduled nets. The trend of speedup keeps increasing as more nets are being routed. A speedup of 73X is seen if both routers are scheduled with 1000 nets (not shown).

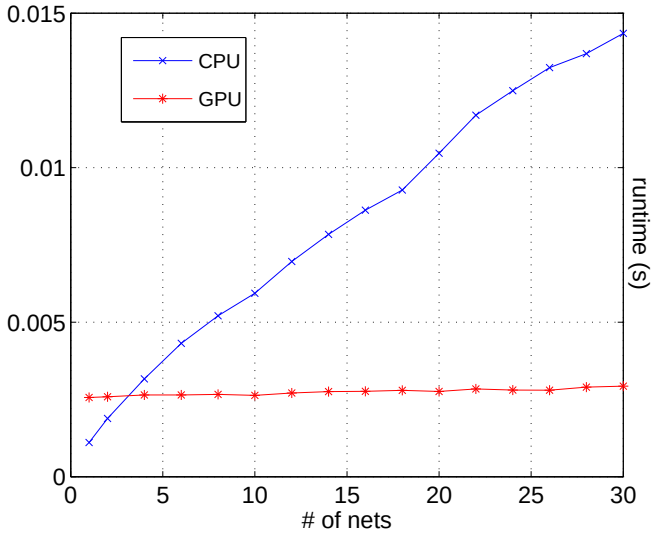


Fig. 5.14: Runtime comparison between CPU A*Search and GPU BFS.

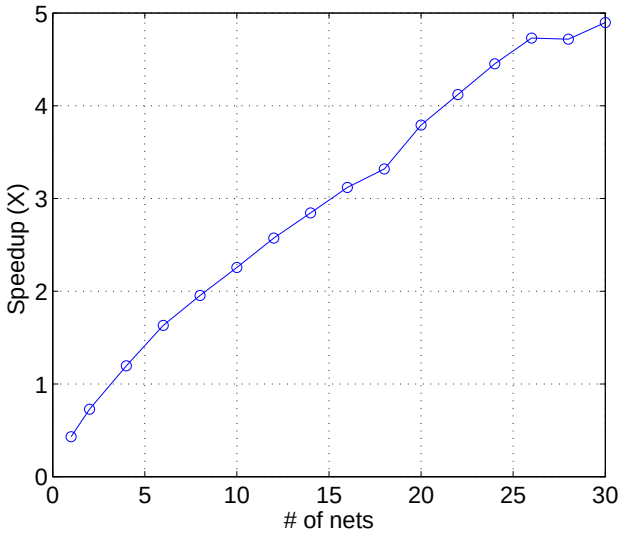


Fig. 5.15: Speedup of GPU BFS over CPU A*Search.

5.9.2 Comparison with NTHU-Route 2.0

Table 5.3 shows a wire length and run time comparison against ISPD 2008 global routing contest winner NTHU-Route 2.0. The GPU-CPU Hybrid global router generates high quality routing solutions within an average of 1.1% wire length increase to that reported by NTHU-Route 2.0. Noticeably, this router yields the same solution quality under different levels of parallelism. The results of the hard-to-route problems are listed in Table 5.4.

The speedups achieved by the GPU-CPU parallel model is examined in Table 5.5, where the run time of the RRR stage is compared under different configurations of the parallel router. The parallel router utilizing four CPU threads achieves an average speedup of 2.74X, while an average speedup of 3.27X is gained with the additional GPU.

The speedup is highly dependent on the benchmarks. Among the 12 overflow-free benchmarks, speedups ranging from 1.68X to 4.49X are received under the quad-core plus GPU platform. This observation indicates that on benchmarks with less number of congested regions, there is often little net level parallelism to exploit; whereas on the benchmarks with widely-spread congestion regions we can extract more concurrent nets.

Although the proposed GPU-CPU hybrid parallel router shows great potential in offering high throughput, the results indicate that the performance of parallel router is heavily

Table 5.3: Wire length and run time comparison with NTHU-Route 2.0 on overflow-free benchmarks.

	1-core		4-core		4-core+GPU		NTHU 2.0	
	WL[a]	time[b]	WL[a]	time[b]	WL[a]	time[b]	WL[a]	time[b]
adaptec1	5.43E6	8.04	5.43E6	2.90	5.43E6	2.67	5.34E6	9.95
adaptec2	5.29E6	1.09	5.29E6	0.70	5.29E6	0.63	5.23E6	2.1
adaptec3	1.31E7	8.04	1.31E7	3.53	1.31E7	3.34	1.31E7	10.86
adaptec4	1.24E7	1.09	1.24E7	0.95	1.24E7	0.94	1.22E7	2.5
adaptec5	1.55E7	30.66	1.55E7	9.98	1.55E7	9.12	1.55E7	21.9
newblue1	4.70E6	7.58	4.70E6	2.87	4.70E6	2.57	4.65E6	6.2
newblue2	7.79E6	0.82	7.79E6	0.68	7.79E6	0.64	7.57E6	1.1
newblue5	2.38E7	17.39	2.38E7	7.87	2.38E7	7.38	2.32E7	19.1
newblue6	1.80E7	13.48	1.80E7	5.67	1.80E7	5.35	1.77E7	17.5
bigblue1	5.63E6	10.64	5.63E6	3.92	5.63E6	3.74	5.59E6	13.1
bigblue2	9.10E6	6.79	9.10E6	3.03	9.10E6	2.78	9.06E6	8.4
bigblue3	1.30E7	4.01	1.30E7	2.39	1.30E7	2.33	1.31E7	4.4

a. wire length in terms of edges consumed
b. wall clock run time expressed in minutes

Table 5.4: Wire length and overflow comparison with NTHU-Route 2.0 on hard-to-route testcases.

	GPU-CPU GR			NTHU-R2		
	MO	TO	WL	MO	TO	WL
newblue3	183	31484	108.5	204	31454	106.49
newblue4	4	167	138.9	4	138	130.46
newblue7	4	78	352.3	2	62	353.35
bigblue4	2	178	232.9	2	162	231.04

Table 5.5: Speedup comparison in RRR stage.

	1-core	4-core		4-core+GPU	
	t_{RRR}	t_{RRR}	Spdup	t_{RRR}	Spdup
adaptec1	7.30	2.16	3.38X	1.77	4.12X
adaptec2	0.71	0.32	2.21X	0.25	2.81X
adaptec3	6.48	1.97	3.29X	1.63	3.98X
adaptec4	0.26	0.12	2.13X	0.11	2.33X
adaptec5	28.79	8.11	3.55X	6.46	4.49X
newblue1	7.23	2.52	2.87X	2.22	3.25X
newblue2	0.44	0.30	1.45X	0.26	1.68X
newblue5	14.78	5.26	2.81X	4.40	3.36X
newblue6	11.88	4.07	2.92X	3.20	3.71X
bigblue1	9.83	3.11	3.16X	2.57	3.82X
bigblue2	6.04	2.28	2.65X	2.03	2.98X
bigblue3	2.72	1.10	2.47X	1.03	2.65X
Average	-	-	2.74X	-	3.27X

dependent on certain properties of the benchmark. In some cases, large number of concurrently routable nets hardly exists. In this situation, the insufficient amount of independent nets causes the router threads to wait in idle. These observations suggest that more fundamental research needs to be done to unveil the connection between the benchmarks and the levels of parallelism that are exploitable.

5.10 Conclusion

As technology continues to scale, computational complexity of many EDA algorithms is growing rapidly. Exploiting the computational bandwidth of high throughput platforms like the GPU is a prominent direction for future EDA. In this paper, we present a hy-

brid GPU-CPU high throughput computing environment as a scalable alternative to the traditional CPU based router. We show that the traditional GRP needs to be revamped for exploiting the new computing environment. The key to our method is using *Net Level Concurrency* guided by a Scheduler. The Scheduler analyzes data dependencies between nets and dynamically generates concurrent routing tasks for the computing environment. Detailed simulation results show an average of **4**X speedup over NTHU-Route 2.0 with negligible loss in solution quality. This framework is a concrete step towards developing next generation global routers geared for high throughput compute architectures.

Chapter 6

GPU-Based Global Router with Fine-Grain Parallelism

The previous chapter introduces a GPU-CPU hybrid parallel global router based on a NLC model to parallelize workloads. The global router is tested on ISPD 2007 and 2008 benchmark and showcases tremendous speedup over the state-of-the-art sequential global router. However, there are several limitations to this approach: (1) The NLC model has shown limitation in extracting enough parallel workload to fully utilize the throughput of hardware; (2) A large bulk of data, mostly the congestion map, must be kept synchronized between the system memory and GPU memory, leading to a significant performance overhead by the frequent traffic over PCI-E interface.

To address the above issues, a new fine-grain concurrency (FGC) model is proposed to mitigate the lack of exploitable concurrency of the NLC model. A GPU only routing engine is developed to pair with the new parallel model. The newly router aims to cleanly separate the workloads and data structures residing on CPU and GPU subsystem, reducing the major overhead in CPU and GPU communication. This chapter focuses on introducing the FGC model and the new GPU-based routing engine.

6.1 Motivation

The key to effectively harness the computation throughput of GPU is to provide it with sufficient parallel workloads. This section studies the exploitable concurrency of the net-level dependency model. Results show that an inherent false data dependency in the model leads to a bottleneck of exploitable concurrency generation, which makes it unable to support sufficient workloads for massive parallel routing on GPU, especially for the newly emerged modern global routing problems.

6.1.1 Insufficient Exploitable Concurrency

The NLC model actively speculates the dependencies of workloads by examining the overlapping of routing regions among workloads. Essentially workloads without any overlapping region are deemed independent from each other, and are scheduled to route in parallel regardless of their original routing order.

However, the NLC model is reported to have insufficient parallel workloads. This effect is studied through an experiment with a series of global routing benchmarks from ISPD 2007, ISPD 2008, ISPD 2011, and DAC 2012 suites. For each suite, its exploitable concurrency is calculated using the model presented by Han et al. [12]. In each iteration of the global router, parallel workloads are extracted based on their data dependencies. Each workload is a 2-pin subnet, which is decomposed from a multi-pin net topology. Since multiple subnets can belong to the same net, the model enforces a coherent policy such that only one subnet from a net is allowed to be routed each time. Assuming enough bandwidth to route all the workloads of each iteration, the exploitable concurrency is quantified as $\frac{N_{subnets}}{N_{iter}}$, where N_{subnet} is the total number of subnets rerouted, and N_{iter} is the iteration count to complete routing them.

As a comparison of the above model, another set of experiments are set to relax the coherent policy enforced on workloads. This model enables subnet level concurrency and allows parallel routing of subnets from the same net. Therefore, conceptually it should release a higher level of exploitable concurrency to the parallel router.

The results are shown in Figure 6.1. The net level concurrency model is denoted as the strict model, and the subnet level model is denoted as the relaxed model. The strict model suffers from limited exploitable concurrency, and such limitation exacerbates in the most recent benchmarks, i.e. the exploitable concurrency of DAC2012 benchmarks are only half of the ISPD2007 benchmarks. In contrast, the relaxed model does not show a similar trend, demonstrating better robustness in the capability of concurrent workloads extraction. In addition, up to 3X exploitable concurrency is observed in the relaxed model.

Unfortunately, although the relaxed model delivers promising exploitable concurrency,

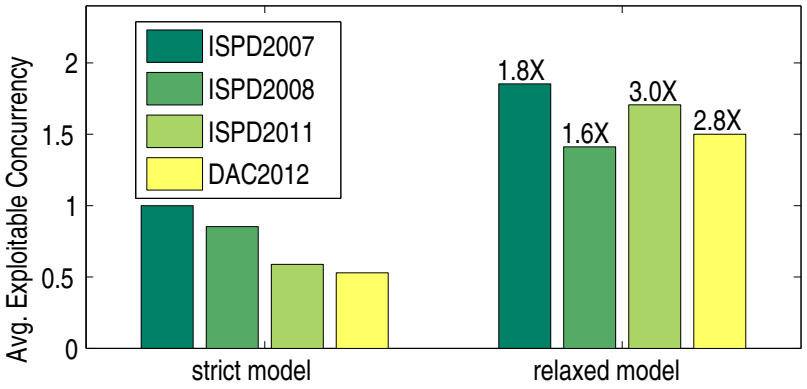


Fig. 6.1: Exploitable concurrency in ISPD2007, ISPD2008, ISPD2011, and DAC2012 benchmark suites. The y-axis is normalized to the strict model of ISPD2007.

it typically leads to a large degradation of solution quality. the following section will elaborate on the cause of degradation, and show that the relaxed model is not a sufficient model for global routing parallelization.

6.1.2 Degradation of Routing Quality

Existing parallel global routers unanimously exploit net level concurrency rather than the subnet level [12, 14, 71]. This design choice is due to an intrinsic *false data dependency* among subnets of the same net. Figure 6.2 shows a typical subnet decomposition from a net, where subnets are allowed to share certain paths to construct the net topology. Global routers extensively exploit these shared paths to generate solutions with competitive wire length. Figure 6.2 illustrates this process. In each step, the router serializes the workload of each subnet, and reuses the existing routed paths to negotiate a new route around the congested regions. The resultant net topology in Step 3 has an optimal wire length as a result of this strategy. One can observe that the additional wires in $subnet_1$ and $subnet_3$ both take the least length to detour around the obstacles.

In contrast to the above strategy, the relaxed model ignores the intrinsic false data dependency among these subnets. In this case, order in which subnets are routed is only based on their data dependencies. That is, if there is no routing region overlapping between

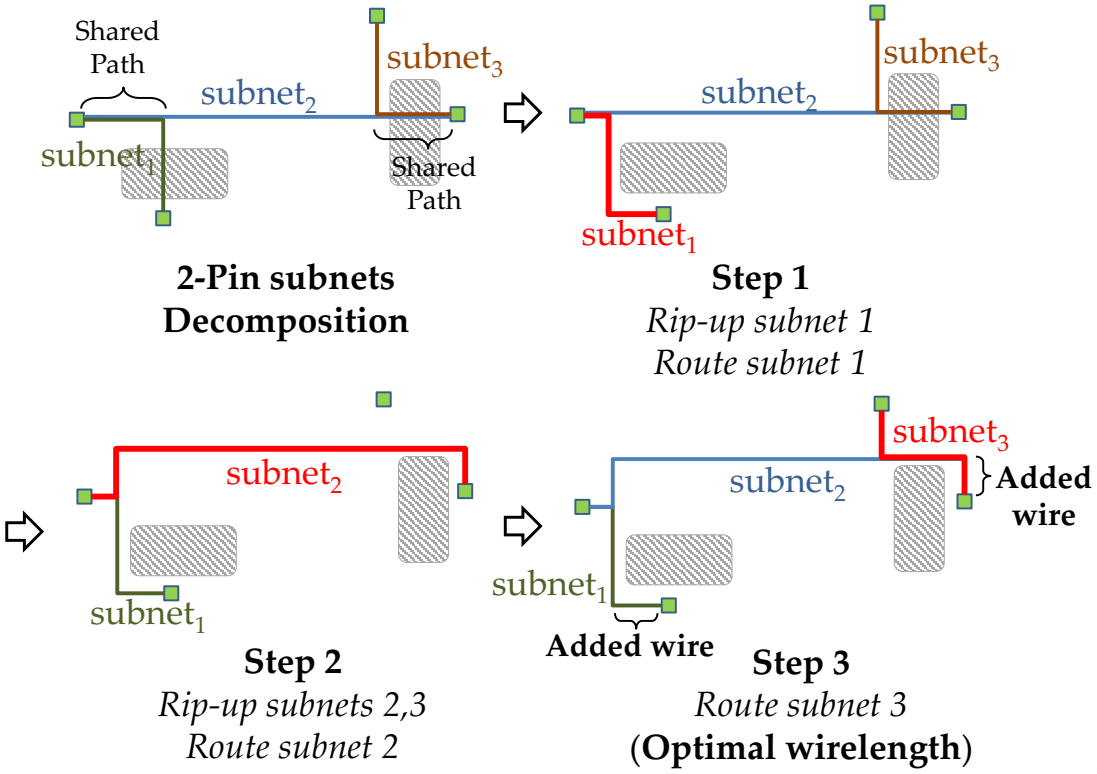


Fig. 6.2: Net level concurrency serializes the subnets of the same net to rebuild the net topology around obstacles, illustrated by the gray areas.

subnets, then they are deemed independent from each other. If there is overlapping, then the net ordering rules apply. As evident in Figure 6.3, $subnet_1$ and $subnet_3$ are routed in parallel because they have no overlapping in routing regions. Subsequently $subnet_2$ is routed to complete the net topology. As a result of this strategy, the chain of false data dependency is disrupted. The global router generates longer overall wire length for the net, leading to degraded solution quality. The additional wire length is much longer than that in Figure 6.2, Step 3.

The above degradation proves that the relaxed model is not sufficient as a parallel model for global routing. Despite its advantage in the amount of exploitable concurrency, it tends to produce degraded solution quality for multi-pin nets. Instead, a design overhaul on workload decomposition is the key to supply enough exploitable concurrency for massive parallelism. In the following section, a parallel model based on a Steiner edge decomposition

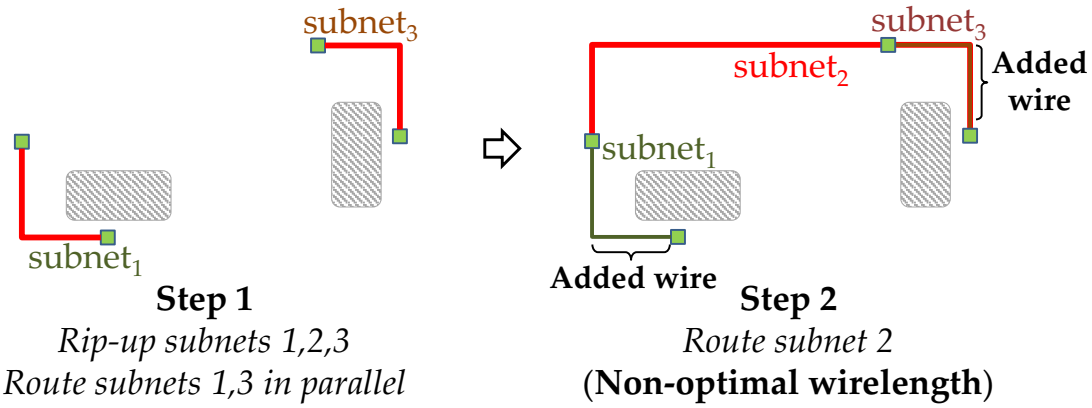


Fig. 6.3: Subnet level concurrency promotes concurrency, but relaxes the false data dependency, leading to degradation of solution quality.

strategy is proposed. This model effectively removes the intrinsic false data dependency, and provides abundant workloads for the parallel GPU router.

6.2 Parallelism on Steiner Edge

This section shows a novel parallel model to mitigate the lack of exploitable concurrency in the existing one. This model has several advantages, listed as follows:

- The model can extract abundant exploitable concurrency;
- The model does not lead to solution quality degradation;
- The model is beneficial for GPU implementation.

6.2.1 Improving the Exploitable Concurrency

The new model exploits parallelism on Steiner edges to mitigate the drawbacks of existing parallel models. Figure 6.4 demonstrates this model, which reuses the example from Figure 6.2. The multi-pin net is decomposed into five Steiner edges. Steiner edge 1 and 4 are routed over two obstacles respectively. Now these Steiner edges are re-routed to create an overflow free topology. Two concurrent threads are used to re-route the Steiner edges. In each thread, the overflowed Steiner edge is replaced completely by a new one

within the routing region marked by the dotted lines. The new Steiner edge subsequently adjusts the location of Steiner point to rejoin the net topology. The resultant net has an optimal wire length. Importantly, the parallel model produces the *same* routing solution regardless of any order in which the parallel threads are executed.

The major difference in this approach from the 2-pin subnet based approach is that Steiner edges have no shared paths. Eliminating the shared paths play a key role in exploiting concurrency. It removes the false data dependency among workloads. Evident from Figure 6.2, exploiting the shared paths creates an implicit chain of false dependencies. Specifically, in Step 1, $subnet_1$ depends on $subnet_2$'s path; in Step 2, $subnet_2$ depends on $subnet_1$; and $subnet_3$ depends on $subnet_2$ in Step 3. By removing the shared paths, Steiner edge based decomposition effectively eliminates the chain of false data dependencies. The routing solution of geographically independent Steiner edges is no longer dependent on the sequence in which they are routed.

This model allows parallelizing fine grain workloads. Since each workload is a Steiner edge, the granularity of each workload is much smaller than the net level concurrency

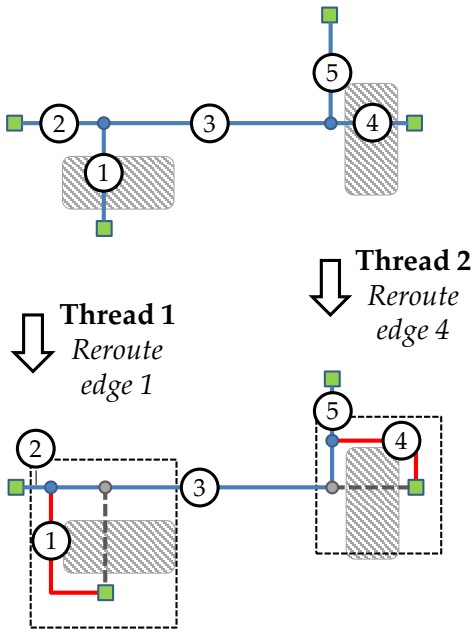


Fig. 6.4: The net is decomposed into five Steiner edges that do not have any shared path. Steiner edges 1 and 4 are re-routed in parallel in an asynchronous manner.

model. Such fine grain concurrency allows multiple workloads from the same net to be routed simultaneously, as long as no data dependency exists. In addition, as illustrated in Figure 6.4, the size of the routing problems are typically smaller than that of the 2-pin subnet. As a result, significantly more workloads can be parallelized.

6.2.2 Data Isolation for GPU Router

The proposed model has several intrinsic advantages to enable global routing on heterogeneous architectures. Typically, a major bottleneck of GPU application lies in the data communication between GPU and CPU through the PCI-E bus. Eliminating frequent data exchange and reducing the size of the exchanging package plays an important role in GPU application optimization. This design achieves such a goal through data isolation.

The Steiner edge based approach facilitates the isolation of two major data structures for global routing: the routing grid edge usage and the net topology representations. First, the array that stores the edge usage of the entire routing grid can reside completely within the GPU device memory, facilitating more computational intensive tasks such as maze routing on GPU. Due to the absence of shared path, GPU can safely update edge usage by replacing the usage of old Steiner edges with newly routed paths. Second, the data structures that represent net topology are stored in the system memory. These data structures often require inherently sequential operation and dynamic allocation, making them more efficient on the CPU. This information is used to examine the data dependencies and create concurrent workloads for GPU router.

Between the two processors, an uplink and downlink data channel is defined for communication of different purposes. The uplink channel sends workload information to the GPU for maze routing. The data package includes only source and sink nodes and bounding box information. By completion of the routing, GPU uses the downlink channel to send the routed paths back to CPU, which updates the net topology representation in system memory. Essentially both the uplink and downlink channels are implemented with the CUDA memory copy APIs. Only essential data required for routing is transmitted through these data channels to limit communication overhead.

Figure 6.5 explains this process in a step by step manner. More details on this implementation will be presented in the following sections.

6.3 GPU Framework Overview

This section explains the GPU based parallel routing framework. This framework is demonstrated in Figure 6.5. It is largely divided into four stages: (1) congested region analysis, (2) searching for and scheduling parallel workloads, (3) routing the workloads and committing their edge usages on GPU, and (4) adjusting topology to accept the new routes.

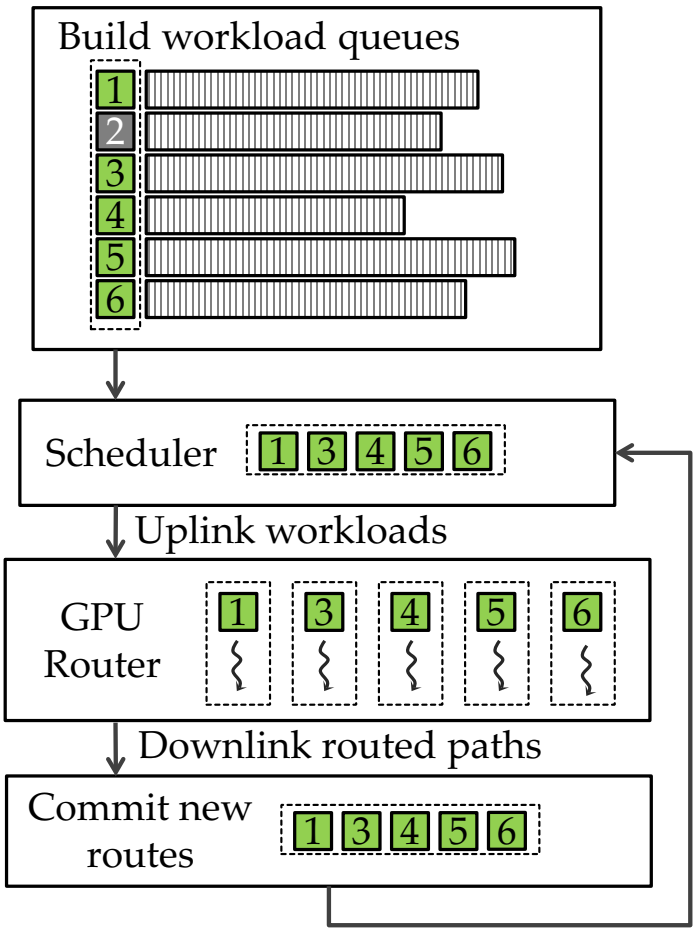


Fig. 6.5: The main GPU routing framework.

6.3.1 Congestion Analysis

The first stage examines edge usage across the entire routing grid given an initially routed global solution. The edges whose usage exceed their capacities are congested. They are distributed coherently on the routing grid, spread in isolated regions. A distribution of congested regions is captured, and the regions are correlated with the corresponding Steiner edges, which will be rerouted to resolve the congestion. Each congested region relates to one set of Steiner edges. The router explicitly creates an order in which these Steiner edges are scheduled to route. Typically, Steiner edges covering larger area are given higher priority to better create detour without introducing excessive wire length overhead.

The Steiner edges are organized in multiple ordered queues instead of a single queue to improve the efficiency of parallel workload extraction. Since these regions essentially partition the entire routing map into multiple independent sub-maps, the Steiner edges from different congested regions are naturally isolated and have no explicit dependencies. Similarly, the Steiner edges within each queue all route pass the same congested region, they are very likely to have mutual data dependencies. Therefore, each ordered queue hides the non-parallelizable workloads within its list, while exposing the concurrent workloads to the scheduler. Using an efficient lookup from the head of each list, the scheduler is highly likely to extract only independent workloads.

6.3.2 Scheduling

The scheduler stage searches for Steiner edges that are independent of each other for parallel routing. When the bounding boxes of two Steiner edges do not overlap, these Steiner edges are considered independent. These Steiner edges are thereby issued to the ready list. In Figure 6.5's example, five Steiner edges from queues 1, 3, 4, 5, and 6 are pushed to the ready list for parallel routing, while workload 2 cannot be issued due to a data dependency. In an actual benchmark, however, the number of ordered queues is often in hundreds, and the ready list is quickly populated with independent Steiner edges.

6.3.3 GPU Routing

The GPU routing stage re-routes all Steiner edges gathered from the ready list. The bounding box used for scheduling of the Steiner edge is re-used by each routing thread on the GPU to restrain the search region. First, the routing thread rips-up the overflowed Steiner edge, reducing the corresponding edge usage on its path. Due to data isolation explained in Section 6.2.2, the edge usage data are easily accessed on the GPU on-board memory. Subsequently, the GPU thread reconnects the net with a multi-source multi-sink A* search engine. The router launches independent search agents for the workloads, and negotiates around the congested edges to identify new paths. Finally, the GPU again changes the global edge usage to reflect the newly created Steiner edge.

6.3.4 Commit Topology

The final stage adjusts the Steiner trees according to new paths found by the GPU router. The new path is returned from the GPU to CPU through downlink communication. The change to the net topology takes place on the CPU, where the newly routed Steiner edges from GPU are committed to their Steiner trees one by one. In addition, the scheduler is informed with the completed Steiner edges. Subsequent independent Steiner edge workloads are released to the ready list according to an update to the data dependencies, hence starting the following iteration. This cycle is repeated until all the Steiner edges in the ordered routing queues are visited and rerouted.

6.4 A* Search on GPU

This section explains the GPU based multi-thread global routing. It is designed to enable multi-source multi-sink maze routing to fit the A* search algorithm for the Steiner edge decomposition. A GPU A* search algorithm is described by Bleiweiss [78]. The GPU implementation in this work is largely based on Bleiweiss's method, but with the following modifications to optimize for the global routing problems.

6.4.1 Routing Grid Textures

The GPU router implementation encapsulates the sparse routing grid graph in an adjacency list data structure. Being read-only, this graph is stored as a set of linear device memory bound to texture references. The CUDA device memory that is read through texture fetching has performance benefits, since the data is cached and potentially enables fast localized access.

The routing grid graph data structure is designed purposefully to enhance the spacial locality of the access. The set of data is optimized for the grid graph of a global routing problem, including an edge list and adjacency list. Each edge entry contains its capacity, usage, blockage, and history cost data. The edge list serializes all the adjacent global edges into one collection of edges. The adjacency directory contains an offset into the edge list. Each adjacency directory corresponds to a tile on the routing grid. Since each tile on a routing layer connects to two other tiles in opposite directions either horizontally or vertically, two consecutive reads in the edge list is required to fetch both edge costs. Since via edges have a constant cost and no via usage is counted[1], via edges do not need to occupy the texture memory.

6.4.2 Shared Memory Management

The shared memory is an on-chip memory that has much smaller access latency than the GPU global memory. All threads within a block have access to the same shared memory, which allows user-managed data caches. The use of these caches can significantly improve the performance of a GPU application. Unfortunately the size of shared memory for each block is very limited. Typically the maximum shared memory capacity per block is 48KB for a Fermi NVIDIA GPU.

In general, the A* search data structure for most routing workloads can fit within the shared memory space, thanks to their relatively small sizes. Through experience, there are more than 99% of the Steiner edge rerouting problems satisfying this condition. To

[1]The via edge cost is defined as a constant cost in the current industry benchmarks, although it might not be the case for real world designs.

maximize the usage of shared memory for each workload, one GPU block is dedicated for each routing instance, allowing the routing thread to maximize the utilization of low latency access on-chip memory. Also all threads within the block can collaborate when possible, maximizing the memory access bandwidth. However, certain problems requires the A* search to propagate through a large quantity of nodes, in which case the storage of nodes in the frontier set (open set) can grow beyond the capacity of the shared memory.

A simple mechanism is designed to the GPU router to facilitate very large search space. The min-heap, with which the priority queue for A* search is implemented, resides in the shared memory till the space is about to be exhausted. Then any additional levels of the heap are stored in the device memory. In this case, the heap operations on the items stored in the device memory might suffer from long access latency. To avoid this performance penalty, the size of the heap is minimized so it will be more likely to fit within the shared memory space. The method sets an upper limit of a path's cost to be allowed entering the frontier set. This threshold is typically set as the path cost of the original Steiner edge before ripping it up. As a result, all paths with cost exceeding that of the ripped-up path are not considered to enter the priority queue, hence reducing the size of the frontier set.

This mechanism is effective for the GPU A* search implementation. The shared memory is typically enough to store a heap of 11 levels deep, containing 2047 items. For common cases this storage space does not draw any limitation to the A* search. Yet for rare corner cases where extra space is required, one additional level of heap can double its capacity, significantly relieving the constraint placed on storage space. For each heap level in the device memory, only one device memory access is required for a heap up or heap down operation, given that coalesce memory access is used on the nodes with the same root. Therefore, a linear increase of performance penalty results in an exponential growth of storage space.

6.4.3 Assistance Processes

Prior to the actual routing, the GPU router needs to initialize the source and sink nodes and bounding box bundle for each routing thread. Source nodes are added to the initial open set for the A* search; sink nodes are marked down as destinations of the search,

and initializes heuristic cost function, which is defined as the shortest distance from any tile to all the sink nodes.

Once the routing of all threads are finished on the GPU, the GPU router needs to update the edge usage. Each path's corresponding edge is found on the global grid and their usages are changed accordingly. The process is straight forward, but since the texture fetch is read-only, this process needs to unbind the texture reference to allow write access to the device memory.

6.5 Experimental Results

This section shows experimental results. The proposed parallel routing framework is coded in C++. The GPU portion is development using NVIDIA CUDA. The experiments are done on an Ivy bridge 8-core (four physical cores) processor with 16GB RAM. The GPU of choice is a Geforce GTX 470 graphics processor. The main focus of these experiments is to show the router's strength in parallelization, and its impact on the solution quality. The performance of the proposed parallel router is compared against two state-of-the-art routers: NCTUgr2 [14] and BFG-R 2.0 [79]. The DAC 2012 benchmark suite is used to conduct the study. The placement solutions of the benchmark are provided by ripple placer in the DAC 2012 contest [52]. Both routers comparing to are highly successful, and adopted by the DAC 2012 contest as the evaluation tools. To the best of the author's knowledge, these are the only available routers with the ability to parse the DAC 2012 benchmarks.

The results are shown in Table 6.1. The first row of the table shows a comparison between the router's CPU and GPU implementations. The implementation on both architectures is made consistent in order to deliver as credible as possible performance comparison. The GPU router delivers a notable improvement in run time through parallelization, an average of 5.3X speedup is achieved. In addition, these speedups across all benchmarks are very uniform, showing excellent robustness of the parallel model.

Inherent from the NLC model, the fine-grain parallel model delivers a "*deterministic*" solution. The scheduling sequence is consistent regardless of the number of parallel routing threads, leading to an exact match in the solution wire length and overflow outcome of

Table 6.1: Routing performance comparison between the CPU and GPU routers. The comparison focuses on solution from the RRR stage.

	FGC Router (CPU)				FGC Router (GPU)			
	MO[a]	TO[b]	run time[c]	2DWL[d]	MO	TO	run time	2DWL
sb11	0	0	38.8787	10885057	0	0	7.3050	10885057
sb12	25	937	317.939	12200901	25	937	53.7688	12200901
sb14	0	0	72.0268	7265608	0	0	14.3244	7265608
sb16	0	0	54.2831	7945358	0	0	10.6978	7945358
sb19	0	0	19.2866	5059133	0	0	3.8462	5059133
sb2	255	90666	502.681	20850372	255	90666	82.0863	20850372
sb3	142	18683	424.346	12608778	142	18683	70.6665	12608778
sb6	0	0	132.884	11130475	0	0	23.3166	11130475
sb7	0	0	155.114	14572748	0	0	27.2013	14572748
sb9	0	0	64.9287	8717550	0	0	12.8402	8717550

a. Maximum **Normalized** Overflow.
b. Total **Normalized** Overflow.
c. Expressed in **Seconds**.
d. The 2D wire length is reported at the end of the RRR stage.

the CPU and GPU implementation. These results have shown that the fine-grain parallel model can effectively parallelize the emerging global routing problems on massive parallel architecture without any impact on the solution quality.

Table 6.2 lists the routing solutions of BFG-R 2.0 and NCTUgr2 global routers under the same test environment. The amount of speedup (in times) and increase of wire length (in percentage) in comparison to the GPU router are listed in Table 6.3. The sb12, sb2, and sb3 cases are marked as N/A since the GPU router is unable to converge on an overflow-free solution.

Comparing the GPU router to BFG-R and NCTUgr2, it is evidential that the GPU router has its strength in run time. On an average, the GPU parallel router is 8.29X and 3.06X faster than BFG-R and NCTUgr2 respectively, with a maximum speedup of

Table 6.2: Routing performance comparison between two state-of-the-art routers. Only RRR stage run time is counted to rule out the effects of pre- and post-routing stages from these routers.

	BFG-R 2.0				NCTUgr2			
	MO	TO	run time	2DWL	MO	TO	run time	2DWL
sb11	3	3147	53.481	10845494	0	0	28.7	10732137
sb12	4	13781	162.52	11900644	0	0	52.32	11756906
sb14	4	2490	77.965	7151014	0	0	46.04	7095935
sb16	3	1817	107.5	7850695	0	0	36.74	7790863
sb19	3	1862	39.216	4990464	0	0	10.37	4918376
sb2	9	31306	789.29	19523801	0	0	515.45	19530541
sb3	8	24821	443.26	11777479	0	0	208.32	11852632
sb6	4	8019	213.22	10854467	0	0	64.26	10769850
sb7	6	14003	196.33	14233563	0	0	68.32	14134562
sb9	5	8172	111.28	8556826	0	0	36.75	8499331

Table 6.3: Normalized speed up and wire length comparison between the GPU router and the other routers. The percentage of wire length increase is denoted as "WL+".

	BFG-R 2.0		NCTUgr2	
	Speedup	WL+	Speedup	WL+
sb11	7.32X	0.36%	3.93X	1.42%
sb12	N/A	N/A	N/A	N/A
sb14	5.44X	1.60%	3.21X	2.39%
sb16	10.05X	1.20%	3.43X	1.98%
sb19	10.20X	1.37%	2.70X	2.86%
sb2	N/A	N/A	N/A	N/A
sb3	N/A	N/A	N/A	N/A
sb6	9.14X	2.54%	2.76X	3.34%
sb7	7.22X	2.38%	2.51X	3.10%
sb9	8.67X	1.87%	2.86X	2.56%
avg.	8.29X	1.62%	3.06X	2.52%

10.20X and 3.93X. It should be noted that only the set of run time from the rip-up and re-route (RRR) stage is collected for a more credible comparison. The other routers both

have sophisticated pre-routing and post-routing stages that take a notable portion of the entire run time, while the GPU router mostly rely on the RRR phase, with only a quick edge-shifting stage prior to the RRR.

The final part of this section compares the GPU router's solution quality with that of the other routers. In a routability comparison, the GPU router stands in between the others, finishing seven out of ten benchmarks with overflow free solutions. The BFG-R router is unable to finish any of the tested benchmarks with overflow free solution, while the NCTUgr2 can finish all the benchmarks without any overflow. In the wire length comparison, the GPU router produces slightly longer total wire length. In average, its total wire length in the routed benchmarks are 1.62% and 2.52% higher than that of BFG-R and NCTUgr2, respectively. This marginal increase of overall wire length exist compared to BFG-R and NCTUgr2, which both have enjoyed years of development and parameter tuning. On the contrary, as a newly developed global router, the GPU router does not yet have the same infrastructure necessary to produce similar routing qualities. Nevertheless, the slightly lesser solution quality is an inherent product of inferior design sophistication, and should not be considered as a result of the parallelization scheme.

Overall, the proposed GPU global router has shown success in parallelizing the global routing problem on the massively parallel architecture. The GPU implementation outperforms the leading global router in run time. The concurrency model does not introduce any impact on the solution quality, delivering a "stable" parallelization.

6.6 Conclusion

This work presents an effective way to parallelize global routing on GPU architecture. It tackles the lack of exploitable concurrency in emerging global routing problems, and mitigate it with a global router designed based on the Steiner edge decomposition scheme. The GPU router exploits fine grain parallelism on the level of Steiner edges. The experimental results confirm that this model produces deterministic solutions, and is capable of achieving significant speedup through parallelization on GPU architecture. The router outperforms some of the most success global routers from open literature. On Geforce GTX 470 GPU we

achieve 2.51X to 3.93X speedup over NCTUgr2, and 5.44X to 10.20X speedup over BFG-R 2.0.

Chapter 7

Congestion Analysis

As routing becoming increasingly challenging with the rapid advance of technology scaling, the layout routability has become a fundamental obstacle in achieving the overall design convergence. In this context, the role of congestion analysis is increasingly important. Typically, the congestion analysis provisions the upcoming routability issues in routing and post-routing processes, and provides actionable items to the upstream tools, such as the placer and floorplanner, to mitigate the identified problems. Consequently, the problems created for the router engines are much easier to route, hence leading to a design that is more likely to converge.

However, due to the limitations of the existing models of congestion, few congestion analysis tools are be able to accurately reveal the problematic regions on the layout within a short time window. By studying the distinction between fundamental causes behind congestion, which have profound implication towards their resolution in upstream or downstream tools, this work finds it important to integrate a new metric to provide an accurate feedback of the causal relationship between layout interconnect patterns and the resultant congestion.

In the light of the above notion, this study proposes a framework called dynamic orthogonal congestion (DOC) analysis. DOC studies the patterns of layout interconnects to predict the congestion and problematic regions. It encompasses two major components: (1) an orthogonal congestion correlation coefficient that effectively captures the hard-to-route regions, and (2) a routing framework based on dynamic edge shifting technique. To present DOC framework, the rest of the chapter is organized as follows: Section 7.1 defines the problem formulation; Section 7.2 outlines the motivation of this work; Section 7.3 describes the orthogonal congestion correlation coefficient in detail; Section 7.4 explain the routing

mechanism of the DOC framework; Section 7.5 showcase the experimental results of DOC, and Section 7.6 concludes this chapter.

7.1 Problem Formulation

In this section, a problem formulation is given to explain the congestion analysis problem, and a set of denotations is presented for use throughout the rest of this chapter. The global routing problem is formulated by partitioning the routing region into a 2D grid map. Each tile is considered a global cell (gcell), in which all physical pins within its region are mapped to cell's center. A network topology is defined as a grid graph $G = (V, E)$, in which the vertex set V denotes a set of gcell tiles, and the edge set E denotes the boundaries between two adjacent tiles.

Each edge e has a capacity limit, denoted as C_e. When net topologies pass through an edge, they use certain resources of the edge, denoted as usage U_e. *Congestion* G_e on edge e is defined to be the ratio between the edge usage and capacity:

$$G_e = \frac{U_e}{C_e}. \tag{7.1}$$

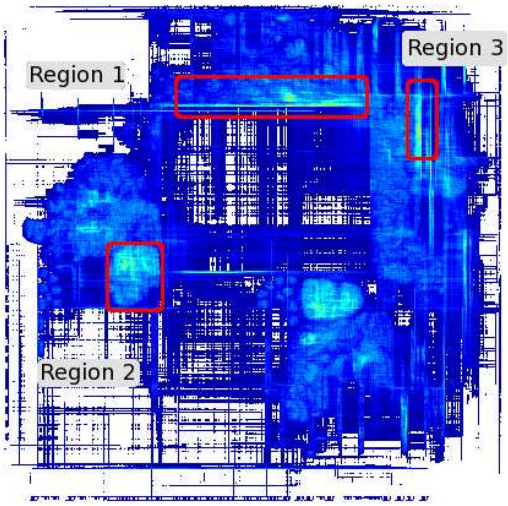
Therefore, a congested edge has $G_e \geq 1$.

In addition, the *overflow* O_e on edge e is defined as following:

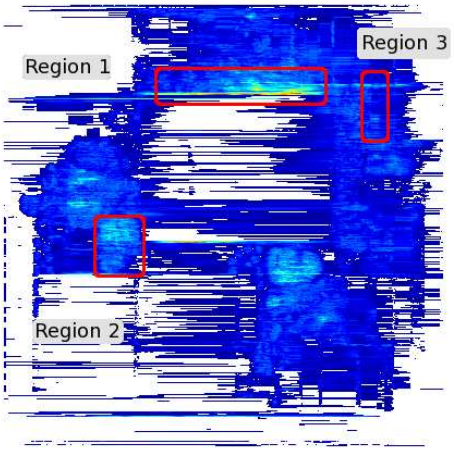
$$O_e = \begin{cases} 0 & U_e \leq C_e \\ U_e - C_e & U_e > C_e \end{cases}. \tag{7.2}$$

7.2 Motivation

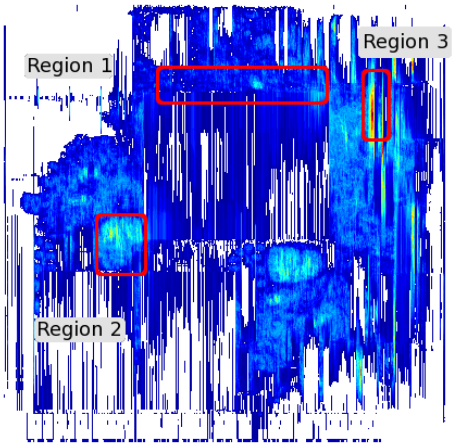
The most commonly used method to present the congestion for global routing problem is by generating a congestion heat map. Figure 7.1(A) present heat map uses visible spectrum to represent the congestion on the entire routing grid. Warm spectrum translates to highly congestion region, and cold spectrum to low congestion. The white spaces in this map presents regions with no routing utilization.



(A) Congestion heat map.



(B) Congestion heat map on horizontal layer.



(C) Congestion heat map on vertical layer.

Fig. 7.1: Comparing congestion heat map and orthogonal congestion heat maps. Benchmark adaptec2 from ISPD 2007 suite.

The heat map in Figure 7.1(A) is generated with *adaptec2* benchmark from the ISPD 2007 global routing suite [66]. The map is rendered after the initial routing stage, during which each net is routed with rectilinear Steiner minimal tree (RSMT) using flute [72]. Each pixel on the map can mirror to a corresponding gcell on the routing grid. A typical method to calculate the congestion of a gcell is to take the average of the congestion on its vertical and horizontal edge.

The congestion heat map presented in Figure 7.1(A) has limitations in abstracting

congestion information. Although it captures the geographical distribution of congested regions, the physical meaning of calculating the gcell congestion by averaging its orthogonal layers is very obscure. On a routing grid the horizontal and vertical tracks represent physically separated interconnect layers. Each layer houses interconnect going in one direction. Therefore, when an interconnect occupies one layer, its usage has no spatial correlation with that of another layer connecting in the orthogonal direction. As a result, the typically method to visualize a congestion map oversimplifies the abstraction with false correlation, which might mislead the designers and tools with inaccurate information.

Alternatively, separating the congestion on horizontal and vertical layers presents a much clearer vision of the congestion. This observation is shown in Figure 7.1(B) and Figure 7.1(C), which represents horizontal and vertical congestion, respectively. Interestingly, compared to the original heat map in Figure 7.1(A), several different congested regions can be identified with distinctive attributes, listed as the following:

- **Region 1** highlights the congestion region in the top middle portion in the three heat maps. In the first heat map a region of highly congested region is identified. However, the rest of the heat maps reveal that the congestion only exist in the horizontal layer, but not the vertical layer.

- **Region 2** highlights the congestion region in the lower left portion in the three heat maps. The same signature of highly congested region is observable in all three heat maps. Indicating that both horizontal and vertical directions have high density of routing usage.

- **Region 3** highlights the congestion region in the upper right portion in the three heat maps. Similar to region 1, the heat signature only appears in the first heat map and the vertical heat map. The horizontal layer is relatively congestion-free.

The heat signature patterns identified in Region 1 and 3 are treated very differently from that of Region 2. In these two regions, since the congestion is only dominant in one layer, the empty usage in the orthogonal direction allows the global router to distribute

the routing resource elsewhere via detours. Therefore, these congested regions are likely to be solved with ease by a global router. In contrast, since both directions in Region 2 are highly congested, creating detours in this region is significantly more difficult, indicating a hard-to-route area for the global router.

As a summary, the observed congested regions can be categorized into two types of congestion.

- **Branch Congestion**: The congested region exists in long stripy pattern, and only consumes routing resources in one direction. The cause of the branch congestion is typically long nets sharing a common path. Therefore, by redirecting the long nets with detours, global routers can easily solve the branch congestion.

- **Bush Congestion**: The congested region appears in clustered regional patterns, and shows high correlation of congestion between the orthogonal layers. The cause of the bush congestion is dense population of pins that are connected with relatively short interconnects. The routing resource is consumed in both directions, making the bush congestion difficult to solve.

7.3 Orthogonal Congestion Correlation

This section proposes the orthogonal congestion correlation (OCC) coefficient to capture the difficulty of routing In the light of the above observation. Based on the previous observations, this section focuses the discussion on the definition of OCC model.

The purpose of OCC coefficient is to abstract and quantify a "route difficulty" metric for global routing. The example shown in Section 7.2 asserts that using a fusion of orthogonal congestion to represent the routing difficulty can be misleading. For instance, the amount of usage overflow in a branch congestion region might surpass the one from bush congestion, but the former can usually be solved, while the later can lead to very poor routability. OCC coefficient is defined purposely to expose the bush congestion, and is less sensitive to the branch congestion. Therefore, the coefficient can effectively capture the poor routability regions that are responsible for the hard-to-route problems.

To achieve this purpose, OCC is defined as the correlation of usage overflow in horizontal and vertical edges within a square region r. The gcell in the center of r is assigned with the value of OCC. OCC is calculated using the following equation:

$$OCC_r = \frac{\sum_{e \in r} O_e^h \times \sum_{e \in r} O_e^v}{\frac{1}{N_O^h - 1} \sum O_e^{h\,2} \times \frac{1}{N_O^v - 1} \sum O_e^{v\,2}}, \tag{7.3}$$

where e denotes all edges within region r, and O_e denotes the overflow of an edge, the superscript h or v denotes horizontal or vertical layer. N_O denotes the number of overflow edges in the layer.

By generating an OCC heat map, the congestion analysis is able to pinpoint the difficult-to-route regions. Figure 7.2 is a OCC heat map rendered after the initial routing stage of *adaptec2* benchmark.

As evident from the OCC heat map, several regions for the initial routing solution are highlighted. These regions effectively pinpoints the hard-to-route portions of the current layout. However, the information presented in the early routing stage is still inaccurate since the routing solutions are created with RSMT. Therefore, its important to integrate

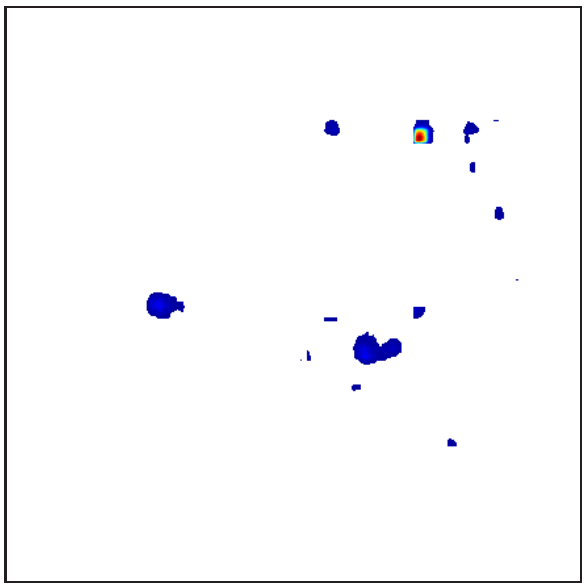


Fig. 7.2: OCC heat map after initial routing. Heat map highlights major hard-to-route areas.

a global router to the DOC framework. The router targets to iteratively resolve branch congestion and expose bush congestion, which are typically hard to route. The routing difficulty is monitored with the OCC coefficient, and stop the routing process when the OCC reading converge to a steady level.

7.4 Routing Technique

This section explains the routing techniques of DOC framework. A distinctive characteristic of DOC router is to rapidly eliminate the branch congestion, while solving the bush congestion as much as possible. DOC is designed from ground-up to use dynamic pattern based moves to generate new net topologies. These moves are rendered to avoid congested regions, and limit solution wire length by taking the net topology into consideration.

The DOC router is based on the widely used rip-up and reroute (RRR) framework for global routing. For a given net, the basic flow of DOC routing technique is listed as the following:

- *Identify the congested portion of the net.* Using the RSMT representation to rapidly pin point the congested portion of the net.
- *Generate detour to avoid the congestion.* A combination of patterns and dynamic edge movements are used to create detours. In comparison to maze routing, these techniques can generate new paths much faster.
- *Commit the new path to the net topology.* After a new path is found to replace to original one, it is committed back to the RSMT representation to renew the multi-pin topology.

A collection of moves are presented in this section. These moves are performed to overflowed Steiner edges, and relocate these edges to a congestion free region. The advantages of using moves instead of a routing heuristic is that dynamic edge shifting is much more runtime efficient. The complexity of each edge move is constant, yet effective to resolve the majority of congestion without introducing scenic detours.

Figure 7.3 highlights five basic moves that can be generated in a net topology. The green squares represent nail nodes, and blue circles represent anchors. From Figure 7.3(A) to 7.3(E), we demonstrate *Parallel Detour*, *Z-bend Detour*, *L-shape Move*, *H-shape Move*, and *Z-shape Move* transformations.

Parallel detour and Z-bend detour both use pattern routing [37, 80] techniques to generate a new net topology. As explained in Figure 7.3(A), a *parallel detour* of CD is created at EF location, then edges AC and BD are removed, replaced with EA and FB. Parallel detour is the backbone approach to redirect resource consumption from long stripy congested regions to less congested ones. The *Z-bend detour*, shown in Figure 7.3(B), reshapes the original L-shape bend to a Z-bend detour. This detour allows us to generate topology with more zigzag patterns to adapt to the complex routing terrain. Figure 7.3(C) illustrates an L-shape path shift from one corner of the rectangle to the other. Figure 7.3(D) shows a parallel move based on an H-shape topology. Figure 7.3(E) shows another parallel move based on a different topology (Z-shape). Importantly, most of these moves do not introduce additional wire length while making changes to a connected net topology.

7.5 Results

This section presents the experimental results. DOC framework is implemented with C++ on an Intel Quad-core 3.5 GHz Sandy Bridge processor with 8GB of RAM. The performance of DOC router is shown using the ISPD 2007 and 2008 suite [66, 67]. The router stops once the reported OCC value converges to a steady level. Then it records the run time, the percentage of resolved overflow, and the OCC_{avg} coefficient. These results are listed in Table 7.1.

All benchmarks are finished within approximately one minute, averaging 28 seconds for the 16 benchmarks. These results show the fast performance of the DOC framework, which is a key property of congestion analyzer. Within the short time window, the router also manages to resolve the majority of overflow, range from 91% to 99.5% of the total overflow have reduced across all benchmarks, effectively revealing the congestion hot-spot.

The OCC_{avg} coefficient is reported to show the overall correlation of the remaining

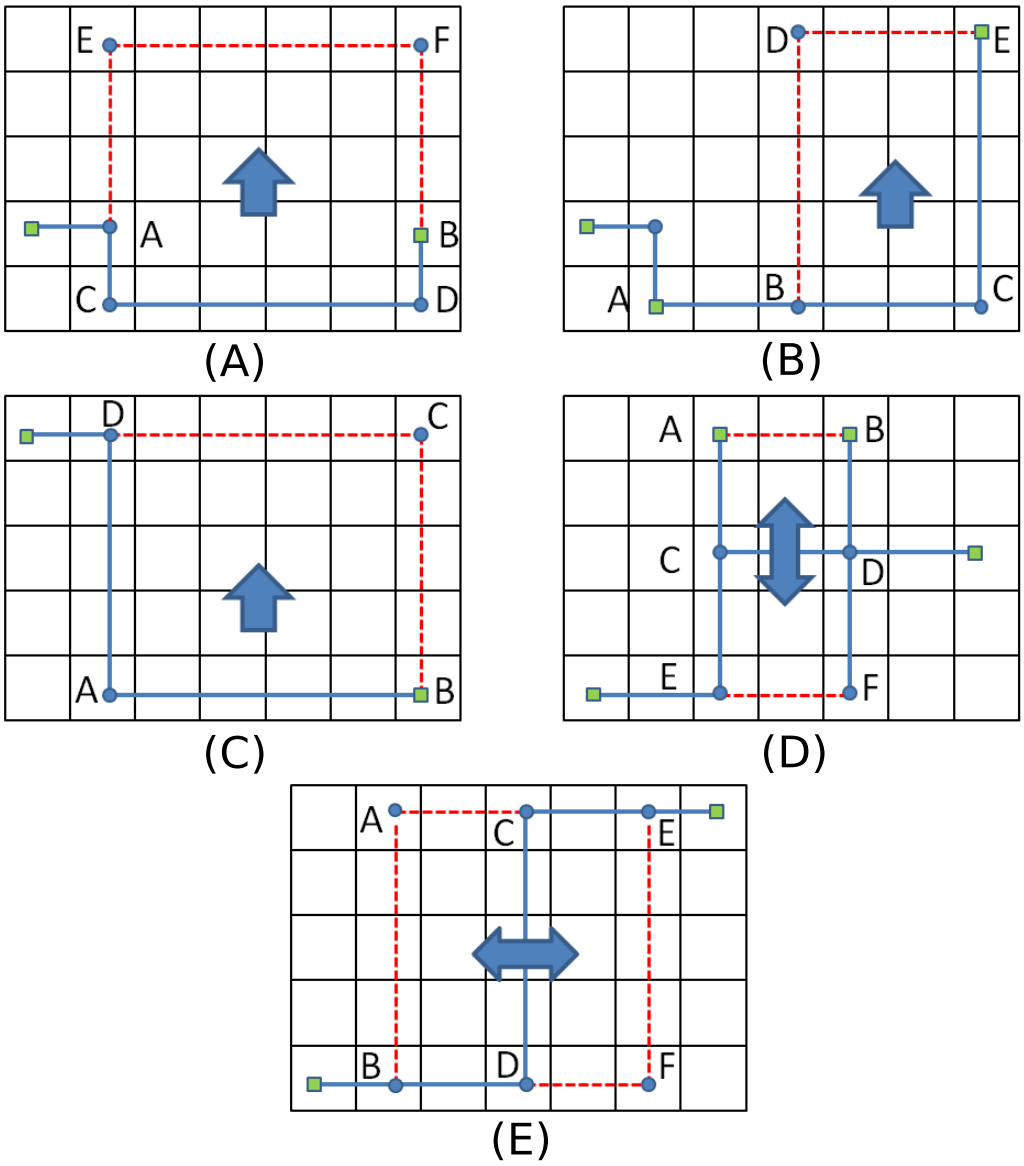


Fig. 7.3: Different types of moves to be generated on the net topology.

congestion. The average coefficient is calculated as $OCC_{avg} = \frac{1}{N_O} \sum OCC_r$ where N_O is the number of overflow edges. The value of OCC_{avg} is a general guideline of the routing difficulty of the benchmark. For example, the easiest benchmark newblue2 have reported the lowest correlation percentage. However, since the coefficient does not reveal the absolute value of congestion, it is not a standard for comparison of routing difficulty among different routing problems.

Table 7.1: ISPD benchmark results.

	Runtime	Resolved Overflow	OCC_{avg}
adaptec1	15.9 sec	96.46%	6.32%
adaptec2	9.6 sec	97.92%	10.06%
adaptec3	34.5 sec	98.06%	3.90%
adaptec4	10.3 sec	99.51%	4.87%
adaptec5	40.9 sec	97.77%	8.65%
newblue1	9.1 sec	92.51%	6.81%
newblue2	8.5 sec	97.38%	0.06%
newblue3	25.4 sec	96.46%	4.49%
bigblue1	25.6 sec	94.03%	10.00%
bigblue2	30.5 sec	91.46%	4.62%
bigblue3	9.8 sec	98.89%	4.79%
bigblue4	37.3 sec	95.81%	6.11%
newblue4	28.4 sec	96.79%	7.79%
newblue5	45.3 sec	97.66%	9.10%
newblue6	61.8 sec	94.08%	8.98%
newblue7	52.9 sec	98.08%	2.91%

The major purpose of DOC is to identify congested regions. Figure 7.4 shows the OCC heat map of the hard-to-route newblue7 benchmarks from ISPD 2008. The DOC analysis pin-pointed the regions that are causing the congestion. These congestion information can feedback to the global placer for cell allocation adjustment, which is beyond the scope of this dissertation.

7.6 Conclusion

This work presents DOC, a fast and accurate congestion analysis framework that effectively identifies the hard-to-route portion of the global routing problem. The proposed

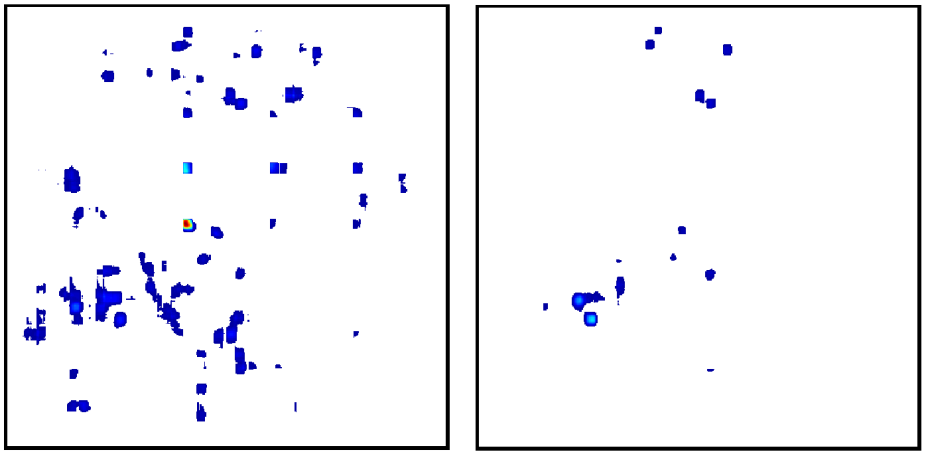


Fig. 7.4: OCC heat map of newblue7 Benchmark before and after DOC analysis.

approach uses novel OCC coefficient to capture the hotspots of global routing problem. An efficient dynamic pattern based router is included in the DOC framework to facilitate accurate calculation of the OCC heat map. The experimental results show that the proposed approach can accurately abstract regions with high usage density within one minute of run time through all tested benchmarks. The proposed framework can be further integrated to a global placement framework and improve the congestion awareness of the design.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This work has covered methodologies to parallelize EDA applications including floorplanner and global router, and implemented these parallel algorithms on GPU architecture. Experimental results have shown that several inherently sequential algorithms can benefit from massively parallel architecture through an algorithmic overhaul.

The parallel simulated annealing floorplanner breaks the long chain of data dependencies to uncover concurrency. This work presents a novel solution space and design space exploration to strike for a balance between performance and solution quality. The GPU floorplanner renders a significant 6-188X speedup compared to the sequential algorithm for a range of MCNC and GSRC benchmarks, while delivering comparable or better solution quality.

Two GPU based global routing have been proposed. The first one uses a GPU-CPU hybrid framework to maximize the computational throughput. A novel net level concurrency model is presented to allow scalable and deterministic parallel routing. An average of 4X speedup over NTHU-Route 2.0 is reported in the experiments using this approach. The second GPU global router identifies several limitations in the previous work: (1) a lack of exploitable concurrency of the net level concurrency model, and (2) GPU-CPU communication overhead of the hybrid framework. To tackle these issues, a new GPU routing engine is developed based on a GPU A*search implementation. A novel fine grain concurrency model is used to extra parallel workload, and is shown superior to the net level concurrency model in terms of scalability.

Lastly, a congestion analysis framework called dynamic orthogonal congestion analysis

is proposed. It targets to quickly identify hard-to-route regions on a routing map. The difficulty is evaluated using a novel orthogonal congestion correlation factor. Coupled with a fast routing engine using only dynamic edge shifting move, this congestion analysis is shown with the ability to pinpoint hard-to-route areas with tremendous speed.

8.2 Future Work

This research has a substantial impact on the future EDA design flow. As suggested in previous chapters, the fast global router based on GPU architecture can be modified for a high performance congestion analysis tool. In an comparison with existing probability or rectilinear Steiner minimal tree (RSMT) based congestion analysis tools, the global router based design can significantly improve the congestion prediction accuracy, which is a key factor in the current and future EDA design flow. Historically, the global router has not been considered as the mainstream technique for congestion analysis due to its extensive run time. But with the parallel models presented in this work, global router engine can scale much better on modern commodity multi-core as well as GPU systems, allowing a significant reduction of run time.

Future EDA design flow can directly benefit from the aforementioned advantage by incorporating the proposed multi-threaded routing engine with upstream physical design tools such as floorplanning and placement. The parallel routing engine can produce accurate congestion information to design tools in an iterative and interleaved fashion, allowing IC designers to avoid potential design flaws in the early stages. As a whole, the EDA design flow can enjoy a faster design convergence as well as shortened turn-around time.

As final remarks, although a successful implementation of the parallel global router is presented, it can still be further improved to extend the abilities in designs of various specifications. For this purpose, a few suggestions are discussed in the following sections.

8.2.1 Parameter Tuning of Global Router

Tuning plays an important role in designing a sophisticated and error-resilient global routing. A well-tuned router not only delivers good solution quality in the selected bench-

marks, but also needs to excel in a variety of workloads. The various parameters throughout different stages of global routing can be tuned to give the router different characteristics. When considering a wide range of routing workloads, it is interesting to see whether the global router can adapt to the workload, which self-configures with the characteristic that is best matched for the problem.

In addition, although several tuning methods for sequential routers have been proposed in the past decade and steadily improved the performance of global router. The parallel router proposed in this work (Chapters 5 and 6) exhibits different characteristics from the sequential router in many subtle ways. For example, creating and expanding bounding boxes, and the use of congested regions for work queue generation, both call for very different tuning measures that tradeoff between performance and solution quality. In-depth understanding of these tradeoffs can help optimizing the parallel router to achieve superior solution quality, as well as runtime efficiency.

8.2.2 Tackling Limitations of Bounding Box

Both the NLC and FGC parallel models in Chapters 5 and 6 rely on the use of bounding box for all routing tasks. Although showcasing excellent ability to ensure a stable and deterministic parallelization in the tested benchmarks, routing with bounding box has several limitations for the real world applications.

- Tackling routing obstacles. The bounding box method assumes that routing resources are distributed uniformly on a routing grid. But it is not true in a routing problem where obstacles exist. Since no interconnect is allowed to route through these obstacles, they reduce the available routing resources within their regions. This constraint leads to an irregular routing resource distribution. Due to this reason, applying bounding box to restrict routing resources can degrade solution quality for some nets. The resources within the bounding box can be insufficient to fulfill a connectivity that would otherwise be more optimal if the bounding box restriction was not applied, or if the bounding box's shape had taken the obstacles into consideration.

- Bounding box expansion strategy. The current bounding box is expanded when a net is found unable to route with an expected wirelength. By giving the net larger area to negotiate a route, it is often able to find a more optimal solution. To some degree, this approach mitigates the aforementioned issue of routing obstacles. But increasing sizes of bounding boxes reduce the exploitable concurrency of the entire routing problem. Typically, the larger each net becomes, the higher likelihood that it can overlap with another net, which leads to a data dependency that must be solved in a sequential order.

Potentially, the above issues can be resolved with the following proposals: (1) A directional bounding box expansion/reduction strategy that takes the routing resource distribution and net topology into account; (2) Relax the data dependency caused by bounding box overlapping to a degree that does not hamper the stable and deterministic properties of parallelization, but allows more parallel workloads.

References

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics, State of the Art Reports*, pp. 21–51, Aug. 2005.

[2] J. Shi, Y. Cai, W. Hou, L. Ma, S. X. Tan, P.-H. Ho, and X. Wang, "GPU friendly fast poisson solver for structured power grid network analysis," in *Proceedings of 46th IEEE/ACM Design Automation Conference (DAC)*, pp. 178–183, 2009.

[3] K. Gulati and S. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proceedings of 45th IEEE/ACM Design Automation Conference (DAC)*, pp. 822–827, 2008.

[4] Z. Feng and P. Li, "Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 647–654, 2008.

[5] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Proceedings of 46th IEEE/ACM Design Automation Conference (DAC)*, pp. 557–562, 2009.

[6] Y. Liu and J. Hu, "GPU-based parallelization for fast circuit optimization," in *Proceedings of 46th IEEE/ACM Design Automation Conference (DAC)*, pp. 943–946, 2009.

[7] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 681–688, 2009.

[8] Y. Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 539–546, 2009.

[9] Y. Han, K. Chakraborty, S. Roy, and V. Kuntamukkala, "A GPU algorithm for IC floorplanning: Specification, analysis and optimization," in *IEEE International Conference on VLSI Design*, pp. 159–164, 2011.

[10] Y. Han, S. Roy, and K. Chakraborty, "Optimizing simulated annealing on GPU: A case study with ic floorplanning," in *IEEE International Symposium on Quality Electronic Design (ISQED)*, pp. 1–7, 2011.

[11] Y. Han, K. Chakraborty, S. Roy, and V. Kuntamukkala, "Design and implementation of a throughput-optimized GPU floorplanning algorithm," *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 3, pp. 23:1–23:21, June 2011.

[12] Y. Han, D. M. Ancajas, K. Chakraborty, and S. Roy, "Exploring high throughput computing paradigm for global routing," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 298–305, 2011.

[13] Y. Han, K. Chakraborty, and S. Roy, "A global router on GPU architecture," in *IEEE International Conference on Computer Design (ICCD)*, pp. 1–6, 2013.

[14] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "Multi-threaded collision-aware global routing with bounded-length maze routing," in *Proceedings of 47th IEEE/ACM Design Automation Conference (DAC)*, pp. 200–205, Anaheim, California, June 2010.

[15] Y. Frishman and A. Tal, "Multi-level graph layout on the GPU," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 13, no. 6, pp. 1310–1319, 2007.

[16] ——, "Online dynamic graph drawing," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 14, no. 4, pp. 727–740, 2008.

[17] S. N. Adya and I. Markov, "Fixed-outline floorplanning through better local search," in *IEEE International Conference on Computer Design (ICCD)*, pp. 328–334, 2001.

[18] ——, "Fixed-outline floorplanning: enabling hierarchical design," *IEEE Transactions on VLSI Systems (TVLSI)*, vol. 11, no. 6, pp. 1120–1135, 2003.

[19] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on B*-tree and fast simulated annealing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 25, no. 4, pp. 637–650, 2006.

[20] C. Zeng and Y. Chen, "Optimal random search, fractional dynamics and fractional calculus," *eprint arXiv:1310.7687*, Oct. 2013.

[21] J. Cong, J. Wei, and Y. Zhang, "A thermal-driven floorplanning algorithm for 3D ICs," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 306–313, 2004.

[22] T.-C. Chen, Y.-W. Chang, and S.-C. Lin, "IMF: interconnect-driven multilevel floorplanning for large-scale building-module designs," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 159–164, 2005.

[23] S. A. Kravitz and R. A. Rutenbar, "Multiprocessor-based placement by simulated annealing," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 567–573, 1986.

[24] J. S. Rose, W. M. Snelgrove, and Z. G. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 7, no. 3, pp. 387–396, 1988.

[25] D. J. Ram, T. H. Sreenivas, and K. G. Subramaniam, "Parallel simulated annealing algorithms," *Journal of Parallel and Distributed Computing*, vol. 37, no. 2, pp. 207–212, 1996.

[26] T. Lengauer, "Global routing and area routing," in *Combinatorial Algorithms for Integrated Circuit Layout*, pp. 379–454. New York: John Wiley & Sons, Inc., 1992.

[27] E. F. Moore, "The shortest path through a maze," in *Proceedings of the International Symposium on the Theory of Switching, and Annals of the Computation Laboratory of Harvard University*, pp. 285–292. Cambridge, MA: Harvard University Press, 1959.

[28] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961.

[29] F. O. Hadlock, "A shortest path algorithm for grid graphs," *Networks*, vol. 7, no. 4, pp. 323–334, 1977.

[30] E. W. Dijkstra, "A note on two problems in connexion with graphs." *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[31] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[32] J. A. Roy and I. L. Markov, "High-performance routing at the nanometer scale," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 6, pp. 1066–1077, June 2008.

[33] J.-R. Gao, P.-C. Wu, and T.-C. Wang, "A new global router for modern designs," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 232–237, 2008.

[34] Y.-J. Chang, Y.-T. Lee, J.-R. Gao, P.-C. Wu, and T.-C. Wang, "NTHU-Route 2.0: a robust global router for modern designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 29, no. 12, pp. 1931–1944, Dec. 2010.

[35] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "Boxrouter 2.0: Architecture and implementation of a hybrid and robust global router," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 503–508, 2007.

[36] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for FPGAs," in *ACM International Symposium on field-Programmable Gate Arrays (FPGA)*, pp. 111–117, Feb. 1995.

[37] M. Pan and C. Chu, "FastRoute: a step to integrate global routing into placement," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 464–471, 2006.

[38] ——, "FastRoute 2.0: a high-quality and efficient global router," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 250–255, 2007.

[39] Y. Zhang, Y. Xu, and C. Chu, "FastRoute 3.0: a fast and high quality global router based on virtual capacity," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 344–349, 2009.

[40] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: global router with efficient via minimization," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 576–581, 2009.

[41] K.-R. Dai, W.-H. Liu, and Y.-L. Li, "Efficient simulated evolution based rerouting and congestion-relaxed layer assignment on 3-D global routing," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 582–587, 2009.

[42] H.-Y. Chen, C.-H. Hsu, and Y.-W. Chang, "High-performance global routing with fast overflow reduction," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 570–575, 2009.

[43] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: scalable 3D global routing using integer programming," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 30, no. 1, pp. 72–84, Jan. 2011.

[44] M. Cho and D. Pan, "Boxrouter: a new global router based on box expansion and progressive ilp," in *Proceedings of 43rd IEEE/ACM Design Automation Conference (DAC)*, pp. 373–378, 2006.

[45] H. Shojaei, A. Davoodi, and J. Linderoth, "Congestion analysis for global routing via integer programming," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 256–262, 2011.

[46] A. B. Kahng and X. Xu, "Accurate pseudo-constructive wirelength and congestion estimation," in *Proceedings of the 2003 International Workshop on System Level Interconnect Prediction (SLIP)*, pp. 61–68, 2003.

[47] J. Lou, S. Thakur, S. Krishnamoorthy, and H. S. Sheng, "Estimating routing congestion using probabilistic analysis." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 21, no. 1, pp. 32–41, 2002.

[48] J. Westra, C. Bartels, and P. Groeneveld, "Probabilistic congestion prediction." in *International Symposium on Physical Design (ISPD)*, pp. 204–209, 2004.

[49] J. Westra and P. Groeneveld, "Is probabilistic congestion estimation worthwhile?" in *Proceedings of the 2005 International Workshop on System Level Interconnect Prediction (SLIP)*, pp. 99–106, 2005.

[50] M.-C. Kim, J. Hu, D.-J. Lee, and I. L. Markov, "A simplr method for routability-driven placement," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 67–73, 2011.

[51] M.-K. Hsu, S. Chou, T.-H. Lin, and Y.-W. Chang, "Routability-driven analytical placement for mixed-size circuit designs," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 80–84, 2011.

[52] X. He, T. Huang, L. Xiao, H. Tian, G. Cui, and E. F. Young, "Ripple: An effective routability-driven placer by iterative cell movement," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 74–79, 2011.

[53] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[54] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, "B*-trees: a new representation for non-slicing floorplans," in *Proceedings of 37th IEEE/ACM Design Automation Conference (DAC)*, pp. 458–463, 2000.

[55] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, "An O-tree representation of non-slicing floorplan and its applications," in *Proceedings of 36th IEEE/ACM Design Automation Conference (DAC)*, pp. 268–273, 1999.

[56] D. Cross, E. Nequist, and L. Scheffer, "A DFM aware, space based router," in *International Symposium on Physical Design (ISPD)*, pp. 171–172, 2007.

[57] N. Kaul, "Design planning trends and challenges," in *International Symposium on Physical Design (ISPD)*, p. 5, 2010.

[58] C. J. Alpert, Z. Li, M. D. Moffitt, G.-J. Nam, J. A. Roy, and G. Tellez, "What makes a design difficult to route," in *International Symposium on Physical Design (ISPD)*, pp. 7–12, 2010.

[59] R. C. Johnson. (2010, Mar.) EE Times: IBM warns of 'design rule explosion' beyond 22-nm. [Online]. Available: http://www.eetimes.com/electronics-news/4088244/IBM-warns-of-design-rule-explosion-beyond-22-nm

[60] P. Groeneveld, "Going with the flow: bridging the gap between theory and practice in physical design," in *International Symposium on Physical Design (ISPD)*, p. 3, 2010.

[61] M. D. Moffitt, J. A. Roy, and I. L. Markov, "The coming of age of (academic) global routing," in *International Symposium on Physical Design (ISPD)*, pp. 148–155, 2008.

[62] J. Croix and S. Khatri, "Introduction to GPU programming for EDA," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 276–280, Nov. 2009.

[63] K. Gulati and S. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proceedings of 45th IEEE/ACM Design Automation Conference (DAC)*, pp. 822–827, June 2008.

[64] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 260–265, 2009.

[65] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "A parallel integer programming approach to global routing," in *Proceedings of 47th IEEE/ACM Design Automation Conference (DAC)*, pp. 194–199, 2010.

[66] "ISPD 2007 global routing contest and benchmark suite." [Online]. Available: http://www.sigda.org/ispd2007/rcontest/

[67] G.-J. Nam, C. Sze, and M. Yildiz, "The ISPD global routing benchmark suite," in *International Symposium on Physical Design (ISPD)*, pp. 156–159, 2008.

[68] Z. Cao, T. Jing, J. Xiong, Y. Hu, L. He, and X. Hong, "DpRouter: a fast and accurate dynamic-pattern-based global routing algorithm," in *Proceedings of Asia-Pacific Design Automation Conference (ASP-DAC)*, pp. 256–261, 2007.

[69] M. M. Ozdal and M. D. F. Wong, "Archer: a history-driven global routing algorithm," in *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 488–495, San Jose, California, Nov. 2007.

[70] M. D. Moffitt, "MaizeRouter: Engineering an effective global router," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 11, pp. 2017–2026, Nov. 2008.

[71] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "A parallel integer programming approach to global routing," in *Proceedings of 47th IEEE/ACM Design Automation Conference (DAC)*, pp. 194–199, 2010.

[72] C. Chu and Y.-C. Wong, "FLUTE: fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 1, pp. 70–83, Jan. 2008.

[73] J. Hu and S. S. Sapatnekar, "A survey on multi-net global routing for integrated circuits," *Integration, the VLSI Journal*, vol. 31, pp. 1–49, 2001.

[74] S. Khanna, S. Gao, and K. Thulasiraman, "Parallel hierarchical global routing for general cell layout," in *Proceedings of 5th ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 212–215, 1995.

[75] I.-L. Yen, R. Dubash, and F. Bastani, "Strategies for mapping Lee's maze routing algorithm onto parallel architectures," in *Proceedings of 7th International Parallel Processing Symposium (IPPS)*, pp. 672–679, Apr. 1993.

[76] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *International Conference on High Performance Computing (HIPC)*, pp. 197–208, 2007.

[77] L. Luo, M. Wong, and W.-M. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of 47th IEEE/ACM Design Automation Conference (DAC)*, pp. 52–55, Anaheim, California, June 2010.

[78] A. Bleiweiss, "GPU accelerated pathfinding," in *Proceedings of 23rd ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware (GH)*, pp. 65–74, 2008.

[79] J. Hu, J. A. Roy, and I. L. Markov, "Completing high-quality global routes," in *International Symposium on Physical Design (ISPD)*, pp. 35–41, 2010.

[80] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh, "Pattern routing: use and theory for increasing predictability and avoiding coupling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 21, pp. 777–790, 2001.

Vita

Yiding Han

Published Journal Articles

- Exploring High Throughput Computing Paradigm for Global Routing, Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy, *IEEE Transactions on Very Large Scale Integration Systems*, *Accepted*
- Analysis of Intermittent Timing Fault Vulnerability, Saurabh Kothawade, Koushik Chakraborty, Sanghamitra Roy, Yiding Han, *Microelectronics Reliability*, Volume 52, Issue 7, July 2012, Pages 1515–1522
- Design and Implementation of a Throughput Optimized GPU Floorplanning Algorithm, Yiding Han, Koushik Chakraborty, Sanghamitra Roy and Vilasita Kuntamukkala, *ACM Transactions on Design Automation of Electronic Systems*, Volume 16, Issue 3, No. 23, June 2011, Pages 23:1–23:21
- Random Delay Effect Minimization on a Hardware-in-the-loop Networked Control System Using Optimal Fractional Order PI Controllers, V. Bhambhami, Y. Han, S. Mukhopadhyay, Y. Luo, and Y. Q. Chen, *Communications in Nonlinear Science and Numerical Simulation-Special Issue*, 15(9), Sep. 2010, Pages 2486–2496

Published Conference Papers

- A Global Router on GPU Architecture, Yiding Han, Koushik Chakraborty, and Sanghamitra Roy, *IEEE International Conference on Computer Design*, 2013, Pages 74–80
- GPU Based Computer Aided Design Algorithms for EDA, Yiding Han, Ph.D. Forum Poster Session, *Design Automation Conference*, 2013

- DOC: Fast and accurate congestion analysis for global routing, Sanghamitra Roy, Yiding Han, Koushik Chakraborty, *IEEE 30th International Conference on Computer Design*, 2012, pages 508–509

- Exploring High Throughput Computing Paradigm for Global Routing, Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, Sanghamitra Roy, *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, November 2011, San Jose, Pages 298–305

- Optimizing Simulated Annealing on GPU: A Case Study with IC Floorplanning, Yiding Han, Sanghamitra Roy and Koushik Chakraborty, *12th IEEE International Symposium on Quality Electronic Design (ISQED)*, March 2011, Pages 1–7

- A GPU Algorithm for IC Floorplanning: Specification, Analysis and Optimization, Yiding Han, Koushik Chakraborty, Sanghamitra Roy and Vilasita Kuntamukkala, *24th IEEE/ACM International VLSI Design Conference*, 2011, Pages 159–164

- Fractional Order Networked Control Systems and Random Delay Dynamics: A Hardware-In-The-Loop Simulation Study, Shayok Mukhopadhyay, Yiding Han and YangQuan Chen, *American Control Conference*, 2009, Pages 1418–1423

- Random delay effect minimization on a hardware-in-the-loop networked control system using optimal fractional order PI controllers, V. Bhambhani, Y. Han, S.Mukhopadhyay, Y. Luo., and Y. Q.Chen, *Third IFAC Workshop on Fractional Differentiation and its Applications (FDA)*, Pages 1418–1423

- Mapping River Changes Using Low Cost Autonomous Unmanned Aerial Vehicles, Yiding Han, Huifang Dou, Yangquan Chen, research poster in *AWRA 2009 Spring Specialty Conference - Managing Water Resources Development in a Changing Climate*, May 4-6, 2009 in Anchorage, Alaska

- Programmable Multispectral Imager Development as Light Weight Payload for Low Cost Fixed Wing Unmanned Aerial Vehicles, Yiding Han, Austin Jensen, Huifang

Dou, *ASME 2009 International Design Engineering Technical Conferences (IDETC)*, Aug. 2009, San Diego

- Aggieair: An Integrated and Effective Small Multi-UAV Command, Control and Data Collection Architecture, Calvin Coopmans, Yiding Han, *ASME 2009 International Design Engineering Technical Conferences (IDETC)*, Aug 2009, San Diego

- Using Aerial Images to Calibrate The Inertial Sensors of a Low-cost Multispectral Autonomous Remote Sensing Platform (AggieAir), Austin Jensen, Yiding Han, YangQuan Chen, *IGARSS 2009*, Pages 555–558

Master Thesis

- An Autonomous Unmanned Aerial Vehicle-Based Imagery System development and Remote Sensing Images Cassification for Agricultural Applications, Yiding Han, *Utah State University* 2009.