Deterministic, Weak-scaling Parallelism for Wirelength- and Timing-driven FPGA Placement, suitable for Multicore and Manycore Architectures

by

Christian Fobel

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Doctor of Philosophy
in
Computer Science

Guelph, Ontario, Canada

ABSTRACT

Deterministic, Weak-scaling Parallelism for Wirelength- and Timing-driven FPGA Placement, suitable for Multicore and Manycore Architectures

Christian Fobel
University of Guelph, 2015

Advisors:
Professor G. Grewal and Professor D. Stacey

Field-Programmable Gate Arrays (FPGAs) enable rapid-prototyping of digital logic designs in-house, without the significant up-front expense of building custom fabrication facilities. However, as the number of resources on each new generation of FPGAs continues to grow rapidly, enormous pressure is placed on the development of algorithms to reduce hardware compilation times to maintain the competitive advantage of using FPGAs. Approximately half of the compilation time for FPGA designs is spent performing *placement* (which is NP-hard). Serial simulated annealing is typically used for placement in practice, where runtime unfortunately grows exponentially with circuit size to be placed. Therefore, development of parallel placement algorithms that can harness the increasingly abundant throughput of modern manycore architectures to improve runtimes remains a critical concern.

Parallel FPGA placement methods in literature provide no theoretical basis for scalability, and reported runtime results suggest non-parallelizable work scaling with the size of the problem, preventing scaling. We propose a parallel FPGA placement methodology based on "completely parallelizable" patterns, leading to a weak-scaling isoefficiency function in $\Theta(p \log p)$, while maintaining determinism. For placement, this means deterministic results where linear speedups are expected as the number of parallel worker threads (p) increases as long as the problem size (i.e., netlist size) grows accordingly. Using parallel patterns, we also propose the first scalable algorithms for timing analysis, which are ideally suited for modern manycore architectures, such as GPUs.

Our experimental results show that our proposed wirelength- and timing-driven placement tools achieve mean absolute runtime improvements of 19× and 31×, respectively, on a commodity GPU over a state-of-the-art academic placer (VPR). With respect to quality, our wirelength-driven tool improves solution quality by 5% over VPR, while our timing-driven placer improves critical-path delay by 20% compared to our proposed wirelength driven method. Our results also indicate increased parallel efficiency as the size of the problem grows. Since both the parallel worker count on modern commodity parallel architectures and the number of resources available on modern FPGAs are growing rapidly, weak-scaling is an ideal fit for parallel FPGA placement to provide sustainable performance for future FPGA designs and manycore architectures.

Acknowledgements

First and foremost, I want to thank my advisors Gary Grewal and Deborah Stacey who promoted strong critical thinking and helped me to become a better research scientist, and whose funding assistance helped significantly. Your feedback and advice was invaluable while writing papers and my thesis dissertation. I particularly want to thank you for supporting my ideas and research tangents that eventually lead to the contributions in this thesis.

Further, I would like to thank my advisory committee members Dilip Banerji and Andrew Morton for their insightful commentary and critiques of my research. I would also like to thank William Gardner for his suggestions which lead me to strengthen the theoretical foundation of my thesis work. In addition, my thanks goes out to Kenneth Kent for his time and for lending his expertise as examiner at my defense.

I would like to extend a warm thanks to my labmates, Max Walton, Sama Sheikh Nia, and Ryan Pattison for lending an ear when needed and for helping to blow off some steam now and then.

I would also like to recognize the financial assistance of NSERC and the University of Guelph School of Computer Science (Teaching Assistantships, Graduate Research Scholarships) and express my gratitude to those agencies.

To my good friends and housemates Frank Hanshar and Brad Chruszcz, I would like to express my gratitude. Frank, thank you for all of our late night intellectual discussions and for helping me to stay motivated. Brad, thanks for listening to me yammer on far too many times about FPGA placement and for helping to keep me sane by continuing to invite me to participate in events outside my research bubble, though I was seldom able to return the favour.

To my family, words cannot express how thankful I am of your unconditional love and

support throughout my research. Ryan, you took the time to listen and provided insights from an outside perspective. Mom and Dad, you have instilled in me a great sense of wonderment and a love of learning. You believed in me whenever I started to doubt myself and were always ready to listen, especially during the rough parts of my research when nothing seemed to work as expected.

Finally, to my loving fiancée, Nancy. Thank you for being patient and making considerable personal sacrifice supporting me during my research. On more than one occasion you pulled me back to normalcy when I was having a meltdown. You were a shoulder to lean on and could always bring a smile to my face when I needed it the most. I couldn't have done it without you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) are a key enabling technology for rapid prototyping of digital logic designs. Compared to *Application-Specific Integrated Circuit* design, FPGAs represent a significant reduction in non-recurring engineering cost, since the devices are pre-manufactured and may be purchased in any volume without the need to build a custom fabrication facility. Furthermore, since FPGA devices are reprogrammable, repeated design iterations may be quickly performed in-house without requiring fabrication. However, as the number of resources continues to increase significantly with every new generation, advanced *Computer-Assisted Design (CAD)* tools are more crucial than ever.

The typical automated software flow for FPGA design has four main stages, as depicted in Fig. 1.1. Logic synthesis is performed on a description of the circuit to be implemented, to translate from a high-level hardware description language, such as Verilog, into units of logic that correspond to the types of resources available on the target FPGA architecture. The result of logic synthesis is a netlist representation of the circuit to be implemented, which consists of various types of blocks, including inputs, outputs, combinational logic, synchronous logic, memories, etc. as well as the connections between the blocks. The next step in the flow is called "placement", which is the primary focus of this thesis. In placement, the blocks in the netlist representation are mapped onto the resources available on the target architecture, while attempting to minimize one or more objective costs, the most common of which is estimated wirelength. Once all blocks have been placed, routing is performed to define the configuration of the programmable interconnect network to make all of the required connections for the

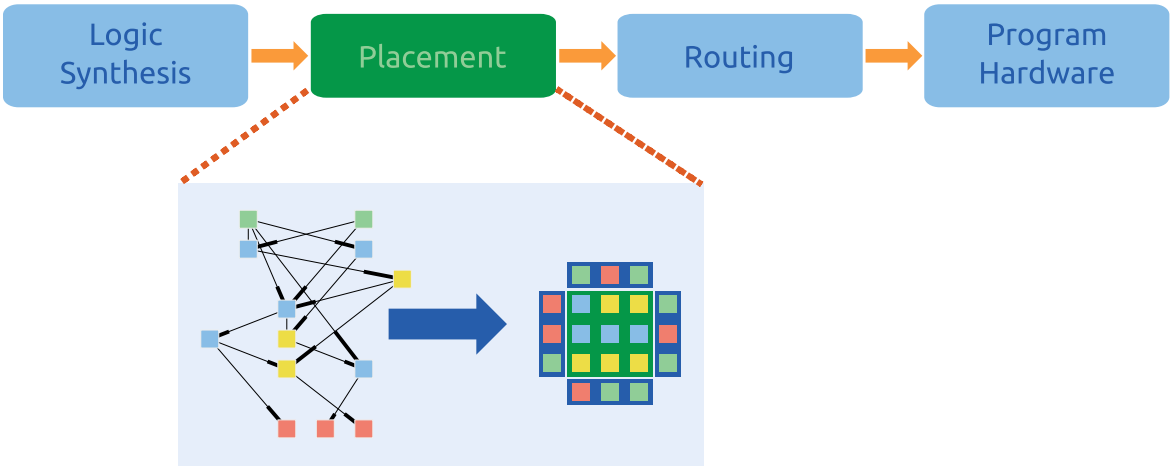


Figure 1.1: Automated *Computer-Aided Design (CAD)* flow for *Field-Programmable Gate Arrays (FPGAs)*.

netlist. The final step in the FPGA design flow is the generation of what is referred to as a "bitstream" file, which is a binary-encoded representation of the circuit implementation that may be programmed onto the FPGA using vendor-supplied tools.

While there are many methods for performing FPGA placement, the most widely used methods rely on simulated annealing, which is a serial algorithm where the runtime scales exponentially with the size of the problem. Since the number of resources available on modern FPGAs continues to grow significantly with each new generation, and the most commonly used algorithms for placement are serial, the only runtime improvement for placement tools afforded by new computing hardware is due to increased single-core performance. However, since 2005, single-core performance has not improved significantly, as it had for several decades prior. Instead, over the last 10 years manufacturers of commodity processors have focused on adding more and more processing cores, operating at the same or lower frequencies. In this thesis, we present a parallel placement strategy composed of "best known" parallel programming patterns, such that the amount of parallel work scales directly with the size of the netlist to be placed, while the amount of serial work remains fixed. Furthermore, our proposed approach leads to theoretical weak-scaling, predicting linear speedup with constant efficiency as parallel workers are added, assuming the problem size increases accordingly. Therefore, the method that we propose for parallel placement provides a sustainably scalable solution for performing

FPGA placement for future FPGA architectures and manycore processing architectures on which to perform the placement. Moreover, by composing our approach using parallel patterns, we make it possible to parallelize placement while maintaining determinism, which ensures quality remains consistent, regardless of the number of parallel workers. In the context of parallel placement based on simulated annealing, determinism ensures that: a) the same placement result is obtained for multiple runs starting from the same random seed value, and b) the same placement result is obtained regardless of the number of parallel workers.

1.1 Contributions

The key contributions of our work include:

Concurrent Associated-Moves Iterative Placement (CAMIP)

We propose our *Concurrent Associated-Moves Iterative Placement (CAMIP)* model, which maps all non-constant time operations to "best known" [1] parallel patterns with defined work and span complexity leading to an isoefficiency function [2][1] in $\Theta(p \log p)$ (where p is the number of parallel workers and $\Theta(x)$ is the set of all functions with the same grow rate as x [3]). In other words, speedups are expected to scale linearly as the number of parallel workers is increased as long as the size of the problem grows at the rate $\Theta(p \log p)$. Our proposed model is general enough to operate on any set of concurrent moves that are grouped into subsets, which when applied, ensure a valid placement is maintained. Furthermore, any objective cost model may be used, so long as an aggregate cost may be assigned to an entire netlist, and a difference in cost can be computed relative to each block based on moving the block to a new position. Any method of move generation with any objective cost model that meet these criteria can be evaluated and applied within our general search model that is capable of executing a simulated annealing schedule. The work complexity of all operations (not including the cost model) is $\Theta(B)$ and the span complexity is $\Theta(\log B)$, where B is the number of blocks in the netlist.

Associated move-pair patterns

To address one of the key issues facing existing parallel placement methods, we propose

[1] We discuss scalability with respect to isoefficiency in Section 2.3.2.

a novel method of generating very large sets of concurrent moves where the number of moves scales along with the size of the netlist to be placed and all moves are guaranteed not to conflict with one another. Moreover, our approach allows the move for each block to be computed independently and in $\Theta(1)$ complexity. This makes our pattern-based move generation method suitable for application using the map parallel pattern to provide sets of moves to be used within the context of our CAMIP model.

Parallel wirelength cost model

We present a formulation of the Star+ wirelength model, where the Star+ cost associated with each net in the netlist may be computed concurrently, using a category-reduce parallel pattern. In addition, our model supports calculating the difference in cost for nets connected to each block based on each block moving to a new proposed position. Note that the difference in cost for each block is computed as if no other blocks are moving. We map all calculations related to Star+ wirelength cost onto structured parallel patterns with work complexity $\Theta(C)$ and span complexity $\Theta(\log C)$, where C is the number of connections in the netlist, predicting weak-scaling with an isoefficiency function [2] in $\Theta(p \log p)$.

Parallel timing cost model

We propose parallel methods for computing all timing-based calculations required for the connection-based timing model from T-VPlace [4] (i.e., VPR in timing-driven mode). This includes calculating the longest incoming path and longest outgoing path for each block in the netlist, computing a criticality cost for each connection in the netlist, and computing the estimated difference in connection costs due to moving each block to a new proposed position. The resulting per-block differences in connection costs may be combined with the differences in wirelength costs by block to perform multi-objective placement. Note that all timing calculations are implemented using best known parallel patterns, where the longest path delays for each block may be computed with $\Theta(C)$ work complexity and $\Theta(L \log C)$ span complexity, and L is the number of synchronous delay levels in the netlist. The connection cost values may be computed with $\Theta(C)$ work complexity and $\Theta(\log C)$ span complexity. The isoefficiency function [2] of all timing calculations in our proposed approach is in $\Theta(p \log p)$, predicting weak-scaling. Our algorithms, based on efficient,

structured parallel patterns, present the *first* FPGA timing analysis algorithms suitable for targeting modern manycore architectures, such as GPUs.

Wirelength CAMIP

We propose a GPU-based, parallel, wirelength-driven placement tool, based on our CAMIP model and our parallel Star+ wirelength cost model, which we call "W-CAMIP". Our proposed W-CAMIP has work complexity of $\Theta(C)$, span complexity $\Theta(\log C)$, and an isoefficiency function [2] in $\Theta(p \log p)$, suggesting that speedup is expected to increase along with the number of parallel workers (p) as long as the netlist size grows at a rate of $\Theta(p \log p)$. Our experimental results show that across a set of benchmarks from the literature, our proposed wirelength-driven tool run on a commodity GPU gained mean improvements in absolute runtime of 19× over VPR in `bounding_box` mode run on a commodity workstation CPU, for the same temperature schedule and number of moves at each temperature. Moreover, our results show improvement of 5% in wirelength and critical-path delay compared to VPR. Note that by using structured parallel patterns, we make it possible to maintain high levels of parallelism while maintaining determinism, ensuring that, given the same random seed, the same placement result is produced regardless of the number of parallel workers.

Timing CAMIP

Based on our proposed parallel timing cost model, and our CAMIP model, we present a multi-objective, timing-driven placement tool which we call "T-CAMIP". Note that, to the best of our knowledge, our proposed implementation is the first to perform *any form* of timing calculations for FPGA placement on a GPU. All calculations, including critical-path delay, connection costs, wirelength costs, and move assessments, are mapped to highly efficient parallel patterns where the overall work complexity is $\Theta(C)$, the span complexity is $\Theta(L \log C)$, and the isoefficiency function is in $\Theta(p \log p)$, where L is the number of synchronous delay levels in the netlist. Our experimental results show a mean improvement in absolute runtime of 31× for our parallel T-CAMIP implementation running on a commodity GPU over VPR in timing-driven mode running on a workstation CPU. Moreover, our timing-driven placer improves mean critical-path delay by 20% over our wirelength-driven implementation.

1.2 Thesis organization

The remainder of this thesis is organized as follows. In Chapter 2, we describe relevant background information including a formal description of the FPGA placement problem and categories of previously proposed FPGA placement methods. We also discuss trends in parallel computing, parallel complexity theory (including scalability with respect to isoefficiency [2]), and the high-level, structured parallel computing patterns used to compose our proposed placement method. In Chapter 3, we review the most successful parallel FPGA placement methods proposed in the literature to-date and identify key outstanding issues, which we address in our work. In Chapter 4, we propose our *Concurrent Associated-Moves Iterative Placement* methodology, which utilizes patterns to generate very large sets of moves that may be evaluated concurrently without conflicts, addressing a key issue with existing parallel placement approaches. Furthermore, all operations in our proposed method are mapped to parallel patterns with known complexity. In Chapter 5 and Chapter 6, we propose models for computing both wirelength costs and timing costs using only structured parallel patterns, and present theoretical complexity analysis which predicts scaling runtime speedup as the size of the problem grows. We also present empirical results based on the GPU implementations of our proposed wirelength-driven and timing-driven parallel placement algorithms, respectively. Our experimental results demonstrate that our proposed wirelength-driven placement tool achieves a mean absolute runtime improvement of 19× on a commodity GPU over a state-of-the-art academic placer [4] run on a commodity workstation CPU, while improving solution quality by 5%. Moreover, our proposed parallel timing-driven placer run on a commodity GPU exhibits absolute runtime improvements of 31× over T-VPlace run on a commodity workstation CPU, and improves mean critical-path delay by 20% over our wirelength-driven placer. In Chapter 7, we summarize our conclusions and present future work based on our proposed contributions. Publications relating to FPGA placement and the work in this thesis include [5–14].

Chapter 2

Background

Placement is one of the most important and time-consuming steps in *Field Programmable Gate Array (FPGA)* design. Figure 2.1 shows the general architecture of an FPGA. The input to the FPGA placement problem is a netlist describing various types of logic blocks and the interconnections between them, along with an architecture to which the circuit is to be mapped. The result of placement is a mapping of all blocks onto physical resources (*I/O Pads* and *Configurable Logic Blocks* **(CLBs)**), on the target FPGA, such that one or more objective functions are minimized. The most basic objective is to minimize the amount of interconnect (estimated by wirelength) required to route the signals between blocks. Wirelength reduction is an important goal since the amount of interconnect resources on an FPGA is limited. Thus, if the interconnect required exceeds the routing resources available on the FPGA, the circuit will not be routable. Minimizing the routing length will also reduce several related design parameters. For example, lower routing resource usage leads to a reduction in power dissipation for the circuit and shorter routing paths result in less delay due to resistance and capacitance. In addition to wirelength, other objectives may be considered[1] including minimizing the critical-path delay for the circuit, or routing congestion. Since the vast majority of circuit designs are sequential and use a clock, reducing the delay along the critical-path will increase the maximum operating clock frequency and will thus maximize the performance of the design. Reducing congestion helps to increase the routability of the circuit by balancing resource usage across the chip.

Unfortunately, with the latest FPGA lines from Xilinx and Altera reaching over 1,995,000

[1]Wirelength, timing, and congestion objectives are discussed in Section 2.2.

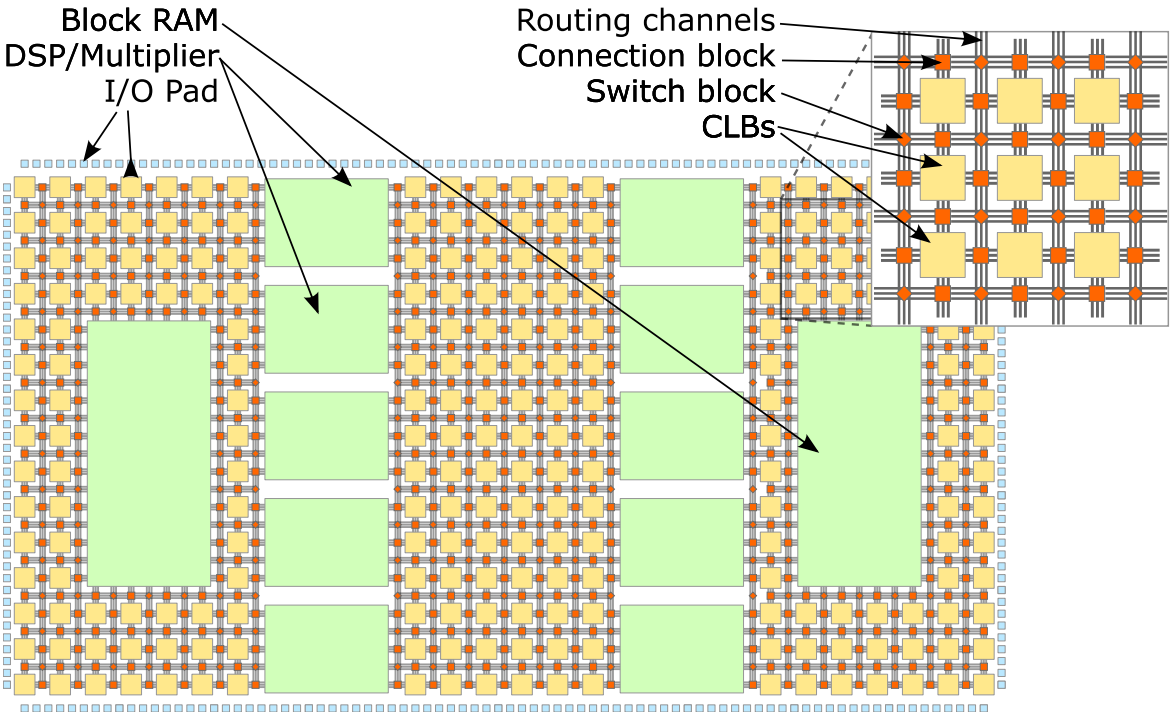


Figure 2.1: Generic FPGA architecture.

logic cells [15, 16], compilation times can easily take tens of hours to complete, where placement accounts for roughly half of the total compilation time. This provides compelling motivation to explore new, faster methods for performing placement.

Many different approaches to FPGA placement have been proposed in literature within the last few decades. Most of the proposed placement methods can be grouped into one of four classes: i)*analytic methods* [17–21]; ii)*partitioning-based* [22–24]; iii)*evolutionary algorithms* [25, 26]; or iv)*iterative improvement* [4, 27–31][2]. In partitioning-based placement, a circuit is recursively bisected, minimizing the number of cuts of nets (i.e., sets of pins that must be connected) that connect blocks between partitions, while leaving highly-connected nets within the same partition and balancing the number of blocks in each partition. The process continues until the number of blocks in each partition is sufficiently small, at which point the blocks are assigned to locations/resources on the FPGA. Partitioning-based tools are usually very fast, but since cut size is not a function of wirelength, the quality of solution is not as good as other methods. Analytic placement techniques model the placement problem as a set of equations and typically

[2]Section 2.2 discusses these placement methods in further detail

utilize fast numerical solvers to minimize an objective, such as wirelength. Analytic methods solve a system of equations, producing real value coordinates for block positions. Since the logic resources on an FPGA are in fixed positions on the chip, legalization must be performed to assign each block to a valid position on the FPGA. Analytic methods typically have much shorter runtimes compared to other placement techniques, but solution quality usually is lower compared to other techniques. Limited research has also been done applying *Evolutionary Algorithms* **(EAs)** to FPGA placement, as proposed by Y. Men et al. [26]. While the solution quality of the experimental results reported were comparable to VPR [32], no runtime comparison was reported, likely indicating much longer runtimes for the GA.

In practice, the best results are produced by *Simulated-Annealing* **(SA)** based methods. These methods begin with an initial placement and iteratively search for small perturbations to the placement resulting in better solutions. An example of such a method is the well-known *VPlace*, which is part of the VPR tool-set [4]. VPR employs the basic template for SA, but with several very effective placement-specific enhancements. Overall, VPR provides very good results, and is widely used in the FPGA research community. However, the simulated annealing approach employed in VPR cannot execute moves in parallel without some mechanism to deal with conflicts that may occur, as we discuss later in Section 3.2. The only deterministic variants of simulated annealing that support parallel move evaluation require a serialized queue [33, 34] that does not allow constant parallel efficiency [3] to be maintained as the number of parallel workers increases since contention over the shared queue grows along with the number of workers.

In this chapter, we provide a formal description of the FPGA placement problem, including objectives considered during placement, such as wirelength or timing. In Section 2.2, we discuss several classes of approaches to placement. In Chapter 3, we extend the discussion of iterative placement methods, by covering *parallel* simulated annealing methods for FPGA placement, where we identify key issues that prevent the parallelism in existing approaches from maintaining constant parallel efficiency as the number of workers is increased. Our contributions presented in Chapters 4-6 directly address the issues identified in the previous work.

[3] We define parallel efficiency in Section 2.3.2.

2.1 Overview of FPGA placement

The FPGA placement problem typically begins with a netlist describing logic blocks and the interconnections between them, along with an architecture to which the circuit is to be mapped. The result of placement is a mapping of logic blocks from the netlist to the physical resources available on the FPGA. A formal definition of the FPGA placement problem follows.

Given:

$$\begin{array}{ll} B = \{b_1, b_2, b_3, \ldots, b_n\} & \text{a set of logic blocks,} \\ S = \{s_1, s_2, s_3, \ldots, s_m\} & \text{a set of signals,} \\ L = \{l_1, l_2, l_3, \ldots, l_k\} & \text{a set of physical locations on the FPGA} \end{array} \tag{2.1}$$

where $|B| \leq k$,

$$\forall\ b_i \in B\ \exists\ S_{b_i} = \{s_j : s_j \text{ sent or received by } b_i\} \tag{2.2}$$

$$\forall\ s_i \in S\ \exists\ B_{s_i} = \{b_j : b_j \text{ sends or receives } s_i\} \tag{2.3}$$

In other words, $S_{b_i} \subset S$ is the set of all signals that are sent or received by block b_i, while $B_{s_i} \subset B$ is the set of all blocks that send or receive signal s_i. The signals in S correspond to wire "nets," which are connected to multiple blocks (as defined in the *netlist*). The locations in L correspond to either *Configurable Logic Blocks* **(CLBs)** or *I/O pads* on the FPGA. Each CLB or I/O pad has one or more pins, which act as potential end-points for net connections. The goal of placement is to find the mapping of set B to L that minimizes one or more cost objectives.

As previously mentioned, several objectives can be considered when performing placement. The most basic objective is to minimize wirelength of the connections defined in the final routing layout. Tools that seek to minimize the routing cost during placement are referred to as *wirelength-driven*. Some tools, such as the state-of-the-art academic FPGA placement tool VPR [4], may include an additional term in the objective cost function to drive the search towards minimizing the critical-path delay throughout a circuit. Tools that employ such techniques are said to be *timing-driven* [4, 35, 36].

Wirelength is the most common objective used in most placement tools. Computing and maintaining exact wirelength requirements for a placement requires detailed routing information. However, placement tools typically must calculate wirelengths over and over throughout

the placement process. Since performing detailed routing before each wirelength calculation is prohibitively expensive computationally, wirelength estimation models are used practice. Various wirelength models exist, including: *Half-Perimeter WireLength* **(HPWL)** [37]; Steiner Tree; Star; and Star+ [31]. Each of these models provides a means of estimating the routing resources required to route a net between a set of given block positions. While HPWL is the most commonly used wirelength model in placement, it is not differentiable, and thus, is not well-suited for some placement methods. The Star wirelength model uses a quadratic distance measure, which is differentiable. Since HPWL and the Star model can be inaccurate for some nets, other models, such as the minimum Steiner tree model, aim to provide higher accuracy wirelength estimates. However, the minimum Steiner tree problem is NP-hard [38]. The Star+ model [31] provides higher estimation accuracy compared to HPWL while maintaining similar runtime. The Star+ model [31] was developed at the University of Guelph by Dr. Ming Xu, under the supervision of Dr. Gary Grewal. The wirelength models mentioned here are discussed further in Section 2.2.5.

2.2 Placement methods

Over the last few decades, many FPGA placement methods have been proposed in the literature. Most approaches to placement can be grouped into one of four classes: i)*analytic methods* [17–21]; ii)*partitioning-based* [22–24]; iii)*evolutionary algorithms* [25, 26, 39]; or iv)*iterative improvement* [4, 27–31]. In Sections 2.2.1-2.2.4, we provide an overview of these classes of placement methods. While the goal of this section is to provide a general background of approaches to FPGA placement, in Chapter 3 we cover the most relevant previous work to our proposed contributions, related to *parallel* iterative placement methods.

2.2.1 Analytic methods

Analytic methods take a top-down approach to placement, utilizing a global perspective of the connectivity among blocks. The main appeal of these methods is they typically have significantly shorter runtimes than other methods, such as the simulated annealing-based VPlace algorithm [4]. However, the shorter runtimes usually come at the cost of degraded solution

quality. Analytic methods solve a system of equations, producing real value coordinates for block positions. Since logic resources on FPGA architectures have fixed positions legalization must be performed to obtain a feasible solution. In addition to the required mapping of coordinates to valid FPGA grid positions, blocks must be spread out to eliminate any overlapping. In [18], it is reported that in quadratic programming placement methods 85% to 98% of blocks overlap. The most common means of legalizing a placement is to perform some type of recursive partitioning [18, 40–42]. The basic idea is to divide the placement area and shift blocks within the created partitions such that highly connected blocks remain close to one another. Another approach is proposed by H. Eisenmann and F. Johannes [43] uses forces to coerce blocks from dense regions to sparse areas.

Analytic Process Overview

Figure 2.2 shows the general procedure for an analytic method using recursive partitioning as a legalization strategy[4]. Algorithm 1 shows pseudocode for the procedure, where l is the current partitioning level, $|B_l|$ is number of blocks per region at level l, $R(l)$ is the set of partitioned regions at level l, and r' and r'' are the newly created partitions at level l. On line 2, the initial global optimization is performed across the entire chip. Lines 3-10 perform the recursive partitioning, performing global optimization within each partitioned region at the current level. Once the partitions only hold at most one block, legalization is performed on line 11 to produce the final placement by mapping each block to the appropriate FPGA grid position.

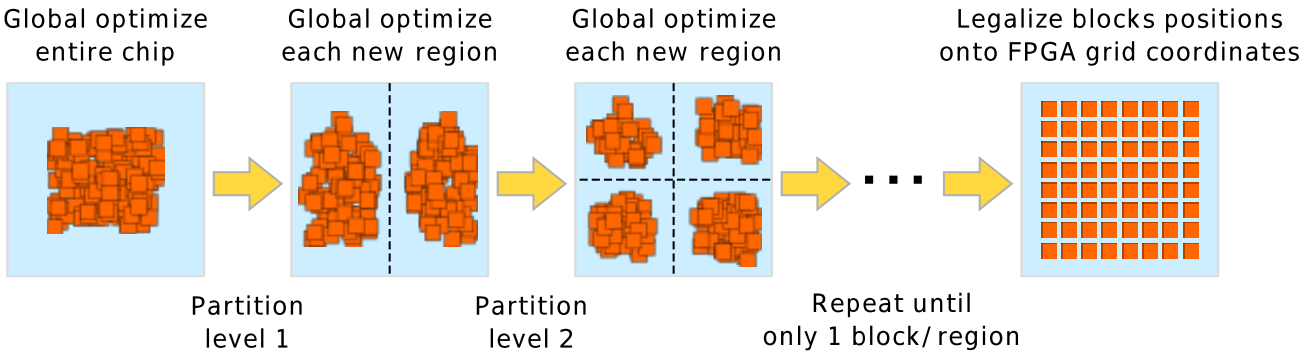


Figure 2.2: Analytic algorithm using recursive partitioning legalization strategy.

[4]This method is similar to the one used by the GORDIAN placement tool [40].

Algorithm 1 Recursive partitioning legalization strategy pseudocode.

```
1: l = 1 {initial partition level}
2: global_optimize(l)
3: while |B_l| > 1 do
4:    for each r ∈ R(l) do
5:       r', r'' = partition(r)
6:       R(l + 1) = R(l + 1) ∪ {r', r''}
7:    end for
8:    l+ = 1
9:    global_optimize(l)
10: end while
11: return legalize(l)
```

Classes of Analytic Methods

Most analytic methods can be grouped in two classes: 1) *force-directed* [17, 18, 43, 44]; and 2) *quadratic programming* [19–21, 35, 40, 42, 45]. Force-directed methods introduce the concept of attractive, repelling, and other forces between blocks. Using these forces, a system of linear equations is extracted from the model and is solved to generate a placement. In [43] and [18], additional forces are introduced to spread cells out from dense areas based on cell distribution. Etawil et al. [44] propose new repulsive forces to discourage overlap among cells connected to the same net, while adding attractive forces to pull cells from dense regions to sparse areas. In [17], B. Hu and M. Marek-Sadowska introduce fixed-points as a means to add forces to encourage the spreading of cells in a controlled manner. While some work has been done regarding force-directed analytic placement for ASIC design, to the best of my knowledge, no work has been done using force-directed analytic placement for FPGAs.

Quadratic programming typically begins with a hyper-graph netlist defining the connections between blocks in the circuit. Given the netlist, the goal is to minimize the total squared wirelength, as defined in Eq. 2.4, where $\Phi(x, y)$ represents a vector of the x and y coordinates of each block, W_{ij} is a weight factor corresponding to the connection between blocks i and j, and x_i/y_i correspond to the coordinates of block i.

$$\Phi(x,y) = \frac{1}{2} \sum_{\forall i,j \in circuit} W_{ij} \left[(x_i - x_j)^2 + (y_i - y_j)^2 \right] \tag{2.4}$$

Equation 2.4 can be written in matrix form as in Eq. 2.5, where A is a symmetrical $n * n$ matrix, called the *Hessian* matrix, and n is the number of blocks in the circuit. Solving Eq. 2.5 as is would result in a trivial solution where all blocks would be placed on top of one another. To prevent this, some block coordinate values must be fixed before solving the equation system. Incorporating fixed values for some x and/or y block coordinates, Eq. 2.5 can be rewritten to the form in Eq. 2.6, where d_x^T and d_y^T are n-dimensional vectors representing fixed x and y values.

$$\Phi(x,y) = \frac{1}{2} \left(x^T A x + y^T A y \right) \tag{2.5}$$

$$\Phi(x,y) = \frac{1}{2} \left(x^T A x + d_x^T x + y^T A y + d_y^T y \right) \tag{2.6}$$

Equation 2.6 can be split into two functions since the set of x variables and the set of y variables are independent of one another. The coordinates in the x dimension are then represented by the function: $\Phi(x,y) = \frac{1}{2} \left(x^T A x + d_x^T x \right)$ (the y dimension is handled similarly). Since this function is strictly convex and definitely positive, its minimum occurs when:

$$0 = Ax + d_x^T x \tag{2.7}$$

Various standard techniques are available to solve this type of linear equation system, including *Conjugate Gradient* **(CG)** [46, 47] and *Successive Over-Relaxation* **(SOR)** [46].

Successive Over-Relaxation (SOR) Placement

Successive Over-Relaxation **(SOR)** [46, 48] is a numerical method for solving linear systems of equations. It is based on the Gauss-Seidel method [46], but SOR adds a relaxation factor to speed convergence. At the University of Guelph, under the supervision of Dr. Gary Grewal, M. Xu developed a new near-linear wirelength estimation model in his thesis [49], called

Star+ [31][5] along with three placement methods. The first two placement methods are pure analytic methods, one using CG and the other using SOR.

As previously mentioned, SOR is based on the Gauss-Seidel method, which is in turn based on the Jacobi method for solving systems of linear equations [46]. The Gauss-Seidel method aims to improve upon the Jacobi method by reducing the memory requirement for computation and by speeding up convergence. SOR modifies the Gauss-Seidel model by introducing a *relaxation factor* to provide a degree of control over the rate of convergence.

On average the CG placement method proposed by M. Xu [49] achieves an improvement of 5x in absolute runtime over VPR, though critical-path delay is increased by 6.6% and wirelength is increased by 12%. The proposed SOR placement method achieves an absolute runtime improvement of 4-40x across the MCNC benchmark circuits compared to of VPR, while improving critical-path delay by 1% and increasing wirelength by 9.1% over VPR. In addition to the two pure analytic placement techniques, a hybrid method is proposed which combines the SOR method followed by a run of the VPR placer. Experimental results show that the hybrid placement method improves critical-path delay by 3.2-6.7% and wirelength by 1-6.5% over SOR alone. However, since a full run of the VPR placement method is run after SOR, there is a 4-40x increase in runtime compared to SOR alone.

Other Analytic Methods

In [19], Xu et al. propose an FPGA placement tool called QPF which uses quadratic programming to minimize the squared distance between connected blocks, followed by a low temperature simulated anneal to refine the final placement. Experimental results show that QPF achieves an average absolute runtime improvement of 5.8x and a 2% increase in wirelength compared to VPR. The analytic placement tool presented in [20] models the FPGA placement problem as a binary quadratic assignment problem. The model used incorporates an abstraction of a target FPGA architecture to provide some awareness of the routing resources during placement. The tool presented employs a *timing-based* goal, which attempts to minimize the critical-path delay for a given design. Preliminary results show that the analytic technique is able to achieve a

[5]The Star+ wirelength model is discussed in detail in Section 2.2.5.

reduction in critical-path delays of up to approximately 10% on some large designs, compared to using VPR's[6] *timing-driven* placement with a defined set of operating parameters.

2.2.2 Partitioning-based

Partitioning-based (or *min-cut*) tools recursively bisect the placement area, placing blocks in one of the partitions while attempting to minimize the number of cut nets across the boundary of the two partitions. The partitioning algorithm attempts to place highly-connected nets within the same partition, while balancing the number of blocks in each partition. The bisecting process is repeated recursively until each partition only contains a few blocks. The main advantage of partitioning-based tools is that they are very fast, due to their divide-and-conquer nature. However, the solution quality produced by min-cut techniques is typically much lower than other leading tools. This is mainly due to the fact that min-cut partitioning does not directly optimize wirelength or critical-path delay.

P. Maidee et al. proposed a partitioning-based placement algorithm called PPFF [50]. PPFF utilizes the state-of-the-art hMetis [51] to perform breadth-first recursive bi-partitioning. A terminal alignment technique is also introduced which attempts to align terminals of each net within the same horizontal or vertical channel. Accurate path delay estimation is achieved by using a lookup table containing routing information from empirical routing results. After the partitioning phase is complete, a low temperature simulated annealing phase is run to refine the final placement. Experimental results [50] show that PPFF achieves a 3-4x improvement in absolute runtime compared to VPR with a mean 1% improvement in delay, when using routing information from previous runs on a selected subset of representative netlists.

Another novel hierarchical partitioning-based approach was proposed in [52, 53], which utilizes a *Greedy Stochastic Algorithm* **(GSA)** with a short-term search history along with an adaptive update schema. Experimental results show a mean absolute runtime improvement of 4.5x and 1.96% reduction in quality of result compared to VPR [32].

[6] *Versatile Place and Route* **(VPR)** is described in Section 2.2.4

2.2.3 Evolutionary algorithms

While *Evolutionary Algorithms* **(EAs)** have been applied to FPGA placement in the literature [25, 26, 39], the proposed approaches have failed to produce results competitive with VPR in terms of runtime and quality, as we discuss in this section. Rubio et al. propose a distributed, multi-island *Genetic Algorithm* **(GA)** for deployment across a grid computing environment [25]. Results for the algorithm were only reported for a single benchmark circuit taken from ITC [54]. For the selected benchmark, the parallel distributed implementation achieved speedups of up to 9.55x running on 10 grid nodes versus running on a single node. The quality of solutions obtained using 10 nodes was comparable to using a single node. Unfortunately, there was no comparison to VPR, either in terms of runtime or solution quality. Therefore, it is difficult to assess the efficacy of the algorithm.

Y. Meng et al. propose another GA for performing FPGA placement [26]. Similar to VPR, the fitness evaluation function is based on HPWL. The encoding of a chromosome is such that each gene position represents a logic cell position on the FPGA grid. The value of a gene is an integer corresponding to the block currently assigned to that logic cell. A modification of single-point crossover is used to ensure that each solution is feasible by avoiding duplicate or missing blocks in offspring solutions. Experimental results on several MCNC [55] benchmarks show that the placement costs obtained by the GA are within 1% to those obtained with VPR. Unfortunately, they do not specifically state what they are using for a placement cost function (e.g., wirelength, critical-path delay?) and no runtime comparison results versus VPR are reported.

P. Jamieson et al. propose several variations of genetic algorithms for placement [39]. However, the reported results show that more work is necessary to make GAs competitive for FPGA placement, since reported solution quality is 21-54% worse than VPR [4] for the same runtime.

2.2.4 Iterative improvement

Iterative improvement tools begin with initial placement and seek improvements within a *neighbourhood* of the current placement. Modifications to the current placement are done by performing swaps with CLBs in a target position, or by moving a CLB to a new, unoccupied location.

Local searches are a subset of iterative improvement algorithms that only allow moves throughout the search space that result in an improvement to the objective function. These searches, if implemented well, have the potential to run quite quickly. However, since possible moves are limited to those which are improving and are within a neighbourhood of the current solution, local searches are subject to getting caught in local optima. In placement, the number of local optima is high, so local search techniques usually produce inferior solutions compared to other techniques with a more global perspective.

Simulated annealing is a *stochastic* search algorithm which accepts any solution under consideration subject to a defined probability. The search attempts to emulate the process of annealing metal in which the size of crystalline structures in the material is increased by heating and gradually cooling it. The initial heating of the material frees an atom from its initial position to one with a lower internal energy. The gradual cooling tries to provide many opportunities for the atom to find configurations that result in lower internal energy. In the context of FPGA placement, simulated annealing mimics this process by allowing blocks to move to new locations "nearby" their current location subject to a difference in the cost function and a global 'temperature' parameter, T, that is gradually decreased. Specifically, while any improving move encountered is accepted, a non-improving move will only be accepted with probability $e^{\delta C/T}$, where δC is the change in cost. During the initial portion of the search, many non-improving solutions should be accepted to promote exploration of the search space. However, as the search progresses, the number of accepted non-improving moves should be decreased to allow blocks to settle in near their final (good) positions. This process is analogous to the slow cooling of molten metal, such that proper crystal formation may occur. By allowing non-improving moves, simulated annealing promotes hill-climbing to escape local optima, which present an issue for many other techniques. VPR [4] is a simulated annealing implementation tailored specifically to FPGA placement.

Versatile Place and Route (VPR)

Vaughn Betz et al. present *Versatile Place and Route* (VPR) [4] as a flexible CAD tool for FPGA architectural studies based on the simulated annealing template described in Section 2.2.4. For the VPR implementation, given an initial placement, blocks are considered for movement

within a distance of D_{limit} surrounding its current position. Throughout the progression of the algorithm D_{limit} reduces from the width/height of the placement grid down to 1. This ensures that as the modules move closer and closer to their final positions, less disruptive position changes are considered. Once a target position is chosen for evaluation, the resulting cost function value is evaluated to determine if the move should be made. Even if the change in cost function is not desirable, the move may still be taken with a probability proportional to the current value of T, $e^{\delta C/T}$. The value of T is reduced throughout the process to allow fewer and fewer non-improving moves as the search progresses closer and closer to the final solution. Algorithm 2 shows the pseudocode for the VPR placement heuristic.

Algorithm 2 VPR pseudocode

```
1: generate_random_placement()
2: get_initial_values(T, D_limit)
3: while T > 0 do
4:    for i in 1 to moves_per_temperature do
5:       move = do_move()
6:       new_cost = calculate_cost()
7:       if accepted(move) then
8:          cost = new_cost
9:       else
10:         undo(move)
11:      end if
12:   end for
13:   get_updated_values(T, D_limit)
14: end while
```

VPR is a state-of-the-art tool which produces very good results for both *wirelength-driven* and *timing-driven* placement. In fact, it remains the *de facto* academic placement tool against which most new algorithms or heuristics are compared. While VPR delivers high solution quality, runtime scales exponentially with the size of the netlist to be placed, where the placement of complex circuits can take up to tens of hours. In Chapter 3, we discuss some of the most successful parallel variations of simulated annealing for FPGA placement, which attempt to

reduce placement runtime while limiting loss of solution quality.

2.2.5 Wirelength estimation

In the context of FPGA placement, in order to maintain exact knowledge of the final routing length, intricate details of the routing fabric would need to be taken into account during the placement search. This would significantly increase the complexity of the placement problem and would lead to much larger runtimes. To alleviate this, wirelength estimates are typically used to compute approximate routing costs for candidate placements. Several wirelength estimation models are discussed in this section.

Minimum Steiner Tree

A net may be modelled as a minimum Steiner tree where each input or output pin of a CLB or I/O pad represents a terminal in the graph. Minimizing the wirelength of a net is then equivalent to finding the minimum Steiner tree connecting the terminals belonging to a net. While Fig. 2.3a shows a Steiner for an example four terminal net, the Steiner tree in Fig. 2.3b is a minimum Steiner for the corresponding net, and thus, shows the optimal way to route the net. While the Steiner tree model may provide an accurate estimation, the minimum Steiner problem is NP-hard [38], limiting the model's practical use in placement algorithms.

(a) Steiner tree (length: 9)

(b) Minimum Steiner tree (length: 7)

Figure 2.3: Steiner tree example for a 4-block net

Half-Perimeter Wirelength (HPWL)

The *Half-Perimeter Wirelength* **(HPWL)** model is commonly used to estimate the wirelength of a net [37]. This method approximates the length of a net by half the perimeter of the smallest bounding rectangle that contains all terminals connected to the net. Figure 2.4 shows a graphical example of a HPWL for the net shown. A bounding rectangle has been placed around the blocks connected to the net. The HPWL is labelled on the figure as the combined length of the height and width of the bounding rectangle (i.e., half the box perimeter). Equation 2.8 shows how the HPWL of net i is calculated, given a block b, with coordinates (x_b, y_b).

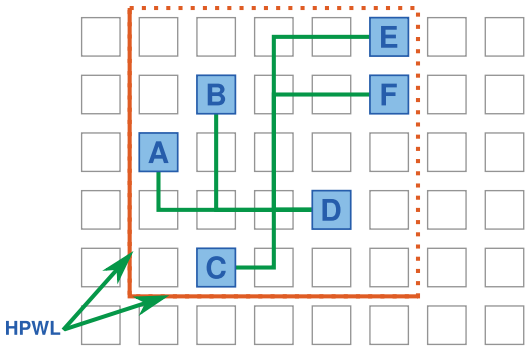


Figure 2.4: Half-perimeter wirelength (HPWL) example.

$$HPWL_i = (MAX_{b \in i}\{x_b\} - MIN_{b \in i}\{x_b\} + 1) + (MAX_{b \in i}\{y_b\} - MIN_{b \in i}\{y_b\} + 1) \quad (2.8)$$

The HPWL provides a reasonable estimate of the length of a net, but as more blocks are connected by a net, the accuracy of the estimate decreases. To compensate for the under-estimation of the HPWL for nets connected to more than three blocks, a $q(i)$ scaling factor is introduced which varies with the cardinality of each net [56]. For nets connecting up to 3 blocks, q is 1, while the value of q increases slowly to a value of 2.79 for nets connected to 50 blocks.

Star

Although tools such as VPR produce high-quality results using HPWL, the HPWL model has some disadvantages. For instance, HPWL is non-differentiable and thus, is not suited for analytic

placement methods. Instead, researchers typically use *quadratic distance* to estimate wirelength for analytic methods. The Star model is an example of a wirelength model that uses a quadratic distance measure.

Figure 2.5 shows a graphical representation of the Star model. The x and y dimensions are handled equivalently, though separately, so for the sake of simplicity, only the x dimension will be discussed here. An important concept of the Star model is that of a *center of gravity*. The *center of gravity*, or c_l, refers to a virtual point that is connected to each block belonging to net l. Roughly speaking, each connection produces a force on c_l, such that c_l becomes a target for the nets of a block to be pulled towards. Formally, the x coordinate of c_l is defined as in Eq. 2.9, where k_l is the cardinality of Net_l (i.e., the number of blocks connected to Net_l), x_i is the x coordinate of block i, and $\forall i \in Net_l$ refers to all blocks connected to Net_l indexed by i.

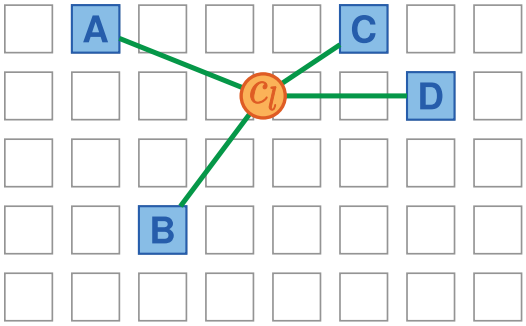


Figure 2.5: Example net representation for Star and Star+.

$$x_{cl} = \frac{1}{k_l} \sum_{\forall i \in Net_l} x_i \tag{2.9}$$

Given x_{cl} and the set of x_i values, the Star estimated wirelength (in the x-dimension) of Net_l is defined as in Eq. 2.10.

$$\|Net_{lx}\| = \sum_{\forall i \in Net_l} (x_i - x_{cl})^2 \tag{2.10}$$

The y-dimension is calculated similarly.

Star+

As previously mentioned, HPWL under-estimates wirelength for nets connected to more than three blocks, requiring a scaling factor function to compensate. Moreover, the Star wirelength model uses the quadratic distance model, which places a much larger emphasis on blocks that are far away. In response to these issues, M. Xu et al. propose a *near-linear* wirelength estimation model called Star+ [31], which tries to solve the deficiencies of the HPWL model and the Star model. Experimental results [31] show that the Star+ model performs comparably to HPWL in with respect to runtime and quality of results when used in the VPR placement tool.

Star+ is represented graphically the same way as the Star model, as shown in Fig. 2.5. As in the Star model, Star+ uses the concept of a *center of gravity*, calculated as in Eq. 2.9. Star+ differs from the Star model in the distance calculation used. Given x_{cl} and the set of x_i values, the Star+ estimated wirelength (in the x-dimension) of Net_l is defined as in Eq. 2.11. In Eq. 2.11, two parameters are present: α and β. The α factor is analogous to the $q(i)$ scaling factor in the HPWL model, in that it compensates for wirelength underestimation. By including a positive-valued β term, $\|Net_l\|$ is guaranteed to be differentiable. The y-dimension is calculated similarly.

$$\|Net_{lx}\| = \alpha \sqrt{\sum_{\forall i \in Net_l} (x_i - x_{cl})^2 + \beta} \tag{2.11}$$

The fact that $\|Net_l\|$ is differentiable allows Star+ to be used in analytic methods. However, the authors also demonstrate that Star+ is well-suited for iterative approaches as well since it can be updated in constant time (i.e., with a runtime bound of $O(1)$) [31].

2.2.6 Timing analysis

While minimizing wirelength aims to reduce routing resources and may help to group highly-connected blocks closer together, it does not guarantee maximum performance of the circuit. In order to improve circuit performance, timing must be taken into account. Timing analysis techniques may be grouped into two classes: (1) *path-based*; and (2) *connection-based*. Path-based timing analysis computes the delay of each path in the circuit to determine which delay

is the longest. This path is taken to be the most critical and is considered for optimization during the next iteration of placement. The computation of all delays is completed again on the updated placement. While path-based approaches provide accurate timing estimates, computing delays for all paths through a circuit is extremely expensive computationally. In practice, connection-based approaches are typically used since they still provide high-quality results, while exhibiting much lower runtime complexities. In this section, we discuss the connection-based timing analysis method used in T-VPlace [4, 36][7].

Connection-based timing analysis begins with modelling the circuit as a directed graph. The nodes of the graph correspond to the input and output pins of the I/O pads, combinational logic (i.e., look-up tables (LUTs)) and registers of the circuit. Connections between the pins are modelled as edges in the graph, each of which is annotated with a weight corresponding to the physical delay between the nodes connected by the edge. Figure 2.6b shows an example circuit along with the corresponding timing graph, with the following elements labelled: *synchronous driver* nodes (i.e., input pins and register outputs); *synchronous sink* nodes (i.e., output pin or register input); and intermediate nodes.

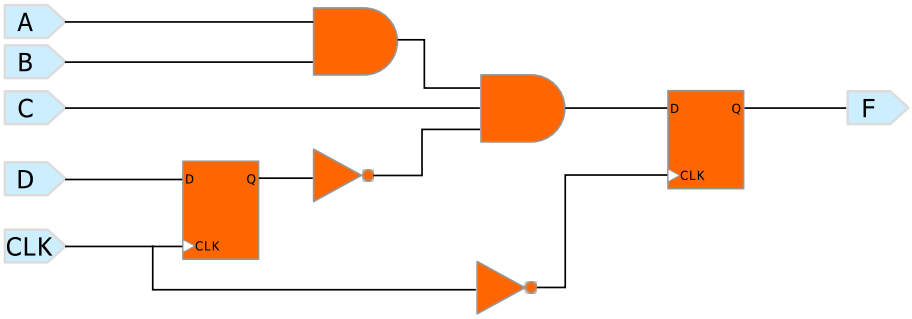
Based on the graph model of the circuit, the timing of a circuit is characterized by the longest path between any two synchronous components. Therefore, in general, timing analysis begins by computing the longest delay path through a circuit. The longest path is referred to as the "critical path", since the delay of this path determines the maximum operating frequency of the circuit. To calculate the critical path delay of the circuit, a breadth-first traversal is performed starting at the synchronous driver nodes until a synchronous sink node is reached on each path.

Based on the graph model of the circuit, the length of the longest incoming delay for each node i, or ℓ_{Ii} is introduced as defined as in Eq. 2.12, where i is the node being computed and $delay(j, i)$ is the delay value of the edge connecting node j to node i.

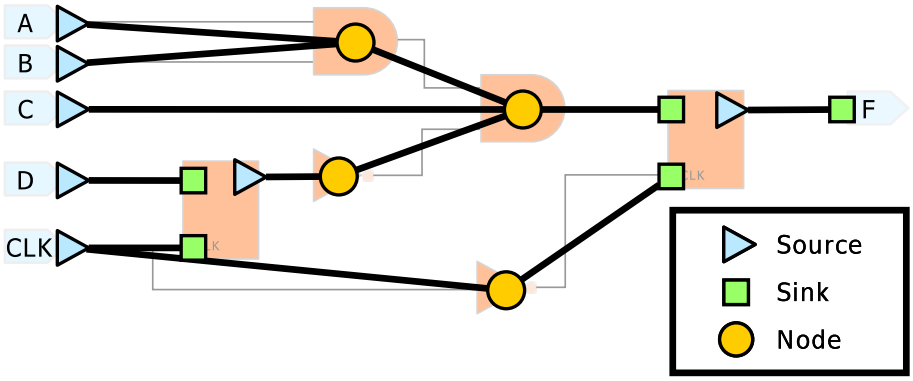
$$\ell_{Ii} = \max_{\forall j \in fanin(i)} \{\ell_{Ij} + delay(j, i)\} \tag{2.12}$$

The delay of the circuit is then computed as the maximum longest incoming delay of all nodes in the graph, as defined in Eq. 2.13.

[7]T-VPlace is the updated timing-driven implementation of the original VPlace tool [32].



(a) Circuit



(b) Timing analysis graph

Figure 2.6: Timing analysis example

$$D_{max} = \max_{\forall j \in sinks} \{\ell_{Ij}\} \tag{2.13}$$

Since the delay along a connected path depends on the position of the nodes along the path, as nodes are moved, the delay along any paths passing through one or move affected blocks will change. To help make informed choices during placement, in addition to computing the critical path delay of the circuit, the longest path passing through each block is typically recorded as well. Specifically, for each block, the following two values are recorded:

1. The length of the longest path from a synchronous driver to an input pin of the block.

2. The length of the longest path from a synchronous sink to an output pin of the block.

Once D_{max} has been calculated, for flexibility during placement, it is useful to determine the ratio of the length of the longest path passing through a connection to the critical-path length,

D_{max}. We refer to this ratio for a connection as the "criticality ratio", since it provides a relative measure of how critical the connection is.

In order to compute the criticality ratio for the edges in the graph, the *longest outgoing path*, or ℓ_{Oi}, of each node must be calculated according to Eq. 2.14, where i is the node currently being computed and $delay(i, j)$ is the delay value of the edge connecting node i to node j. The computation is performed through a breadth-first traversal which begins at the sink nodes and propagates backwards until a driver node is reached on each path.

$$\ell_{Oi} = Min_{\forall j \in fanout(i)} \{\ell_{Oj} + delay(i, j)\} \tag{2.14}$$

Once the longest outgoing path has been computed for each node, the criticality ratio of a connection from node i driving node j can be calculated using Eq. 2.15.

$$r_{crit}(i, j) = \frac{\ell_{Ii} + delay(i, j) + \ell_{Oj}}{D_{max}} \tag{2.15}$$

Using the criticality ratio function defined in Eq. 2.15[8], T-VPlace [36] defines a function for calculating the relative *criticality cost* of the connection between nodes i and j as in Eq. 2.16, where *criticality_exp* is a user-defined parameter used to allow critical connections to be heavily weighted. The timing cost of the entire circuit can then be computed by accumulating the timing costs of all connections in the circuit, as in Eq. 2.17.

$$timing\_cost(i, j) = delay(i, j) \cdot r_{crit}(i, j)^{criticality\_exp} \tag{2.16}$$

$$timing\_cost_{circuit} = \sum_{\forall i,j \in circuit} timing\_cost(i, j) \tag{2.17}$$

In Chapter 6, we present an efficient parallel method for computing the timing costs presented here, where the opportunities for parallelism scale with the size of the netlist.

[8] Technically, T-VPlace uses a "slack" function which refers to the amount of delay that can be added to a connection before it becomes the critical-path, but we feel that it is more natural to describe delays in more general terms as longest paths. The two representations produce equivalent costs.

2.2.7 Congestion

In addition to reducing estimated wirelength and critical-path delay, research has been done on reducing congestion during placement. The basic idea is to guide the placement of blocks such that routing resource usage is spread evenly across the chip. Doing so can especially help to promote the routability of a placement. K. Tsota et al. propose an analytic ASIC placement tool which incorporates a routing congestion model to identify and help alleviate congestion in affected regions of the chip [57]. The model used for congestion estimation is described in this section.

The congestion estimation model used in [57] divides the chip area up into a grid array of "bins," as shown in Fig. 2.7. A congestion estimate is calculated for each *bin* within the grid, based on the relative estimated wirelength usage compared to the other bins.

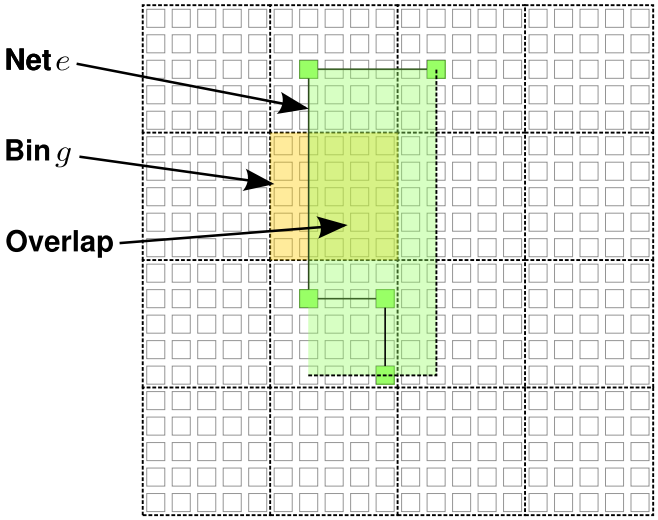


Figure 2.7: Wirelength congestion grid

Formally, for computing the congestion for net e which overlaps with bin g, the estimated congestion cost, $d_g(e)$ is calculated as in Eq. 2.18, where $A_g(e)$ is the area of bin g overlapped by the bounding box of net e and $d_n(e)$ is the estimated wire density of net e. The wire density is calculated by $d_n(e) = \dfrac{wl(e)}{A_{bb}(e)}$, where $wl(e)$ is the estimated wirelength for net e and $A_{bb}(e)$ is the bounding box area of net e. Based on this, Eq. 2.19 is used to calculate the total estimated congestion level within bin g, where $\forall e \in g$ refers to all nets with a bounding box area that overlaps bin g.

$$d_g(e) = d_n(e) \cdot A_g(e) \tag{2.18}$$

$$Sd(g) = \sum_{\forall e \in g} d_g(e) \tag{2.19}$$

The congestion cost of the entire placement is then calculated as in Eq. 2.20, where SD is the average estimated wirelength per bin. By minimizing *congestion_cost*, the search will promote placements which spread estimated routing evenly across the chip.

$$congestion\_cost = \sum_{\forall g \in circuit} (Sd(g) - SD)^2 \tag{2.20}$$

While we do not employ the congestion model in this section in our proposed placement methodology in Chapters 4-6, we discuss the possibility of adding additional objectives, such as congestion, as future work, in Chapter 7.

2.3 Parallel programming

The majority of modern commodity processors are parallel. Over approximately the last ten years, there has been a trend towards improving computational throughput through the use of more and more parallel computing cores. In this section, we discuss the general implications of the paradigm shift to pervasive parallel computing. Specifically, we present evidence that single-threaded performance is no longer improving at a significant rate, but that overall parallel throughput continues to grow steadily. Furthermore, we discuss how, as the number of hardware threads continues to grow well beyond the 2-4 cores of early multicore processors, threading-based programming quickly becomes unmanageable as developers strive to write parallel code that is maintainable and portable to various platforms, including future architectures. We describe recent advances in structured parallel programming resulting in parallel "algorithmic skeletons" [58, 59], allowing developers to describe efficient parallel algorithms without explicitly mapping to a particular architecture.

2.3.1 Pervasive parallel architectures

Up until approximately 2005, the single-threaded performance of commodity CPUs was rapidly increasing [60, 61]. However, shortly thereafter, improvement in single-threaded performance came to an abrupt halt. Several factors contributed to this turning point, including the convergence of three practical limits [1]:

1. *Instruction-Level Parallelism (ILP)* [62].
2. Memory latency.
3. Power.

Instruction-level parallelism refers to parallel work that is extracted on-the-fly at the architecture-level through out-of-order execution. This type of parallelism allows instructions in serial codes to execute in parallel under certain conditions. While ILP has provided significant performance improvements overall, it is widely accepted that this type of parallelism has reached its practical limit [1].

Along with instruction-level parallelism, memory latency was also lagging behind historical trends. Memory latency refers to the time between a memory operation request and the completion of the request. Thankfully, memory bandwidth has fared better overall. Moreover, by explicitly exposing high levels of parallel work, much greater than the number of available cores or threads, it is possible to actually *hide* memory latency. The latency can be hidden by quickly switching to another waiting task while waiting for a memory request to be serviced, allowing the processor to keep doing useful work while waiting for the memory operation to complete.

While reduced levels of ILP and high memory latency contributed to capping single-threaded processor throughput, the biggest problem facing historical trends was what was referred to as the "power wall" [1]. Even though advances in fabrication reduced the power needed to drive each individual transistor, around 2005, the number of transistors packed into a CPU made it impossible to continue to drive up the processor frequency due to an inability to prevent the processor from overheating.

The industry has responded to these barriers by increasing computational throughput, not through single-threaded performance, but rather through adding more and more processing

cores to each processor chip. Below, we present evidence supporting this, both in mainstream *multicore* processors, but also in commodity *manycore* architectures, in the form of *Graphics Processing Units (GPUs)*.

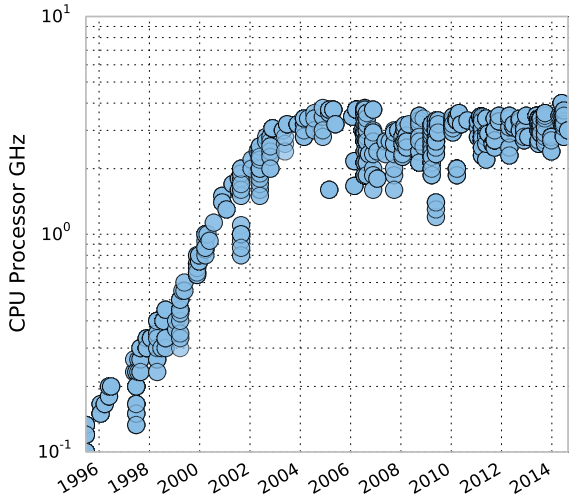
Multicore processors

Figure 2.8a plots the operating frequency of processors released by Intel between 1995-2014. The data in Fig. 2.8 is derived from the SPEC1995, SPEC2000, and SPEC2006 benchmark result sets [60, 61]. As shown in Fig. 2.8a, the operating frequency of processors manufactured by Intel increased exponentially over the last few decades of the twentieth century. However, the increase of processor frequency ceases in approximately 2005.

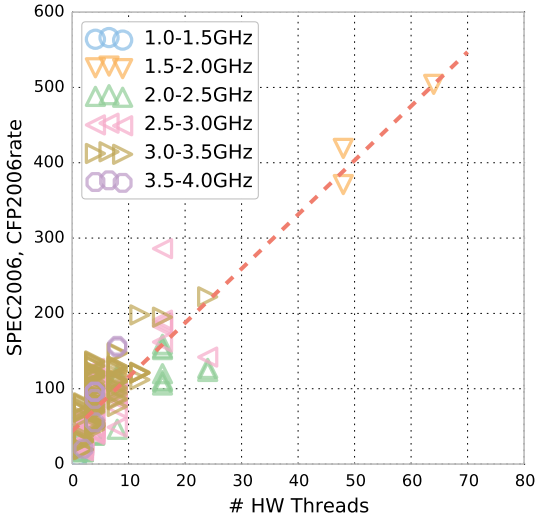
While processor frequencies have stopped growing, peak computational throughput has continued to grow significantly through the addition of more hardware threads. Note that we refer to hardware threads rather than cores here, since most modern Intel cores may actually execute two or more threads simultaneously. Figure 2.8b plots the SPEC2006 `CFP2000rate` result for various Intel processors released between 2006-2014. The SPEC2006 `CFP2000rate` benchmark aims to stress the floating-point throughput of an architecture, by attempting to saturate all available hardware threads with as many floating-point operations as possible. Note that there is a positive correlation between the number of hardware threads and the peak floating-point rate. While there is some variation from the general trend due to clock speed and small improvements to ILP, the most significant influence is the hardware thread count. Figure 2.8c supports this conclusion as well. Note that the floating-point rate for processors with the same number of hardware threads increases with clock rate, but again, the most significant correlation is between floating-point rate and hardware thread count. Although we only show Intel processors here, the trends are also representative of other mainstream CPUs.

Manycore processors

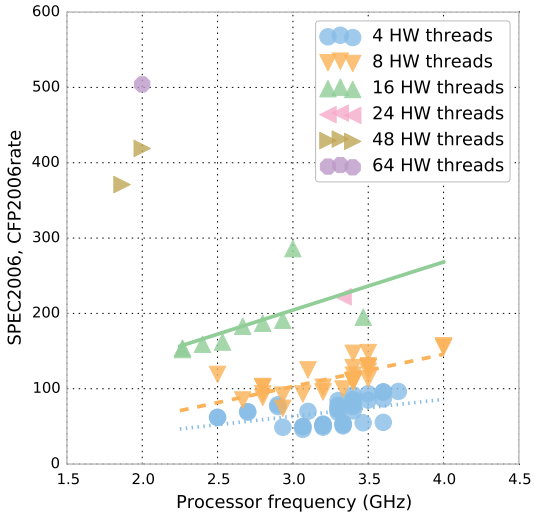
In addition to multicore architectures, over the last decade, manufacturers have introduced general-purpose *manycore* architectures, with hundreds or thousands of simple processing cores, designed for highly data-parallel workloads. Prime examples of these architectures are NVIDIA's



(a) Intel processor frequency (GHz) for processors released between 1995-2014



(b) SPEC 2006 `CFPrate` floating-point results for processors (released 2006-2014), with varying numbers of hardware threads.



(c) SPEC 2006 `CFPrate` floating-point results for processors (released 2006-2014), with varying frequencies.

Figure 2.8: Peak performance trends for Intel multicore CPUs. Derived from the SPEC 95, SPEC 2000, and SPEC 2006 benchmark result sets [60, 61].

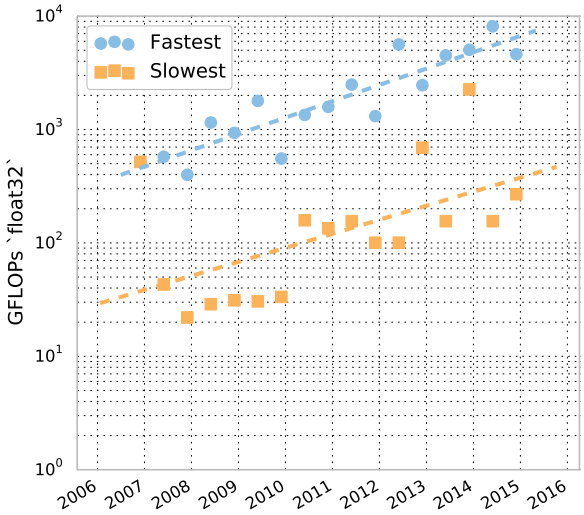


Figure 2.9: Peak floating-point throughput for the fastest and slowest NVIDIA GPU released during each 6-month interval between 2007-2014.

CUDA-capable *Graphics Processing Units (GPUs)*. NVIDIA's CUDA platform enables application developers to compile and run custom code on NVIDIA's manycore GPUs. Figure 2.9 shows the peak floating-point performance of the fastest and slowest NVIDIA GPU released during each six month time interval between 2007-2014 [63]. Note that the peak floating-point throughput has been rising exponentially across NVIDIA's entire product line, with both top-end and bottom-end performance increasing at approximately the same rate.

Based on the trends of peak *parallel* throughput of recent commodity hardware releases, it seems clear that throughput is expected to increase steadily for the foreseeable future. Unfortunately, the majority of this processing power is only accessible through codes that explicitly express parallelism. While much work has been focused on automatic parallelization of serial codes, such techniques have had limited success [1]. We suspect that a major issue with automatic parallelization is that, while certain critical sections of code may be favorable for detection, unless all portions of the code that scale with the size of the problem are auto-parallelized, runtime performance will not scale. In Section 2.3.2, we discuss parallel performance theory from the literature that supports this line of reasoning and provides a basis to predict the runtime behaviour of parallel algorithms.

2.3.2 Parallel performance theory

While parallel computing has been an active area of research for over thirty years [64], the findings were of little relevance to mainstream computing until approximately ten years ago. As we showed in the previous section, single-core processor speeds were growing exponentially until 2005. With this steady performance increase of new architectures, it was often much easier and cheaper to wait for a next generation single processor than to develop parallel algorithms for expensive, industrial large scale parallel computing systems [65]. However, the introduction of ubiquitous, commodity parallel architectures within the last decade has presented renewed relevance of past findings to mainstream computing.

We begin this section by presenting relevant terminology from the peer-reviewed literature [2, 64, 66] [9]. Following the definitions of key terms, we discuss analytic asymptotic analysis techniques from the literature [2, 66, 67] based on these terms which predict how a parallel algorithm will scale relative to the problem size and the number of parallel workers. It is important to note that although the models presented here (from the literature [2, 66, 67]) may not be appropriate in all cases (e.g., parallel algorithms using locks), they are applicable to algorithms composed from the patterns we describe in Section 2.3.4. Since the algorithms we propose in Chapters 4-6 are composed entirely from these patterns, we use the analytic performance models presented in this section to analyze the parallel runtime complexity of our proposed placement method. In Chapters 5-6, we compare the absolute runtime performance of implementations of our proposed placement algorithms against the runtime performance of VPR [4].

Parallel system

As defined in [2] a parallel system is the combination of a parallel algorithm and a parallel architecture.

Worker

While it is common to describe parallel algorithms as targeting a number of *processors*, most modern processors contain multiple processing cores, where some cores are also capable of executing multiple threads (e.g., hyper-threading [1]). To avoid confusion,

[9] Table 2.1 summarizes the symbols associated with key terms presented in this section.

Term	Symbol
Problem size	W
Basic operation execution time	t_c
Sequential execution time	T_1
Parallel execution time	T_P
Total overhead	T_O
Speedup	S_P
Parallel efficiency	E_P

Table 2.1: Parallel terms from peer-reviewed literature.

we instead adopt the term "worker" [65] to describe a single processing unit in a parallel architecture.

Problem size (W)

We adopt the definition of "problem size" from [2], where Grama et al. define the problem size W as the total number of basic operations (e.g., memory access, addition, multiplication, etc.) in the problem. For example, W is $\Theta(n^3)$ for the problem of $n \times n$ matrix multiplication while for $n \times n$ matrix addition W is $\Theta(n^2)$ (where $\Theta(x)$ is the set of all functions with the same growth rate as x [3]).

Basic operation execution time (t_c)

We adopt the term t_c from the work of Grama et al. [2] to denote the average time taken to execute each basic operation in a parallel system. Note that as defined by Grama et al. [2], t_c is assumed to be independent of the number of workers p. This assumption may not hold, for example, in systems utilizing locks where wait times may grow exponentially with the number of workers. Nevertheless, t_c does *not* increase with p for architectures such as NVIDIA CUDA GPUs when employing the parallel patterns we describe in Section 2.3.4. These patterns break work into steps of tasks, where tasks within each step are completely independent of one another and no locks are required. Furthermore, the independent tasks during each step can be scheduled to execute in any order, affording flexibility in implementations to, for example, arrange memory accesses to complement favorable access

patterns for the target architecture.

Sequential execution time (T_1)

Grama et al. [2] define the sequential execution time (T_1) as the time taken by an algorithm to execute on a single worker. Given W and t_c, the sequential execution time is formally defined as [2]:

$$T_1 = t_c W \tag{2.21}$$

Parallel execution time (T_P)

Given a sequential algorithm, Grama et al. [2] define the parallel execution time T_P [2] as the time taken to execute by a corresponding parallel algorithm on an architecture with p workers.

Total overhead (T_O)

In addition to T_1 and T_P, Grama et al. [2] define T_O as the time overhead incurred due to parallelization, including any additional communication, memory accesses, arithmetic operations, etc. Note that T_O is typically *not* constant, but rather is a function that varies according to the parallel system. For the patterns we describe in Section 2.3.4, T_O is a function of the number of parallel workers, p, but does *not* grow with the problem size [2, 64]. Moreover, in the case of the implementations we discuss in Chapters 5-6 and Appendix D, problem data is stored entirely in GPU memory throughout the placement process. While time is required to transfer the initial problem state to GPU memory prior to placement and to transfer the final solution from GPU memory after placement, we do not consider these times when analyzing the scalability of our proposed algorithms for two key reasons. First, for the experiments discussed in Chapters 5-6, the transfer times between CPU and GPU memory are insignificant compared to even one out of hundreds of iterations of the main program loop. Second, NVIDIA has already announced that their next GPU generation, named "Pascal" (to be released in 2016), introduces a new technology called "NVLINK" [68] which allows GPUs to access main memory (i.e., host CPU memory) as fast as the CPU, completely eliminating any overhead associated with GPU to CPU transfers.

Given T_O and T_1, the parallel execution time is formally defined such that [2]:

$$pT_P = T_1 + T_O \tag{2.22}$$

$$T_P = \frac{1}{p}(T_1 + T_O) \tag{2.23}$$

Speedup (S_P)

Grama et al. [2] define the speedup of a parallel system S_P as the ratio of the sequential execution time to the parallel execution time, i.e.:

$$S_P = \frac{T_1}{T_P} = \frac{pT_1}{T_1 + T_O} \tag{2.24}$$

Efficiency (E_P)

Based on the speedup, Grama et al. [2] define the efficiency of a parallel system as the ratio of the speedup to the number of parallel workers:

$$E_P = \frac{S_P}{p} = \frac{T_1}{pT_P} \tag{2.25}$$

Efficiency represents the fraction of time spent by parallel workers doing operations that are not considered overhead.

Amdahl's Law

Amdahl's Law [69] assumes that the number of parallel workers p is increased while the problem size W is held constant. In other words, T_1 is fixed and T_P is only a function of p. Amdahl differentiates between two distinct portions of a sequential algorithm:

- The non-parallelizable portion of the program.
- The parallelizable portion of the program.

According to Amdahl's law, for a problem size W:

$$W = W_{SER} + W_{PAR} \tag{2.26}$$

where W_{SER} is the *size* of the non-parallelizable portion of the problem and W_{PAR} is the *size* of the parallelizable portion of the problem. Given the definition of W_{SER} and W_{PAR}, T_1 and T_P can be defined as follows:

$$T_1 = t_c \left(W_{\text{SER}} + W_{\text{PAR}}\right) \tag{2.27}$$

$$T_P \geq t_c \left(W_{\text{SER}} + \frac{W_{\text{PAR}}}{P}\right) \tag{2.28}$$

Note that Amdahl's law only provides a *lower bound* on the parallel execution time, representing the best case scenario where $T_O = 0$.

If we define f as the fraction of the sequential algorithm that is non-parallelizable, then we can define W_{SER} and W_{PAR} as follows:

$$\begin{aligned} W_{\text{SER}} &= \frac{1}{t_c} f T_1 \\ W_{\text{PAR}} &= \frac{1}{t_c}(1-f) T_1 \end{aligned} \tag{2.29}$$

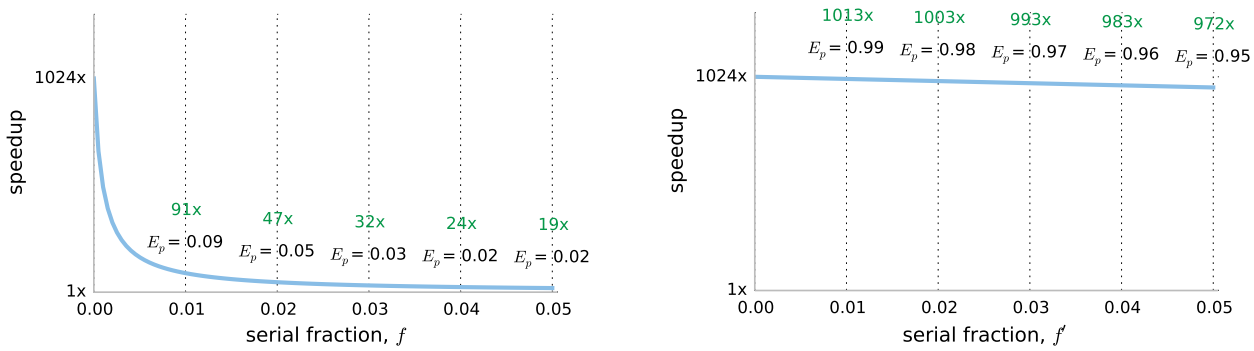
Substituting the definitions in Eq. 2.29 into Eq. 2.28, we end up with the following speedup formulation:

$$\begin{aligned} S_P &\leq \frac{T_1}{f T_1 + \frac{(1-f)T_1}{p}} = \frac{T_1}{T_1\left(f + \frac{(1-f)}{p}\right)} \\ S_P &\leq \frac{1}{f + \frac{(1-f)}{p}} \end{aligned} \tag{2.30}$$

Consider Eq. 2.30 in the limit where the number of parallel workers approaches infinity:

$$S_\infty \leq \lim_{p \to \infty} \frac{1}{f + \frac{(1-f)}{p}} = \frac{1}{f} \tag{2.31}$$

The speedup formulation in Eq. 2.31 captures the essence of Amdahl's law. For a fixed problem size, the speedup of a parallel program is limited by the fraction of the sequential algorithm that is non-parallelizable. For example, if the serial fraction is 10% the maximum speedup is 10. Recall that parallel efficiency, E_P, is defined as $\frac{S_P}{p}$ [2]. According to Amdahl's law [69], provided the problem size remains fixed, parallel efficiency is highest with a single worker but decreases towards zero as the number of parallel workers p increases.



(a) Amdahl's paradigm [69] (fixed W)

(b) Gustafson-Barsis' paradigm [70] (scaled W)

Figure 2.10: Speedup on 1024 workers for varying serial fraction

Gustafson-Barsis' Law

Amdahl's law is based on the implicit assumption that the amount of parallel work is independent of the number of workers. However, Gustafson et al. [70] suggest that in scientific computing, one generally does not take a fixed size problem and run it on a varying number of workers, but rather that the size of the computing problem scales with the available number of workers. Following this logic, rather than ask how fast a sequential program would take to run on a parallel system, Gustafson et al. instead ask how long a parallel program would take to run on a single processor. Formally, where f' denotes the fraction of the *parallel execution time* spent doing serial work on p workers, Gustafson-Barsis' law [70] defines speedup (i.e., the execution time on a single processor normalized to parallel execution time on p workers) as:

$$S_P = f' + (1 - f')p \tag{2.32}$$

Figure 2.10 illustrates how speedup varies for a parallel system with 1024 workers (i.e., $p = 1024$) according to a) Amdahl's law, and b) Gustafson-Barsis' law. As shown in Fig. 2.10a, according to Amdahl's law, where the problem size is assumed to be fixed, speedup is a steep function of f near $f = 0$. In contrast, as illustrated in Fig. 2.10b, where Gustafson-Barsis' law implies the problem size scales relative to the number of parallel workers, the speedup function is simply a line. Gustafson et. al present results from several parallel programs used at Sandia National Laboratories [70] where, for the large problem sizes encountered *in practice*, reported

speedups of over 1015× on 1024 parallel workers are achieved, leading to parallel efficiency over 99%.

Gustafson-Barsis' law leads to the key observation that efficient parallel performance can be achieved for a very large number of parallel workers provided the serial fraction of the parallel execution time remains low. This observation motivates *scalability* analysis, which we discuss in the following section. Scalability analysis is used to determine how the efficiency of a given parallel algorithm is expected to scale as both the problem size *and* the number of parallel workers are increased, as is the case in the Gustafson-Barsis paradigm.

Scalability

As shown in the previous section, Gustafson-Barsis' law implies that if the problem size if permitted to scale with p, efficient parallel performance is achievable for large numbers of parallel workers provided the serial fraction of the parallel execution time remains low. Thus, rather than focusing directly on parallel runtime for a *fixed* problem size, it is typically more useful to analyze the *scalability* of an algorithm, i.e., the ability of a parallel system[10] to increase speedup as the size of the problem increases along with the number of parallel workers [2].

In this thesis, we consider a parallel algorithm to be *scalable* if it is *completely parallelizable*, as defined [64] by Gottlieb et al. Where the *problem size* W is defined as the number of basic operations (e.g., memory access, addition, etc.) [2], Gottlieb et al. [64] define an algorithm as completely parallelizable if there exists a relationship between W and the number of parallel workers p, such that speedup is expected to be unbounded at a constant efficiency. Such an algorithm ensures that speedup continues to grow as the number of parallel workers is increased provided there is sufficient work (i.e., the problem size W is large enough).

For example, a speedup of 90× for a parallel system with 100 parallel workers results in a parallel efficiency of 90%. For any completely parallelizable algorithm, constant parallel efficiency can be maintained as the number of workers increases provided the size of the problem increases accordingly. This does not mean that the speedup remains constant, but rather that the speedup is expected to grow linearly along with p such that constant parallel efficiency is

[10]Recall that a parallel system is the combination of a parallel algorithm and a parallel architecture.

maintained. For instance, assuming $E_P = 0.9$, the example algorithm considered above would result in a speedup of 900× with 1000 parallel workers.

The speedup of *completely parallelizable* algorithms approaches linear if the size of the problem is permitted to increase for a fixed number of parallel workers. Completely parallelizable algorithms are said to exhibit *weak scaling* [1], since linear speedup is maintained as p increases, so long as the problem size W is increased accordingly (not necessarily at the same rate as p). In the following section, we discuss the *isoefficiency* metric proposed by Grama et al. [2] to describe the scalability of a completely parallelizable algorithm, i.e., the rate at which the problem size W must scale relative to the number of parallel workers p to maintain constant parallel efficiency. **Note that the isoefficiency metric is only defined for algorithms that are completely parallelizable**.

Isoefficiency function

The isoefficiency function is a metric proposed by Grama et al. [2] which relates problem size to the number of parallel workers required to maintain a system's efficiency. A system's isoefficiency function can be derived from Eq. 2.25. To start, we substitute Eq. 2.23 into Eq. 2.25 to obtain Eq. 2.33 [2]. Given Eq. 2.21, we obtain Eq. 2.34 [2] which expresses efficiency E_P in terms of parallel overhead T_O, problem size W, and the constant-time cost of a basic operation t_c.

$$\begin{aligned} E_P &= \frac{T_1}{pT_P} \\ &= \frac{T_1}{T_1 + T_O} \\ &= \frac{1}{1 + \frac{T_O}{T_1}} \end{aligned} \tag{2.33}$$

$$E_P = \frac{1}{1 + \frac{T_O}{t_c W}} \tag{2.34}$$

Equation 2.35 can be obtained by rearranging Eq. 2.34 to isolate W [2], where $K = \frac{E_P}{t_c(1-E_P)}$ is a constant that depends on the efficiency.

$$W = \frac{1}{t_c}\left(\frac{E_P}{1-E_P}\right)T_O$$
$$W = KT_O \tag{2.35}$$

Since, for completely parallelizable algorithms [64], the overhead T_O is a function of the number of parallel workers, Eq. 2.35 can be used to obtain W as a function of p. This function is defined by Grama et al. [2] as the *isoefficiency function*. The isoefficiency function dictates how the problem size must grow in relation to the number of parallel workers to maintain a fixed efficiency.

Grama et al. illustrate that parallel reduction has an isoefficiency function in $\Theta(p \log p)$ [2]. To arrive at this result, first consider adding n numbers on a sequential machine where reading or writing each number in memory takes t_c time and adding two numbers also takes t_c time. Summing all numbers sequentially requires loading n numbers from memory, performing $n-1$ additions, and writing the final result back to memory. The problem size W is then equal to $2n$, resulting in a sequential execution time of $T_1 = Wt_c = 2nt_c$.

Now consider summing n numbers using p parallel workers, as shown in Fig. 2.11. In stage i, the numbers are assigned evenly across the p parallel workers. Each worker loads $\frac{n}{p}$ numbers from memory, performs $\frac{n}{p} - 1$ additions, and writes the result back to memory, leading to an execution time of $2\frac{n}{p}t_c$ for each worker. In stage ii, the remaining problem is to add the partial sums of the workers together. This can be done in $\log p$ steps, where in each step one or more workers concurrently load 2 numbers from memory, add them together, and write the result to memory, for an execution time of $(2t_c + t_c + t_c)\log p = 4t_c \log p$. The total parallel execution time of both stages is $T_P = (2\frac{n}{p} + 4\log p)t_c$. Substituting T_1 and T_P into the definition of speedup in Eq. 2.24, results in the efficiency shown in Eq. 2.36.

$$E_P = \frac{S_P}{p} = \frac{2n}{2n + 4p\log p} = \frac{W}{W + 4p\log p} = \frac{1}{1 + \frac{4p\log p}{W}} \tag{2.36}$$

Based on the form of efficiency shown in Eq. 2.34, the parallelization overhead time for the reduction is $4t_c p \log p$, leading to an isoefficiency function in $\Theta(p \log p)$. In other words, if the problem size grows at the rate of $\Theta(p \log p)$, **the efficiency will not decrease** as p increases. Moreover, this parallel reduction algorithm can be applied with associative binary operators

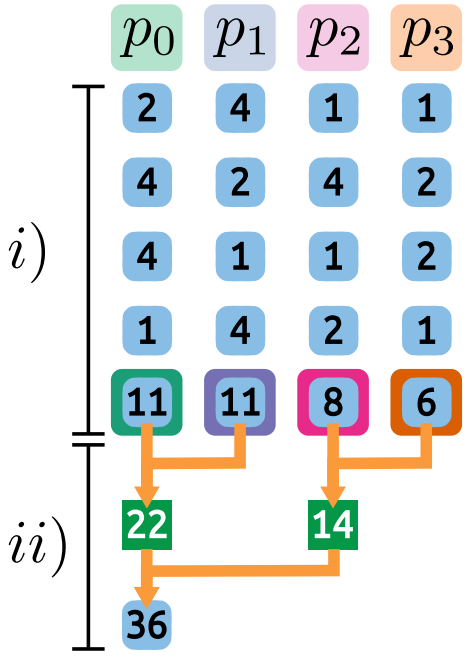


Figure 2.11: Add 16 numbers using 4 parallel workers.

other than addition, including minimum, maximum, etc., which all result in the same isoefficiency function. All such reductions are *completely parallelizable* [64, 66] where constant parallel efficiency can be maintained by increasing the problem size along with the number of parallel workers at the rate of $\Theta(p \log p)$, resulting in efficient *weak scaling* [1].

While the reduction pattern acts as a useful example for understanding isoefficiency, it has been used as the basis for composing higher level parallel patterns including *scan*, which can in turn be used to implement even higher level parallel patterns. In Section 2.3.4, we describe scan along with the following patterns that can be composed from reduction and scan [71]: category reduce, sort-by-key, pack, and split. Note that efficient implementations of all of the patterns we describe in Section 2.3.4 are available in parallel computing libraries such as NVIDIA's Thrust library [72]. In the implementations we discuss in Chapters 5-6, we employ the Thrust library to target NVIDIA CUDA GPUs using only patterns that can be composed from map, reduction, and scan, leading to an isoefficiency of $\Theta(p \log p)$.

Work-span model

In this section, we introduce the *work-span* [67] model for analyzing the runtime behaviour of parallel algorithms. The work-span model is relatively straightforward, yet is much more useful than Amdahl's Law in practice, since work-span takes into account imperfect parallelism and varying problem size. Moreover, whereas the overhead T_O is a function of the number of parallel workers, work and span only analyze parallel execution time for the two asymptotic extremes of parallel worker count (rather than as a function of p), making work-span analysis often easier than analyzing T_O directly. Furthermore, as shown in [66], in some cases equivalent T_O expressions can be derived based on the span, making isoefficiency analysis possible. For example, as shown in [2, 66], **a parallel algorithm with work in $\Theta(n)$ and span in $\Theta(\log n)$ leads to an isoefficiency function in $\Theta(p \log p)$.**

In the work-span model, tasks are arranged in a directed acyclic-graph (shown in Fig. 2.12b), representing the data dependencies between tasks, which we will refer to as the *"data-dependency graph"*. A task is ready if all tasks corresponding to incoming dependencies are finished.

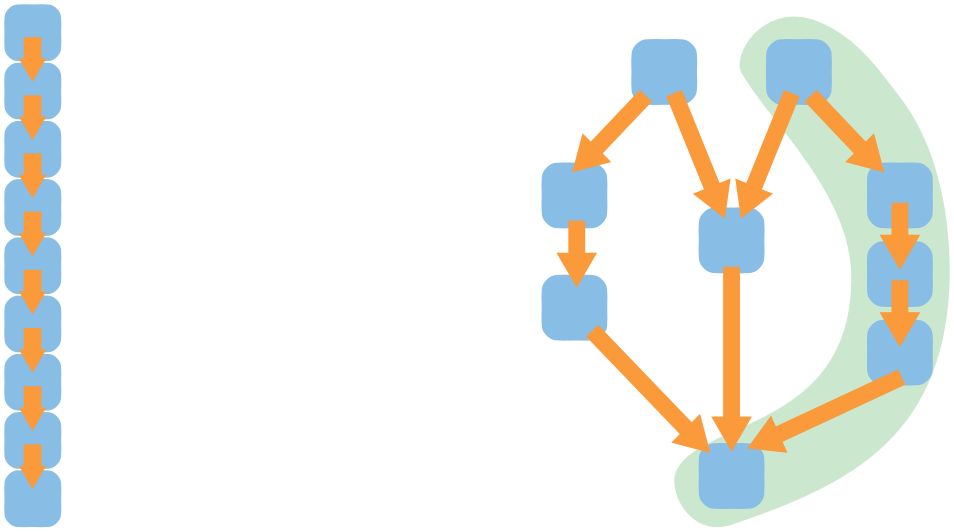
The data-dependency graph is analyzed under two conditions, $p = 1$ and $p = \infty$, that is, when the tasks are executed with a single worker and when the tasks are executed with an infinite number of parallel workers. The execution time associated with each of these scenarios is assigned a special name. The execution time when $p = 1$ is referred to as the *"work"* and corresponds to the time to run all tasks sequentially (Fig. 2.12a). The execution time when $p = \infty$ is referred to as the *"span"*, and corresponds to the "critical path" through the data-dependency graph (Fig. 2.12b). We denote the *"work"* as T_1, and we denote the *"span"* as T_∞.

Leiserson et al. [67] define two laws, Eq. 2.37 and Eq. 2.38, which place lower bounds on the parallel execution time based on the work and span.

$$T_P \geq \frac{T_1}{p} \tag{2.37}$$

$$T_P \geq T_\infty \tag{2.38}$$

The reasoning behind the *work law* (Eq. 2.37) is that, assuming equivalent worker architecture, a single worker can be used to simulate a p-worker system in p time steps, while the p-worker



(a) Work (T_1): Single worker.

(b) Span (T_∞): Infinite number of workers.

Figure 2.12: Work and span for a parallel algorithm

system may incur additional overhead due to imperfect parallelism. Since any practical parallel system has a finite number of parallel workers, the *span law* (Eq. 2.38) states that the parallel execution time on p workers must be at least as long as the parallel execution time with an infinite number of workers. The span law results from the observation that a parallel system with an infinite number of workers could theoretically simulate any p-worker system by only using p workers [67].

Note that the work-span model is particularly useful when composing parallel programs from existing parallel computing patterns with known work and span. Specifically, by organizing a program by composing pre-analyzed patterns, programmers can easily derive these useful metrics (i.e., the work and span of the program) without resorting to direct analysis. In Section 2.3.4 we describe a set of such parallel programming patterns, also called "algorithmic skeletons" [58, 59], providing the "best known method" for various parallel operations. Each of the patterns we discuss has known work and span for the "best known" method, allowing work-span analysis to be conducted for programs constructed from the patterns. In Chapters 4-6, we apply work-span analysis to our proposed placement method and objective cost algorithms, which are composed entirely of the patterns described in Section 2.3.4. In particular, we show that, **of the patterns used in our proposed placement methodology, the highest work complexity is $\Theta(n)$ and the highest span complexity is $\Theta(\log n)$, resulting in an overall isoefficiency**

function of $\Theta(p \log p)$, leading to weak scaling [1, 2, 66]. In the next section we briefly cover explicit thread-based parallel programming and present some reasons why expressing scalable parallelism can be difficult using low-level thread-based parallel programming.

2.3.3 Explicit thread-based programming

Threads are one of the most common mechanisms used to express explicit parallelism in parallel codes. Thread-based programming has several advantages that have contributed to its widespread adoption, including:

- Threads support can be added to established languages (e.g., `C`, `C++`) through software libraries, without requiring changes to the languages themselves.
- Threads extend the concept of well-established serial programming to parallel execution.

Unfortunately, while the characteristics listed above originally made threads an attractive model for expressing explicit parallel execution, in practice there are several fundamental issues with programming custom parallel programs directly at the level of threads [73].

One key issue with thread programming is that can be difficult to ensure determinism, since scheduling of thread execution is handled by the operating system and is out of control of the application. Although determinism can be enforced through mechanisms such as a shared, ordered queue, contention for such a resource quickly rises along with the number of threads. While *lock* mechanisms can be used to control access to a single shared resource, locks typically prevent scaling beyond a few threads.

Since the performance of a multi-threaded parallel program on a particular architecture can depend greatly on the number of threads used, the thread count is typically empirically tuned and fixed at compile-time for the target processing architecture. This greatly reduces portability of thread-based parallel codes. Moreover, when the number of utilized hardware threads is fixed speedup does not increase as the number of available hardware threads increases, causing parallel efficiency to decline. Furthermore, contention over locks increases with the number of parallel workers, causing parallel efficiency to decrease as the number of parallel workers grows.

While programming parallel applications by explicitly managing threads extends serial semantics to execute tasks in parallel, typical usage prevents such programs from maintaining determinism and constant parallel efficiency as the number of parallel workers increases. In the next section, we discuss general parallel programming patterns that have been proposed in the literature to provide an alternative, higher-level means of expressing scalable, and in most cases deterministic, parallelism.

2.3.4 Structured parallel programming

Prior to the introduction of parallel computing architectures in mainstream workstations, simulations of parallel algorithms targeted main-frame parallel computers such as the "Connection Machine" [65]. Due to the constraints of such architectures, much of the parallel computing literature [64, 66, 71, 74] focused on data-parallel algorithms which promoted fine-grained, *independent* parallelism. Such research lead to *completely parallel* [64, 66] algorithms for important recurring patterns such as parallel reduction and sorting. Recently, in response to the deficiencies of programming applications at the low-level of threads [73], particularly when targeting architectures with hundreds or thousands of hardware threads (e.g., GPUs), there has been renewed interest in these structured parallel patterns for composing scalable parallel algorithms [1].

These patterns, also called "algorithmic skeletons" [58, 59] in the literature, are abstracted away from any particular architecture and represent "best known" methods for developing highly-efficient parallel algorithms that scale to massively parallel systems. We want to stress that these structured parallel patterns are not tied to any particular architecture or language, but that highly-optimized implementations of the patterns (or at least the building blocks necessary to implement the patterns) described in this section are now available in existing libraries targeting modern mainstream parallel architectures. For example, all patterns described below are available as functions in the NVIDIA Thrust C++ template library [72], which can be compiled to to target one of several back-ends, including serial C++, OpenMP, Intel Threading Building Blocks [75], or NVIDIA CUDA GPUs. The patterns may also be implemented directly using Intel Threading Building Blocks [75] or Cilk Plus [1, 76].

In addition to abstracting away architecture-level details of parallel programming, the structured parallel patterns have been characterized using the work-span model described in Sec-

tion 2.3.2. This enables the structured patterns [1, 71] to be composed to express scalable parallelism to form complex, efficient applications targeting highly parallel architectures. In fact, our parallel placement methodology, described in Chapters 4-6, maps *all* non-constant-time operations to the structured parallel patterns described in this section.

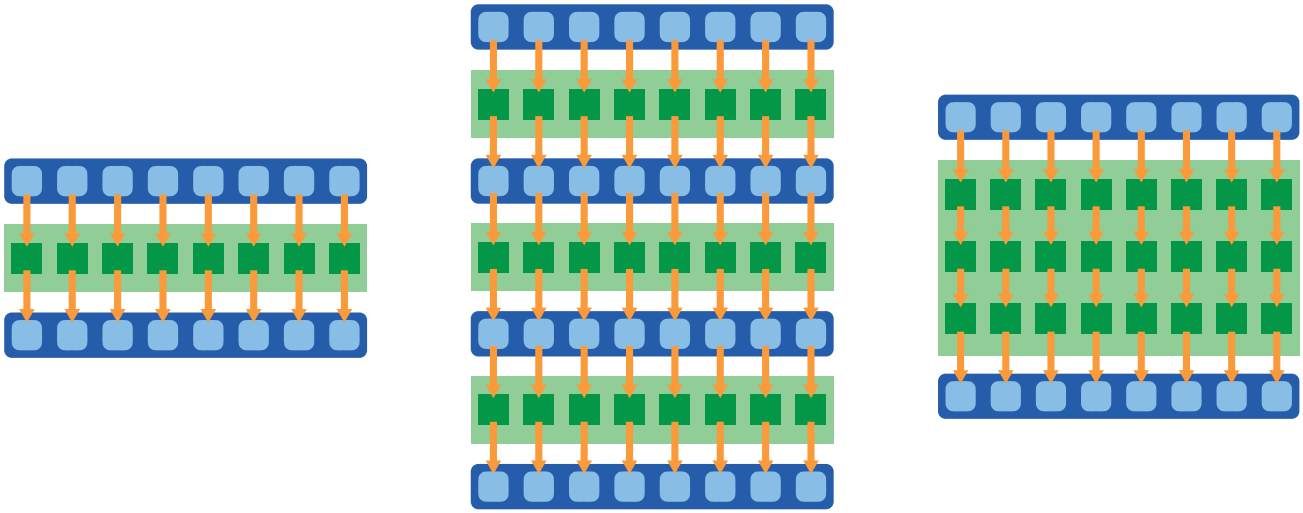
All parallel patterns described in this section are *completely parallelizable* [64, 66] with isoefficiency functions in either $\Theta(p)$ or $\Theta(p \log p)$ (depending on the span complexity of the pattern). Moreover, all of the patterns presented here can be implemented without the use of locks (e.g., NVIDIA Thrust C++ template library [72]). Therefore, expressing a parallel application in terms of these patterns ensures:

- Weak-scalability, where speedup is expected to grow with $\Theta(p)$ as long as the problem size grows with $\Theta(p \log p)$.
- The algorithm is not architecture or language specific.

Both of these points are noteworthy. By composing a parallel algorithm using these parallel patterns with known work and span, the algorithm is expressed in a common vocabulary without targeting any specific architecture. Furthermore, the speedup of the resulting application will increase along with the size of the problem as additional parallel workers are added. Table 2.2 lists the patterns used in our proposed parallel algorithms in Chapters 4-6, along with the corresponding work, span, and isoefficiency function of the best known algorithm for each pattern.

Map

The *map* pattern is the fundamental mechanism for describing, regular data-parallelism [1], *i.e.*, parallelism that scales with the problem size. The map pattern, also known as *transform*, applies a sequence of one or more unary operations to each item in an ordered input list, to produce an output list of the same size. In other words, the transform pattern is a one-to-one mapping applied to each item in an input list. Figure 2.13a illustrates the general map pattern, where a single unary operator is applied to each element in an input list. Note that in cases where several consecutive map operations are applied to a list, it is usually possible to combine the



(a) Map (b) Sequence of several maps (c) Fused sequence of maps

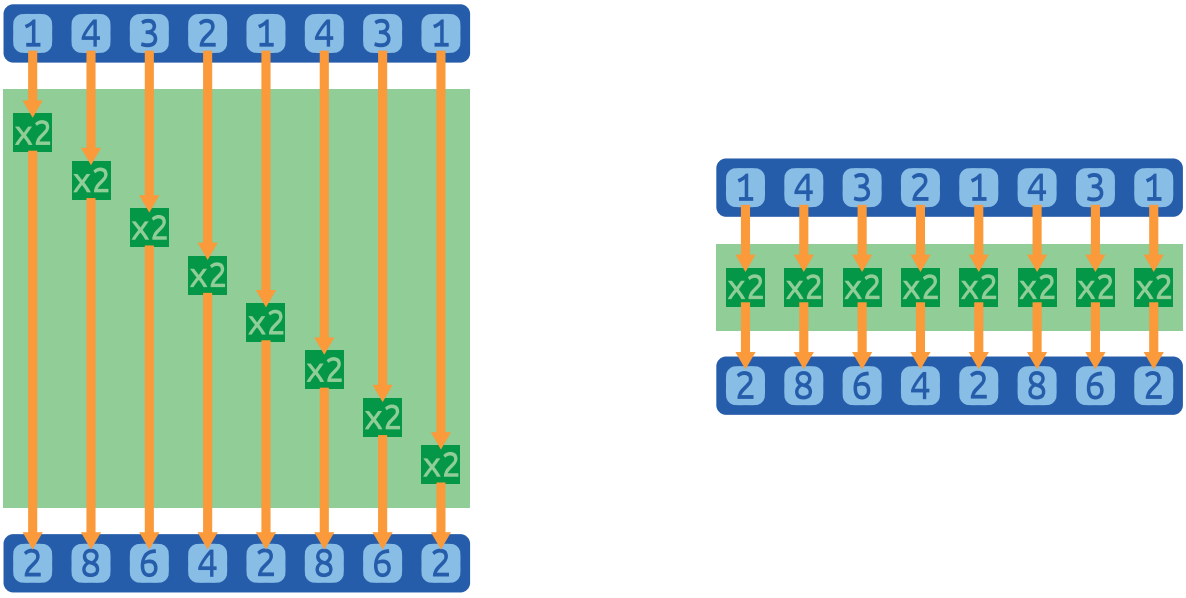
Figure 2.13: Map pattern

Pattern	Work	Span	Isoefficiency
Map	$\Theta(n)$	$\Theta(1)$	$\Theta(p)$
Gather	$\Theta(n)$	$\Theta(1)$	$\Theta(p)$
Scatter	$\Theta(n)$	$\Theta(1)$	$\Theta(p)$
Reduce	$\Theta(n)$	$\Theta(\log n)$	$\Theta(p \log p)$
Category reduce	$\Theta(n)$	$\Theta(\log n)$	$\Theta(p \log p)$
Sort-by-key	$\Theta(n)$	$\Theta(\log n)$	$\Theta(p \log p)$
Pack	$\Theta(n)$	$\Theta(\log n)$	$\Theta(p \log p)$
Split	$\Theta(n)$	$\Theta(\log n)$	$\Theta(p \log p)$
Scan	$\Theta(n)$	$\Theta(\log n)$	$\Theta(p \log p)$

Table 2.2: Work, span, and isoefficiency function complexity of structured parallel patterns

functionality of all unary operators into a single operator, such that multiple applications of map may be replaced by a *single* map application. This algorithm transformation is called "fusion", where we say that the consecutive map operators are "fused" together [1]. Figure 2.13b and Fig. 2.13c illustrate a sequence of maps and the corresponding fused map, respectively.

To help understand the map pattern, consider the operation in Fig. 2.14, where each element



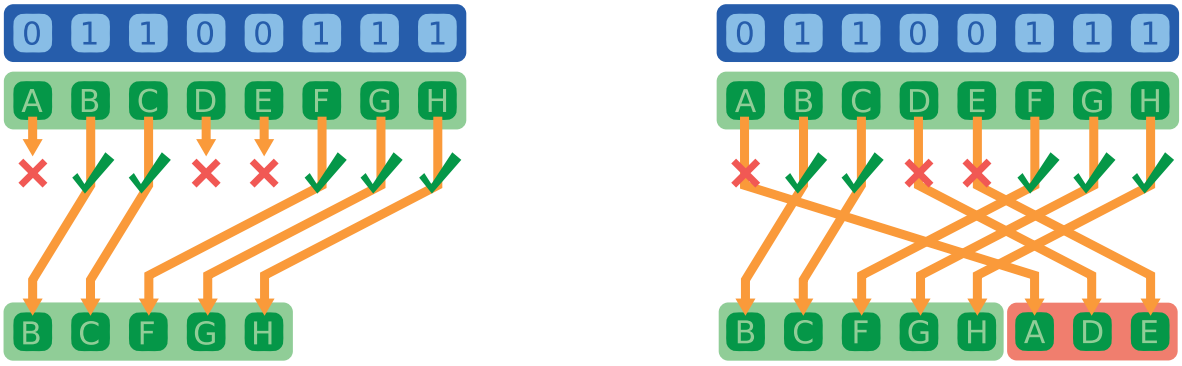
(a) Serial (b) Parallel

Figure 2.14: Map pattern example

in the input list is doubled. Fig. 2.14a shows how such a pattern may be implemented in serial, where each element is processed one-after-another, in sequence. Figure 2.14b illustrates how, given a sufficient number of parallel workers, all elements may be processed concurrently. Note that the work is 8 for the parallel map in Fig. 2.14b because the size of the data set is 8, and the span is 1 because there is just one computation step regardless of the data set size. This example highlights a key characteristic of all the parallel patterns we will look at. Any of the patterns we cover express "optional parallelism" [1], that is, a well-defined set of tasks that *may* be executed in parallel *if* it is beneficial to do so for a particular input set on a particular architecture. For instance, in circumstances where there are many more short tasks than parallel workers, it is often beneficial for each parallel worker to process more than one element at a time to reduce scheduling overhead. However, we want to stress again that structured parallel patterns abstract away these types of low-level, architecture-specific details.

Split

Since scalable parallel patterns typically exhibit regular data-parallelism, typical control flow structures like those used in serial code (e.g., `if`, `switch`, etc.) must often be replaced in



(a) Pack (b) Split

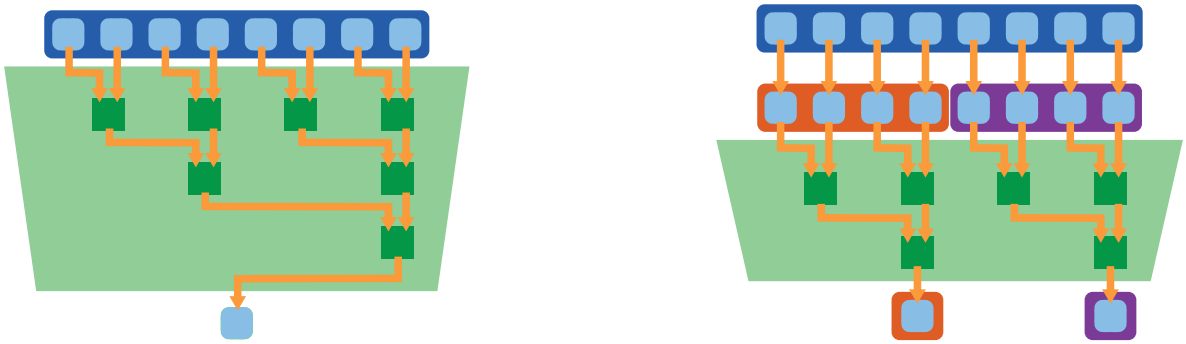
Figure 2.15: Binary partitioning/stream compaction

parallel algorithms with patterns that reorganize data such that identified sections of data may be handled differently. The most straight-forward example of this is what is called *stream compaction*, or the *pack* pattern [1]. Figure 2.15 illustrates two variations of stream compaction.

Figure 2.15a shows the *pack* pattern, which applies a binary mask to an input list to gather all elements for which the mask is set to the front of the list. The pack pattern is useful for filtering out data from a list according to some criteria. In the case of pack, the length of the resulting list is equal to the number the packed elements, and the result is typically written a separate structure. Figure 2.15b shows another variation of *pack*, called *split*, where instead of only gathering the masked elements to the front of the output list, the elements in the input list are partitioned according to the mask. Implementations of the *split* pattern exist that may operate in-place on the input list [72]. Note that both pack and split also compute the number of items set in the mask. This enables later patterns to operate on a subset of the resulting list as necessary.

Reduce

The *reduce* pattern recursively applies an *associative* binary operator to the elements of an input list to produce a single output value. Examples of binary operators include addition, multiplication, minimum, maximum, etc. Figure 2.16a illustrates how a reduction may be applied in parallel in the form of a tree with depth $log_2 n$. Note that this depth corresponds to the *span* of the reduce pattern.



(a) Reduce

(b) Category reduce

Figure 2.16: Reduce recursively applies a "combiner" binary operator to produce a single value. Category reduce applies reduction to all input elements assigned to the same category.

Figure 2.16b illustrates a variation of the *reduce* pattern, called *"category reduce"*. Like the basic *reduce* pattern, the *category reduce* pattern recursively applies an *associative* binary operator to the elements of an input list. The key difference is that each element in the input list is assigned a category beforehand, and all elements within each category are reduced to produce a single output value, resulting in one output value per category.

It is important to note that category reduction can be used to efficiently apply a reduction to each row or column in a *sparse matrix*. A sparse matrix is a matrix where the majority of elements in the matrix are zero (or some other constant value, though for the sake of clarity, we assume "empty" matrix positions correspond to a value of zero). Rather than storing all elements of such a matrix, special encodings may be used to only store the non-zero elements. For example, consider the example matrix in Fig. 2.17, which is encoded using what is called a "Coordinate list" (or COO, for short). Note that there are only four non-zero matrix elements. For each element, along with the element value, the row index and column index are stored.

Given a COO-encoded sparse matrix, it is possible to apply a *reduce* pattern (not to be confused with matrix row reduction) along either rows or columns of the matrix using a *category reduce* pattern. For example, consider the 12x10 sparse matrix depicted in Fig. 2.18a (note that only non-zero elements are shown). Each row is assigned an index value between 0 and 11, and each column is assigned an index between 0 and 9.

To help understand how the sparse, COO-encoded matrix representation is formed, consider Fig. 2.18b-Fig. 2.18d. First, in Fig. 2.18b, we spread out the matrix such that each non-zero

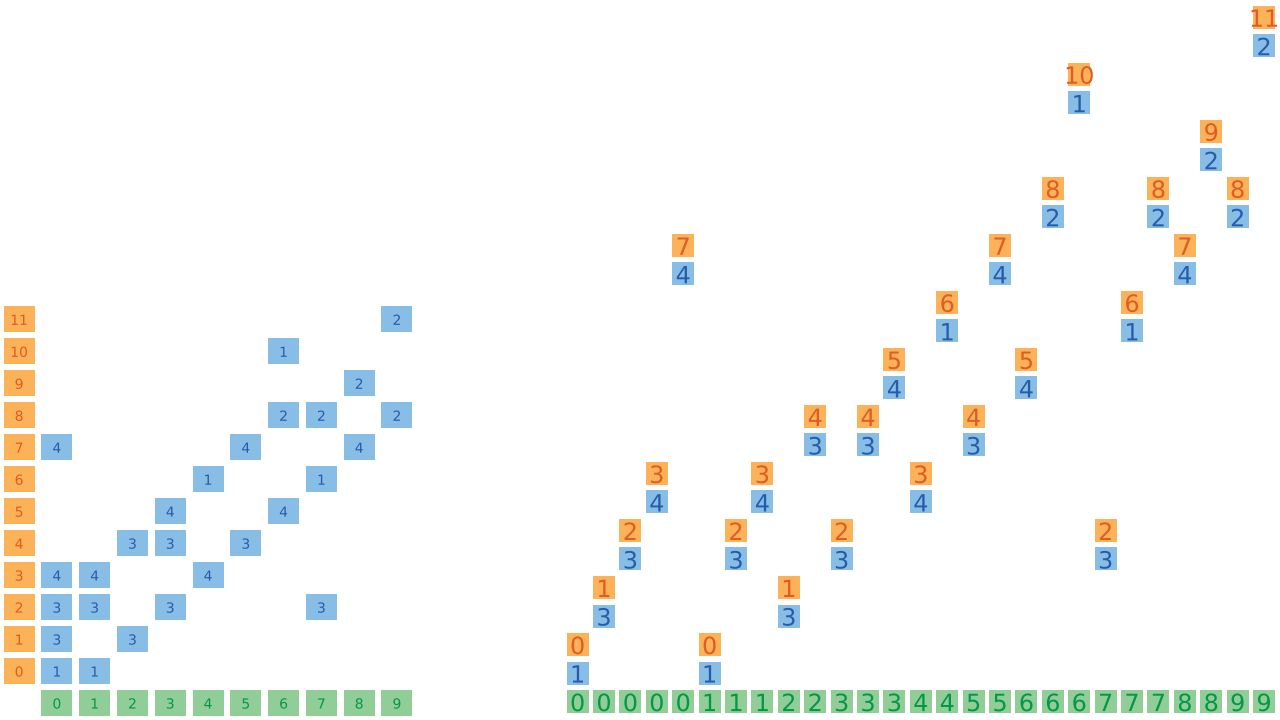
```
>>> row = [0, 3, 1, 0]
>>> col = [0, 3, 1, 2]
>>> data =[4, 5, 7, 9]

Resulting sparse matrix:

        [[4, 0, 9, 0],
         [0, 7, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 5]]
```

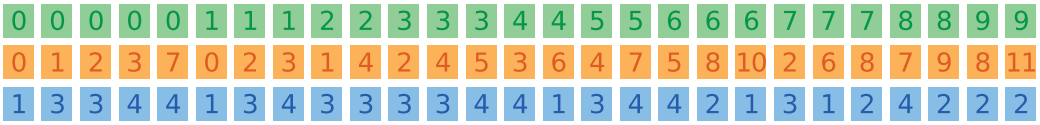
Figure 2.17: Sparse matrix represented using a coordinate-list encoding (i.e., "COO" encoding). For each element, the following data is stored: row index, column index, and element value.

element occupies its own column. To keep track of which column each element was originally located in, we replicate the original column index as necessary, labelling each element with a column index. In addition, we tag each element with the index of the row in which it is located. Next, in Fig. 2.18c, we collapse the matrix along the columns, resulting in three lists of the same length, corresponding to: a) the column index, b) the row index, and c) the element value, for each non-zero matrix element. Note that this corresponds to the COO-encoding of the sparse matrix. Finally, in Fig. 2.18d, using the column index as a category marker, we apply a category reduce pattern using an addition operator, to compute the sum of all elements within each column of the sparse matrix. We use this approach to perform various sparse matrix operations in Chapters 4-6, applying operators including sum, maximum, and minimum. For example, as part of a wirelength calculation we use a sparse matrix containing the x coordinate of each block (columns indexed by block index) connected to each net (rows indexed by net index). We then perform a category reduce with an addition operator along the rows of the matrix, to compute the sum of the x coordinates of all blocks connected to each net. See Section 5.2 in Chapter 5 for details.

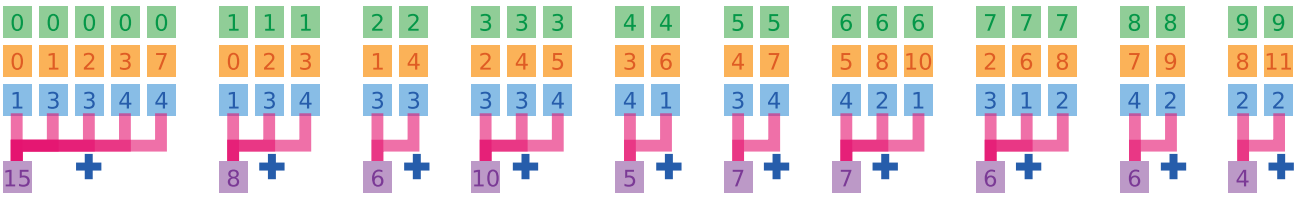


(a) Sparse matrix with 12 rows and 10 columns.

(b) Spread elements of matrix, tagging each element with row index, and replicating column indexes as necessary.



(c) Flattened matrix elements, *i.e.*, COO-encoded sparse matrix representation.



(d) Category reduction of elements by column.

Figure 2.18: Using category reduction to efficiently compute sum of each column in a sparse matrix.

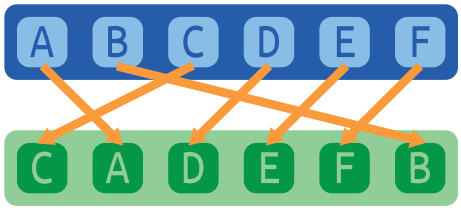


Figure 2.19: Permutation scatter reorders any number elements of the input list.

Permutation scatter

The permutation scatter pattern reorders the elements in an input list according to a permutation of list indexes into an output list. Figure 2.19 illustrates an example of a permutation scatter. Note that each index position must occur *exactly once* in the index permutation list, otherwise the result is undefined.

In Chapter 4, we describe how we use the permutation scatter pattern to efficiently update a placement according to a set of accepted moves.

Scan

Unlike most of the other patterns described above, a version of the *scan* pattern does not typically show up in serial code. However, scan is an invaluable pattern that is actually used to form many other parallel patterns, including other patterns mentioned above such as category reduce. Figure 2.20 shows the equivalent serial code for computing a scan using addition, which corresponds to computing the prefix-sum of a list. As an example, consider an array with the following contents:

$$5, 0, 3, 3, 7, 9, 3, 5, 2, 4$$

Applying a prefix sum scan to the array would result in the following sequence:

$$5, 5, 8, 11, 18, 27, 30, 35, 37, 41$$

For the scope of this thesis, it is not important to know the exact details of the scan pattern and where it is used, but rather that the work complexity is $\Theta(n)$ and the span complexity is $\Theta(\log_2 n)$, implying that the pattern exhibits weak-scaling, according to Gustafson-Barsis' Law.

```
1     int data[N];
2
3     /* ... fill `data` ... */
4
5     for (int i = 0; i < N - 1; i++) {
6       data[i + 1] += data[i];
7     }
```

Figure 2.20: Serial implementation of prefix sum, which is a specific application of the scan pattern using a binary addition operator.

We use the scan pattern in Chapter 4 during the assessment of groups of moves that must be applied together to maintain a valid placement.

2.4 Benchmark netlists

To evaluate the efficacy of our placement strategies that we propose in Chapters 4-6, we use a set of 7 large benchmark netlists originally reported in [77], ranging from approximately 0.6-1.4 million block-to-block connections, which are based on the IWLS 2005 benchmark set [78]. While many results in literature compare results on much smaller netlists from the MCNC set [55], we selected this set of benchmark netlists due to their large size to better evaluate the runtime of our proposed methods for large circuit designs, closer to the scale of modern FPGA architectures. Table 2.3 summarizes the benchmark circuits in terms of block count of each type, connection count, synchronous delay level count, and mean fan-out (i.e., the number of blocks driven by each net).

2.5 Summary

In this chapter, we covered several topics. In Section 2.1, we presented an overview of the FPGA placement problem. In Section 2.2, we discussed several categories of methods proposed in the literature for performing FPGA placement. We also discussed common objective-cost models considered when performing placement, including wirelength, timing, and congestion.

Netlist	Connections	Inputs	Outputs	Logic blocks	Delay levels	Mean fanout
`uoft_raytracer`	575395	542	425	170069	64	3.4
`b17_1_new_mapped`	581521	193	321	124660	29	4.7
`leon3mp`	614752	158	250	216562	17	2.8
`netcard`	623926	61	70	203727	13	3.1
`b18_new_mapped`	727922	182	116	156623	56	4.6
`b18_1_new_mapped`	729741	182	111	156955	56	4.6
`leon2`	772751	124	197	273921	16	2.8
`b19_1_new_mapped`	1428437	108	162	306719	56	4.7

Table 2.3: Benchmark netlists based on IWLS 2005 circuits.

In Section 2.3, we covered several aspects related to parallel programming. First, we presented evidence to show that single-core processor performance is no longer increasing at a significant rate, but peak parallel computational throughput is still growing steadily with new generations of both multicore and manycore architectures. In Section 2.3.2 we described asymptotic analysis of parallel algorithms in terms of a *work-span* model, and scalability in the form of isoefficiency for completely parallelizable [64, 66] algorithms which promote the concept of weak-scaling by maintaining constant parallel efficiency by scaling the problem size according to the number of parallel workers.

Finally, we identified some fundamental issues that prevent most programs that express parallelism at the thread level from scaling to larger problem sizes or to new parallel architectures. However, we also presented structured parallel patterns that have been proposed in the literature with known work complexity and span complexity that can be composed to create

completely parallelizable [64, 66] algorithms. Such parallel algorithms exhibit weak scalability, with isoefficiency functions in $\Theta(p \log p)$ where speedup is expected to grow with $\Theta(p)$ as long as the problem size grows with $\Theta(p \log p)$.

Chapter 3

Parallel iterative placement

In practice, the most common methods for performing FPGA placement employ iterative approaches, such as simulated annealing, to repeatedly apply incremental improvements to a placement. Such methods tend to produce high quality results, but at the expense of extensive runtimes. In traditional iterative placement methods each move is randomly generated independently from previous moves. In such methods, moves must be applied serially, one after another to prevent logical inconsistencies that may arise due to conflicts which we discuss in Section 3.2. When moves are evaluated serially, one at a time, placement runtime is dependent on the operating frequency of a single core. However, as we discussed in Section 2.3.1, advances in computational throughput for the foreseeable future will be through the use of increasingly abundant parallel processing cores, while operating frequencies are held constant or are potentially decreased. Therefore, it remains crucial to develop parallel variations of iterative methods for placement to exploit all available parallel computing resources to reduce placement times. While several parallel variations of simulated annealing for FPGA placement have been proposed in the literature [33, 34, 79, 80], **none of the existing methods exhibit weak-scaling parallelism**. Moreover, the approaches that report the most promising runtime improvements suffer significant reductions in solution quality, where quality decreases as parallelism increases [79, 80].

In this chapter, we discuss parallel iterative methods in general, and identify key complications that arise when conducting iterative placement in parallel. Specifically, in Section 3.1, we review the keys steps involved in traditional, serial, iterative placement and describe a general

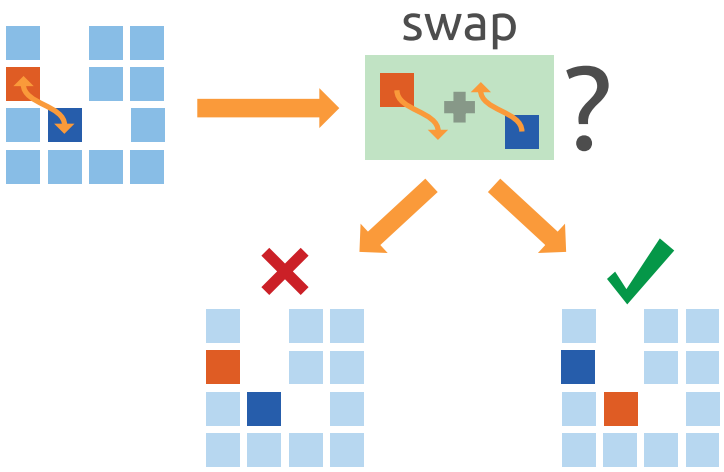


Figure 3.1: Serial iterative placement.

extension of iterative placement where multiple moves are evaluated in parallel. In Section 3.2, we describe types of conflicts that may arise when updating the position of more than one block concurrently. In Section 3.3 we discuss previous work in the area of parallel iterative placement, identify how conflicts are managed, and highlight fundamental issues with the described approaches. In Chapters 4-6, we propose our parallel placement methodology that addresses the key issues affecting the approaches described in this chapter. In our approach, all non-constant-time operations are conducted using scalable, structured parallel patterns.

3.1 Overview

Before discussing parallel iterative placement methods, we first review the steps involved in performing serial iterative placement. As illustrated in Fig. 3.1, two locations within the placement area are selected. The contents of those locations are then considered to swap places. An estimated difference in cost is then calculated based on the swap being applied. Based on the difference in cost, the swap is either accepted or rejected. If the swap is accepted, the placement is updated accordingly. The process is then repeated until a stopping condition is met.

When performing iterative placement in parallel, instead of considering a single swap between a pair of locations, we would *ideally* like to consider *many* swaps concurrently, as illustrated in Fig. 3.2. Each swap may be evaluated independently and either accepted or rejected according to the same acceptance criteria used in the serial algorithm. Any accepted swaps are then

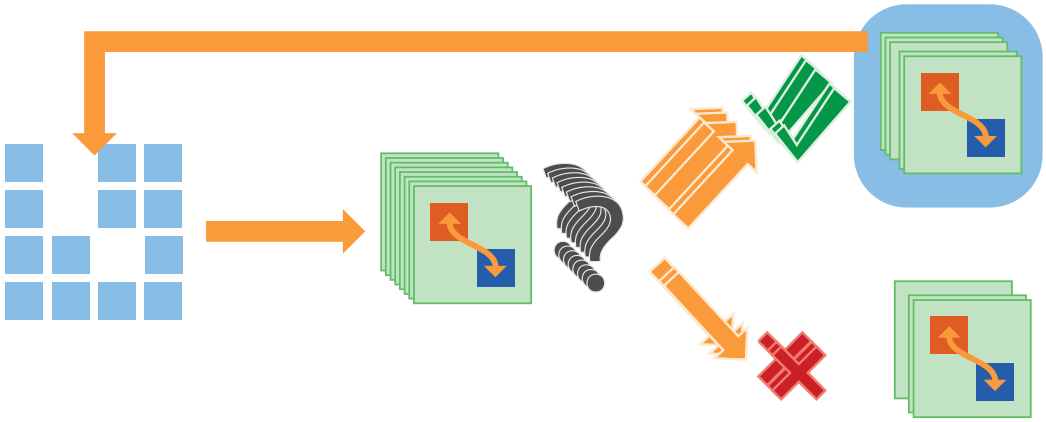


Figure 3.2: Parallel iterative placement.

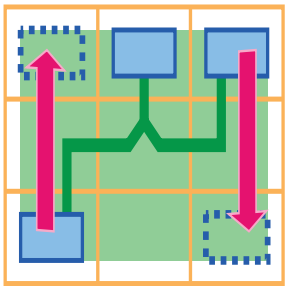
applied to the placement and the process is repeated, again, until a stop condition is met.

To achieve high levels of parallelism, we would ideally like to consider as many swaps as possible in an iteration. However, if a set of swaps is randomly generated, there is a chance that a location may be involved in more than one swap, which may result in conflicts if multiple swaps involving the same location are accepted during the same iteration. In the next section, we outline the types of conflicts that may occur during parallel placement in detail.

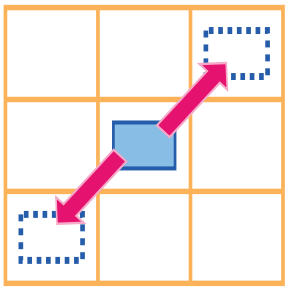
3.2 Move conflicts

When applying multiple moves concurrently, there are two main categories of conflicts that may occur. The first category we refer to as "soft conflicts". As illustrated in Fig. 3.3a soft conflict occurs when two blocks are moved that are connected to the same net. Soft conflicts may be ignored and the resulting placement will still be valid. However, the change in cost due to moving the involved blocks will be based on stale data, where the cost of moving each block is evaluated as if no other block is changing position. However, in practice, soft-conflicts have not been shown to significantly impact solution quality [79]. Therefore, in our parallel placement methods proposed in Chapters 4-6, we ignore soft conflicts.

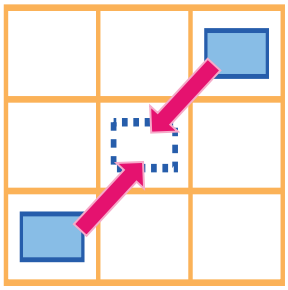
We refer to the second category of conflicts as "hard conflicts". These types of conflicts arise in one of two situations. The first situation, shown in Fig. 3.3b, occurs when a block is assigned two or more concurrent moves to different locations. The second type of hard conflict, as shown



(a) "Soft conflict": two blocks connected to the same net.



(b) "Hard conflict": two moves assigned to the same block.



(c) "Hard conflict": Two blocks assigned to the target location.

Figure 3.3: Types of move conflicts.

in Fig. 3.3c occurs when two or more blocks are assigned concurrent moves that target the same location. Unlike soft conflicts, if hard conflicts are ignored, the resulting placement is undefined, since non-deterministic data races may arise, leading to potential logical inconsistencies. In Section D.3, we discuss a method for verifying the validity of a placement.

In practice, the most complex aspect of extending serial iterative placement methods to parallel variations is determining how to deal with both soft conflicts and hard conflicts. In the next section, we present previous parallel iterative placement methods from the literature which resolve these types of conflicts in different ways. We discuss the methods these approaches use for dealing with both soft conflicts and hard conflicts, and note the performance and quality exhibited by each approach. Moreover, we identify key weaknesses with existing approaches to parallel iterative placement.

3.3 Previous work

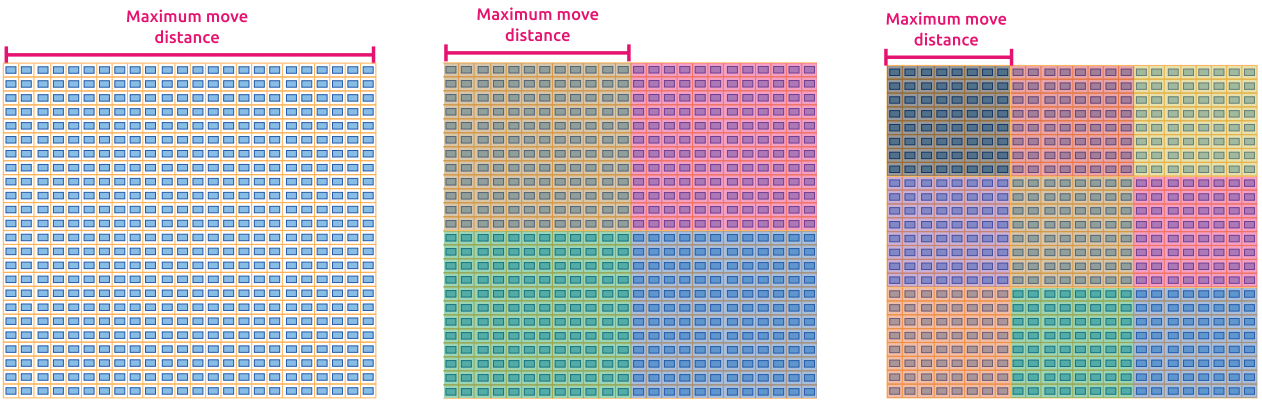
Since the placement methods that produce the best quality in literature are based on serial simulated annealing [32], runtimes have grown rapidly over the last decade since, as we presented in Chapter 2, single-core performance has not increased significantly since approximately 2005. Therefore, within the last ten years, there have been many attempts to parallelize traditional serial-annealing-based iterative placement, with varying degrees of success. The most successful proposed methods can be classified into two main categories, based on how they deal with the conflicts described in Section 3.2. In this section we describe these categories of approaches:

- Partitioning-based.
- Independent sets of moves.

The methods described in this section are summarized in Table 3.1.

Reference	Runtime improvement	Deterministic	Scalable	Parallel timing
Choong (2010) [77]	12x on GTX480	No	No	Yes[a]
Goeders (2011) [79, 80]	17.5x on 60+ cores	For a given number of cores	No	Yes
Ludwin (2011) [34]	3.8x[b] on 4+ cores	Yes	No	No[b]
An (2014) [33]	33x[b] on 64 cores	No	No	No[b]
An (2014) [33]	5.9x[b] on 8+ cores	Yes	No	No[b]
This thesis	**25-51x** on GTX980	**Yes**	**Yes**	**Yes**

Table 3.1: Parallel approaches to iterative placement. [a] Net-based timing was used, which is a simplified model compared to connection-based timing. [b] Although timing analysis was used, any timing-related code was *not included* when calculating runtime improvements. Since timing calculations are one of the most time-consuming operations in placement, overall runtime improvements 1) are not expected to scale, and 2) are likely significantly lower than reported in practice.



(a) Entire placement area. (b) Partitioned into 4 regions. (c) Partitioned into 9 regions.

Figure 3.4: Avoiding hard conflicts through spatial partitioning of the placement area.

3.3.1 Partitioning-based

The first category of methods relies on partitioning of the placement area. Placement is then carried out concurrently within each region, avoiding any possibility of conflicts with blocks in other regions. Additional mechanisms are then employed to allow blocks to migrate from one region to another, either through alternating repartitioning using overlapping areas [79, 80], or by finalizing the placement using a low-temperature anneal [29, 81]. While these approaches cleanly divide up the work for placement, there are two key issues, a) loss of quality, and b) non-scalable parallelism. As illustrated in Fig. 3.4, as the number of partitioned regions increases, the maximum distance that any block can move in a single move is reduced considerably. The impact of this is a dramatic reduction in quality as the number of regions is increased. Since the number of regions must be increased as parallel workers are added, the number of workers is limited for a given target quality of solution. Alternatively, as the number of parallel workers is increased, the quality is reduced. In the reported results [79], the authors state that quality drops by up to 20% due to the reduced maximum move length. With respect to runtime performance, the best speedup of partitioning-based methods is presented in [79], where 60 workers exhibited a speedup of 17.5× over a single worker. Another variation of partitioning-based approaches employs a hybrid CPU-GPU parallel strategy [77]. The proposed approach uses a CPU worker to partition the placement area into random sets of locations (not necessarily adjacent). Each set of locations is then assigned to a group of worker threads on a GPU, where each group of threads serially generates, evaluates, and conditionally applies random swaps involving the

locations in the assigned set. It is difficult to ascertain the quality and runtime performance of the proposed algorithm relative to existing standard methods from the literature [32], since the reported results were only compared against the proposed method running on a single CPU worker. The presented results show speedups of up to 25× relative to a single worker, but it should be noted that there is a significant amount of work still executed on the CPU, which scales with the size of the netlist to be placed. Furthermore, the proposed approach offers no support for timing-based objective cost calculation.

Although reported speedups for the partitioning-based parallel placement methods described above are relatively high, all of the described methods have a non-parallelizable portion of the algorithm that scales with the size of the netlist to be placed. This characteristic prevents this class of algorithm from scaling according to Gustafson-Barsis' Law, since the serial portion of the code grows at the same rate as the amount of parallel work as the problem size increases.

3.3.2 Independent sets of moves

The second category of parallel iterative FPGA placement methods proposes many random parallel swaps, but then must analyze the swaps to detect any potential conflicts. Of note is a deterministic parallel placement algorithm proposed by Altera [34], which produces identical results to an existing serial algorithm. This is important, since deterministic programs are much easier to debug, and are guaranteed to maintain consistent quality regardless of the number of parallel workers. However, the mechanism employed for detecting conflicting moves is a serialized queue, where parallel workers push accepted moves to. Figure 3.5a illustrates how, after evaluating moves concurrently, a set of parallel workers push accepted moves into a shared, serialized queue to enable conflict detection. Workers take turns temporarily becoming the master to process the queue, applying moves and broadcasting information about any detected conflicts to the parallel workers, signalling for the workers to reevaluate the conflicted moves based on the updated placement state. As the number of parallel workers is increased, contention over the shared queue is also increased, limiting the reported speedups to 2.4× on four cores. An extension of this work was proposed which detects move conflicts *before* evaluating the difference in cost due to the moves [33], as depicted in Fig. 3.5b. This change improved the observed speedups to 5.9× on 9 threads. However, speedups, again, were limited due to contention over

(a) Detect conflicts in accepted moves.

(b) Detect conflicts in proposed moves.

Figure 3.5: Randomly generate sets of moves. Detect and repair conflicts.

the shared serialized queue, where adding more than 9 parallel workers failed to scale the speedup beyond 5.9×. A variation was proposed as well, which uses hardware transactional memory to replace the serialized queue [33]. While this significantly improved performance, up to 34× speedup on over 64 cores, issues remain with such an approach. First, transactional memory is still imposing serialization, albeit accelerated in hardware. This serialization, again, prevents the parallel performance from scaling beyond 34×, as reported. Second, when using hardware transactional memory, the results are no longer deterministic, and, thus, the resulting quality is uncertain, and verifying and debugging the algorithm becomes much more difficult. Third, hardware transactional memory is not a standard feature in commodity processors, limiting the immediate benefit of the proposed approach, in practice.

All of the approaches discussed in this section lack scalability according to Gustafson-Barsis' Law. Furthermore, the most promising approaches in terms of runtime speedup are either non-deterministic or the behaviour of the algorithm changes depending on the number of parallel workers. In either case, the resulting quality is not consistent for different parallel worker counts.

3.4 Summary

In this chapter, we first reviewed serial, iterative, swap-based placement, followed by a generalized, extended approach to iterative placement using many concurrent moves. In Section 3.2, we identified two categories of conflicts that may occur when assessing and applying multiple moves in parallel. In Section 3.3, we discussed the most promising parallel FPGA placement

methods proposed in the literature, and described how they deal with move conflicts. As shown, partitioning-based approaches have been demonstrated to exhibit relatively high speedups, but these speedups are not scalable according to Gustafson-Barsis' Law. Moreover, in these approaches, quality diminishes as the number of parallel workers increases. Alternative methods in the literature utilize serialized queues or hardware transactional memory to detect and repair conflicts while parallel workers apply moves concurrently. While some of these approaches enable deterministic results regardless of the number of parallel workers used, the serialization mechanisms used to maintain determinism prevent weak-scaling, since the serial portion of the algorithm scales along with the size of the problem. Furthermore, the deterministic methods *do not perform timing calculations in parallel* and they *do not include the runtime spent in timing calculations* when measuring algorithm runtime [33, 34]. Therefore, the reported speedups are likely far off from absolute runtime improvements in practice. We also discussed a hybrid CPU/GPU approach to parallel FPGA placement from the literature. While the reported speedups are promising, a significant amount of work is performed on the CPU, where the amount of work scales with the size of the problem, again, preventing weak-scaling. Moreover, while the GPU-based method incorporates timing, the timing model used is net-based, which is a simplified model compared to connection-based timing, resulting in a negative impact on quality.

In Chapters 4-6, we present our proposed parallel placement methodology, which directly addresses the issues outlined above. First, we propose a novel method of concurrently generating very large sets of moves that may be applied concurrently, but are **guaranteed to be free of hard conflicts**. Secondly, we map all non-constant-time operations to the structured parallel patterns described in Chapter 2, resulting in an algorithm that exhibits work complexity and span complexity such that weak-scaling is achieved according to Gustafson-Barsis' Law. In Chapter 5 and Chapter 6, we present empirical results running our proposed algorithm on an NVIDIA CUDA GPU. Note that unlike the GPU-accelerated method discussed in this chapter, our proposed GPU implementation is the **first to perform all non-constant-time operations on the GPU and store all placement data structures in GPU memory, requiring data transfers between the CPU memory and GPU memory only at the beginning and end of the entire search**. Moreover, we are the first to present **scalable, deterministic, connection-based timing calculations suitable for a GPU**.

Chapter 4

Concurrent Associated-Moves Iterative Placement

In this chapter, we introduce a model for performing iterative FPGA placement, where all steps map to one or more of the structured parallel patterns discussed in Chapter 2. We call our model, *Concurrent Associated-Moves Iterative Placement (CAMIP)*. In the Section 4.1, we describe our *CAMIP* model at a high-level and demonstrate how traditional, serial annealing placement can be emulated using the model. In Section 4.1.3, we discuss how the CAMIP model may be used to implement a variation of traditional annealing to evaluate *many* groups of associated-moves concurrently, and we propose a novel method to efficiently generate many such groups of associated-moves, exposing a *very high* level of potential parallelism that scales along with the size of the netlist to be placed.

In Section 4.3, we describe a formal mathematical model for all operations of *CAMIP* that do not depend on the objective cost model, using matrix and vector operations. In Chapter 5 and 6, we extend the formal model from Section 4.3 to *efficiently* handle calculations related to *wirelength* cost and *timing* cost, respectively. Note that all operations proposed in our formal model, including the objective cost extensions, can be implemented *efficiently* using the parallel primitives described in Chapter 2.

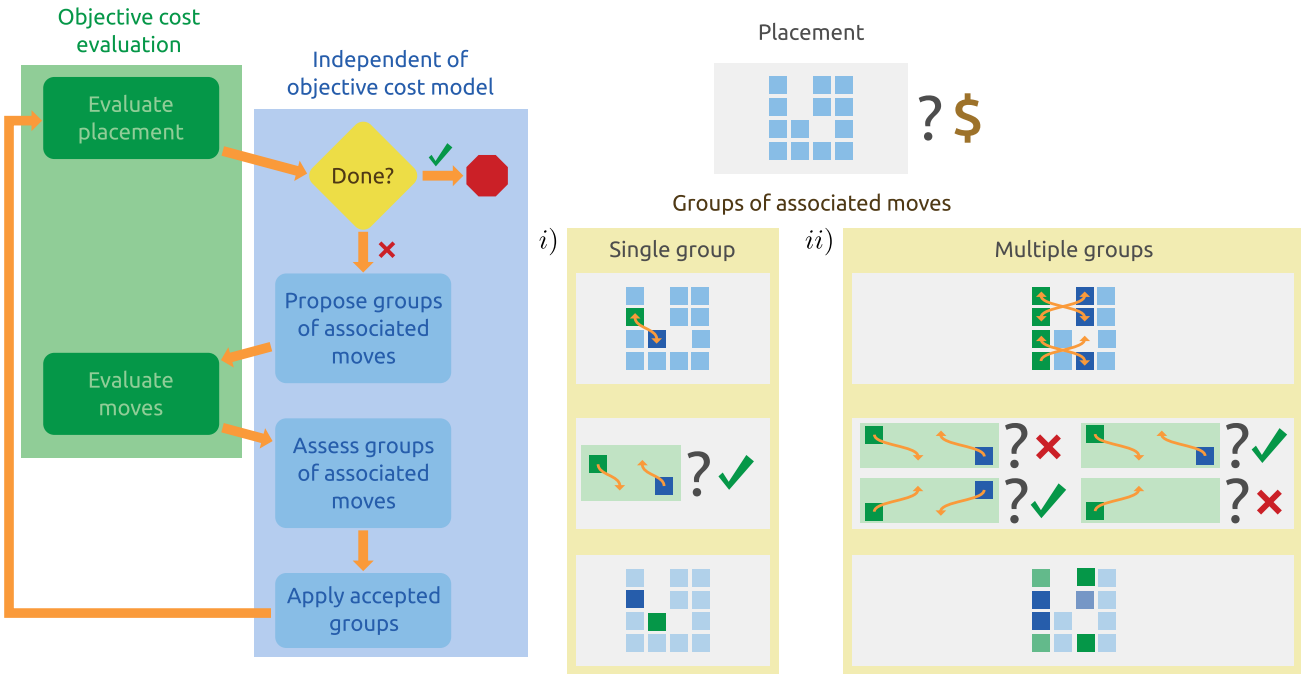


Figure 4.1: Concurrent associated-moves iterative placement *(CAMIP)*

4.1 Overview

Figure 4.1 shows an overview of our proposed *Concurrent Associated-Moves Iterative Placement (CAMIP)* model. Note that the model performs iterative placement using the following steps:

1. Evaluate placement.
2. Stop search if exit criteria is met.
3. Propose groups of associated moves.
4. Evaluate proposed moves individually.
5. Assess estimated cost of each group of associated-moves.
6. Apply moves from accepted groups.

Steps 1-6 are repeated until the exit condition is met in step 2. In this chapter, we discuss steps 2, 3, 5, and 6. In Section 4.1.1, we give our definition of a group of *associated-moves*. In Section 4.1.2 demonstrates how a traditional simulated annealing placer may be emulated using the CAMIP model by considering each swap as a group of associated-moves. In Section 4.1.3, we propose a straight-forward extension of the serial anneal (Section 4.1.2) allowing many

groups of associated-moves to be *concurrently* evaluated and conditionally applied, along with a novel method of efficiently generating very large sets of associated-moves groups.

Note that steps 2, 3, 5, and 6 in Fig. 4.1 are independent of the objective cost used to evaluate either the placement as a whole, or the estimated change in cost due to applying a move to the placement. The next two chapters of this thesis discuss steps 1 and 4 from above, which involve evaluating the cost of the placement and proposed moves. Chapter 5 describes how steps 1 and 4 can be modelled as efficient parallel operations for evaluating a placement, and proposed moves to a placement, using a *wirelength* cost model. Chapter 6 describes a similar approach, but incorporates a *path-based timing-delay* cost model.

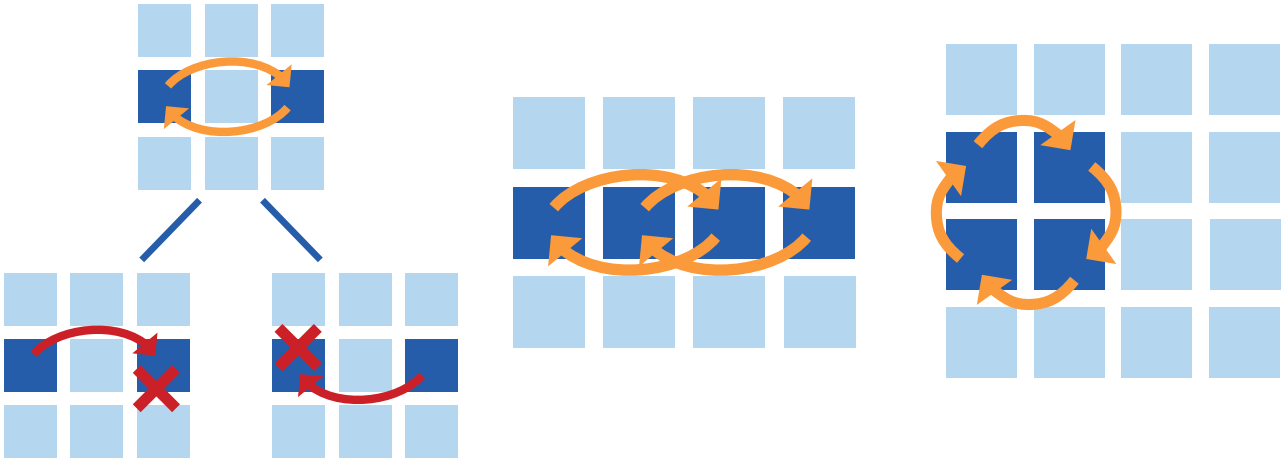
4.1.1 Associated-moves

Within CAMIP, we define a group of associated moves as a set of moves proposed against a placement that must be applied or rejected *together* to maintain a valid placement. For example, in the case of traditional, serial, simulated annealing placement, the two moves involved in swapping the location of two blocks would constitute a group of *associated-moves*, since moving either block without moving the other would result in an invalid placement due to a hard conflict (as shown in Fig. 4.2a). Note that a group of associated moves may consist of any number of moves, so long as the target location of each move overlaps with a source of another move in the set, and failing to apply *all* moves in the set would result in an invalid placement. For example, consider the group of four associated-moves depicted in Fig. 4.2c. In Section 4.1.3, we describe a novel method to efficiently define a very large set of pairs of associated-moves that may be applied without any possibility of a hard-conflict.

4.1.2 Serial anneal as CAMIP

As discussed in Chapter 2, in a traditional, serial, simulated annealing placer, moves are performed either one or two at-a-time. Each move iteration consists of the following steps:

1. Randomly select a *source block* to move.
2. Propose a move by randomly selecting a *target location* within the vicinity of the *source block*.



(a) Pair of associated moves (b) Two pairs of associated moves (c) Four associated moves

Figure 4.2: Example groups of associated moves

3. If the *target location* is occupied by a block propose a *second* move to relocate the *target block* to the current location of the *source block*, resulting in a proposed swap between the locations of the *source* and *target* blocks.
4. Evaluate proposed moves and aggregate combined estimated change in cost due to applying proposed move(s) (at most two moves).
5. Based on the aggregated change in cost, assess whether the move(s) should be applied.
6. If the assessment passes, update the placement according to the move(s).

As shown in Fig. 4.1i , we can emulate the typical annealing move iteration by using steps 1-3 of the move iteration described above to form a *single group* of associated-moves, containing either one or two moves, corresponding to a *swap* in a traditional annealer. The proposed group of associated-moves can then be evaluated using the steps illustrated in the flow chart in Fig. 4.1. While this demonstrates that a traditional annealing placer can be emulated using the CAMIP model, evaluating moves at-most two at-a-time greatly limits the amount of parallelism available. The next section decribes how CAMIP can extend to an arbitrary number of non-overlapping groups of associated-moves.

4.1.3 Concurrent associated-move groups

While a serial annealer may be implemented using the CAMIP model by evaluating a single swap at a time, where each swap corresponds to a single group of associated-moves, the amount of potential parallel work is quite limited. However, the amount of exposed parallelism may be greatly increased by evaluating *multiple* groups of associated-moves *concurrently*. The next section discusses the constraints that must be met when applying multiple groups of associate-moves concurrently to ensure that no hard conflicts occur.

Constraints

As described in Section 4.1.1, we define a group of *associated-moves* such that all moves within a group either must *all* be accepted or must *all* be rejected in order to maintain a valid placement. Applying a partial set of moves from a group of associated-moves results in one or more hard-conflicts. When considering multiple groups of *associated-moves* to apply *concurrently*, there are additional constraints that must be met to ensure no hard-conflicts occur:

- Each *block* may belong to *at most one group* of associated-moves.
- Each *grid-location* may be targetted by *at most one move* across all groups.

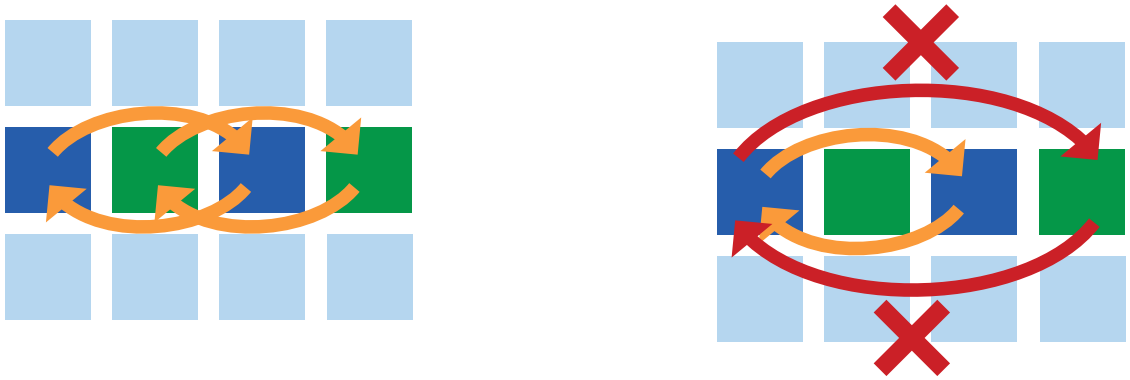
By ensuring the two constraints above are met, no hard-conflicts will occur since:

- Each block is involved in *at most* one move.
- Each grid-location is targetted by *at most* one move.

For example, Fig. 4.3a shows two associated-moves pairs that qualify for concurrent application, since they meet the above criteria. However, the two moves in Fig. 4.3b violate the constraints since the left-most location in the middle row is **a)** the source of more than one move, and **b)** the target of more than one move.

Associated move-pair patterns

Generating moves that meet the constraints outlined above allows for potentially hundreds or thousands of moves to be evaluated concurrently without any risk of hard-conflicts, exposing



(a) Valid

(b) Invalid (conflict)

Figure 4.3: Concurrent associated moves

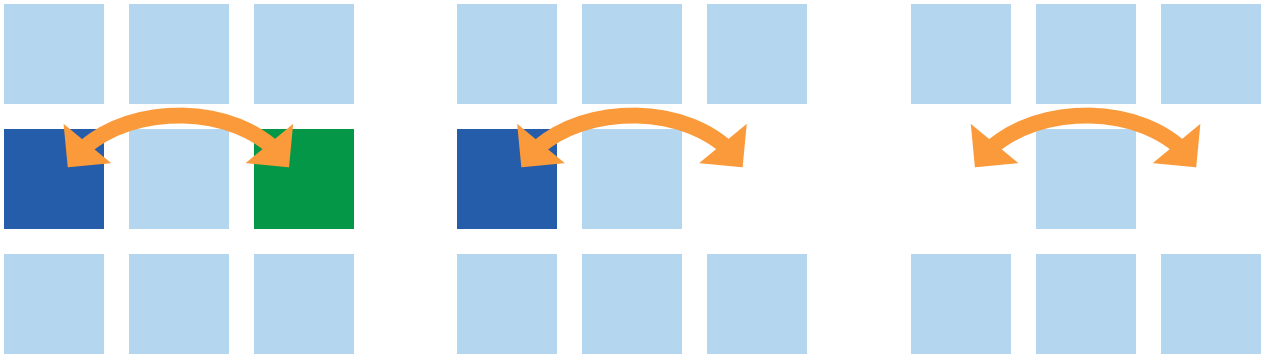
a high level of parallelism. However, in order to improve performance overall, a method for generating groups of associated-moves must *also* provide opportunities for parallelism. In this section, we propose a method for generating *very large sets* of groups of associated-moves, exposing parallelism that scales with the number of locations on the target FPGA.

We propose a method for generating what we refer to as *Associated Move-Pairs (AMPs)*, where each *associated move-pair* corresponds to a pair of complimentary moves between two locations on the target FPGA. In other words, each AMP is analogous to a *swap* in a traditional simulated annealing placer. Since each AMP defines moves between two FPGA locations, we can assign AMPs into one of three categories, based on the number of blocks occupying the locations involved. Each category corresponds to one of the following scenarios:

1. Both locations are occupied by placed blocks.
2. Only one location is occupied by a placed block.
3. Both locations are unoccupied.

Figure 4.4 illustrates categories 1-3. Note that we use a double-ended arrow to represent both moves belonging to an associated move-pair, to reduce clutter.

AMPs belonging to category 1 or category 2 result in one or more blocks being assigned a move to a new position. Note, however, that our approach assigns moves based on locations of the FPGA grid, regardless of whether or not the corresponding locations are occupied. In the



(a) Category 1: Two blocks (b) Category 2: Single block (c) Category 3: No blocks

Figure 4.4: Associated move-pair (AMP) categories

case where a location is unoccupied, the move can be thought of as "moving" the empty space to a new target.

To apply multiple AMPs concurrently, we must meet the constraints laid out on page 71. In our approach, we use geometric *patterns* to define a large set of AMPs which satisify the following objectives:

1. Each location is targeted by at most one block.
2. Each block is involved in at most one move.
3. Moves within each AMP are complimentary (i.e., swap between two locations).

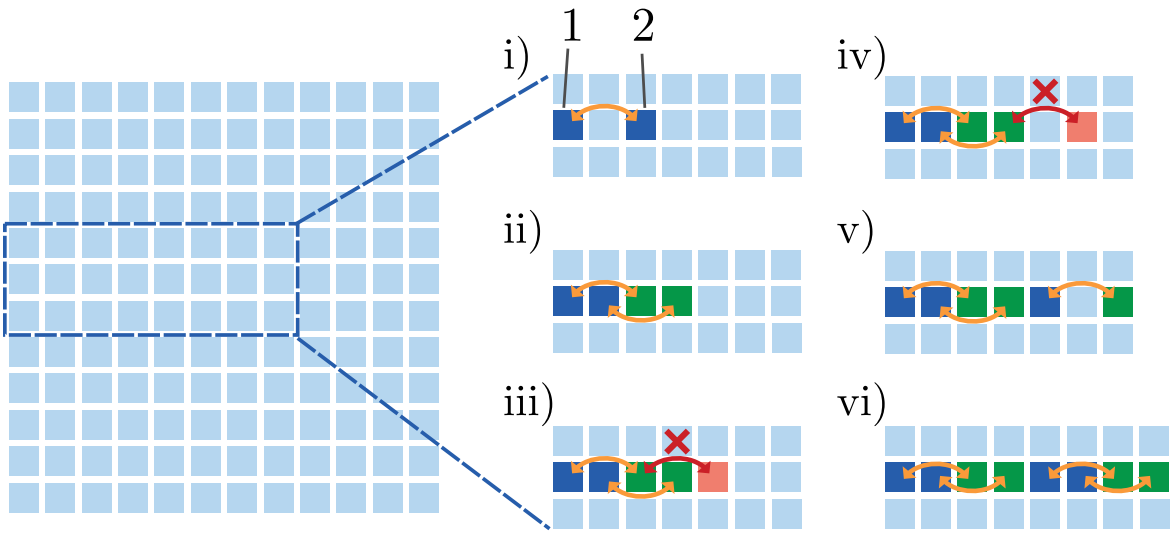
Point 3 ensures that no hard-conflict occurs within each AMP. Given points 1 and 2, it follows that no hard-conflict will occur between any moves between multiple AMPs meeting objectives 1-3. Note that each AMP qualifies as a group of associated-moves.

Our pattern-based approach to generating large sets of AMPs is highly parallel, where for a given pattern, the target position for any block in the netlist can be computed independently and in constant-time. We refer to each pattern defining a set of concurrent AMPs as an *Associated Move-Pair Pattern (AMPP)*.

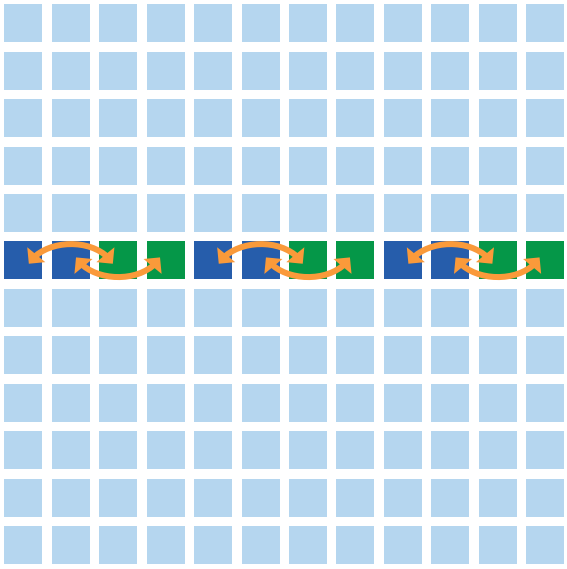
Associated move-pair pattern methodology

To illustrate our associated move-pair pattern methodology, let us first consider a random move that might be proposed by a traditional simulated annealing placer. As shown in Fig. 4.5ai,

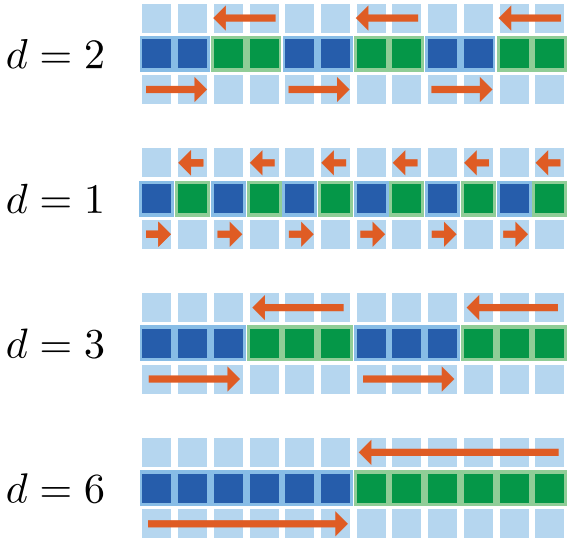
1) we randomly select a block within the FPGA, after which, *2)* we randomly select a target position within the vicinity of the current location of block (two positions to the right of the source position, in this case). The selected block and target corresponds to a move of a block from one location to another in the FPGA. Since the target location is occupied by another block, a complimentary move is proposed to ensure the placement remains valid. Note that the complimentary pair of moves between the source and target location is equivalent to an *Associated Move-Pair (AMP)*, as defined above. Let us refer to the pair of moves denoted by the double-ended arrow in Fig. 4.5ai as AMP 1. Given AMP 1, Fig. 4.5aii shows another *associated move-pair*, which we will refer to as AMP 2, that can be formed by assigning a move of length two to the second block from the left, and a complimentary move to the block occupying the target location. If we continue assigning moves of the same distance along the row we see in Fig. 4.5aiii that we encounter a hard-conflict, since the third block is already represented in AMP 1 as both a *source* and a *target*. A similar scenario is encountered in Fig. 4.5aiv, where the fourth block in the row is already involved in AMP 2. However, if we continue along the row, the fifth block is not involved in any moves, so it can be assigned a move two locations to the right, resulting in a third AMP. Carrying on, we can see in Fig. 4.5avi that a fourth AMP can be constructed in a similar manner. In fact, as shown in Fig. 4.5b, a recurring pattern emerges, where a pattern of AMPs can be constructed by concatenating several interleaved pairs of AMPs. This concept can be applied generally across a row by assigning a contiguous section of locations moves of the same distance and direction, as shown in Fig. 4.5c, where d denotes the distance of the moves in the *associated move-pair pattern (AMPP)* corresponding to each row. The case where $d = 2$ corresponds to the pattern of AMPs in Fig. 4.5b. Figure 4.5c also demonstrates how patterns of equidistant moves may be applied along a row, where contiguous sections of locations are assigned moves in the same direction, where each section is of the same length as the distance of the moves in the row. Note that the sections of moves alternate in direction, and that the patterns repeat along the row, and can be extended as long as necessary until the end of the row is reached.



(a) Forming structure of row pattern ($d = 2$)



(b) Full row pattern ($d = 2$)



(c) Row patterns for varying move distances

Figure 4.5: Associated move-pair patterns (AMPP)

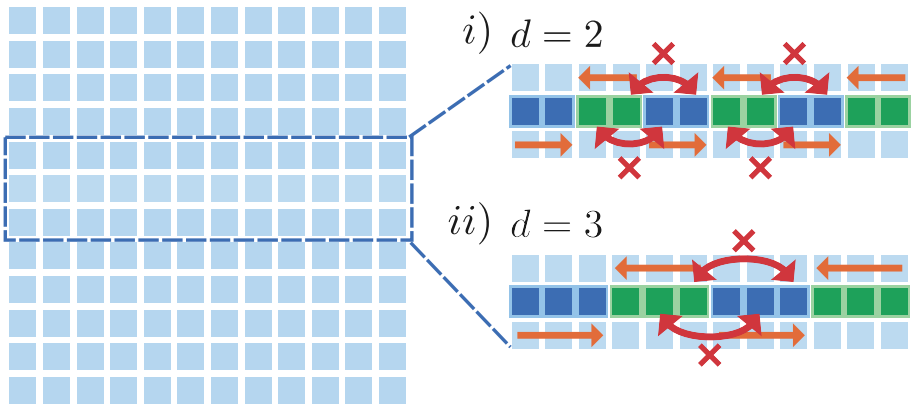


Figure 4.6: All moves are not possible if AMPP always starts at beginning of row.

Pattern shift

As shown in Fig. 4.5b, an *associated move-pair pattern (AMPP)* may be extended to fill an entire row of the FPGA. In the case shown in Fig. 4.5b, the AMPP begins at the left-most position in the row. However, even though we can adjust the distance of proposed moves by adjusting the value of d, as shown in Fig. 4.5c, if AMPPs are always applied starting at the beginning of a row, certain moves will never be proposed. For example, consider the scenarios in Fig. 4.6, where the AMPPs are applied starting at the beginning of the row. Fig. 4.6i, indicates four moves with a distance of 2 that are not proposed by the pattern in this situation. Fig. 4.6ii shows a similar scenario when $d = 3$. To handle this issue, we introduce a *shift* parameter to change the starting position where the AMPP is applied, rather than always starting the AMPP at the beginning of a row. We define the *shift* of an AMPP, which we denote here as s, such that $s = 0$ corresponds to applying an AMPP where the first position in a row is assigned a move to the right, while the the second position begins a contiguous section of moves to the left. For example, consider the AMPP applied to each row in Fig. 4.7a, where $d = 2$ and $s = 0$. The first block in the row is assigned a move two positions *to the right*, while the second position in the row is assigned a move two positions *to the left*. However, since moving the second block in the row two positions to the left would place the block beyond the edge of the FPGA grid, we reject the move and assign *no move* to the second position in each row. The AMPP applied to each row in Fig. 4.7b is equivalent to the AMPP in Fig. 4.7a shifted one position *to the right*. In this case, the first two locations in each row are assigned a move two positions to the right, while the third and fourth location in each row are assigned a move two positions to the left. Note

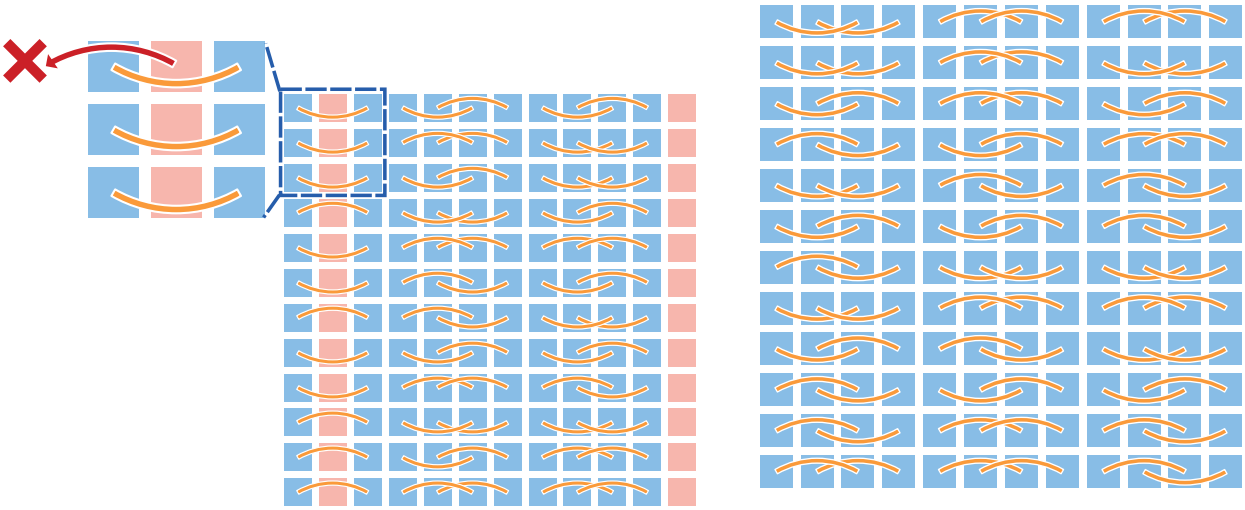
that in Fig. 4.7b, all moves remain within the FPGA grid. Figure 4.7c and Fig. 4.7d show the AMPP from Fig. 4.7a shifted 2 positions to the right and 3 positions to the right, repectively. Note that shifting the AMPP from Fig. 4.7a four positions to the right would result in the same AMPP, so three is the maximum effective shift for an AMPP with a move distance of two. In Section 4.2, we present a general mathematical definition of AMPPs in terms of move distance (d) and pattern shift (s), including valid parameter ranges.

Two-dimensional associated move-pair patterns

Figure 4.7 illustrates how repeated patterns of AMPs with the same move-distance may be interleaved and concatenated to extend across an entire row of the FPGA placement grid, resulting in an AMPP. Moreover, by applying varying *shift* values to an AMPP, all possible moves of the respective distance may be applied along a row. However, in practice, moves are applied in two-dimensions during FPGA placement, i.e., a block may target a new column *and* a new row in a single move. Fortunately, the approach illustrated in Fig. 4.7 can easily be extended to two (or more dimensions), as we will demonstrate in this section.

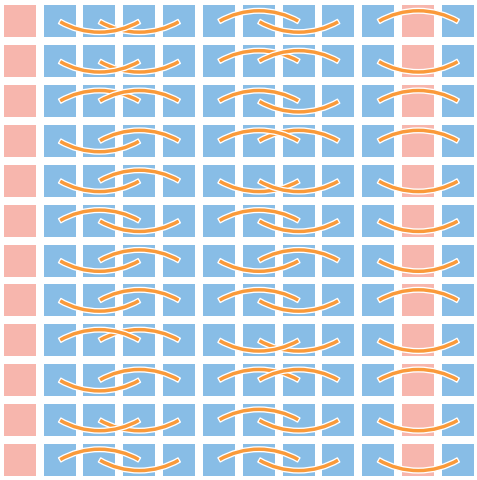
First, consider the AMPP applied to the rows in Fig. 4.8a (the same as in Fig. 4.7c), where the move distance is 2 and the shift is 2. Now, consider a second AMPP, with a move distance of 3 and the shift of 1, applied to each of the *columns* of the FPGA, as depicted in Fig. 4.8b. The *row* AMPP in Fig. 4.8a and the *column* AMPP in Fig. 4.8b may be added together to form a two-dimensional move for each location in the FPGA, as shown in Fig. 4.8c. More details on combining two orthogonal AMPPs are provided in the formal AMPP model, in Section 4.2.

In general, a two-dimensional AMPP may be formed by combining any two orthogonal, one-dimensional AMPPs. In Section 4.2, we present a mathematical definition of AMPPs and describe how the move proposed by any two-dimensional AMPP for any FPGA location may be computed independently, in constant-time. Independent, constant-time move generation for each location leads to a move proposal method that is *a)* highly parallel, and *b)* is compatible with our CAMIP model, described in Section 4.1.

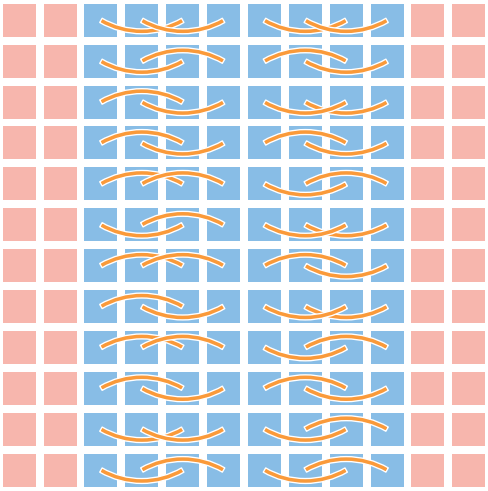


(a) $s = 0$

(b) $s = 1$

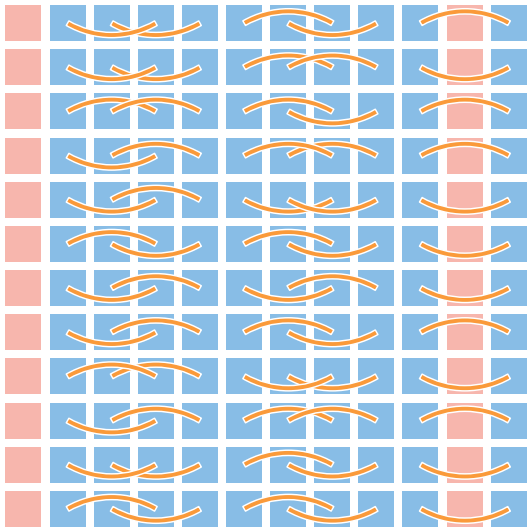


(c) $s = 2$

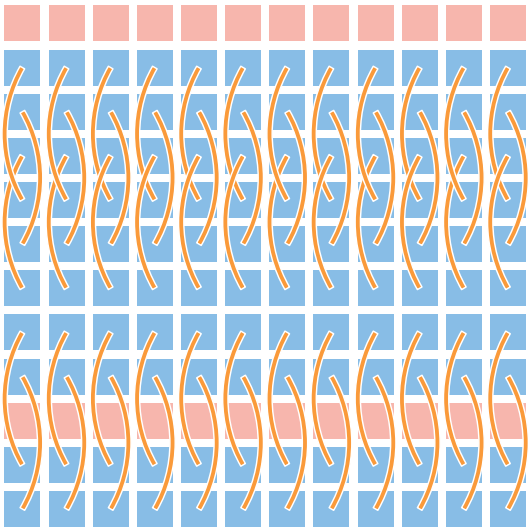


(d) $s = 3$

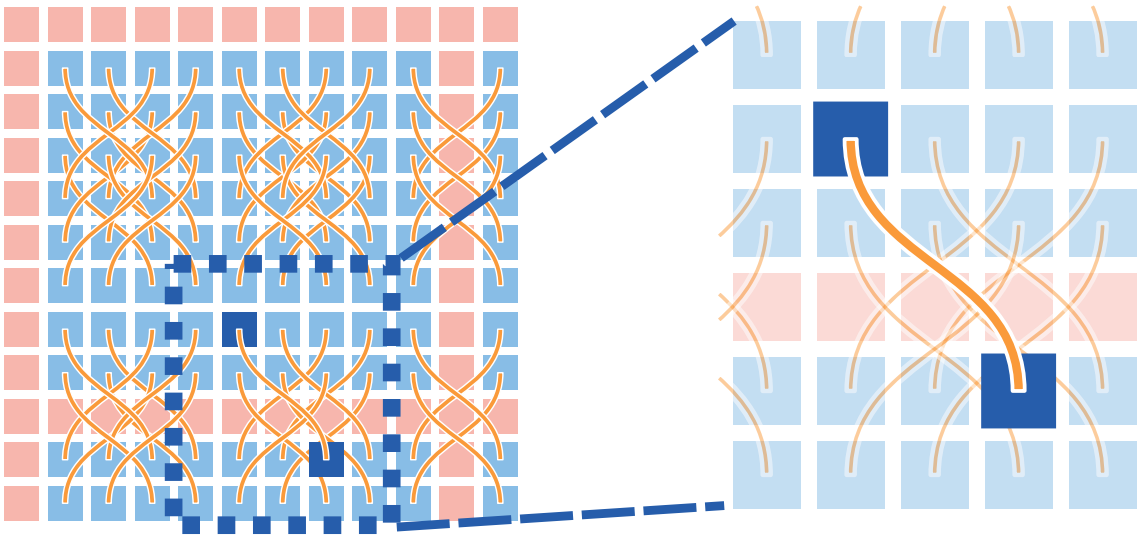
Figure 4.7: Associated move-pair pattern: move-distance of 2, shift 0-3.



(a) Row AMPP: $d = 2$, $s = 2$



(b) Column AMPP: $d = 3$, $s = 1$



(c) Two-dimensional AMPP: $\{\text{row} : d = 2, s = 2\}$, $\{\text{column} : d = 3, s = 1\}$

Figure 4.8: Construct a 2D AMPP from two orthogonal 1D AMPPs.

4.1.4 Input/output blocks

The method that we describe for generating two-dimensional associated move-pair patterns applies to the regular logic block grid area of the FPGA. The blocks surrounding the perimeter of the FPGA may be configured as either inputs or outputs. Since there are significantly fewer inputs and outputs compared to logic cells on a FPGA architecture, we employ a simple mechanism to place the input and output blocks. Using such a simple mechanism seems to work well in practice, likely because the placement is not sensitive to minor deviations in the position of input and output blocks due to their limited number. Rather than generate a two-dimensional associated move-pair pattern for the IO-blocks, we simply consider the perimeter of the FPGA as a one dimensional region of target positions. We then generate AMPPs that are one-dimensional for the IO region of the FPGA. Thus, when applying all move-related operations in the CAMIP method described in this chapter, we simply assign moves based on the AMPP for the appropriate block type. In other words, each logic block is assigned a move based on the current two-dimensional AMPP, and each IO block is assigned a move based on the current IO, one-dimensional AMPP. Note that this mechanism can easily be extended to support any number of block types where a distinct AMPP may be generated for each block type based on the geometry of the positions available as targets for blocks of that type.

4.2 Formal AMPP model

As described in Section 4.1.3, an *Associated Move-Pair Pattern (AMPP)* can be defined by a *move-distance* (d) and a *shift* (s). In this section we define how to compute the move for each resource location in the FPGA using these two parameters.

Similar to the way a traditional simulated annealing placer would randomly propose a *swap*, in our approach, we randomly generate an AMPP for each search iteration. To do so, we use the following procedure:

1. Randomly select a move distance, d_x, for the x-dimension.
2. Randomly select a pattern shift, s_x, for the x-dimension.
3. Randomly select a move distance, d_y, for the y-dimension.

4. Randomly select a pattern shift, s_y, for the y-dimension.

Therefore, only four random-numbers must be generated to define an AMPP, which in turn, defines a move to apply to each location in the FPGA grid.

4.2.1 Random selection of move-distance and pattern-shift

In this section, we formally describe the random selection of parameters used to define an AMPP. Let us denote the length of a *row* in the FPGA grid as ℓ_x, and the length of a *column* as ℓ_y. Furthermore, let us denote a random integer variable uniformly distributed on $[a, b]$ as $U([a,\ b])$.

Starting with the x-dimension, we define a random variable, D_x, as shown in the Eqn. 4.1

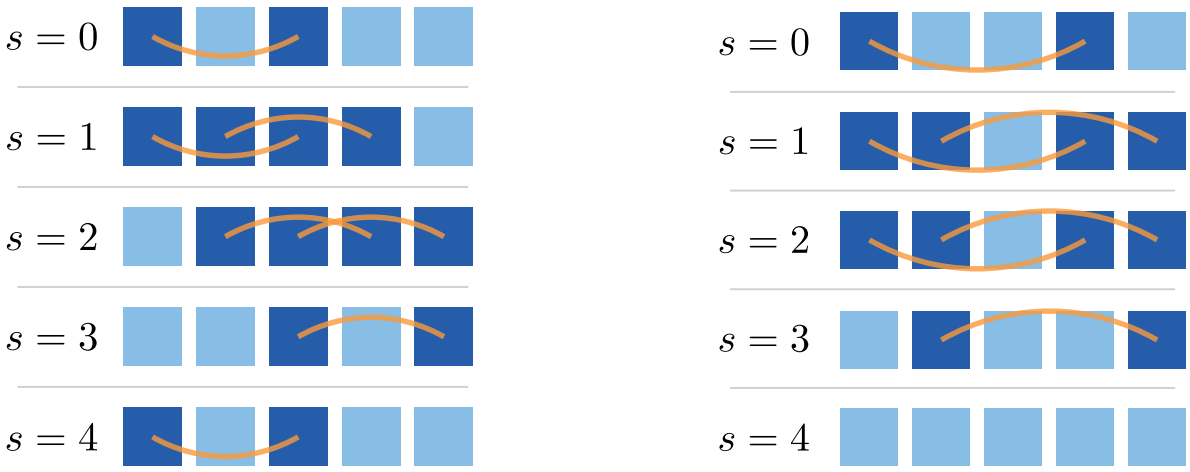
$$D_x \sim U([0,\ \ell_x - 2]) \tag{4.1}$$

The move distance in the x-dimension, d_x, is randomly sampled from the random variable D_x. The limits on the range of D_x ensure that the move distance is less than the length of row.

Given a sampled d_x value, we define another random variable, S_x, as shown in Eqn. 4.2.

$$S_x \sim \begin{cases} U([0,\ 2d_x - 1]), & \text{if } \ \ell_x > 2d_x, \\ U([0,\ \ell_x - 2]), & \text{otherwise} \end{cases} \tag{4.2}$$

A pattern shift value for the x-dimension, which we denote as s_x, is sampled from the random variable S_x. Note that the range of the random variable S_x is dependent on magnitude of the sampled d_x parameter. This ensures that *at least one* location within the row will be assigned a move that targets a location within the FPGA boundaries. For example, consider the row of length 12 shown in Fig. 4.9a. Figure 4.9a depicts an AMPP with $d = 2$ for *shift* values between 0 and 4, inclusive. Note that according to the equation above, since a move distance of 2 is less than half the length of the row in Fig. 4.9a, shift values are restricted to the range $[0, 3]$. Moreover, it is evident that, for a move distance of 2, a shift value of 4 is equivalent to a shift value of 0. In constrast, consider the AMPP shown in Fig. 4.9b, with a move distance of 3. In this case, the move distance is greater than half the row length, so the range of shift values is



(a) $l = 5$, $d = 2$ (b) $l = 5$, $d = 3$

Figure 4.9: Restricted AMPP shift ranges.

restricted to $[0, 3]$. As shown in Fig. 4.9b, selecting a shift value outside of this range causes all moves to target locations beyond the edges of the FPGA row.

Parameters are selected for the y-dimension in a similar process to the x-dimension. Specifically, we define a random variable, D_y, as shown in Eqn. 4.3

$$D_y \sim U([0,\ \ell_y - 2]) \tag{4.3}$$

The move distance in the y-dimension, d_y, is randomly sampled from the random variable D_y. Given a value for d_y, we define another random variable, S_y, as shown in Eqn. 4.4.

$$S_y \sim \begin{cases} U([0,\ 2d_y - 1]), & \text{if } \ \ell_y > 2d_y, \\ U([0,\ \ell_y - 2]), & \text{otherwise} \end{cases} \tag{4.4}$$

As in the case of the x-dimension, a pattern-shift value for the y-dimension, which we denote as s_y, is sampled from the random variable S_y. As we will show in the next section, the randomly sampled values d_x, s_x, d_y, and s_y, are sufficient to define a two-dimensional AMPP that assigns a move to every location in the FPGA grid.

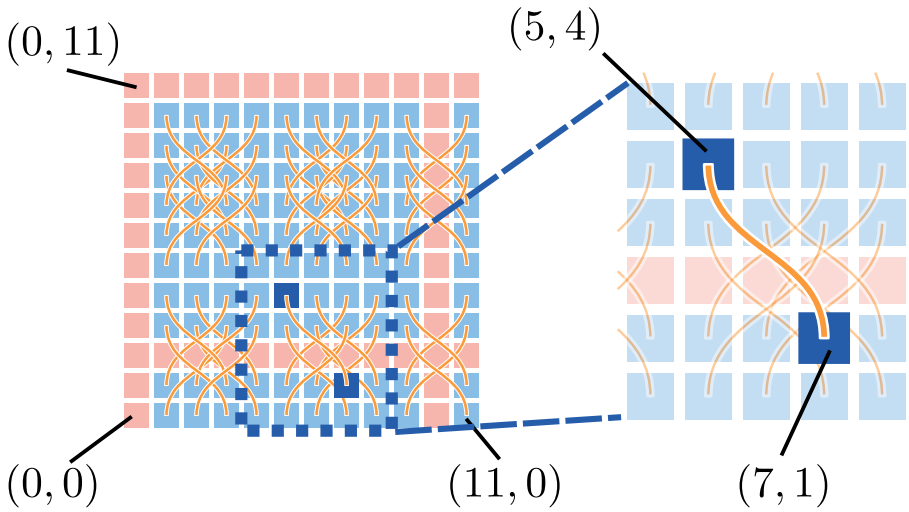


Figure 4.10: Two-dimensional AMPP: $\{d_x = 2, s_x = 2\}$, $\{d_y = 3, s_y = 1\}$

4.2.2 Proposed move based on pattern definition

Given the parameters d_x and s_x, we define a function λ_x as follows:

$$\lambda_x(x) = \begin{cases} d_x, & \text{if } \; d_x > (x + 2d_x - (s_x + d_x + 1)(mod\ 2d_x))\,(mod\ 2d_x), \\ -d_x, & \text{otherwise} \end{cases} \tag{4.5}$$

We define a similar function, λ_y, based on the the parameters d_y and s_y:

$$\lambda_y(y) = \begin{cases} d_y, & \text{if } \; d_y > (y + 2d_y - (s_y + d_y + 1)(mod\ 2d_y))\,(mod\ 2d_y), \\ -d_y, & \text{otherwise} \end{cases} \tag{4.6}$$

Using the functions $\lambda_x(x)$ and $\lambda_y(y)$, from Eq. 4.5 and Eq. 4.6, respectively, we may compute a move to be applied to each (x, y) location in the FPGA. For example, consider the example placement and AMPP depicted in Fig. 4.9. Note that in the example, the x-dimension maps to the rows of the FPGA, while the y-dimension maps to the columns of the FPGA.

Consider the location $(5, 4)$ (i.e., $x = 5, y = 4$) in Fig. 4.9. We can compute $\lambda_x(5)$ (i.e, the relative move to apply in the x-dimension) as shown in Eq. 4.7 and Eq. 4.8.

$$\lambda_x(5) = \begin{cases} d_x, & \text{if } \ d_x > (x + 2d_x - (s_x + d_x + 1)(mod\ 2d_x))\,(mod\ 2d_x), \\ -d_x, & \text{otherwise} \end{cases}$$
$$= \begin{cases} 2, & \text{if } \ LHS > RHS, \\ -2, & \text{otherwise} \end{cases} \tag{4.7}$$

$$\begin{aligned} \text{LHS} &= d_x \\ &= 2 \\ \text{RHS} &= (x + 2d_x - (s_x + d_x + 1)(mod\ 2d_x))\,(mod\ 2d_x) \\ &= (5 + 2(2) - (2 + 2 + 1)(mod\ 2(2)))\,(mod\ 2(2)) \\ &= (5 + 4 - 5(mod\ 4))\,(mod\ 4) \\ &= (9 - 1)\,(mod\ 4), \\ &= 0 \\ LHS &> RHS \\ \therefore \lambda_x(5) &= 2 \end{aligned} \tag{4.8}$$

Similarly, we can compute $\lambda_y(4)$ (i.e, the relative move to apply in the y-dimension) as shown in Eqn. 4.9 and Eqn. 4.10.

$$\lambda_y(4) = \begin{cases} d_y, & \text{if } \ d_y > (y + 2d_y - (s_y + d_y + 1)(mod\ 2d_y))\,(mod\ 2d_y), \\ -d_y, & \text{otherwise} \end{cases}$$
$$= \begin{cases} 3, & \text{if } \ LHS > RHS, \\ -3, & \text{otherwise} \end{cases} \tag{4.9}$$

$$
\begin{aligned}
\text{LHS} &= d_y \\
&= 3 \\
\text{RHS} &= (y + 2d_y - (s_y + d_y + 1)(mod\ 2d_y))\,(mod\ 2d_y) \\
&= (4 + 2(3) - (1 + 3 + 1)(mod\ 2(3)))\,(mod\ 2(3)) \\
&= (4 + 6 - 5(mod\ 6))\,(mod\ 6) \\
&= (10 - 5)\,(mod\ 6), \\
&= 5 \\
LHS &< RHS \\
\therefore \lambda_y(4) &= -3
\end{aligned}
\tag{4.10}
$$

Now, consider location $(7, 1)$ (i.e., $x = 7, y = 1$), which is the *target* location of the move from $(5, 4)$. We can compute $\lambda_x(7)$ as follows:

$$
\begin{aligned}
\lambda_x(7) &= \begin{cases} d_x, & \text{if } \ d_x > (x + 2d_x - (s_x + d_x + 1)(mod\ 2d_x))\,(mod\ 2d_x), \\ -d_x, & \text{otherwise} \end{cases} \\
&= \begin{cases} 2, & \text{if } \ LHS > RHS, \\ -2, & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.11}
$$

$$
\begin{aligned}
\text{LHS} &= d_x \\
&= 2 \\
\text{RHS} &= (x + 2d_x - (s_x + d_x + 1)(mod\ 2d_x))\,(mod\ 2d_x) \\
&= (7 + 2(2) - (2 + 2 + 1)(mod\ 2(2)))\,(mod\ 2(2)) \\
&= (7 + 4 - 5(mod\ 4))\,(mod\ 4) \\
&= (11 - 1)\,(mod\ 4), \\
&= 2 \\
LHS &= RHS \\
\therefore \lambda_x(7) &= -2
\end{aligned}
\tag{4.12}
$$

Similarly, for $\lambda_y(1)$:

$$
\begin{aligned}
\lambda_y(1) &= \begin{cases} d_y, & \text{if } \; d_y > (y + 2d_y - (s_y + d_y + 1)(mod\ 2d_y))\,(mod\ 2d_y), \\ -d_y, & \text{otherwise} \end{cases} \\
&= \begin{cases} 3, & \text{if } \; LHS > RHS, \\ -3, & \text{otherwise} \end{cases} \\
\text{LHS} &= d_y \\
&= 3 \\
\text{RHS} &= (y + 2d_y - (s_y + d_y + 1)(mod\ 2d_y))\,(mod\ 2d_y) \\
&= (1 + 2(3) - (1 + 3 + 1)(mod\ 2(3)))\,(mod\ 2(3)) \\
&= (1 + 6 - 5(mod\ 6))\,(mod\ 6) \\
&= (7 - 5)\,(mod\ 6), \\
&= 2 \\
LHS &> RHS \\
\therefore \lambda_y(1) &= 3
\end{aligned}
\tag{4.13}
$$

Note that the moves for locations $(5, 4)$ and $(7, 1)$ are complimentary, and form a group of associated moves, analagous to a *swap* in a traditional annealing placer. More specifically, the moves for locations $(5, 4)$ and $(7, 1)$ form an *Associated Move-Pair (AMP)*. In general, each move proposed by an AMPP forms half of an AMP.

Compute moves in parallel

As shown on pages 83-86, for a given (x, y) location, we can compute $\lambda_x(x)$ and $\lambda_y(y)$ to produce a move with a runtime complexity of $O(1)$. Moreover, the move for any location can be computed independently. It follows that we can use the *map* pattern described in Chapter 2 to concurrently compute the moves for all blocks in a placement, based on the current location of each block. Furthermore, only four random integers (corresponding to d_x, s_x, d_y, and s_y) must be generated to define an AMPP to cover a two-dimensional grid, *regardless of the size of the grid*. Therefore, our proposed method of randomly generating AMPPs is *highly* parallel and is ideally suited for use in our proposed CAMIP model.

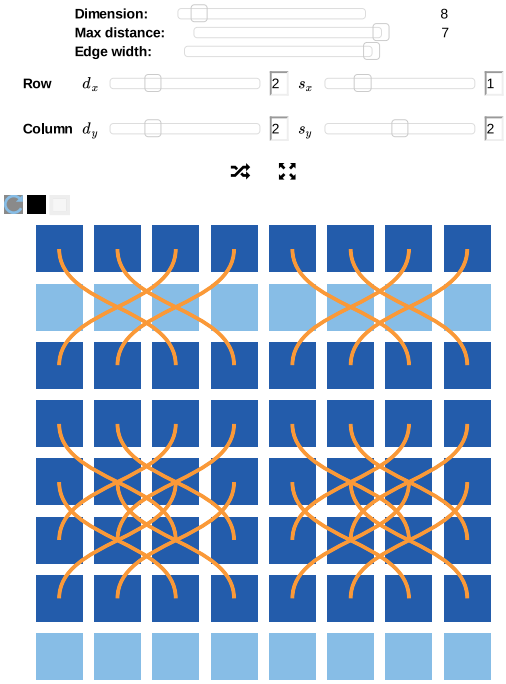


Figure 4.11: Interactive AMPP demo interface.

Interactive demonstration

To assist in understanding our proposed AMPP generation method, we have developed an interactive demonstration of AMPP generation. Figure 4.11 shows the user interface. AMPPs may be generated based on explicitly specified move distance and pattern shift values, or may be generated randomly. The tool provides widgets to adjust the parameters used to construct the displayed two-dimensional associated-move-pairs pattern, including the distance (i.e., d_x/d_y) and the shift (i.e., s_x/s_y). The interactive tool is available online at:
`http://cfobel.github.io/ampp-gui/www/ampp.html`.

4.3 Formal CAMIP model

In this section, we present a mathematical model for performing the following three steps of the CAMIP process:

1. Propose groups of associated moves.

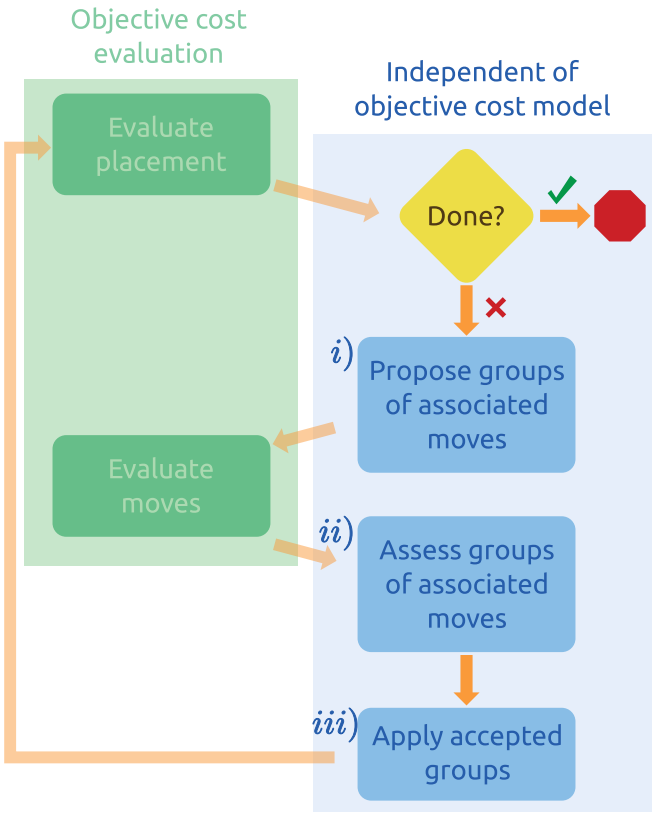


Figure 4.12: Concurrent associated-moves iterative placement *(CAMIP)*

2. Assess groups of associated moves.

3. Apply accepted groups of moves.

These steps correspond to the steps labelled *i*, *ii*, and *iii*, in Fig. 4.12. First we present a representation of the current placement (i.e., the position of each block), followed by a description of each of the three steps listed above. Recall that the steps in the flow chart from Fig. 4.12 that evaluate the placement and individual moves with respect the objective cost will be covered in the next two chapters.

4.3.1 Placement representation

A placement corresponds to the mapping of each block in a netlist to a specific location on the target FPGA. In our CAMIP model, we define the vectors $\vec{p_x}$ and $\vec{p_y}$, in Eqn. 4.14 to store the

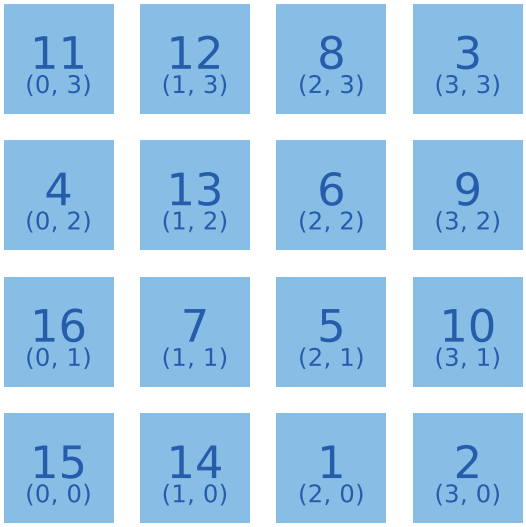


Figure 4.13: Example placement of 16 blocks

location assigned to each block, where p_{xi} and p_{yi} denote the location index of block i in the x-dimension and y-dimension, respectively.

$$\begin{aligned} \vec{p_x} &\in \left(\mathbb{Z}_0^+\right)^m,\ \vec{p_x} = \left(p_{x1},\ \ p_{x2},\ \ p_{x3},\ \ \ldots,\ \ p_{xi},\ \ \ldots,\ \ p_{xm}\right) \\ \vec{p_y} &\in \left(\mathbb{Z}_0^+\right)^m,\ \vec{p_y} = \left(p_{y1},\ \ p_{y2},\ \ p_{y3},\ \ \ldots,\ \ p_{yi},\ \ \ldots,\ \ p_{ym}\right) \end{aligned} \tag{4.14}$$

For example, consider the placement in Fig. 4.13, where each block is labelled with the corresponding block index and (x, y) position. The placement in Fig. 4.13 may be represented by the two following vectors:

$$\begin{aligned} \vec{p_x} &= \begin{pmatrix} 2, & 3, & 3, & 0, & 2, & 2, & 1, & 2, & 3, & 3, \\ 0, & 1, & 1, & 1, & 0, & 0 \end{pmatrix} \\ \vec{p_y} &= \begin{pmatrix} 0, & 0, & 3, & 2, & 1, & 2, & 1, & 3, & 2, & 1, \\ 3, & 3, & 2, & 0, & 0, & 1 \end{pmatrix} \end{aligned} \tag{4.15}$$

4.3.2 Propose groups of associated moves

In our CAMIP model, we propose moves to apply to a placement in terms of groups of *associated-moves*. These groups of associated-moves may be represented as *a)* a pair of vectors corresponding to a new (x, y) position for each block (which may be the same as the current position), along with *b)* a set containing a sub-set for each group of associated-moves.

Proposed position vectors

For the new block positions, we define the vectors $\vec{p'_x}$ and $\vec{p'_y}$, shown below:

$$\begin{aligned}
\vec{p'_x} \in \left(\mathbb{Z}_0^+\right)^m,\ \vec{p'_x} &= \vec{p_x} + \vec{d_x} \\
&= \left(p_{x1} + d_{x1},\ \ p_{x2} + d_{x2},\ \ \ldots,\ \ p_{xm} + d_{xm}\right) \\
&= \left(p'_{x1},\ \ p'_{x2},\ \ \ldots,\ \ p'_{xi},\ \ \ldots,\ \ p'_{xm}\right) \\
\vec{p'_y} \in \left(\mathbb{Z}_0^+\right)^m,\ \vec{p'_y} &= \vec{p_y} + \vec{d_y} \\
&= \left(p_{y1} + d_{y1},\ \ p_{y2} + d_{y2},\ \ \ldots,\ \ p_{ym} + d_{ym}\right) \\
&= \left(p'_{y1},\ \ p'_{y2},\ \ \ldots,\ \ p'_{yi},\ \ \ldots,\ \ p'_{ym}\right)
\end{aligned} \tag{4.16}$$

where m is the number of blocks in the placement, p'_{xi} and p'_{yi} denote the new proposed location index for block i in the x-dimension and y-dimension, respectively, and d_{xi} and d_{yi} denote the proposed move for block i in each dimension. When generating proposed moves using an AMPP, the new position for block i can be computed as shown in the equation below.

$$\begin{aligned}
p'_{xi} &= \begin{cases} p_{xi} + \lambda\left(p_{xi}\right), & \text{if } \ 0 \le p_{xi} + \lambda\left(p_{xi}\right) < \ell_x \\ 0, & \text{otherwise} \end{cases} \\
p'_{yi} &= \begin{cases} p_{yi} + \lambda\left(p_{yi}\right), & \text{if } \ 0 \le p_{yi} + \lambda\left(p_{yi}\right) < \ell_y \\ 0, & \text{otherwise} \end{cases}
\end{aligned} \tag{4.17}$$

Note the condition that assigns a move of zero in each dimension if the corresponding move would otherwise target a position that is outside the FPGA grid. This ensures that only moves that target positions within the FPGA grid will be considered.

For example, consider the placement from Fig. 4.13 with an AMPP applied, as shown in Fig. 4.14, where $d_x = 2$, $d_y = 1$, and there is no shift in either dimension. The vectors $\vec{p'_x}$ and $\vec{p'_x}$ can be computed as follows:

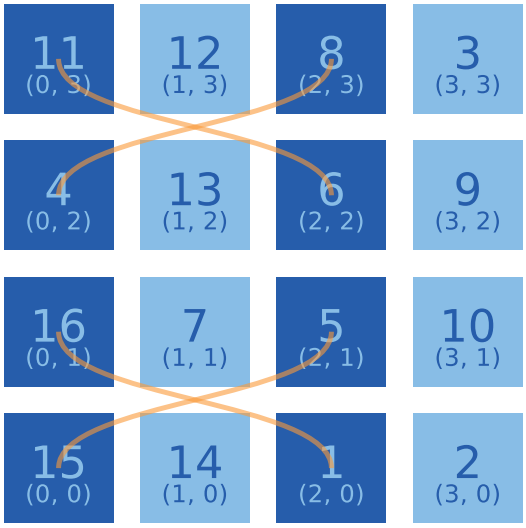


Figure 4.14: Groups of associated-moves proposed by AMPP ($d_x = 2$, $d_y = 1$)

$$
\begin{aligned}
\vec{p_x} &= \big(2,\ 3,\ 3,\ 0,\ 2,\ 2,\ 1,\ 2,\\
&\qquad 3,\ 3,\ 0,\ 1,\ 1,\ 1,\ 0,\ 0\big)\\
\vec{p_y} &= \big(0,\ 0,\ 3,\ 2,\ 1,\ 2,\ 1,\ 3,\\
&\qquad 2,\ 1,\ 3,\ 3,\ 2,\ 0,\ 0,\ 1\big)\\
\vec{p'_x} &= \big(p_{x1} + d_{x1},\ \ p_{x2} + d_{x2},\ \ \ldots,\ \ p_{x16} + d_{x16}\big)\\
&= \big(2-2,\ 3,\ 3,\ 0+2,\ 2-2,\ 2-2,\ 1,\ 2-2,\\
&\qquad 3,\ 3,\ 0+2,\ 1,\ 1,\ 1,\ 0+2,\ 0+2\big)\\
&= \big(0,\ 3,\ 3,\ 2,\ 0,\ 0,\ 1,\ 0,\\
&\qquad 3,\ 3,\ 2,\ 1,\ 1,\ 1,\ 2,\ 2\big)\\
\vec{p'_y} &= \big(p_{y1} + d_{y1},\ \ p_{y2} + d_{y2},\ \ \ldots,\ \ p_{y16} + d_{y16}\big)\\
&= \big(0+1,\ 0,\ 3,\ 2+1,\ 1-1,\ 2+1,\ 1,\ 3-1,\\
&\qquad 2,\ 1,\ 3-1,\ 3,\ 2,\ 0,\ 0+1,\ 1-1\big)\\
&= \big(1,\ 0,\ 3,\ 3,\ 0,\ 3,\ 1,\ 2,\\
&\qquad 2,\ 1,\ 2,\ 3,\ 2,\ 0,\ 1,\ 0\big)
\end{aligned}
\tag{4.18}
$$

Note that any move that would target a location outside the FPGA grid is rejected, i.e., the position of the corresponding block is unchanged. For example, block 12 was originally assigned

a move two positions to the left (and one position down) based on the AMPP parameters. However, since that move would place block 12 outside the FPGA grid, block 12 is assigned the same position, $(1, 3)$, for the current CAMIP iteration.

Move groups

While the vectors $\vec{p'_x}$ and $\vec{p'_y}$ define the new positions based on the proposed moves, we must also define the groups of moves that are associated with one another, i.e., which moves must be applied together to maintain a valid placement. For each group of *associated-moves*, we define a set of block indexes, s_k, corresponding to the blocks moved within the group. In the case where moves are proposed using an AMPP, each group of associated-moves s_k corresponds to an *Associated Move-Pair (AMP)*, where s_k contains either one or two blocks, depending on whether the AMP is category 1 or category 2, as defined in Section 4.1.3. We denote the total number of associated-move sets as q. For example, the AMPs in Fig. 4.14 correspond to the following set definitions (in Eqn. 4.19), implying that each group of two blocks must be moved *(or not moved)* in tandem to maintain a valid placement.

$$\begin{aligned} s_1 &= \{1, 16\} \\ s_2 &= \{4, 8\} \\ s_3 &= \{5, 15\} \\ s_4 &= \{6, 11\} \end{aligned} \tag{4.19}$$

4.3.3 Assess groups of associated moves

After moves have been proposed, as shown in Fig. 4.1 and Fig. 4.12, each move is evaluated to determine the estimated difference in cost due to applying the move, assuming it is the only move taking place. In this section, we assume that move evaluation has been completed using an objective cost, such as *wirelength* (see Chapter 5) or *timing-delay* (see Chapter 6). Note that operations described in this chapter are independent of the objective cost used, and that objective cost evaluation *does not need to take the association of moves into account.* **The only objective cost requirements for the CAMIP model are that *a)* a cost can be assigned**

to each proposed move, and *b)* a cost can be assigned to the overall placement to determine when to terminate the search.

Proposed associated-moves sets costs

We define the vector $\vec{\delta_b} \in \mathbb{R}^m$, such that δ_{bi} corresponds to the estimated difference in cost due to applying the move assigned to block i. Recall that m corresponds to the number of blocks in the placement, and that a block may be assigned a "zero" move, which effectively leaves the block in its current position. In this section, we assume that $\vec{\delta_b}$ is available, computed using an objective cost model such as the ones described in Chapter 5 and Chapter 6.

The moves within each set of associated-moves, s_k, must be applied or rejected in tandem. Thus, to assess each set s_k, we compute the combined estimated difference in cost due to moving all blocks in the set. Let us define the matrix G, as shown in Eqn. 4.20 . Note that each row in G corresponds to a block in the placement, while each column corresponds to a set of associated moves.

$$G \in \mathbb{M}_{mq}$$

$$G = \begin{pmatrix} g_{11} & g_{12} & g_{13} & \cdots & \cdots & g_{1k} \\ g_{21} & g_{22} & g_{23} & \cdots & \cdots & g_{2k} \\ \vdots & & \ddots & & & \\ \vdots & & & g_{ik} & & \\ \vdots & & & & \ddots & \\ g_{m1} & g_{m2} & g_{m3} & \cdots & \cdots & g_{mq} \end{pmatrix} \quad g_{ik} = \begin{cases} \delta_{bi}, & \text{if } i \in g_k \\ 0, & \text{otherwise} \end{cases} \tag{4.20}$$

Each entry $g_{ik} \in G$, holds the cost of moving block i if block i is in set s_k.

Consider the AMPP moves shown in Fig. 4.14. Let us assume that the move applied to each of the blocks is evaluated according to some objective cost (e.g., wirelength of connected nets) to compute an estimated difference in cost. In other words, let us assume that $\vec{\delta_b}$ is available, as defined below in Eqn. 4.21, where each element δ_{bi} corresponds to the difference in cost due to applying the proposed move for block i.

$$\begin{aligned}\vec{\delta_b} &= \begin{pmatrix}\delta_{b1}, & \delta_{b2}, & \ldots, & \delta_{bm}\end{pmatrix} \\ &= \begin{pmatrix}-2, & 0, & 0, & 1, & -1, & -1, & 0, & -3, \\ 0, & 0, & -4, & 0, & 0, & 0, & 3, & 1\end{pmatrix}\end{aligned} \tag{4.21}$$

Using the definition given above, the corresponding G matrix is shown below, in Eqn. 4.22. Note that we replace all zero elelements with "–" to accentuate the non-zero elements.

$$G = \begin{pmatrix}-2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0\end{pmatrix} = \begin{pmatrix}-2 & - & - & - \\ - & - & - & - \\ - & - & - & - \\ - & 1 & - & - \\ - & - & -1 & - \\ - & - & - & -1 \\ - & - & - & - \\ - & -3 & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & - & -4 \\ - & - & - & - \\ - & - & - & - \\ - & - & - & - \\ - & - & 3 & - \\ 1 & - & - & -\end{pmatrix} \tag{4.22}$$

Next, we define the vector $\vec{\delta_g}$, such that δ_{gk} corresponds to the estimated difference in cost corresponding to each set s_k, where $\vec{\delta_g}$ may be computed as shown in Eqn. 4.23 below.

$$\begin{aligned}\vec{\delta_g} &\in \mathbb{R}^q \\ \vec{\delta_g} &= \vec{1}_n G \\ &= \begin{pmatrix}\delta_{g1}, & \delta_{g2}, & \ldots, & \delta_{gq}\end{pmatrix} \\ \delta_{gk} &= \sum_i g_{ik}\end{aligned} \tag{4.23}$$

Continuing the example from Fig. 4.14, the resulting $\vec{\delta_g}$ vector is shown in Eqn. 4.24 below, where each element corresponds to the sum of values in the respective column of G.

$$\vec{\delta_g} = \begin{pmatrix}-1, & -2, & 2, & -5\end{pmatrix} \tag{4.24}$$

Note that G can be formed in parallel using the *map* pattern and $\vec{\delta_g}$ (i.e., the sum of each column of G) can be computed efficiently in parallel by applying a category-reduction on a sparse-matrix representation of the the G matrix (see Chapter 2).

Assess combined cost of each associated-moves set

Based on the estimated cost differences in $\vec{\delta_g}$, we determine whether or not to move the blocks in each set s_k based on the same annealing schedule used in [4]. We define the vector $\vec{a}$, as shown in Eqn. 4.25, such that each element, a_k, represents whether or not the moves corresponding to set s_k should be applied. For each set s_k where a_k is equal to one, *all* blocks in set s_k must be moved. Otherwise, *none* of the moves are applied. Note that in the equation below, u_k denotes a sampling from the random variable $U([0,1])$, where each set s_k is assigned an independent value on each search iteration. To reiterate, since, by definition, the moves corresponding to each set s_k are associated with one another, the blocks within each set *must* be moved in an *all or nothing* manner to avoid hard-conflicts.

$$\begin{aligned} \vec{a} &= \begin{pmatrix} a_1, & a_2, & \ldots, & a_k, & \ldots & a_q \end{pmatrix} \\ a_k &= \begin{cases} 1, & \text{if } \delta_{gk} \leq 0 \text{ or } u_k < e^{-\frac{\delta_{gk}}{t}}, \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{4.25}$$

Given the example $\vec{\delta_g}$ above for the AMPP in Fig. 4.14, let us assume that $t \sim 0^+$, such that $u_k < e^{-\frac{\delta_{gk}}{t}}$ will essentially always be false, since $\lim_{t \to 0^+} \left(e^{-\frac{\delta_{gk}}{t}} \right) = 0$. This scenario is typical near the end of a simulated annealing run, where assessment is essentially greedy, only accepting reductions in cost. In this case, the $\vec{\delta_g}$ from Eqn. 4.24 would result in the $\vec{a}$ vector in Eqn. 4.26 below, indicating that moves associated with sets s_1, s_2, and s_4 must be applied.

$$\vec{a} \ = \begin{pmatrix} 1, & 1, & 0, & 1 \end{pmatrix} \tag{4.26}$$

Note that $\vec{a}$ can be computed in parallel using the *map* pattern to the assess each entry in the $\vec{\delta_g}$ vector concurrently.

4.3.4 Apply accepted groups of associated moves

The final step in each iteration of the CAMIP model entails updating the position of each block in the placement based on the moves accepted during the current iteration. The position of each block i can be updated as shown in the equation below, where $p_{xi}^{(+1)}$ denotes the new x-dimension position for block i (which may be the same as p_{xi}), and $p_{yi}^{(+1)}$ denotes the new y-dimension position for block i.

$$p_{xi}^{(+1)} = \begin{cases} p'_{xi}, & \text{if } \exists\, k : i \in s_k \text{ and } a_k = 1, \\ p_{xi}, & \text{otherwise} \end{cases} \tag{4.27}$$

$$p_{yi}^{(+1)} = \begin{cases} p'_{yi}, & \text{if } \exists\, k : i \in s_k \text{ and } a_k = 1, \\ p_{yi}, & \text{otherwise} \end{cases} \tag{4.28}$$

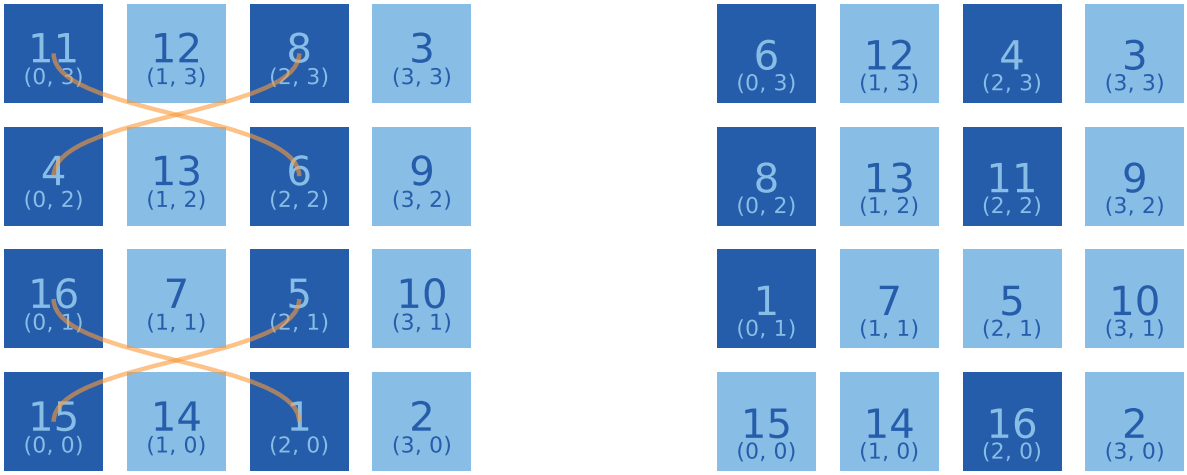
To illustrate the application of accepted moves resulting from the example in Fig. 4.14, here we review the structures involved. First, based on the AMPP from Fig. 4.14, we define four sets in Eqn. 4.19, where each set s_k consists of indexes of blocks involved in associated-moves of an AMP. The set definitions are repeated below, in Eqn. 4.29, for easy reference.

$$s_1 = \{1, 16\} \quad s_2 = \{4, 8\} \quad s_3 = \{5, 15\} \quad s_4 = \{6, 11\} \tag{4.29}$$

Second, given $\vec{\delta_b}$ from Eqn. 4.21 above, containing the evaluated difference in cost due to applying the proposed move for each block in the placement, the matrix G is constructed. By taking the sum of each column in G, we obtain the vector $\vec{\delta_g} \in \mathbb{R}^q$ (Eqn. 4.24), where q is the number of sets of associated moves ($q = 4$ in the example from Fig. 4.14) and each element δ_{gk} corresponds to the total difference in cost due to applying all proposed moves for the blocks in set s_k. The vector $\vec{\delta_g}$ is repeated in Eqn. 4.30 for easy reference.

$$\vec{\delta_g} = \begin{pmatrix} -1, & -2, & 2, & -5 \end{pmatrix} \tag{4.30}$$

Based on the difference in cost computed for each set s_k, we apply an assessment to construct the vector $\vec{a}$, as shown below in Eqn. 4.31 (repeated from Eqn. 4.26), indicating that any block belonging to sets s_1, s_2, and s_4 must be moved to its proposed new position.



(a) AMPP $d_x = 2$, $d_y = 1$.

(b) Blocks $1, 16, 4, 8, 6, 11$ moved.

Figure 4.15: Application of *accepted* associated-moves.

$$\vec{a} = \begin{pmatrix} 1, & 1, & 0, & 1 \end{pmatrix} \tag{4.31}$$

Finally, we apply the proposed move for each block where the corresponding element $a_k \in \vec{a}$ is equal to 1. As shown in Fig. 4.15b, each block belonging to one of the accepted sets (s_1, s_2 and s_4) has been moved to the corresponding proposed new position according to the AMPP in Fig. 4.15a, while blocks 5 and 15 were not moved, since set s_3 was not accepted (i.e., $a_3 = 0$). **Note that the placement can be updated efficiently in parallel based on the vector $\vec{a}$ along with the position vectors by using the *permutation scatter* parallel pattern from Chapter 2**.

4.4 Work, span, and isoefficiency complexity

Table 4.1 lists the work complexity, span complexity, and isoefficiency function complexity of the operations performed according to our proposed CAMIP methodology. As shown in Table 4.1, the overall work complexity of the CAMIP operations (not including objective cost evaluation) is $\Theta(B)$, and the overall span complexity is $\Theta(\log B)$, where B is the number of blocks to be placed. As discussed in Section 2.3.2, based on work and span complexity, the CAMIP operations are "completely parallelizable" [64, 66] with an isoefficiency function in $\Theta(p \log p)$. Note that an

isoefficiency function in $\Theta(p \log p)$ implies weak scaling, where constant parallel efficiency can be maintained as the number of parallel workers increases as long as the problem size grows at at the rate $\Theta(p \log p)$.

Operation	Pattern	Work	Span	Isoefficiency
Propose moves	Map	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
Evaluate moves (not including objective cost complexity)	Map	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
Assess groups	Map	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
delta-cost	Reduce	$\Theta(B)$	$\Theta(\log B)$	$\Theta(p \log p)$
	Category sort	$\Theta(B)$	$\Theta(\log B)$	$\Theta(p \log p)$
	Scan	$\Theta(B)$	$\Theta(\log B)$	$\Theta(p \log p)$
	Scatter	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
	Category reduce	$\Theta(B)$	$\Theta(\log B)$	$\Theta(p \log p)$
	Pack	$\Theta(B)$	$\Theta(\log B)$	$\Theta(p \log p)$
Apply groups	Scatter	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
Total complexity		$\Theta(B)$	$\Theta(\log B)$	$\Theta(p \log p)$

Table 4.1: Work, span, and isoefficiency complexity of calculations related to CAMIP, based on number of blocks (B) and number of parallel workers (p). *N.B.,* "Evaluate moves" does not include the complexity of the objective cost calculations (see Chapters 5-6).

As we will see in Chapters 5-6, work and span complexity analysis of our proposed wirelength and timing calculations also predicts weak-scaling for objective-cost evaluation.

4.5 Summary

In this chapter, we presented our *Concurrent Associated-Moves Iterative Placement* model. Our model iteratively improves a placement by evaluating and conditionally applying very large sets of moves concurrently. Some key characteristics of our CAMIP model include:

1. Deterministic behaviour ensuring that, given the same random seed, the same placement result is produced regardless of the number of parallel workers.

2. Support for arbitrary objective cost function, as long as a cost can be assigned to:

   (a) The entire placement (for exit criterion).

   (b) Each proposed move.

3. Support for arbitrary moves during each search iteration, assuming moves are defined in terms of groups of *associated-moves*.

4. Highly parallel, where all non-constant-time operations are expressed in terms of load-balanced, "completely parallelizable" [64, 66] parallel patterns with very low overhead, implicit, synchronization (no locking required).

We also presented a novel mechanism for efficiently generating many groups of associated-moves in parallel using what we call *Associated Move-Pair Patterns (AMPPs)*. Each AMPP can be defined with $O(1)$ runtime complexity by randomly generating only four random integers (two for each dimension), regardless of the size of the placement grid. Moreover, the move for each block can be computed *independently* in constant-time, which allows moves to be processed efficiently in parallel using the *map* pattern.

In the next two chapters, we present implementations of the CAMIP model presented here, using two different objective cost functions. In Chapter 5, we use the Star+ wirelength model [82] to assign a cost to each net in the netlist and to estimate the difference in cost due to moving a block to a proposed target position. In Chapter 6, we combine the Star+ wirelength cost with a timing-delay cost based on the connection-based timing model used in [4]. Moreover, in Chapter 5 and Chapter 6, we present experimental results demonstrating the high runtime performance

and solution quality exhibited by our CAMIP model using wirelength and connection-based delay objective functions, respectively.

Chapter 5

Parallel, wirelength-driven placement using CAMIP

5.1 Introduction

As discussed in Chapter 2, the most common optimization objective used in FPGA placement is estimiated *wirelength*. In this chapter, we propose an implementation of the CAMIP model from Chapter 4 using the Star+ [82] wirelength model. While the implementation presented in this chapter could easily be adapted to use other wirelength models (including bounding-box/HPWL), we chose to use the Star+ model since it has been shown to provide improved quality of results compared to other models [82]. Furthermore, as we will discuss in this chapter, characteristics of the Star+ model enable efficient update of the cost of a net due to a move based *only* on the positions involved in the move, whereas in the worst case, the HPWL cost of a net requires the position of *every* block connected to the net.

The remainder of this chapter is laid out as follows. Section 5.2 discusses how structured parallel patterns can be used to efficiently compute *a)* the cost of an entire placement, *b)* the cost of all nets connected to a block, and *c)* the difference in cost of nets connected to block i based on moving block i to a new position. In Section 5.4, we discuss our experimental methodology, along with results comparing our parallel implementation of the proposed wirelength-driven CAMIP model to VPR, both in terms of runtime and solution quality. Our proposed implementation

is GPU-based, and demonstrates the improvements in absolute execution time compared to the most commonly used sequential algorithm (16-28× on commodity hardware) made possible by utilizing structured parallel patterns for *all* operations in our CAMIP model, while maintaining the same quality as an equivalent serial implementation.

5.2 Parallel Star+

As discussed in Chapter 4, all steps of the CAMIP model that do not involve objective cost evaluation can be completed using the structured parallel patterns introduced in Chapter 2, leading to predicted weak-scaling parallelism based on parallel performance theory. To maintain a high degree of parallelism, it follows that a selected objective cost model must also permit efficient parallel calculation, which exhibits weak-scaling. In this section, we demonstrate that the following costs based on the Star+ model may be calculated efficiently using structured parallel patterns that exhibit weak-scaling:

1. Cost of the entire placement.
2. Cost of all nets connected to a block.
3. Difference in cost due to move applied to each block.

Moreover, we show that the above calculations, in combination with the steps described in Chapter 4, are sufficient to implement a complete algorithm based on the CAMIP model using *only* structured parallel patterns for *all* steps.

5.2.1 Formal model

The remainder of this section describes our proposed mathematical model for efficiently computing Star+ costs using matrix operations that can be performed using structured parallel patterns. We first present relevant notation used, followed by a description of matrices and vectors defined in our model.

Notation

Before describing our method of computing Star+ costs, we first introduce some notation used in the remainder of this chapter. In describing our matrix Star+ calculations, we use the following notation to denote various element-wise matrix operations, based on the Hadamard product [83]:

Element-wise multiplication

- $C = A \circ B \implies c_{ij} = a_{ij}b_{ij}$
- A and B must be the same size, and C will be the same size as A and B.

Element-wise square

- $B = A \circ A \implies b_{ij} = a_{ij}^2$
- Equivalent to squaring every element in matrix A.

Element-wise reciprocal

- $B = A^{\circ(-1)} \implies b_{ij} = \frac{1}{a_{ij}}$.
- Equivalent to computing the reciprocal of every element in matrix A.

Element-wise square-root

- $B = A^{\circ\frac{1}{2}} \implies b_{ij} = \sqrt{a_{ij}}$.
- Equivalent to taking the square-root of every element in matrix A.

Much of the work done to compute Star+ consists of computing either the sum of each row in a matrix, or the sum of each column. To concisely describe such operations, we define $\vec{1_r}$ as a *row* vector, containing all ones, and we define $\vec{1_c}$ as a *column* vector, containing all ones, where $\vec{1_r} \in \mathbb{R}^m$, $\vec{1_c} \in \mathbb{R}^n$, m is the number of blocks in the netlist and n is the number of nets. In other words, the rank of $\vec{1_r}$ is the same as the number of *nets* while the rank of $\vec{1_c}$ is the same as the number of *blocks*. Given $\vec{1_r}$ and $\vec{1_c}$ note the following:

- $\vec{u} = \vec{1_r}A \implies u_j = \sum_i a_{ij}$
  - Equivalent to computing the sum of each *column* in matrix A.
- $\vec{u} = A\vec{1_c} \implies u_i = \sum_j a_{ij}$
  - Equivalent to computing the sum of each *row* in matrix A.

We make heavy use of these dot-product operations within our parallel Star+ calculations.

Netlist adjacency matrix

To compute Star+ using matrix operations, first we need to represent the connections defined by the netlist in the form of a matrix. Let m and n be the number of blocks and the number of nets in the netlist, respectively. As shown in Eq. 5.2, we define $C \in \mathbb{M}_{mn}$ as a netlist adjacency matrix, such that each c_{ij} corresponds to a potential connection between block i and net j. Note that our implementation described in Section 5.4 section uses sparse matrix encoding, similar to all matrices discussed in Chapter 4, such that the storage requirements *and* computational complexity scale with the *number of connections (i.e., the number of non-zero entries in C)*, which is typically *much, much* less than $O(mn)$.

$$c_{ij} = \begin{cases} 1, & \text{if block } i \text{ is connected to net } j \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & \dots & \dots & c_{1n} \\ c_{21} & c_{22} & c_{23} & \dots & \dots & c_{2n} \\ \vdots & & \ddots & & & \\ \vdots & & & c_{ij} & & \\ \vdots & & & & \ddots & \\ c_{m1} & c_{m2} & c_{m3} & \dots & \dots & c_{mn} \end{pmatrix} \tag{5.2}$$

Net rank vector

To compute the Star+ cost of net j, it is necessary to know the number of blocks connected to net j. We refer to the number of blocks connected to net j as the *rank* of net j. We define the

vector $\vec{r}$ to contain the rank of each net. The vector $\vec{r}$ can be computed using the C matrix as shown in Eq. 5.3.

$$\vec{r} \in \mathbb{R}^n,\ r_j = \vec{1} \cdot \vec{C_{*j}} \tag{5.3}$$

Block positions

Recall from Chapter 4 that the vectors $\vec{p_x}$ and $\vec{p_y}$ contain the x and y position of each block in the placement, where p_{xi} and p_{yi} denote the location index of block i in the x-dimension and y-dimension, respectively.

Given the vectors $\vec{p_x}$ and $\vec{p_y}$, we define the following matrices:

$$X \in \mathbb{M}_{mn} : x_{ij} = \begin{cases} p_{xi}, & \text{if } c_{ij} = 1, \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

$$Y \in \mathbb{M}_{mn} : y_{ij} = \begin{cases} p_{yi}, & \text{if } c_{ij} = 1, \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

In other words, X is the same shape as the adjacency matrix, where each element contains the x-dimension position index of the block corresponding to the row of the matrix, *if* the block is connected to the respective net. If the block i is *not* connected to net j, the x_{ij} element is equal to 0. The Y matrix is formed the same way, but holds y-dimension block position indexes instead.

Star+ cost of entire placement

Recall from Chapter 2 that the Star+ cost of a net may be calculated using the following equation, where r_j is the number of blocks connected to net j, and $i \in \text{net}_j$ denotes each block, i, connected to net j.

$$n_{cj} = \alpha \left\{ \sqrt{\sum_{\forall i \in \text{net}_j} p_{xi}^2 - \frac{\left(\sum_{\forall i \in \text{net}_j} p_{xi}\right)^2}{r_j} + \beta} + \sqrt{\sum_{\forall i \in \text{net}_j} p_{yi}^2 - \frac{\left(\sum_{\forall i \in \text{net}_j} p_{yi}\right)^2}{r_j} + \beta} \right\} \tag{5.6}$$

Given the matrices X, and Y, and vectors $\vec{1_r}$, and $\vec{r}$, it is possible to compute the Star+ [82] wirelength cost for all nets by performing a matrix variation of the equation above. We define the vector $\vec{n_c} \in \mathbb{R}^n$ in Eq. 5.7, where n_{cj} denotes the Star+ wirelength cost of net j.

$$\begin{aligned}
\vec{n_c} \in & \mathbb{R}^n \\
\vec{n_c} = & \alpha \left\{ \left[(\vec{1_r}X) \circ (\vec{1_r}X) - (\vec{1_r}(X \circ X) \circ (\vec{r}^{\circ(-1)}) + \beta \right]^{\circ \frac{1}{2}} \right. \\
& \left. + \left[(\vec{1_r}Y) \circ (\vec{1_r}Y) - (\vec{1_r}(Y \circ Y) \circ (\vec{r}^{\circ(-1)}) + \beta \right]^{\circ \frac{1}{2}} \right\} \\
= & \left(n_{c1}, \quad n_{c1}, \quad \ldots, \quad n_{cj}, \quad \ldots, \quad n_{cn} \right)
\end{aligned} \tag{5.7}$$

By summing all elements of the vector $\vec{n_c}$ the cost of the entire placement is obtained. **Note that the matrix calculation above can be performed efficiently in parallel using a category reduction parallel pattern to perform the sparse matrix operations. Furthermore, the sum of $\vec{n_c}$ can be computed efficiently using a parallel reduction sum pattern.**

Star+ cost of all nets connected to a block

Since the CAMIP model proposed in Chapter 4 iteratively modifies the placement by moving blocks, we need a means to calculate an estimated difference in placement cost based on moving a block from one position to another. With this goal in mind, we first define the cost of block i as the sum of costs of all nets connected to block i, where we use b_{ci} to denote the cost of block i. To prepare for calculating the cost of each block, we also define the matrix $\Omega \in \mathbb{M}_{mn}$, where the value of each element $\omega_{ij} \in \Omega$ is defined as shown in Eq. 5.8.

$$\omega_{ij} = \begin{cases} n_{cj}, & \text{if } c_{ij} = 1, \\ 0, & \text{otherwise} \end{cases} \tag{5.8}$$

Given the matrix Ω, we can compute the vector $\vec{b_c} \in \mathbb{R}^m$ as shown in Eq. 5.9 (recall that b_{ci} denotes the total Star+ cost of all nets connected to block i).

$$\begin{aligned} \vec{b_c} &= \Omega\left(\vec{1_c}\right) \\ &= \begin{pmatrix} b_{c1}, & b_{c2}, & \ldots, & b_{ci}, & \ldots, & b_{cm} \end{pmatrix} \\ b_{ci} &= \sum_j \omega_{ij} \end{aligned} \tag{5.9}$$

Note that the elements in the matrix Ω may be computed efficiently in parallel using the *map* pattern described in Chapter 2. The vector $\vec{b_c}$ may also be computed efficiently in parallel by applying a *category reduction* pattern to a sparse-matrix representation of Ω matrix to compute the sum of each row concurrently.

Difference in Star+ cost due to moving a block

In addition to computing a cost for each block i based on the current placement, we must compute the difference in cost for each block i based on the set of moves proposed for the current CAMIP iteration. To compute the difference in cost for each block, let us define a matrix $\Omega' \in \mathbb{M}_{mn}$, where each element $\omega'_{ij} \in \Omega$ corresponds to the updated cost of the connection between block i and net j based on the newly proposed position of block i *(i.e.,* (p'_{xi}, p'_{yi})).

Given Ω', it is straight-forward to compute $\vec{b'_c}$ in a similar manner to the way $\vec{b_c}$ is calculated as shown above, where each element b'_{ci} corresponds to the cost of block i at location (p'_{xi}, p'_{yi}). However, we first must provide an efficient method to compute the matrix Ω'.

To compute Ω', we first must revisit the matrix calculation given above to compute the vector $\vec{n_c}$, containing the cost of each net.

$$\vec{n_c} = \alpha \left\{ \left[(\vec{1_r}X) \circ (\vec{1_r}X) - (\vec{1_r}(X \circ X) \circ (\vec{r}^{\circ(-1)}) + \beta \right]^{\circ \frac{1}{2}} + \left[(\vec{1_r}Y) \circ (\vec{1_r}Y) - (\vec{1_r}(Y \circ Y) \circ (\vec{r}^{\circ(-1)}) + \beta \right]^{\circ \frac{1}{2}} \right\} \tag{5.10}$$

Now, let us define four vectors, $\vec{n_x}, \vec{n_{x^2}}, \vec{n_y}, \vec{n_{y^2}} \in \mathbb{R}^n$, as shown in Equations 5.12-5.14. Here, each element n_{xj} corresponds to the sum of the x-dimension position of all blocks connected to net j, while each element n_{x^2j} corresponds to the sum of the *squared* x-dimension position of all blocks connected to net j. Similarly, each element n_{yj} corresponds to the sum of the y-dimension position of all blocks connected to net j, while each element n_{y^2j} corresponds to the sum of the *squared* y-dimension position of all blocks connected to net j.

$$\vec{n_x} = (\vec{1_r}X) \implies n_{xj} = \sum_i x_{ij} \tag{5.11}$$

$$\vec{n_{x^2}} = \vec{1_r}(X \circ X) \implies n_{x^2j} = \sum_i x_{ij}^2 \tag{5.12}$$

$$\vec{n_y} = \vec{1_r}Y \implies n_{xj} = \sum_i x_{ij} \tag{5.13}$$

$$\vec{n_{y^2}} = \vec{1_r}(X \circ X) \implies n_{y^2j} = \sum_i y_{ij}^2 \tag{5.14}$$

Substituting $\vec{n_x}, \vec{n_{x^2}}, \vec{n_y}, \vec{n_{y^2}}$ into Eq. 5.10 gives the equation below.

$$\vec{n_c} = \alpha \left\{ \left[\vec{n_x} \circ \vec{n_x} - \vec{n_{x^2}} \circ (\vec{r}^{\circ(-1)}) + \beta \right]^{\circ \frac{1}{2}} + \left[\vec{n_y} \circ \vec{n_y} - \vec{n_{y^2}} \circ (\vec{r}^{\circ(-1)}) + \beta \right]^{\circ \frac{1}{2}} \right\} \tag{5.15}$$

Given an element n_{xj}, it is straight-forward to compute an updated element due to a single block i moving from (p_{xi}, p_{yi}) to (p'_{xi}, p'_{yi}), as shown in Eq. 5.16 below, where the updated element is denoted as n'_{xij}.

$$n'_{xij} = n_{xj} - p_{xi} + p'_{xi} \tag{5.16}$$

In a similar fashion, n_{x^2j} can be updated in constant-time as shown in the Eq. 5.17 below.

$$n'_{x^2ij} = n_{x^2j} - p_{xi}^2 + \left(p'_{xi}\right)^2 \tag{5.17}$$

The updated y-dimension elements (i.e., n'_{yij} and n'_{y^2ij}) can be computed in the same way. **Note that $n'_{xij}, n'_{x^2ij}, n'_{yij}, n'_{y^2ij}$ are all indexed by both net j and block i, where i corresponds to the block that was moved**. Extending this idea, let us define n'_{cij} as the updated Star+ cost for net j, due to block i moving from (p_{xi}, p_{yi}) to (p'_{xi}, p'_{yi}). Assuming net j is connected to block i, n'_{cij} can be computed in *constant-time*, as shown in Eq. 5.18 below.

$$\begin{aligned} n'_{cij} &= \alpha \left\{ \sqrt{\left(n'_{xij}\right)^2 - \frac{\left(n'_{x^2ij}\right)}{r_j} + \beta} + \sqrt{\left(n'_{yij}\right)^2 - \frac{\left(n'_{y^2ij}\right)}{r_j} + \beta} \right\} \\ &= \alpha \left\{ \sqrt{\left(n_{xj} - p_{xi} + p'_{xi}\right)^2 - \frac{\left(n_{x^2j} - p_{xi}^2 + \left(p'_{xi}\right)^2\right)}{r_j} + \beta} \right. \\ &\quad \left. + \sqrt{\left(n_{xj} - p_{xi} + p'_{xi}\right)^2 - \frac{\left(n_{x^2j} - p_{xi}^2 + \left(p'_{xi}\right)^2\right)}{r_j} + \beta} \right\} \end{aligned} \tag{5.18}$$

Finally, given the equation above for computing n'_{cij} for each connection between a block and a net, we can fill the elements of the Ω' matrix, as shown in Eq. 5.19 below.

$$\omega'_{ij} = \begin{cases} n'_{cj}, & \text{if } c_{ij} = 1, \\ 0, & \text{otherwise} \end{cases} \tag{5.19}$$

It follows that we can compute the vector $\vec{b'_c} \in \mathbb{R}^m$ as show in the equation below, where each element b'_{ci} corresponds to the updated cost of block i based on moving to the new position, (p'_{xi}, p'_{yi}).

$$
\begin{aligned}
\vec{b'_c} &= \Omega\left(\vec{1_c}\right) \\
&= \left(b'_{c1},\quad b'_{c2},\quad \ldots,\quad b'_{ci},\quad \ldots,\quad b'_{cm}\right) \\
b'_{ci} &= \sum_j \omega'_{ij}
\end{aligned} \tag{5.20}
$$

Once we have computed the cost of each block based on the corresponding current and proposed positions, the only remaining step is to compute the difference between the block costs. As defined in Chapter 4, we use $\vec{\delta_b} \in \mathbb{R}^m$ to denote the vector where each element δ_{bi} corresponds to the difference in cost due to moving block i according to the proposed moves for the current CAMIP iteration. The vector $\vec{\delta_b}$ can be computed as show in the equation below, by simply computing the difference between the vectors $\vec{b'_c}$ and $\vec{b_c}$.

$$
\vec{\delta_b} = \vec{b'_c} - \vec{b_c} \tag{5.21}
$$

Summary

Recall that the only requirements for the objective cost evaluation steps in the CAMIP model from Chapter 4 are that they must provide:

1. A total cost based on the entire placement.
2. A difference in cost for each block based on the proposed set of moves for the current search iteration.

The matrix calculations described in this chapter demonstrate how both of these requirements can be met. **In addition, all calculations described in this chapter can be performed efficiently in parallel using structured parallel patterns, including *map*, sparse matrix operations using *category-reduction*, and *reduction sum*.** The remainder of this chapter presents experimental results obtained by running a parallel, NVIDIA CUDA GPU-based implementation of our proposed wirelength-driven CAMIP algorithm, based on Star+.

5.3 Work, span, and isoefficiency complexity

Table 5.1 lists the work complexity and span complexity of the calculations related to wirelength in our proposed W-CAMIP implementation. As shown, the work complexity of the wirelength calculations is $\Theta(C)$, and the span complexity is $\Theta(\log C)$. As discussed in Section 2.3.2, based on work and span complexity, the wirelength calculations are "completely parallelizable" [64, 66] with an isoefficiency function in $\Theta(p \log p)$. Recall that an isoefficiency function in $\Theta(p \log p)$ implies weak scaling, where constant parallel efficiency can be maintained as the number of parallel workers increases as long as the problem size grows at at the rate $\Theta(p \log p)$.

Operation	Pattern	Work	Span	Isoefficiency
Total placement	Map	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
cost	Category reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
	Reduce	$\Theta(N)$	$\Theta(\log N)$	$\Theta(p \log p)$
Per-block cost	Category reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
Per-block move	Category reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
delta-cost	Map	$\Theta(B)$	$\Theta(1)$	$\Theta(p)$
Total complexity		$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$

Table 5.1: Work, span, and isoefficiency complexity of calculations related to wirelength, based on: number of connections (C), number of blocks (B), number of nets (N), and number of parallel workers (p).

Note that the connection count corresponds to the size of a netlist (i.e., the problem size is in $\Theta(C)$).

5.4 Experiments

To evaluate the efficacy of our proposed *wirelength-driven CAMIP (using Star+)* design with respect to runtime performance and quality-of-result, we developed a CUDA implementation using the Thrust parallel library. Within the remainder of this thesis, we will refer to our Star+ wirelength-driven CAMIP implementation as, "W-CAMIP". Appendix D describes details of the W-CAMIP implementation. We used our CUDA W-CAMIP implementation along with VPR to conduct several sets of experiments. First, we ran VPR using the bounding-box cost function with `inner_num=1` [1], across the set of 7 large benchmark netlists described in Section 2.4, ranging from 575395 to 1428437 adjacency connections. Note that running VPR with `inner_num=1` is equivalent to running VPR using the `-fast` flag, which is typical in the experiments reported in the literature [77, 79, 80]. For each netlist, we ran ten trials, each with a different random seed. Next, we ran ten trials using our *W-CAMIP* for each netlist in the same set of benchmarks, on each of the following NVIDIA GPU cards: *a)* GeForce 8800 GT [84], *b)* GeForce GT 430 [85], and *c)* GeForce GTX 480 [86]. For each card, Table 5.2 lists the number of processing cores, the peak memory bandwidth, and the frequency of the core clock. While it is tempting to compare the GPU models based on these parameters, the underlying architecture of each card may vary considerably, making empirical performance benchmarking important. Note that we ran all experiments using on an Intel Core i7-930 2.8GHz processor with 12GB RAM running 64-bit Ubuntu 12.04.3. All host code was compiled using `g++` with the flags `-O3 -msse2`. All CUDA code was compiled using `nvcc` version 5.5.0.

Model	CUDA Cores	Bandwidth (GB/s)	Core clock (MHz)
GeForce 8800 GT	112	57.6	1500
GeForce GT 430	96	28.8	1400
GeForce GTX 480	448	133.9	1215

Table 5.2: Specifications for GeForce 8800 GT [84], GeForce GT 430 [85], and GeForce GTX 480 [86].

[1] The `inner_num` parameter controls the number of moves evaluated at each temperature in the anneal. For example, `inner_num=10` results in 10× more moves at each temperature compared to `inner_num=1`.

5.4.1 Absolute execution time

Table 5.3 shows the mean placement runtimes in seconds, using VPR with `inner_num=1` and our W-CAMIP running on three different GPU cards (with `inner_num=1`), across the ten trials for each of the 7 IWLS-based netlists. By observing the summed total runtime for each placer configuration in the last row, we can see that the W-CAMIP is significantly faster than VPR when run on any of the GPU cards. Specifically, on the highest performing card tested, the GTX480, improvements in absolute execution time range from 16.3× to 27.8×, with a mean improvement of 19×. When comparing runtime performance of our parallel W-CAMIP implementation against VPR, we report absolute execution time of our parallel implementation running on one of several GPUs against VPR running on a workstation CPU (all commodity hardware). While comparing across architectures is not ideal, it is not possible to run VPR directly on a GPU. Furthermore, we would like to note that cross-architecture absolute runtime comparisons for FPGA placement have been reported previously in peer-reviewed studies [77, 79, 80], including the results presented in this section [13].

Netlist	Connections	GeForce8800GT	GT430	GTX480	VPR
`uoft_raytracer`	575395	1.0k	829.8	217.0	3.5k
`b17_1_new`	581521	821.0	711.7	174.7	2.9k
`leon3mp`	614752	1.2k	1.1k	263.9	4.5k
`netcard`	623926	1.2k	1.0k	248.5	4.3k
`b18_new`	727922	1.1k	940.0	217.1	4.2k
`b18_1_new`	729741	1.1k	935.0	218.5	4.1k
`b19_1_new`	1428437	2.5k	2.2k	414.3	11.5k
Sum		8.9k	7.7k	1.8k	35.0k

Table 5.3: Wire-length driven placement run-times in seconds.

It is important to note that, as shown in the plot in Fig. 5.1, there is a strong positive correlation between the observed absolute runtime improvements and the adjacency-list connection

count. Note that the connection-count scales directly with the size of the netlist to be placed. This indicates that as the problem size grows, our approach exhibits a larger improvement in absolute execution time, which suggests that none of the benchmarks used in our experiments were large enough to reach maximum parallel efficiency on any of the GPUs we tested with, since larger netlists were able to utilize more processing resources. Since our results suggest that processing resources are not saturated on any of the GPUs for the size of netlists in our tests, the difference in slope between the improvements in runtime observed for the different GPU cards is likely due to architectural improvements between card generations, including memory bandwidth, per-core performance, etc. Figure 5.2 shows a relative comparison between the specifications of the GPU cards from Table 5.2 and the total runtimes from Table 5.3, normalized by the largest corresponding value.

Due to the weak-scaling behaviour predicted by an isoefficiency function in $\Theta(p \log p)$, as the size of the netlist to be placed increases, we expect parallel efficiency to continue to grow until the GPU resources are saturated. In Chapter 7, we discuss future work that would provide larger netlists to conduct further efficiency scaling experiments with our placement tool.

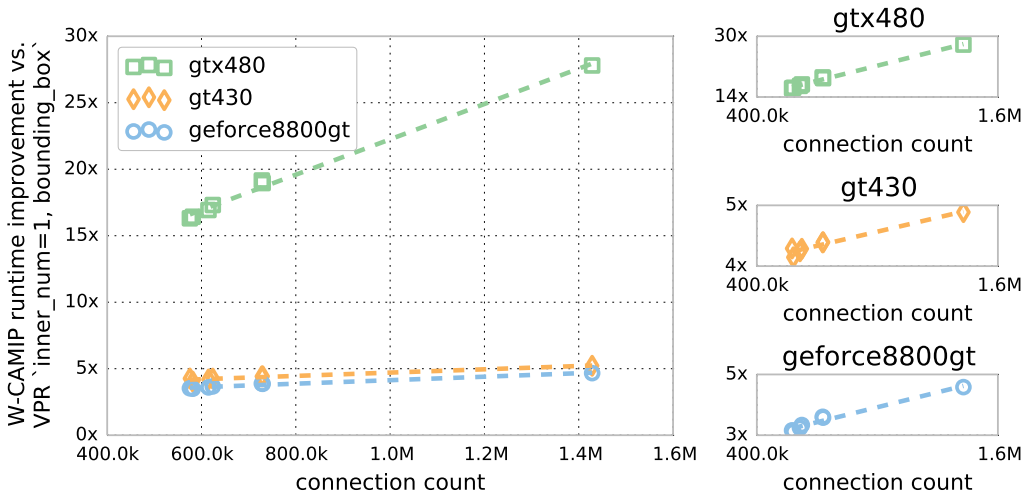


Figure 5.1: W-CAMIP improvements in absolute runtime relative to VPR (*inner-num = 1*).

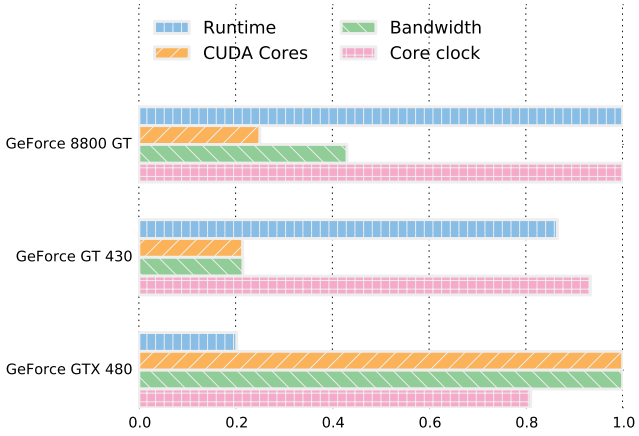
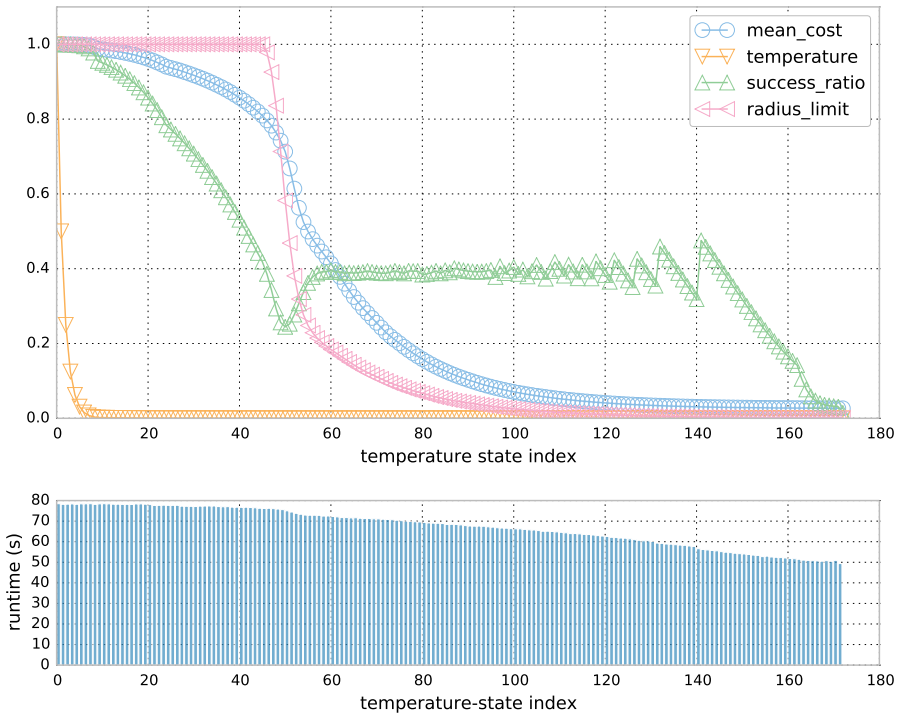


Figure 5.2: Relative comparison of normalized specifications of NVIDIA cards and normalized runtime across all IWLS benchmarks.

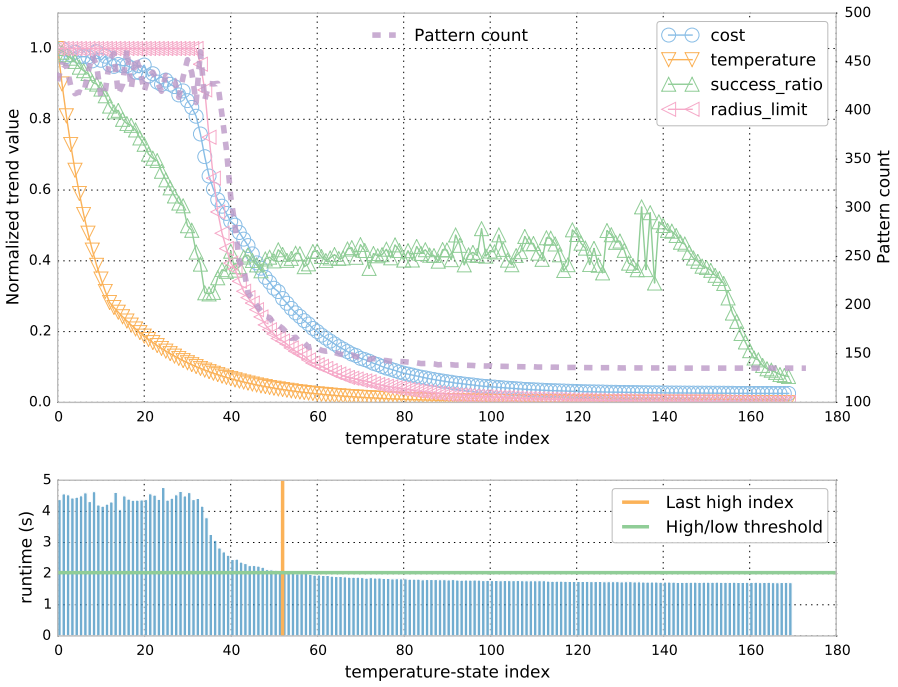
Variation of runtime across temperature states

Here we compare the annealing runtime behaviour of VPR in bounding-box mode, with `inner_num=1`, against our W-CAMIP placer, also with `inner_num=1`. Note that we run VPR and W-CAMIP with the same annealing temperature schedule, and target the same number of moves at each temperature. Recall that the VPR annealer evaluates a set number of moves at each temperature, after which the temperature is updated, starting the next temperature state. Figure 5.3 shows the trend of several values across an entire annealing run, of VPR and our W-CAMIP placer. This figure shows the trends for a single run of only one of the netlists, but it is representative of a typical run of both algorithms, where the number of temperature states is typically between 120-200. The trend lines include the wirelength cost, the radius-limit, the success ratio, and the temperature. For each temperature state, the radius-limit corresponds to the maximum distance that a block may move, the success ratio is the percentage of moves that are accepted, and the temperature is related to how likely it is that non-improving moves will be accepted. Each unit on the x-axis corresponds to a temperature state in the anneal, where a set number moves are evaluated before updating the temperature to transition to the next temperature state. As shown in the figure, both VPR and W-CAMIP run between 160-180 temperature iterations in this example.

The bottom sub-plots in Fig. 5.3a and Fig. 5.3b show the runtime for each temperature state.



(a) VPR: `bounding_box, inner_num=1`



(b) W-CAMIP: `inner_num=1`

Figure 5.3: Simulated annealing trend

Notice that the runtimes for VPR are relatively consistent, but are gradually decreasing towards the end of the anneal, whereas in W-CAMIP, approximately the first 30 states are about double the runtime of the final temperature states.

The reason for this is that, as shown in Chapter 4, when we generate move patterns, some of the moves may target positions that are outside the boundary of the placement area, in which case we ignore such moves. Early on in the anneal, when the radius-limit is high, as shown in the top sub-plot of Fig. 5.3b, there are more moves that target positions that are out-of-bounds than in the later temperature states, where the radius-limit is much smaller. When the radius-limit is large, we must apply more parallel sets of moves to reach the target number of moves that must be evaluated at each temperature. Recall that we stay at a given temperature until we reach a set number of moves. As can be seen from Fig. 5.3b, once the radius-limit begins to the drop, the runtimes of subsequent temperature states decrease as well. To validate our hypothesis regarding the cause of the longer temperature state runtimes at the beginning of the anneal, we plot the number of associated move-pair patterns that must be evaluated at each temperature state as a dashed line in Fig. 5.3b. As shown in the figure, the number of move-pair patterns at each temperature state is strongly correlated with *a)* the radius-limit, and *b)* the temperature state runtime. To emphasize the relationship between temperature state runtime, radius-limit, and the number of move-pair patterns processed at each state, in Fig. 5.4, we plot linear regressions between temperature state runtime and the other factors, where R^2 value is 0.99 in both cases, indicating a strong positive correlation.

In the bottom sub-plot of Fig. 5.3b, we have marked a threshold which is 5% of the difference between the maximum and the minimum temperature state, added to the minimum temperature state runtime. We then count the number of temperature states where the runtime is higher, and count the number of states with a lower runtime. Figure 5.5a plots the high runtime and low runtime count using this approach for each of the netlists in our benchmark set. As we can see, the number of temperature states that are above and below the threshold are relatively consistent across the netlists. Figure 5.5b shows the number of connections in each netlist, where the connection count is representative of the size of the netlist. From Fig. 5.5a and Fig. 5.5b, it is evident that there does not seem to be a correlation between the connection count, i.e., the size of the netlists, and the number of temperature states that are above and below the threshold.

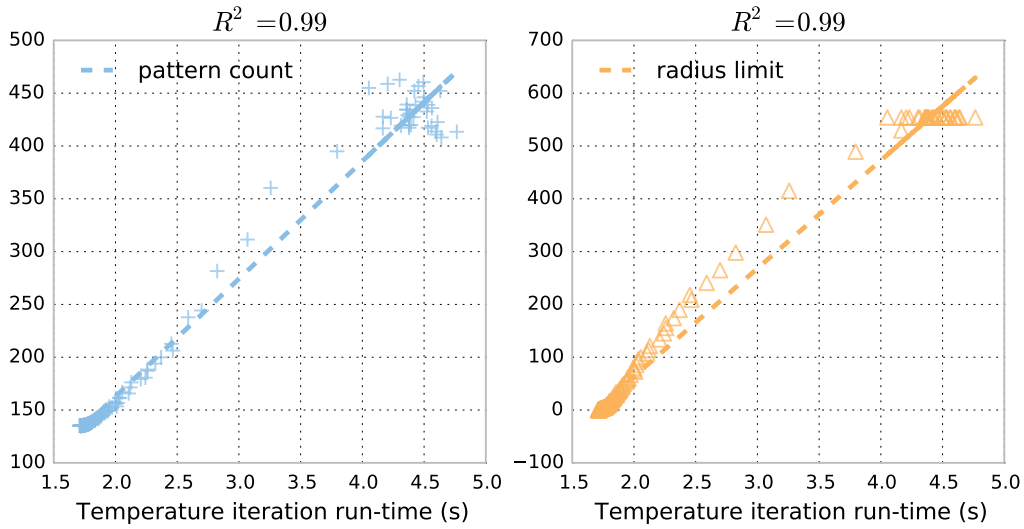
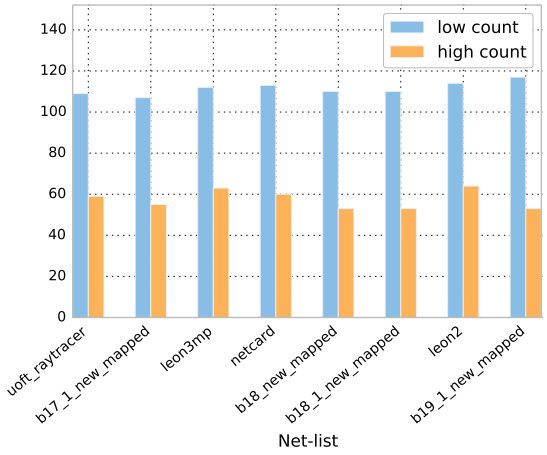


Figure 5.4: W-CAMIP temperature state runtime correlations.

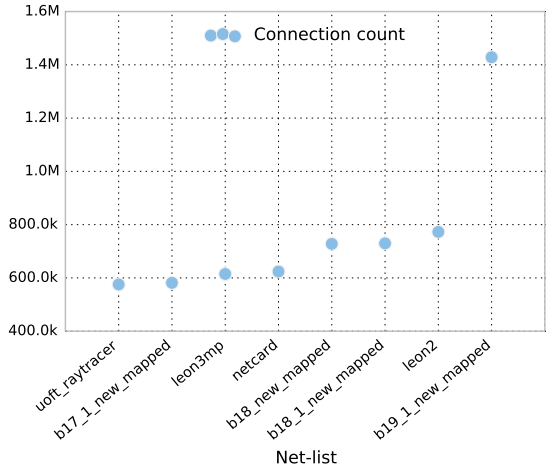
Furthermore, Fig. 5.5c shows the ratio of the mean of the runtime of temperature states *below* the threshold, and the mean of the temperature states *above* the threshold. As we can see, the ratio is relatively consistent, between 47-48% for all of the netlists in our benchmark set, regardless of the netlist size. Since the number of high and low runtime temperature states, and the ratio of the runtimes between the states below and above the threshold are both relatively consistent, this phenomenon should not affect the scalability of our approach. The time spent in the high runtime states relative to the low runtime states is relatively consistent and does not increase with the size of the netlist.

5.4.2 Post-routing wirelength and critical-path-delay

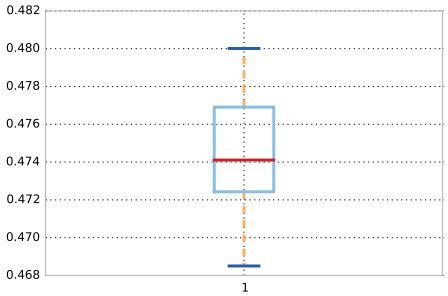
After performing placement under each of the configurations described above, we routed each placement using VPR's router. The resulting routed critical-path-delays and wirelengths for VPR and our proposed W-CAMIP are listed in Table 5.4, where μ_{wl} represents the mean wirelength (in segments), and μ_{cpd} represents the mean critical-path-delay (in nanoseconds), for each placer configuration across 10 trials for each netlist. Note that the quality-of-result for W-CAMIP does not depend on the number of processing cores, and therefore does not change based on the GPU card used. The last two columns in Table 5.4a show the percentage difference in the resulting wirelength and critical-path-delay from VPR versus our parallel annealer,



(a) The number of high/low runtime iterations across benchmarks.



(b) Connection count not correlated to number of high runtime iterations.



(c) Ratio of low runtime iteration length to high runtime iteration length. (˜47-48% across all benchmark netlists)

Figure 5.5: Analysis of high runtime temperature iterations vs. low runtime

denoted as Δ_{wl} and Δ_{cpd}, respectively. Note that a negative value in either the Δ_{wl} column or the Δ_{cpd} column indicates that W-CAMIP improved upon the corresponding metric from VPR.

The box-plots[2] in Fig. 5.6, show head-to-head comparisons between the wirelength and critical-path-delay results from W-CAMIP and VPR. From the plots, we see that our W-CAMIP improves the critical-path-delay and wirelength compared to VPR with statistical significance (determined using Wilcoxon signed-rank test with $\alpha = 0.05$) for 3 out of 7 netlists, with an overall improvement of 5.1% in wirelength and 4.8% in critical-path-delay. These results clearly

[2]In the convention used for the box-plots in this thesis, the ends of the whiskers represent the lowest datum within $1.5IQR$ of the lower quartile, and the highest datum within $1.5IQR$ of the upper quartile, where $IQR = Q3 - Q1$ (i.e., the inner-quartile range), $Q3$ is the upper quartile, and $Q1$ is the lower quartile [87].

show that our W-CAMIP is able to effectively explore the placement solution space, despite the soft-conflicts introduced by evaluating swaps in concurrent sets and any bias introduced by the regularity of the applied displacement-patterns. Furthermore, it is evident that despite the highly parallel nature of our approach, quality is maintained, and even improved in several cases compared to VPR.

Netlist	VPR		W-CAMIP		% diff	
	μ_{wl}	μ_{cpd}	μ_{wl}	μ_{cpd}	Δ_{wl}	Δ_{cpd}
`uoft_raytracer`	1.6M	1.8u	1.5M	1.3u	-2.8	-27.1
`b17_1_new`	2.2M	352.9n	2.2M	368.3n	-1.4	4.4
`leon3mp`	1.6M	1.1u	1.5M	1.1u	-6.5	3.9
`netcard`	1.5M	973.6n	1.4M	918.2n	-5.4	-5.7
`b18_new`	2.7M	589.8n	2.7M	612.3n	-1.0	3.8
`b18_1_new`	2.7M	591.6n	2.7M	634.9n	0.0	7.3
`b19_1_new`	5.4M	891.1n	5.2M	837.8n	-3.4	-6.0
Sum	17.7M	6.3u	17.2M	5.8u		
Mean					5.1	4.8

Table 5.4: Post-routing mean wirelength (μ_{wl}) and critical-path delay (μ_{cpd}). A negative difference (Δ_{wl}, Δ_{cpd}) corresponds to W-CAMIP improvement over VPR.

5.5 Summary

Using our method of generating very large sets of non-overlapping swaps where the number of swaps scales with the size of the target FPGA architecture, in this chapter we proposed a variant of our CAMIP model, which we call *W-CAMIP*, that evaluates and applies the generated sets of swaps based on Star+ wirelength costs. While our approach utilizes the Star+ wirelength model, it is straight-forward to incorporate other wirelength models, such as HPWL. The W-CAMIP design presented in this chapter can be implemented using any parallel framework (e.g., Cilk

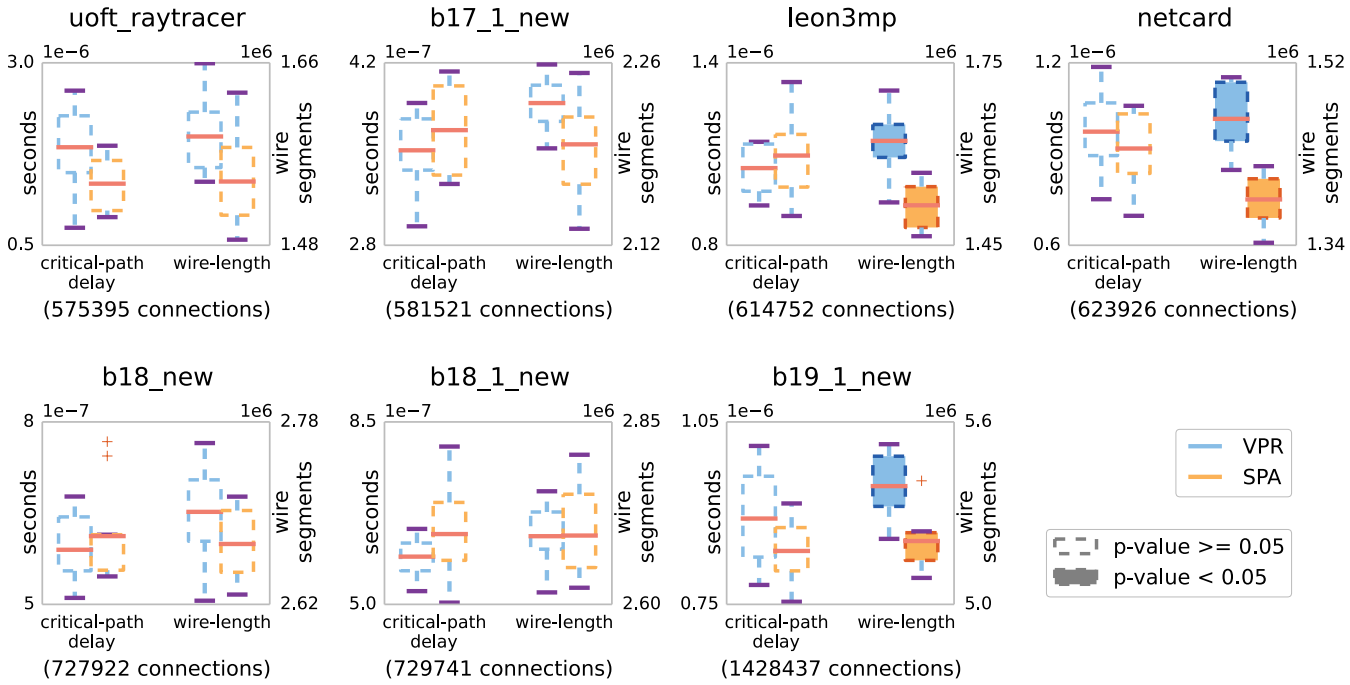


Figure 5.6: W-CAMIP post-routing results

Plus, Thrust, etc.) which provides implementations of the structured parallel patterns used. To demonstrate this, we developed a Thrust-based implementation of W-CAMIP and compared the placement runtime and solution quality of W-CAMIP running on three GPU cards of varying performance with the results from running VPR with `inner_num=1`. Our experimental results showed that not only does the absolute runtime performance of our implementation increase with GPU generation, (e.g., when moving from the GT430 to the GTX480), but it also increases along with problem size. This indicates that the parallel efficiency of our design increases as the problem size grows, exhibiting weak-scaling parallelism, unlike [34, 77, 79, 80]. Moreover, our annealer achieves greatly reduced runtimes without sacrificing quality, unlike in [79, 80], where the quality-of-result degrades as the number of parallel workers increases. In fact, as our results demonstrated, our annealer is capable of out-performing VPR in terms of post-routing wirelength and critical-path-delay on average.

Chapter 6

Parallel timing delay calculation suitable for manycore architectures

In Chapter 5, we presented a model for computing Star+ wirelength using vector and matrix operations that naturally map to the structured parallel patterns discussed in Chapter 2. Our experimental results showed that the improvements in absolute runtime of our wirelength-driven W-CAMIP parallel implementation over VPR *increased* along with the problem size on all GPU platforms we tested. This confirms our hypothesis that designing our approach to *only* use structured parallel patterns leads to predictable, scalable, runtime performance.

The positive runtime results from our wirelength-driven placer provided motivation to apply similar methods to timing analysis. The most commonly used timing analysis used in the literature is connection-based timing, as implemented in VPR's timing-driven mode [4]. As discussed in Chapter 2, there have been no scalable, parallel approaches to timing analysis proposed in the literature. In fact, the most promising deterministic parallel placement methods, while incorporating timing calculations, do not parallelize the timing portion of the code *and* do not include the time spent in timing calculations in runtimes when reporting performance improvements [33, 34]. In other words, the improvements in absolute runtime reported in the literature for such approaches are *only* taking the wirelength calculations and placement updates into account.

In this chapter, we propose a mapping of equivalent timing analysis calculations that are

used in VPR onto the efficient parallel patterns shown to be effective in our wirelength-driven, W-CAMIP placer. Using structured parallel patterns not only predicts weak-scaling parallelism, where the expected speedup scales along with the number of parallel workers p as long as the problem size grows at the rate $\Theta(p \log p)$, but also decouples the implementation of the patterns in terms of language and processing architecture from the description of the algorithm. **Note that the parallel calculations, in particular the critical-path delay calculations, may also be helpful in accelerating the timing analysis portion of existing placement methods, where timing has not been done in parallel**.

Using our proposed model of parallel timing analysis, we present a proof-of-concept timing-driven placement algorithm, which we call "T-CAMIP". T-CAMIP incorporates our scalable timing calculations into our W-CAMIP implementation from Chapter 5. Our T-CAMIP implementation is NVIDIA CUDA-based and represents **the first mapping of timing analysis for FPGA placement onto massively parallel processing architectures, where all non-constant-time operations are performed solely on the GPU, with data that remains entirely in GPU memory**.

We do not intend for our proposed T-CAMIP placer in its current form to be immediately competitive with state-of-the-art timing-driven placers that have been tuned over many years. Instead, we present results showing high absolute runtime performance, and show that critical-path delay is reduced significantly compared to our W-CAMIP placer when incorporating a connection-cost-based timing objective. As the first to introduce parallel connection-based timing analysis suitable for efficient execution on modern manycore architectures, we propose that our work provides a solid foundation for future work on the development of massively parallel timing-driven placement methods.

The remainder of this chapter is laid out as follows. Section 6.1 reviews the timing-based cost model used in VPR [4], and covered briefly in Chapter 2. In Section 6.2 we propose a method to compute the critical-path delay, while computing the longest incoming path delay and longest outgoing path delay for each block in the netlist, using the structured parallel patterns from Chapter 2. In Section 6.3, we describe a method to compute the difference in connection cost for each block moving to a new proposed position, and extend our proposed wirelength-driven simulated annealing-based placement algorithm from Chapter 5 to incorporate

a timing-based minimization objective. In Section 6.4, we demonstrate that our proposed timing calculations have an overall work complexity of $\Theta(C)$ and span complexity of $\Theta(L \log C)$, where C denotes the number of connections and L denotes the number of synchronous delay levels in the netlist, leading to an isoefficiency function in $\Theta(p \log p)$. In Section 6.5, we discuss our experimental methodology, and in Section 6.5.1 we present runtime results showing the speedup of our proposed T-CAMIP placer compared to timing-driven VPR (i.e., T-VPlace) across the IWLS benchmark netlists described in Chapter 2. Our proposed implementation is GPU-based, and demonstrates mean absolute runtime performance improvements of 31.1× on commodity GPU hardware versus VPR on a workstation CPU, made possible by utilizing structured parallel patterns for *all* non-constant-time operations. In Section 6.5.2, we also present experimental results showing that our timing-driven T-CAMIP placer significantly reduces critical-path delay by 20% on average across a large set of netlists, compared to our wirelength-driven placer, W-CAMIP.

6.1 Timing overview

In the context of FPGA architectures, the timing characteristics of a circuit depend on the delays along paths between synchronous *drivers* and synchronous *sinks*. The maximum operating frequency of a synchronous circuit is determined by the delay along the longest such path. This longest path, from a synchronous driver to a synchronous sink, is referred to as the "critical-path" of the circuit, and the delay along that path is called the "critical-path delay". When performing placement, it can be beneficial to maintain an approximate delay estimate for the synchronous paths in the circuit based on the position of the blocks. This can be accomplished by computing the longest *incoming path* from a synchronous *driver* to *each block* and the longest *outgoing path* from *each block* to a synchronous *sink*. By computing these two longest path values for every block in the netlist, the critical-path delay may be found by taking the maximum incoming delay length of all synchronous sink blocks, or the maximum outgoing delay length of all synchronous driver blocks (they are equivalent).

While the critical-path delay is valuable on its own, it is also helpful during placement to know how moving a block from one position to another will impact the delay of the circuit. One

approach to this, as proposed in T-VPlace [36], is a connection-based cost assignment, which assigns a "criticality" value to every connection in the netlist. The criticality of a connection is determined by the ratio of the longest path going through that connection versus the critical-path length. This weights the connections in the netlist based on their delay relative to the critical-path delay. To place more emphasis on the most critical connections, T-VPlace also introduces what they call a "criticality exponent", where the criticality of each connection is raised to the criticality exponent, causing the most critical connections to be emphasized more than less critical connections. When moving a block, from one position to another, the estimated impact on delay may computed by calculating the criticality of all incoming and outgoing connections of the block being moved. Then, the criticality of those connections may be reevaluated based on the proposed new position for the block, which will change the delay of the connections. The difference between these two aggregate connection costs for connections associated with the block provides an estimated difference in timing cost due to moving the block to the proposed position. T-VPlace computes the estimated difference in cost due to moving a block from one position to another by calculating a weighted sum of the estimated difference in wirelength cost and the estimated difference in timing cost. Before combining together, the wirelength cost and timing cost are both normalized against the total wirelength cost of the netlist and the total timing cost of the netlist, respectively. Normalizing the wirelength and timing values before adding them together in a weighted sum provides adaptive weighting of wirelength and timing objectives throughout the search.

In this chapter we propose a parallel method for computing:

- The longest incoming path into every block from a synchronous driver.
- The longest outgoing path from every block to a synchronous sink.
- The critical-path delay.
- The criticality ratio of every connection.
- The difference in connection criticality costs [1] based on moving a block to new proposed position.

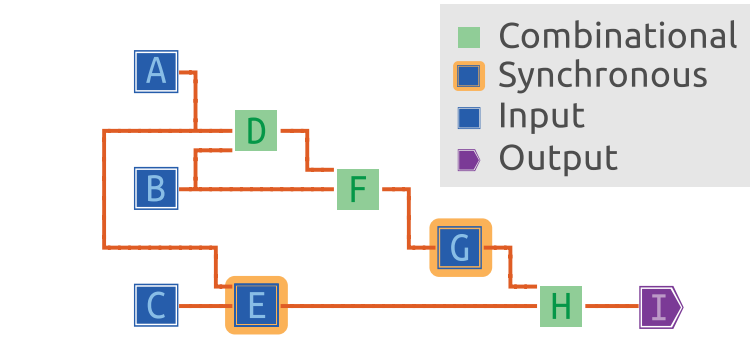
[1] We define the term "criticality cost" to refer to $\left(\frac{d_{ij}}{d_{max}}\right)^{\epsilon}$, where d_{ij} is the delay of the connection between block i and block j, d_{max} is the critical-path delay, and ϵ is the criticality exponent, as defined in Section 2.2.6.

All of the operations described in this chapter are mapped to the parallel patterns from Chapter 2 that have known work complexity and span complexity that lead to weak-scaling. In Section 6.3, we describe a timing-driven variation of our CAMIP model, using a weighted sum of the wirelength costs as computed in Chapter 5 along with the timing costs described in this chapter.

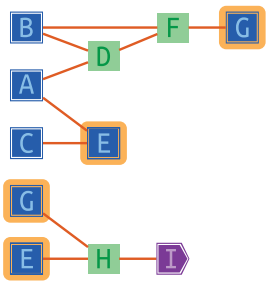
6.2 Critical-path delay

The peak performance of a circuit mapped onto a FPGA architecture is determined by the critical-path delay, which is the longest path from any synchronous driver to a synchronous sink. In this section, we describe how to compute the critical-path using structured parallel patterns from Chapter 2. To compute the critical-path delay, we divide the problem into two sub-problems. First, we compute the longest *incoming path* from a synchronous driver into every block of the netlist, and second, we compute the longest *outgoing path* from every block to a synchronous sink. As we will see, we can use the same algorithm to compute the incoming paths and outgoing paths. In the case of the incoming paths, we will begin the algorithm at the synchronous *driver* blocks and when computing the output paths we will start the algorithm from the synchronous *sink* blocks.

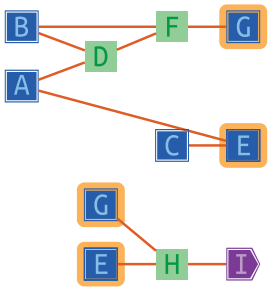
To compute the longest incoming path for every block, consider the netlist in Fig. 6.1a. In this figure, we imply that connections are moving from the left to the right, through the circuit, in that blocks A, B, and C on the left are input blocks in the netlist, block I is an output block, and blocks E and G are synchronous logic blocks. In other words, the output of block E and the output of block G are both registered by the clock. When computing incoming paths, blocks A, B and C are considered synchronous drivers, and blocks E and G are considered synchronous drivers for all of their outgoing connections. Figure 6.1b shows the topological ordering from *left to right* from synchronous driver blocks through any intermediate blocks to the terminating synchronous sink for all paths in the netlist. Similarly, Fig. 6.1c shows the topological ordering from *right to left* from synchronous sink blocks through any combinational logic to the terminating synchronous driver for all paths in the netlist.



(a) Net list



(b) Incoming paths



(c) Outgoing paths

Figure 6.1: Simple netlist example

6.2.1 Terminology

Here, we introduce some terminology to help us refer to the two sides of every block-to-block connection. We define a "source" block as the *driver* of a connection when computing *incoming paths* and the *sink* of a connection when computing *outgoing paths*. This allows us to use the same term when describing our algorithm, whether we are computing incoming paths or outgoing paths. Similarly, we refer to the block for which the delay is being calculated as the "target" block, whether we are computing incoming paths or outgoing paths. For example, when computing the longest incoming path for block D in Fig. 6.1b, there are two incoming connections to consider. For the first connection, we refer to block B as the "source" and block D as the "target". For the second connection, block D is still the target, but block A is the source. When computing outgoing connections, we follow the topological ordering in Fig. 6.1c from right-to-left. In this case, block B has two outgoing connections to consider. In the case of the first outgoing connection, we refer to block F as the source and block B as the target (recall that we are computing the longest *outgoing path* in this case). For the second outgoing

connection, block B is still the target, but block D is the source. For a list of all connections, marked as source or target with respect to *incoming paths*, see the left-most two columns in Fig. 6.2a. For a similar list with respect to *outgoing paths*, see the left-most two columns in Fig. 6.3a.

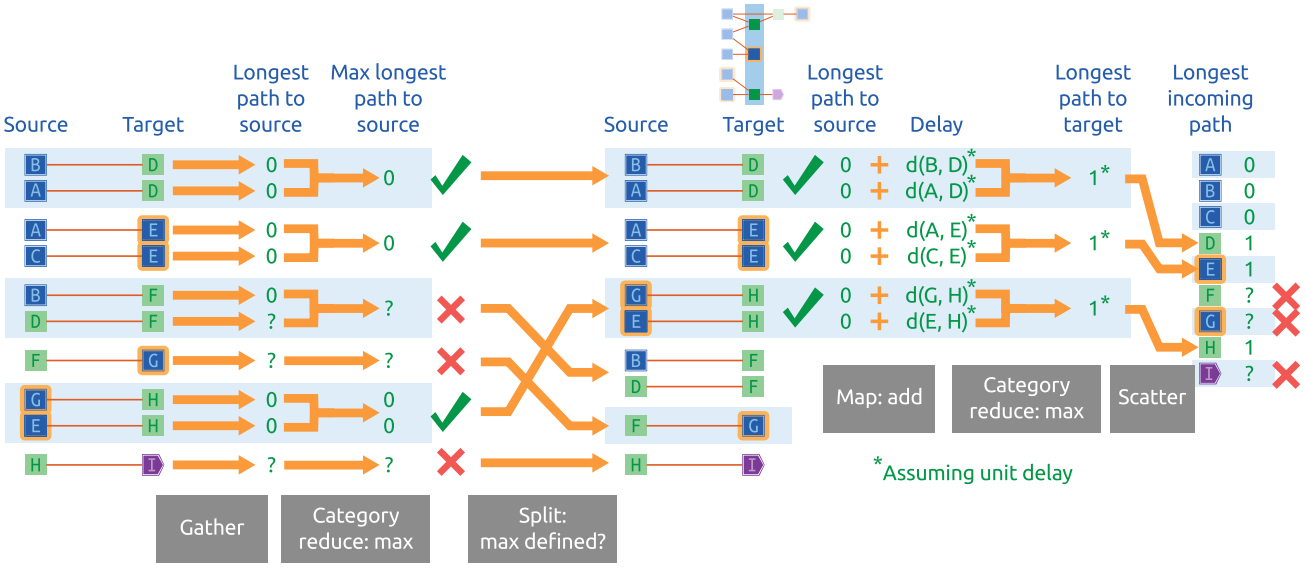
6.2.2 Longest incoming paths

To compute the longest incoming path delay of every net in the netlist, we must perform a breadth-first traversal from the synchronous driver blocks through to the synchronous sink block of every path. Figure 6.2a illustrates the breadth-first traversal process for computing the incoming path lengths of the netlist in Fig. 6.1a. Note that to compute the longest path for any given block, all blocks to the left in the topological ordering (according to 6.1b) must already have their longest incoming path delay calculated.

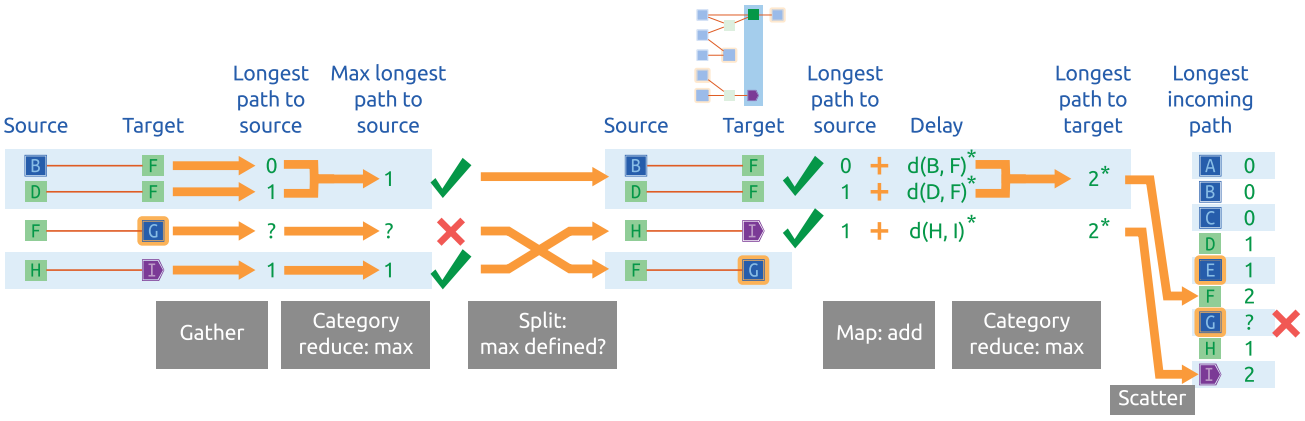
To begin the breadth-first traversal all synchronous drivers are assigned a longest-incoming path delay of zero, since they are the starting point of all synchronous paths. In Fig. 6.2a, this corresponds to blocks A, B, C, G, and E. For each target block, we compute the maximum incoming path length for the source of any incoming connection of the target. If the source of any incoming connection of a target block has not been assigned been a longest incoming path delay yet, the target block is skipped for the current step of the traversal. In the example netlist in Fig. 6.1a, the target blocks D, E, and H, all have incoming connections where the longest incoming path delay of the corresponding source block is known.

To compute the longest incoming path delay for targets D, E, and H, we then add the connection delay between each source and the corresponding target, and take the maximum such total delay for each target. The result is the longest incoming path delay for blocks D, E, and H. Note that in this example, we are assuming a unit connection delay, where the delay of every connection in the netlist is equal to 1. In practice, a look-up table is used, where the delay of a connection is based on positions of the source and target blocks. After the first step of our breadth-first traversal, blocks D, E, and H now have an assigned longest incoming path value.

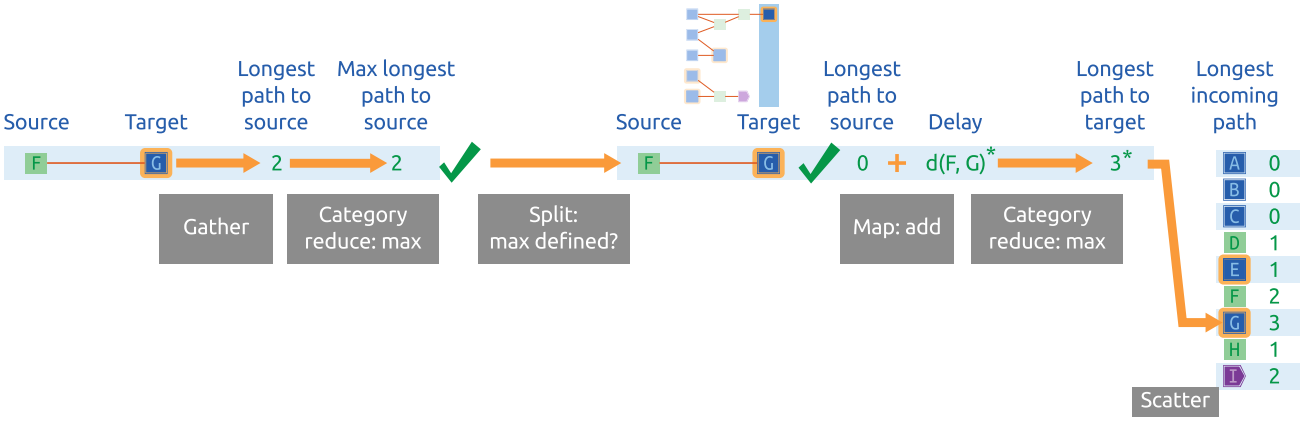
The next step of the traversal only takes into consideration incoming connections to target blocks that do not yet have an incoming path delay assigned. In our example, we look up



(a) Driver delay level 1



(b) Driver delay level 2



(c) Driver delay level 3

Figure 6.2: Calculation of longest incoming path for each block using structured parallel patterns.

the longest incoming path delay to the source of all remaining connections. In the case of target blocks F and I, the source block of all incoming connections have longest incoming path delays assigned. However, in the case of target G, the source block F has not yet been assigned an incoming path delay, and thus, we must post-pone the calculation of the longest incoming path delay for G. At this point, for every incoming connection to targets F and I, we compute the longest path length including each of the corresponding connections. Then, we compute the longest such incoming path for each target and assign that longest incoming path to the respective target accordingly. In this example, again, with unit delay, targets F and I end up with a longest incoming path delay of 2. At this point, we start the final step of our breadth-first traversal, where we have one remaining target block, G. By this step of the traversal, the longest incoming path to the source block F is known. Therefore, we add the delay of the connection between F and G to the longest incoming path of the source, F, to obtain the longest incoming path to the target G, and update the longest incoming path accordingly.

Note that the longest incoming path delays, when using unit delays for all connections, results in the left-to-right ordering shown in Fig. 6.1b. In general, we compute the longest incoming path assuming unit delays once at the beginning of placement to *a)* determine the number of delay levels in the topological ordering of the netlist, and *b)* to order the netlist connections according to the delay levels to avoid having to partition the connections in subsequent timing calculations. This enables us to skip the left three parallel operations at each delay level shown in Fig. 6.2, when computing the longest incoming path lengths, such that we only perform the map, category reduce, and scatter operations on the right for each delay level. By ordering the connections by their unit delay level, we guarantee that all required incoming path delay lengths for source blocks will be available so long as we process the delay levels one-at-a-time and in order, as shown in Fig. 6.4a.

6.2.3 Longest outgoing paths

Figure 6.3 illustrates the process of computing the longest outgoing delay paths for each block in the netlist in a similar manner to how the longest incoming path lengths were computed in Fig. 6.2. To begin computing the longest outgoing path lengths, we assign each synchronous sink a longest outgoing delay of zero, since each synchronous sink block is the terminating for

a synchronous path. In the example in Figure 6.3, target blocks C, F, and H, have all outgoing connections connected to source blocks, where each source has a longest outgoing path delay assigned. In this step of the breadth-first traversal, we take the longest outgoing path delay of the source for any of the out-going connections for the target blocks C, F, and H, and add the delay of along the corresponding connection to compute the total delay along each respective path. Given the longest delay along each outgoing connection from target blocks C, F, and H, we compute the maximum such delay for each target and assign the longest outgoing path length accordingly. In this case, assuming unit delay, blocks C, F, and H, are assigned a longest outgoing path delay of 1.

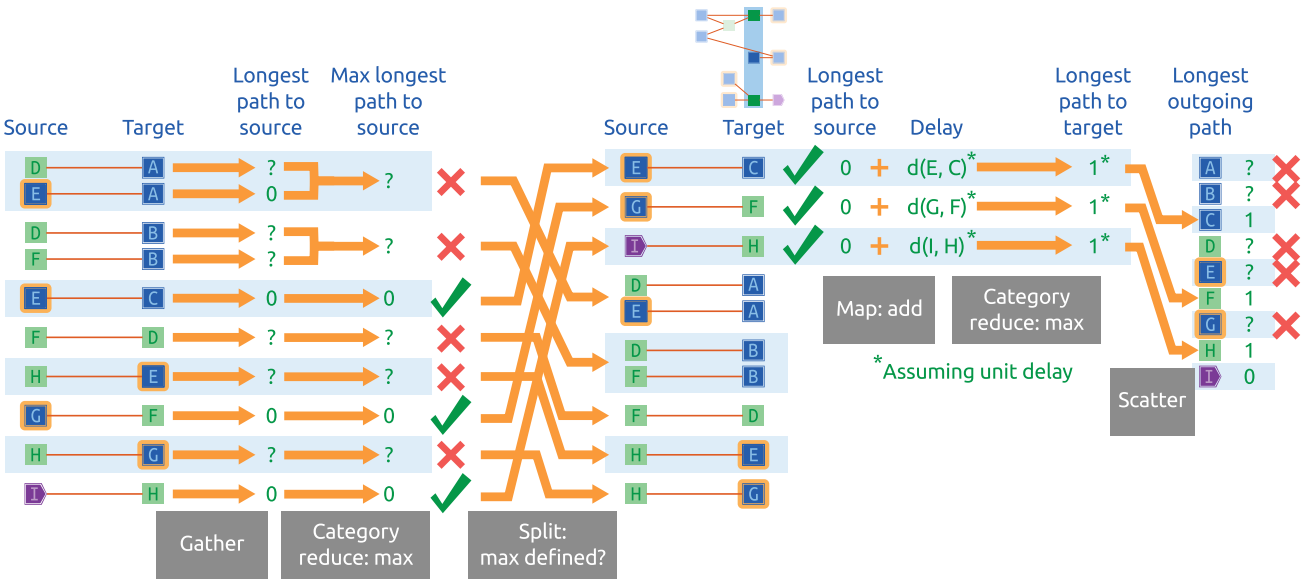
The breadth-first search continues right-to-left according to the topological ordering in Fig. 6.1c. Note that after completing the breadth-first search in Fig. 6.3, the maximum outgoing path delay is equal to the maximum incoming path delay computed in Fig. 6.2. This should always be the case.

Given the longest incoming path delay and the longest outgoing path delay for each block in the circuit, it is straight-forward to compute the delay of the longest path through any block in the circuit by simply adding the longest incoming path delay to the block to the longest outgoing path delay from the block. In the next section, we propose a method to calculate connection costs based on:

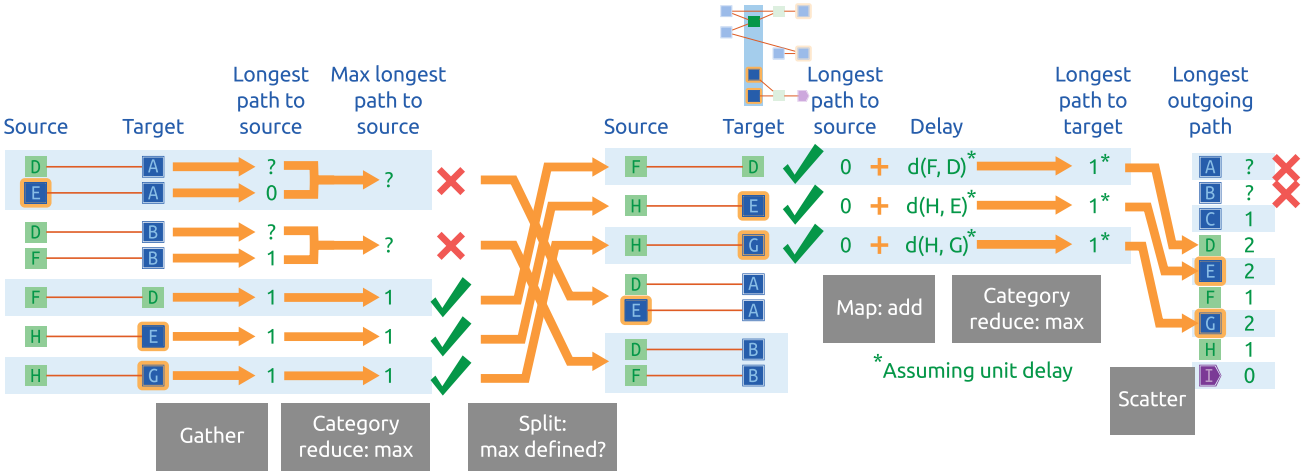
- The longest incoming path delay of each block.
- The longest outgoing path delay of each block.
- The critical-path delay of the placement.
- The position-based delay between each pair of connected blocks.

6.3 Connection costs

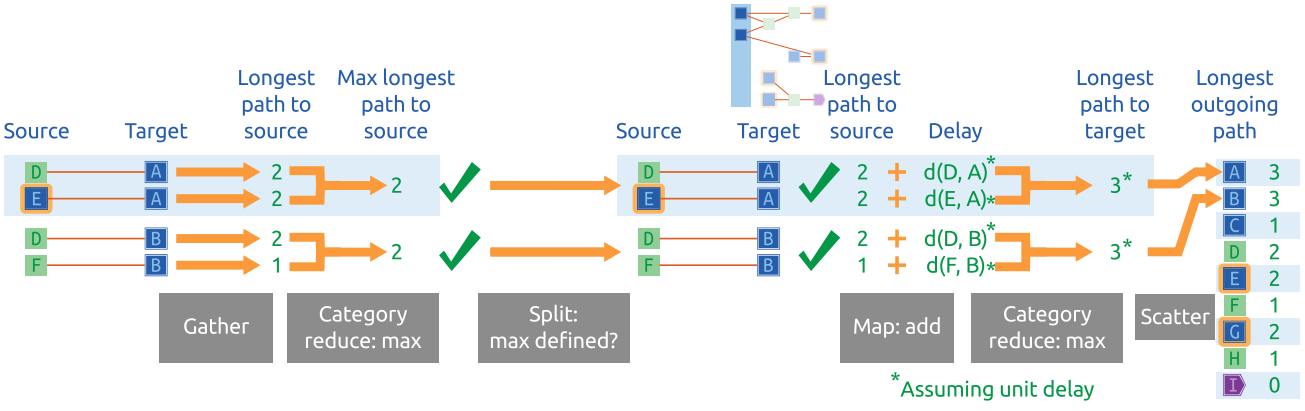
In the previous section we demonstrated how we can compute the longest incoming path and longest outgoing path delay for each block in the netlist using parallel patterns. When performing placement, we want to be able to calculate the estimated difference in timing cost due to



(a) Sink delay level 1

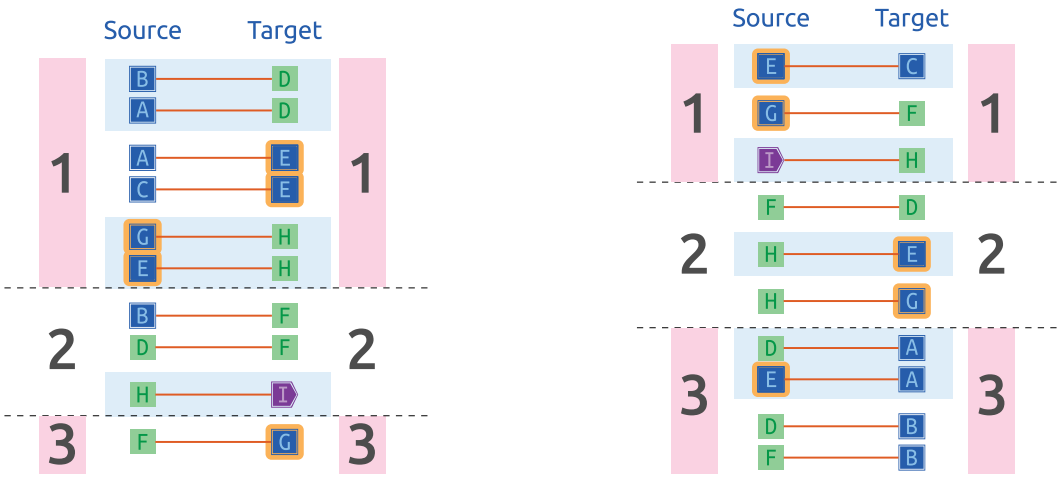


(b) Sink delay level 2



(c) Sink delay level 3

Figure 6.3: Calculation of longest outgoing path for each block using structured parallel patterns.



(a) Connections ordered according to driver delay level (i.e., unit delay arrival time).

(b) Connections ordered according to sink delay level (i.e., unit delay departure time).

Figure 6.4: The order of the connections after calculating the longest incoming path delays (as shown in Fig. 6.2) and the longest outgoing path delays (as shown in Fig. 6.3). Given the connections in order, the incoming or outgoing delays can be computed by applying only a map, a category reduce, and a scatter, so long as the delay levels are processed one at a time, and in order.

moving a block from one position to another. In this section, we demonstrate how this can be done using parallel patterns, where the result is a vector of length m where each element in the vector corresponds to the estimated difference in timing cost due to moving the corresponding block, i.

6.3.1 Prerequisite data

Here, we describe the data that is required before computing the difference in cost for each connection in the netlist based on the target block of each connection moving to a new position.

Recall that as part of the CAMIP model defined in Chapter 4, we have the vectors p_x, and p_y, as well as the vectors p'_x, and p'_y, which contain the current and proposed position of each block in the netlist, respectively. Furthermore, based on the calculations described in the previous section, let us define the vector $\vec{\ell_I} \in \mathbb{R}^m$, where each element ℓ_{Ii} corresponds to the longest incoming delay for block i. Let us also define the vector $\vec{\ell_O} \in \mathbb{R}^m$, where each element ℓ_{Oi} corresponds to the longest outgoing delay for block i.

To mark which blocks drive connections to other blocks, let us define the matrix $A \in \mathbb{M}_{mm}$, where each element is defined as shown below in Eq. 6.2. The row index is denoted by i, j is the column index, and m is the number of blocks in the netlist.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & \dots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \dots & \dots & a_{2m} \\ \vdots & & \ddots & & & \\ \vdots & & & a_{ij} & & \\ \vdots & & & & \ddots & \\ a_{m1} & a_{m2} & a_{m3} & \dots & \dots & a_{mm} \end{pmatrix} \tag{6.1}$$

$$a_{ij} = \begin{cases} 1, & \text{if block } \ i \ \text{ is sink for connection driven by block } \ j \\ 0, & \text{otherwise} \end{cases} \tag{6.2}$$

When computing an incoming connection cost where the source is a synchronous driver, the delay of the driver must be ignored, since it is treated as the start of the synchronous path. Let

us define the vector $\vec{\alpha_I} \in \mathbb{R}^m$, where each element α_{Ii} is defined as in Eq. 6.3, to mark each block i according to whether or not it is a synchronous driver.

$$\alpha_{Ii} = \begin{cases} 1, & \text{if block } \ i \ \text{ is a synchronous driver} \\ 0, & \text{otherwise} \end{cases} \tag{6.3}$$

Similarly, when computing an outgoing connection cost where the source is a synchronous *sink*, the outgoing delay of the sink must be ignored, since it is treated as the end of the synchronous path. Let us define the vector $\vec{\alpha_O} \in \mathbb{R}^m$, where each element α_{Oi} is defined as in Eq. 6.4, to mark each block i according to whether or not it is a synchronous sink.

$$\alpha_{Oi} = \begin{cases} 1, & \text{if block } \ i \ \text{ is a synchronous sink} \\ 0, & \text{otherwise} \end{cases} \tag{6.4}$$

Computing the cost of a connection requires that the delay between any two positions be defined. Let us define the function $delay\left((p_{xi}, p_{yi}),\ (p_{xj}, p_{yj})\right)$ which maps two positions (p_{xi}, p_{yi}) and (p_{xj}, p_{yj}) to a delay value. In our implementation described in Appendix D, we run the tool T-VPlace a priori to precompute timing delay look-up tables for various architectures. Note that the timing delays are only dependent on the architecture and size of the placement area and do not dependent on the netlist. The appropriate precomputed timing delay look-up table is then used by our tool during timing cost calculations.

6.3.2 Formal connection cost model

Incoming connections

Let us define the matrix $D_I \in \mathbb{M}_{mm}$, where each element d_{Iij} is defined as shown below in Eq. 6.5. In other words, d_{Iij} corresponds to the delay of longest incoming path that passes through the connection between blocks i and j where the position of blocks corresponds to the current placement. Note that in the case where the source block of the connection, j, is a

synchronous driver, d_{Iij} is set based on the delay of the connection only (the incoming delay of j is ignored).

$$d_{Iij} = \begin{cases} delay\left((p_{xj}, p_{yj}),\ (p_{xi}, p_{yi})\right), & \text{if } a_{ij} = 1 \text{ and } \alpha_{Dj} = 1 \\ \ell_{Ij} + delay\left((p_{xj}, p_{yj}),\ (p_{xi}, p_{yi})\right), & \text{if } a_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \tag{6.5}$$

Let us define the matrix $D'_I \in \mathbb{M}_{mm}$, where each element d'_{Iij} is defined as shown below in Eq. 6.6. In other words, d'_{Iij} corresponds to the delay of longest incoming path that passes through the connection between blocks i and j, where the target block, i, occupies a proposed new position.

$$d'_{Iij} = \begin{cases} delay\left((p'_{xi}, p'_{yi}),\ (p_{xj}, p_{yj})\right), & \text{if } a_{ij} = 1 \text{ and } \alpha_{Dj} = 1 \\ \ell_{Ij} + delay\left((p'_{xi}, p'_{yi}),\ (p_{xj}, p_{yj})\right), & \text{if } a_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \tag{6.6}$$

Let us denote ϵ as the *criticality exponent*, as defined in timing-driven VPR [4]. The criticality exponent is used to apply non-linear weighting to the criticality ratio of connections. Figure 6.5 illustrates the effect of various values of the criticality exponent. Note that as the criticality exponent increases, the most emphasis is placed on connections with the highest criticality ratios. Using an extension of notation to the element-wise operators used in Section 5.2.1, let us use $A^{\circ\epsilon}$ to denote a matrix element-wise power operator, such that:

$$B = A^{\circ\epsilon} \implies b_{ij} = a_{ij}^{\epsilon} \tag{6.7}$$

Moreover, recall the element-wise reciprocal from Section 5.2.1, where:

$$B = A^{\circ(-1)} \implies b_{ij} = \frac{1}{a_{ij}} \tag{6.8}$$

Given D_I and D'_I, we may compute the estimated change in timing cost of all connections aggregated by the target block of incoming paths using vector and matrix operations as shown in Eq. 6.9.

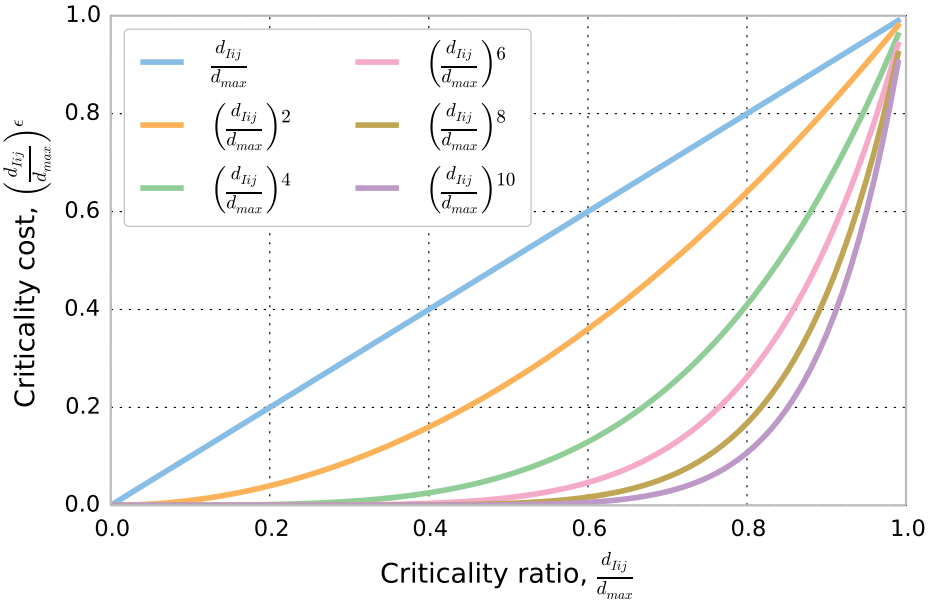


Figure 6.5: The effect of the criticality exponent, ϵ, on criticality ratio $\frac{d_{Iij}}{d_{max}}$.

$$\vec{\delta_I} = \left\{ \left(\frac{1}{d_{max}} D'_I \right)^{\circ\epsilon} \circ D'_I - \left(\frac{1}{d_{max}} D_I \right)^{\circ\epsilon} \circ D_I \right\} \vec{1_c} \tag{6.9}$$

Outgoing connections

Similar to the case of incoming connections, let us define the matrix $D_O \in \mathbb{M}_{mm}$, where each element d_{Oij} is defined as shown below in Eq. 6.10. In other words, d_{Oij} corresponds to the delay of longest *outgoing* path that passes through the connection between blocks i and j where the position of blocks corresponds to the current placement. Note that since when dealing with *outgoing* connections we consider the *driver* of each connection as the *target* block, the conditions in Eq. 6.10 are dependent on elements from the *transpose* of matrix A. In other words, each row of D_O corresponds to the *outgoing* connections of block i. Also, note that in the case where the source block of the connection, j, is a synchronous sink, d_{Oij} is set based on the delay of the connection only (the outgoing delay of j is ignored).

$$d_{Oij} = \begin{cases} delay\left((p_{xj}, p_{yj}),\ (p_{xi}, p_{yi})\right), & \text{if } a^T_{ij} = 1 \text{ and } \alpha_{Sj} = 1 \\ \ell_{Oj} + delay\left((p_{xj}, p_{yj}),\ (p_{xi}, p_{yi})\right), & \text{if } a^T_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \tag{6.10}$$

Let us also define the matrix $D'_O \in \mathbb{M}_{mm}$, where each element d'_{Oij} is defined as shown below in Eq. 6.11. In other words, d'_{Oij} corresponds to the delay of longest outgoing path that passes through the connection between blocks i and j, where the target block, i, occupies a proposed new position. Again, note that the conditions in Eq. 6.11 are dependent on elements from the *transpose* of matrix A.

$$d'_{Oij} = \begin{cases} delay\left((p'_{xi}, p'_{yi}),\ (p_{xj}, p_{yj})\right), & \text{if } a^T_{ij} = 1 \text{ and } \alpha_{Dj} = 1 \\ \ell_{Oj} + delay\left((p'_{xi}, p'_{yi}),\ (p_{xj}, p_{yj})\right), & \text{if } a^T_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \tag{6.11}$$

Given D_O and D'_O, we may compute the estimated change in timing cost of all connections aggregated by the target block of outgoing paths using vector and matrix operations, in the same way we calculated $\vec{\delta_I}$, as shown below.

$$\vec{\delta_O} = \left\{ \left(\frac{1}{d_{max}} D'_O \right)^{\circ\epsilon} \circ D'_O - \left(\frac{1}{d_{max}} D_O \right)^{\circ\epsilon} \circ D_O \right\} \vec{1_c} \tag{6.12}$$

Combined difference in connection costs

Given the two vectors $\vec{\delta_I}$ and $\vec{\delta_O}$, defined in Eq. 6.9 and Eq. 6.12, we have the difference in incoming connection costs per block and the difference in outgoing connection costs per block, respectively. We then add $\vec{\delta_I}$ and $\vec{\delta_O}$ (as shown in Eq. 6.13) to obtain $\vec{\delta_T}$ which corresponds to the total estimated difference in timing cost per block, including both incoming and outgoing connection costs.

$$\vec{\delta_T} = \vec{\delta_I} + \vec{\delta_O} \tag{6.13}$$

In the case where we are including a timing objective in the CAMIP model, let us redefine the vector $\vec{\delta_b}$ from Chapter 5 as $\vec{\delta_W}$ to denote that it corresponds to the difference in wirelength cost per block. We then define $\vec{\delta_b}$ a weighted sum of $\vec{\delta_W}$ and $\vec{\delta_T}$ to combine the wirelength and timing objectives:

$$\vec{\delta_b} = \omega \frac{\vec{\delta_W}}{W_c} + (1 - \omega) \frac{\vec{\delta_T}}{T_c} \tag{6.14}$$

where $\omega \in [0, 1]$ denotes a wirelength trade-off fraction, W_c represents the total wirelength cost for the entire netlist based on the current placement, and T_c represents the aggregate cost of all connections in the netlist based on the current placement. The ω term is a tuning parameter to explicitly adjust the search pressure between wirelength and timing objectives. Normalizing the wirelength costs and timing costs by the corresponding total netlist cost further weights the multiple-objectives adaptively, based on the approach proposed in T-Vplace [36].

The vector $\vec{\delta_b}$ defined in Eq. 6.14 corresponds to the total difference in cost per block due to moving each target block to a new position, assuming all connection sources stay in their current positions. Vector $\vec{\delta_b}$, representing both wirelength and timing objectives, is in the appropriate form to be processed by the group assessment step of our CAMIP method.

Note that all matrices in this section are typically sparse, and that all vector and matrix calculations in this section can be performed efficiently in parallel using either the map pattern or the category reduce pattern.

6.4 Work, span, and isoefficiency complexity

Table 6.1 lists the work, span, and isoefficiency complexity of the calculations related to timing in our proposed T-CAMIP implementation.

Operation	Pattern	Work	Span	Isoefficiency
Arrival times	Map	$\Theta(C)$	$\Theta(1)$	$\Theta(p)$
(Longest	Category reduce	$\Theta(C)$	$\Theta(L \log C)$	$\Theta(p \log p)$
incoming path to	Split	$\Theta(C)$	$\Theta(L \log C)$	$\Theta(p \log p)$
each block)	Scatter	$\Theta(C)$	$\Theta(1)$	$\Theta(p)$

Operation	Pattern	Work	Span	Isoefficiency
Departure times	Map	$\Theta(C)$	$\Theta(1)$	$\Theta(p)$
(Longest	Category reduce	$\Theta(C)$	$\Theta(L \log C)$	$\Theta(p \log p)$
outgoing path	Split	$\Theta(C)$	$\Theta(L \log C)$	$\Theta(p \log p)$
from each block)	Scatter	$\Theta(C)$	$\Theta(1)$	$\Theta(p)$
Per-block	Map	$\Theta(C)$	$\Theta(1)$	$\Theta(p)$
connection cost	Reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
	Category reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
Per-block move	Map	$\Theta(C)$	$\Theta(1)$	$\Theta(p)$
delta-cost	Reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
	Category reduce	$\Theta(C)$	$\Theta(\log C)$	$\Theta(p \log p)$
Total complexity		$\Theta(C)$	$\Theta(L \log C)$	$\Theta(p \log p)$

Table 6.1: Work, span, and isoefficiency complexity of calculations related to timing, based on: number of connections (C), number of synchronous delay levels (L), and number of parallel workers (p).

As shown in Table 6.1, the overall work complexity of the timing calculations is $\Theta(C)$, and the overall span complexity is $\Theta(L \log C)$, where C denotes the number of connections and L denotes the number of synchronous delay levels. The L term is required since, when computing the longest synchronous path delays, each delay level must be processed individually, and in order, as described in Section 6.2. Note, as shown in Table 2.3 (in Section 2.4), the number of levels is typically *much, much* less than the number of connections, i.e., $L \ll C$. As discussed in Section 2.3.2, based on work and span complexity assuming $L \ll C$, the timing calculations are "completely parallelizable" [64, 66] with an isoefficiency function in $\Theta(p \log p)$. Recall that an isoefficiency function in $\Theta(p \log p)$ implies weak scaling, where constant parallel efficiency can

be maintained as the number of parallel workers increases as long as the problem size grows at at the rate $\Theta(p \log p)$. Note that the connection count corresponds to the size of a netlist (i.e., the problem size is in $\Theta(C)$).

6.5 Experiments

To evaluate the efficacy of our proposed *timing-driven CAMIP (using connection-criticalities and Star+ wirelength)* design with respect to absolute runtime performance and quality-of-result, we developed a CUDA implementation using the Thrust parallel library. Within the remainder of this thesis, we will refer to our Star+ wirelength-driven CAMIP implementation as, "T-CAMIP". Appendix D describes details of the T-CAMIP implementation.

We conducted several sets of experiments using our CUDA T-CAMIP implementation to evaluate parallel runtime performance and resulting solution quality. In Section 6.5.1, we compare the absolute parallel runtime of our T-CAMIP placer running on a commodity GPU against VPR in timing-driven mode (i.e., T-VPlace) with `inner_num=1`[2] running on a workstation CPU, across the set of 7 large benchmark netlists described in Section 2.4, ranging from 575395 to 1428437 adjacency connections. In Section 6.5.2, we present results comparing our wirelength-driven W-CAMIP placer versus our timing-driven T-CAMIP placer in terms of estimated wirelength and critical-path delay. When comparing the runtime performance of our parallel T-CAMIP implementation against timing-driven VPR, we report absolute execution time of our parallel implementation running on a commodity GPU against VPR running on a commodity workstation CPU. Again, while we acknowledge that comparing across architectures is not ideal, it is not possible to run VPR directly on a GPU. Moreover, we would like to note that cross-architecture absolute runtime comparisons for FPGA placement have been reported previously in peer-reviewed studies [13, 77, 79, 80].

For each netlist in our experiments, we ran ten trials, each with a different random seed. All runs of our CAMIP placers were run on a NVIDIA GTX 980 [88] card. Note that we ran all

[2]Recall from Section 5.4 that the `inner_num` parameter controls the number of moves evaluated at each temperature in the anneal. For example, `inner_num=10` results in 10× more moves at each temperature compared to `inner_num=1`.

host-code, including VPR, using on an Intel Core i7-930 2.8GHz processor with 12 GB RAM running 64-bit Ubuntu 12.04.3. All host code was compiled using `g++` with the flags `-O3 -msse2`. All CUDA code was compiled using `nvcc` version 6.5.

6.5.1 Absolute execution time

Table 6.2 shows the mean absolute placement runtimes in seconds, using VPR in timing-driven mode with `inner_num=1` on the Intel workstation CPU described above and our T-CAMIP running on a commodity GPU (NVIDIA GTX 980) (with `inner_num=1`), across the ten trials for each of the 7 IWLS-based netlists. We can see that the T-CAMIP is significantly faster than VPR on all of the netlists. Specifically, improvements in absolute runtime range from 24.3× to 51.6×, with a mean improvement of 31.1×.

Netlist	GTX980 (s)	T-VPlace (s)	Connections	Speed-up
`uoft_raytracer`	193.5	4.8k	575.4k	25.0X
`b17_1_new`	159.9	4.2k	581.5k	26.0X
`leon3mp`	216.7	5.3k	614.8k	24.3X
`netcard`	213.7	5.6k	623.9k	26.0X
`b18_new`	190.4	6.0k	727.9k	31.4X
`b18_1_new`	186.1	6.2k	729.7k	33.2X
`b19_1_new`	306.8	15.8k	1.4M	51.5X

Table 6.2: Timing-driven placement runtime results.

Note that, as in the case of our W-CAMIP placer, Fig. 6.6 shows a strong positive correlation between the observed improvements in absolute runtime over VPR and the adjacency-list connection count. Recall that the connection-count corresponds to the size of the netlist to be placed. This indicates that as the problem size grows, our approach exhibits a larger runtime improvement indicating increased parallel efficiency, which agrees with our parallel complexity analysis suggesting weak-scaling. **This result is significant, since this is, to the best of our knowledge, the first demonstrated timing-driven placement method using scal-**

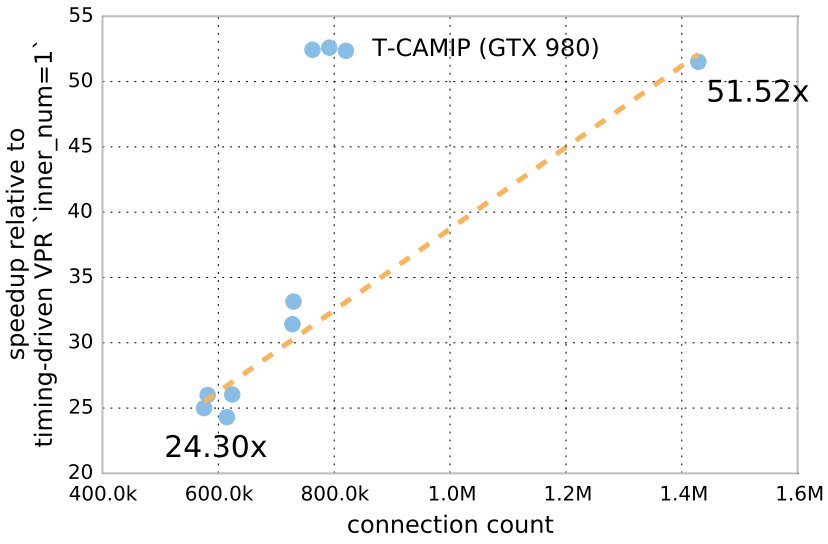


Figure 6.6: T-CAMIP absolute runtime improvement relative to VPR (*inner-num = 1, timing-driven*).

able patterns of parallel computation. In Chapter 7, we discuss future work that would make our approach compatible with a larger set of available netlists to test against to observe further increases in parallel efficiency.

6.5.2 Solution quality

Our main contribution with regards to timing is proposing the first parallel method for connection-based timing analysis that is "completely parallelizable" [64, 66] with an isoefficiency function in $\Theta(p \log p)$, based on work and span complexity analysis. In other words, our proposed approach a) is well-suited for application on manycore architectures (including GPUs), and b) predicts weak-scaling, such that the speedup is expected to increase at a fixed efficiency as parallel workers are added as long as the problem size grows at the rate $\Theta(p \log p)$. However, in this section, we demonstrate the quality improvement afforded by using our timing cost model, by presenting preliminary results showing significant improvements in critical-path delay compared to our wirelength-driven placement method proposed in Chapter 5. Note that our proposed timing-driven placement method is intended as a *proof-of-principle*, which opens the door to massively parallel timing-based placement variants. As such, it our intention to demonstrate the feasibility of incorporating our efficient parallel timing calculations into our CAMIP placement tool.

Parameter selection

While comparing two wirelength-driven placement methods to each other is straight-forward, comparison of timing-driven methods is less so, since timing-driven placement methods are typically multi-objective. For instance, in the simplest form, when considering timing, there is a trade-off between critical-path delay and wirelength. While reducing estimated critical-path delay may lead to improved maximum operating frequency, reducing estimated wirelength helps to increase the "routability" of a placement, since fewer required routing resources helps to reduce contention for the fixed resources available on the target architecture.

To further complicate the situation, the goals may be different depending on the application, and on the characteristics of the target device. Therefore, ideally we would like to have a placement method that may be tuned according to the situation at hand. In our T-CAMIP implementation, we provide the following two parameters, following the approach in VPR [4]:

Timing trade-off

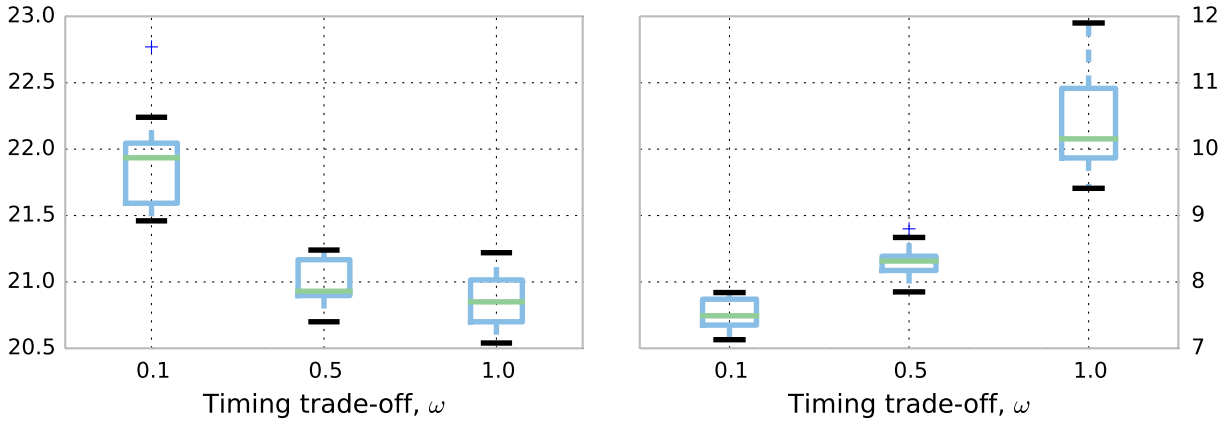
The fractional weighting of the timing cost objective relative to the wirelength objective (see ω in Eq. 6.14).

Maximum criticality exponent

The maximum criticality exponent applied to criticality ratios when computing connection timing costs (see ϵ in Eq. 6.9 and Eq. 6.12). Note that the anneal begins with $\epsilon = 1$. The ϵ value is held at one until the radius-limit starts to change, at which point ϵ is gradually increased at each temperature iteration until the maximum value is met.

To evaluate the effect of the parameters on final critical-path delay and wirelength, we placed a set of netlists with various combinations of values for ω and ϵ. Since the parameter exploration space is large, we elected to use the MCNC [55] set of benchmark netlists, which take less time to process since they are smaller than the IWLS netlists, but have been cited extensively in the literature [26, 39, 49]. Moreover, since there are more netlists in the MCNC benchmark set, we demonstrate that our results apply across a variety of circuits. Table C.1 lists the properties of the MCNC netlists.

First, we consider the timing/wirelength trade-off parameter, ω, while holding ϵ constant. Figure 6.7 shows final estimated wirelength and critical-path-delay for 10 runs at each of the



(a) Wirelength

(b) Critical-path delay

Figure 6.7: Effect of timing trade-off, α, on wirelength and critical-path delay (lower wirelength and critical-path delay values are better).

following values for ω on a representative netlist[3]: 0.1, 0.5, 1.0. Note that $\omega = 1.0$ is equivalent to running without timing analysis enabled, which is equivalent to our wirelength-driven W-CAMIP placer. As shown in Fig. 6.7a, as ω is increased from 0.1 to 1.0, the wirelength cost is reduced. In contrast, as shown in Fig. 6.7b, the critical-path delay is improved as ω is *decreased*, from 1.0 to 0.1.

For the remainder of our timing experiments, based on the results from adjusting ω, we chose a timing trade-off value of 0.5, since it provided a significant reduction in critical-path delay, without a large increase wirelength. Holding ω at 0.5, we placed the MCNC netlists with various maximum values for ϵ. Figure 6.8 shows how final estimated (normalized) wirelength and critical-path delay vary based on different maximum values of ϵ. From Fig. 6.8, we can see that increasing the maximum ϵ above 5 can help to improve both wirelength and critical-path delay. However, note that once the maximum value of ϵ is set too high, critical-path begins to increase again. Based on these results, we chose 20 as a maximum value of ϵ for the remainder of our experiments.

[3]See Fig. B.1 and Fig. B.2 for similar plots for MCNC benchmarks netlists in Table C.1

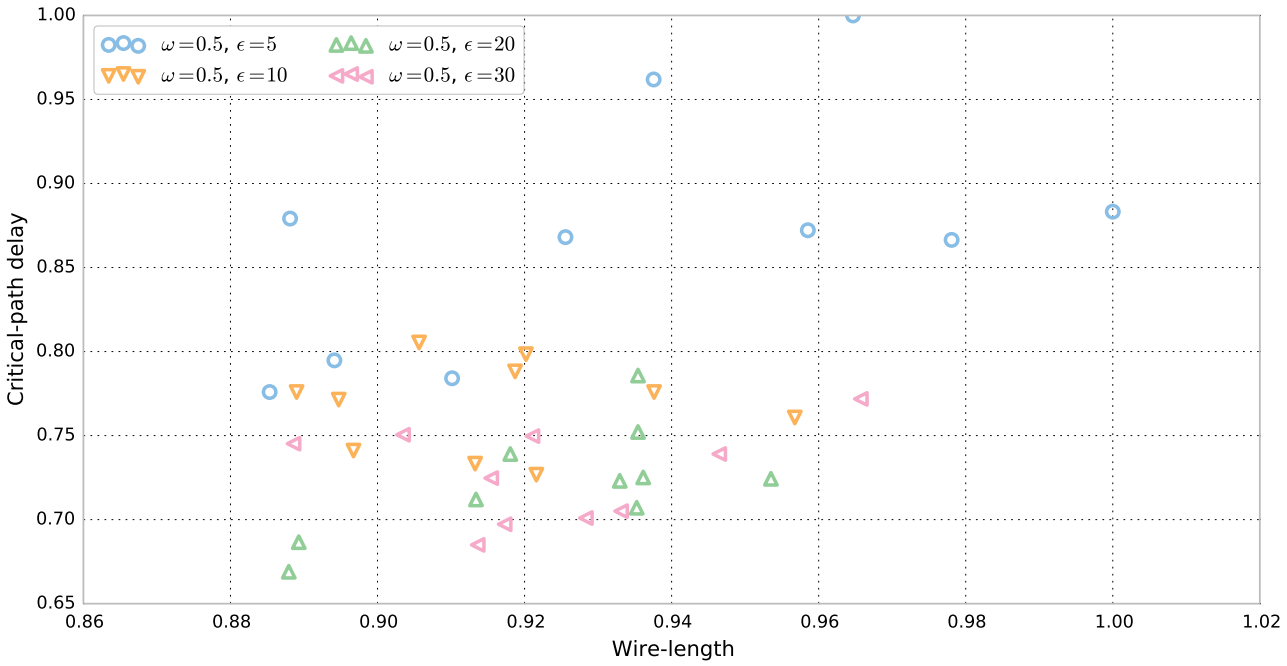


Figure 6.8: The effect of the max criticality exponent, ϵ, on final wirelength and critical-path delay.

Critical-path delay and wirelength

After empirically selecting a timing trade-off value of 0.5 and a maximum criticality exponent of 20, to establish that our timing calculations produce a significant improvement in critical-path delay, we ran both our W-CAMIP and T-CAMIP placers on the set of netlist benchmarks in Table C.1. With each placer, we ran 10 trials for each netlist starting from different random seeds.

Figure 6.9 and Table 6.3 show the wirelength and critical-path delay estimates computed by our W-CAMIP and T-CAMIP placers. Figure 6.9 shows a box-plot for each objective (wirelength and critical-path delay) for each netlist, across 10 random trials of both our wirelength-driven W-CAMIP placer and our timing-driven T-CAMIP placer.

The results in Table 6.3 are the mean costs across 10 trials for each netlist. The last two columns in Table 6.3 show the percentage difference in the resulting wirelength and critical-path-delay from W-CAMIP versus T-CAMIP, denoted as Δ_{wl} and Δ_{cpd}, respectively.

Netlist	W-CAMIP		T-CAMIP		% diff	
	μ_{wl}	μ_{cpd}	μ_{wl}	μ_{cpd}	Δ_{wl}	Δ_{cpd}
ex5p	18.1k	9.8n	18.4k	8.5n	**1.4**	**-14.0**
tseng	10.7k	7.9n	11.3k	6.2n	**5.7**	**-21.9**
apex4	20.5k	10.2n	20.7k	8.4n	**1.3**	**-17.5**
misex3	21.1k	9.0n	21.7k	7.7n	**2.8**	**-14.7**
alu4	20.6k	9.5n	21.0k	8.2n	**1.9**	**-14.4**
diffeq	16.4k	9.6n	17.6k	7.1n	**7.2**	**-26.5**
seq	28.0k	9.7n	28.5k	8.1n	**1.8**	**-16.7**
apex2	30.6k	10.5n	31.2k	9.3n	**1.8**	**-11.6**
s298	21.7k	20.5n	22.9k	15.4n	**5.3**	**-24.8**
frisc	64.9k	23.2n	66.6k	14.9n	2.7	**-35.8**
elliptic	53.0k	21.4n	57.1k	12.4n	**7.8**	**-41.8**
spla	69.6k	15.6n	72.1k	14.3n	**3.6**	**-7.9**
ex1010	69.0k	17.5n	70.6k	14.9n	**2.4**	**-14.7**
pdc	99.9k	20.3n	100.9k	16.0n	**1.0**	**-21.6**
s38584.1	67.1k	10.4n	71.9k	10.7n	**7.2**	2.2
s38417	68.6k	16.0n	73.1k	10.2n	**6.6**	**-36.0**
clma	146.6k	21.9n	155.8k	17.4n	**6.2**	**-20.7**
Sum	826.4k	243.0n	861.5k	189.5n		
Mean					3.9	-19.9

Table 6.3: Timing-driven (T-CAMIP) wirelength and critical-path delay versus wirelength driven (W-CAMIP).

Note that a negative value in either the Δ_{wl} column or the Δ_{cpd} column indicates that T-CAMIP improved upon the corresponding metric from W-CAMIP. Moreover, bold-face values in the right-most two columns indicate a statistically significant difference (determined using Wilcoxon signed-rank test with $\alpha = 0.05$). As shown in both Fig. 6.9 and Table 6.3, our

T-CAMIP placer improved mean critical-path delay by 19.9%, while only increasing mean wirelength by 3.9%. Furthermore, critical-path delay is improved in all but one case, in which case the difference between the W-CAMIP and T-CAMIP results is not statistically significant (p-value of 0.5).

6.6 Summary

In this chapter we presented, to best of our knowledge, the *first* scalable timing analysis for FPGA placement, targeting massively parallel architectures. Our proposed algorithms for computing critical-path delay and connection-based timing costs, which are equivalent to those computed in timing-driven VPR [4], are based on structured parallel patterns with known work and span complexity, resulting in expected weak-scaling behaviour, where parallel efficiency can be fixed as the number of parallel workers increases as long as the problem size grows at the rate $\Theta(p \log p)$. Our experimental results demonstrated the highly parallel nature of our approach allowing mean improvements in absolute runtime of 31.1× on commodity GPU hardware versus VPR running on a workstation CPU, where the relative runtime improvement increases significantly along with the size of the problem. It is important to note, that our proposed timing calculations may be incorporated into existing placement methods to provide parallel timing analysis. This is of particular interest for existing parallel placement methods that currently provide no parallel support for timing analysis [33, 34].

We also demonstrated that our proposed parallel timing calculations are compatible with our *Concurrent Associated-Moves Iterative Placement (CAMIP)* model, and presented a timing variation of the W-CAMIP placer from Chapter 5, which we call "T-CAMIP". We proposed our T-CAMIP as a proof-of-concept, parallel, scalable, timing-driven placement tool. Our experimental results established that by incorporating a connection-cost-based objective into our CAMIP model, we were able to significantly improve critical-path delay across a variety of benchmark netlists, with a mean improvement of 19.9% over our W-CAMIP placer. We propose that our work presented in this chapter provides a strong foundation for future work on developing massively parallel, timing-driven FPGA placement methods. An important extension of this work would be to add support for heterogeneous architectures, as we discuss as future work in

Chapter 7.

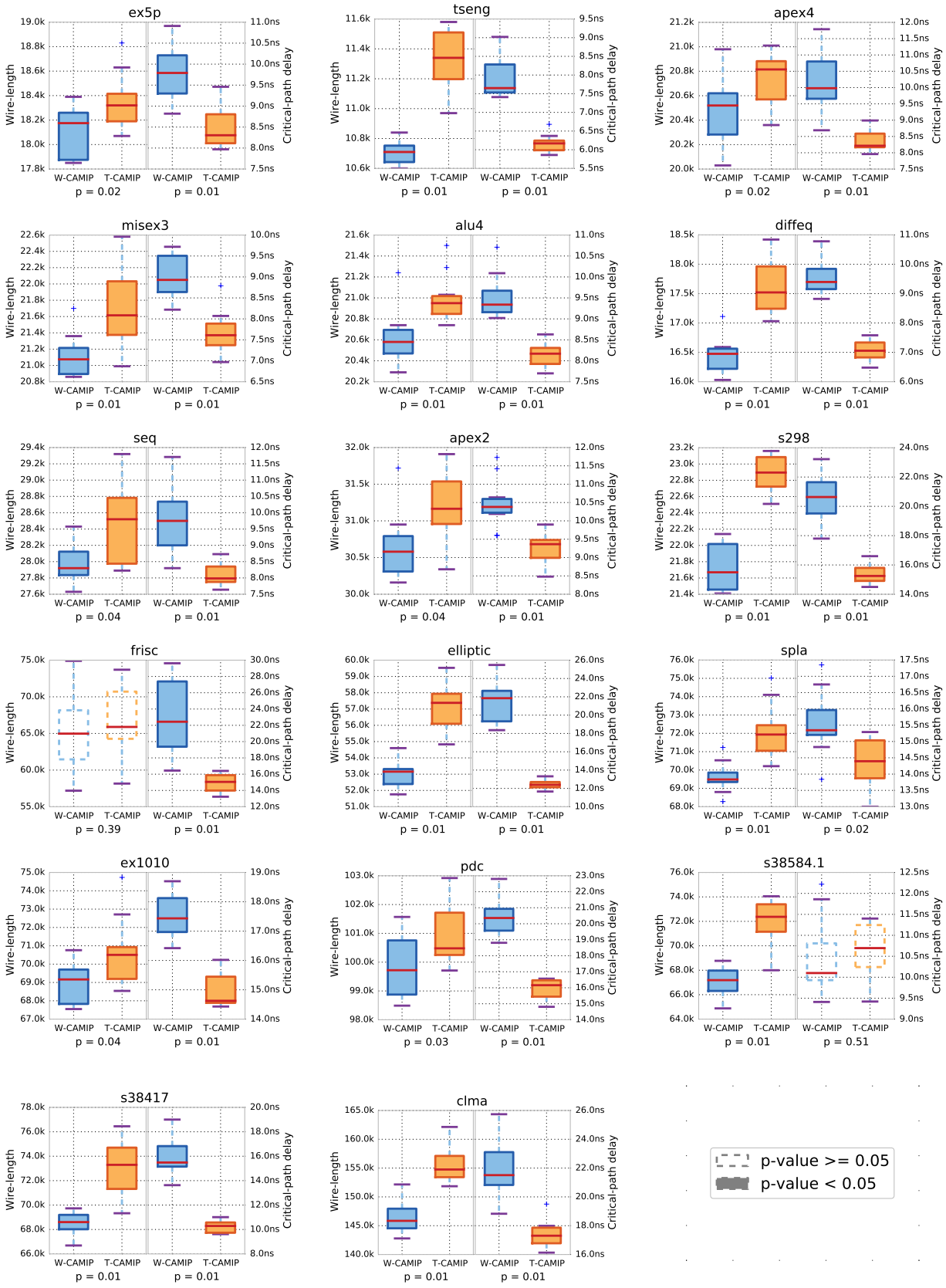


Figure 6.9: W-CAMIP versus T-CAMIP: Estimated wirelength and critical-path delay.

Chapter 7

Conclusion

In this thesis, we proposed the first "completely parallelizable" [64, 66] FPGA placement method with work and span complexity predicting weak-scaling parallelism according to the isoefficiency function $\Theta(p \log p)$. As the number of parallel workers increases the speedups are also expected to grow as linearly as long as the problem size grows at the rate $\Theta(p \log p)$. Furthermore, by using best known parallel patterns we made deterministic execution possible. Determinism is important, not only because of ease of debugging, but more importantly, it ensures that the quality of placement *will not change as more parallel workers are added*, unlike in most parallel FPGA placement methods proposed in literature [33, 77, 79, 80].

To enable the massive amounts of parallelism exposed in our approach, we proposed a novel, pattern-based mechanism for generating very large sets of moves where the size of the sets grows along with the size of the netlist. Moreover, moves within each set are guaranteed to be free of hard conflicts. This method of move generation is the first of its kind and is the key enabling factor for our parallel placement methodology. We also presented the first implementation of timing-based cost calculations which run entirely on a GPU. Table 7.1 summarizes the key contributions presented in this thesis.

In Chapter 4, we presented our *Concurrent Associated-Moves Iterative Placement* methodology, where we defined parallel placement in terms of six key steps:

Contribution	Features
Concurrent Associated Moves Iterative Placement (CAMIP) Model	• Work complexity of $\Theta(B)$. • Span complexity of $\Theta(\log B)$. • Isoefficiency complexity of $\Theta(p \log p)$. • All non-$\Theta(1)$ operations GPU-compatible.
Associated Move-Pair Patterns	• Move for each block can be computed independently in $\Theta(1)$. • Suitable for application using parallel map.
Parallel Star+	• Work complexity of $\Theta(C)$. • Span complexity of $\Theta(\log C)$. • Isoefficiency complexity of $\Theta(p \log p)$. • All non-$\Theta(1)$ operations GPU-compatible.
Wirelength CAMIP	• Work complexity of $\Theta(C)$. • Span complexity of $\Theta(\log C)$. • Isoefficiency complexity of $\Theta(p \log p)$. • All non-$\Theta(1)$ operations GPU-compatible. • Mean speedup of 19× over VPR. • Improved wirelength and critical-path by 5% over VPR.
Parallel Timing	• Work complexity of $\Theta(C)$. • Span complexity of $\Theta(L \log C)$ – L is the number of synchronous delay levels in the netlist. • Isoefficiency complexity of $\Theta(p \log p)$. • All non-$\Theta(1)$ operations GPU-compatible.

Table 7.1: Summary of contributions.

1. Placement Evaluation.
2. Stop criteria assessment.
3. Move proposal.
4. Move evaluations.
5. Associated-move group assessment.
6. Placement update.

We presented our novel *Associated Move-Pair Pattern (AMPP)* method for generating very large sets of non-conflicting moves, and described how these moves can be assessed on a per-group basis to accept or reject all moves within an associated subset to ensure a valid placement is maintained. All non-objective cost steps in our CAMIP algorithm model are mapped to structured parallel patterns, which provide a language-free, architecture-free, efficient parallel description of our algorithm. However, by using best known parallel patterns, with known work complexity and span complexity, we reasoned about the scalability of our approach with respect to isoefficiency [2]. Not including the objective cost calculations, our CAMIP model has a work complexity of $\Theta(B)$ and a span complexity of $\Theta(\log B)$, where B is the number of blocks in the netlist, resulting in an isoefficiency function in $\Theta(p \log p)$. The only parallel FPGA placement approaches proposed in the literature which are deterministic and provide high quality of results, competitive with serial annealing algorithms, do not exhibit weak-scaling since the span grows relative to the size of the problem. This is due to the mechanism used for detecting and repairing move conflicts, whereas in our approach we used a pattern to generate moves, thus avoiding any possibility of hard conflicts from the start.

In Chapter 5, we presented a method for computing all required wirelength-based calculations necessary for incorporation into our CAMIP placement model. These calculations included computing the Star+ [82] cost for each net in the netlist, computing the entire netlist cost, and calculating the difference in cost for moving any number of blocks in the netlist, where the difference in cost for each block is calculated as if no other block is moving. Our proposed parallel wirelength cost calculations have an overall work complexity of $\Theta(C)$ and span complexity of $\Theta(\log C)$, where C is the number of connections in the netlist, resulting in a corresponding

isoefficiency function in $\Theta(p \log p)$. With all of these calculations mapped onto parallel patterns, we developed a GPU-based implementation of our proposed wirelength-driven algorithm, "W-CAMIP", using our pattern-based move generation method from Chapter 4. To establish the efficacy of our wirelength-driven algorithm, we ran VPR [4] to perform place and route on the set of benchmark netlist circuits described in Chapter 2. We then placed the same netlists with our proposed W-CAMIP implementation and routed the results using VPR. For each netlist we ran VPR and W-CAMIP for 10 random trials each, and compared VPR against W-CAMIP in terms of absolute runtime and post-routing wirelength and critical-path delay. Note that we ran VPR in bounding-box mode with `inner_num=1`, and we ran W-CAMIP, also with `inner_num=1`, and using an annealing schedule identical to VPR. Our experimental results showed that our proposed W-CAMIP algorithm improved post-routing critical-path delay and wirelength by approximately 5%, while achieving a mean improvement in absolute runtime of 19×. Note that, as predicted based on the work and span complexity of our algorithm, absolute runtime improvements increased significantly along with the size of the netlist, indicating parallel efficiency increased along with the problem size.

In Chapter 6, we presented parallel methods, composed from the parallel patterns discussed in Chapter 2, to compute the following timing cost calculations:

1. Critical-path delay.
2. The longest incoming path delay for each block in the netlist.
3. The longest outgoing path delay for each block in the netlist.
4. The connection cost for every connection in the netlist.
5. The difference in cost, per block, based on moving each block to a new proposed position, where the connection costs for each block are calculated as if no other block is moving.

These calculations resulted in a list of per-block cost differences based on a set of proposed moves, which may be incorporated into our CAMIP model to optimize for a timing delay objective. Our proposed parallel timing costs calculations have an overall work complexity of $\Theta(C)$ and span complexity of $\Theta(L \log C)$, where C is the number of connections in the netlist and L

is the number of synchronous delay levels, leading to an isoefficiency function in $\Theta(p \log p)$. To establish the absolute runtime performance and the effectiveness of adding a timing objective to our placement method, we developed a multi-objective GPU-based implementation of our CAMIP model, "T-CAMIP", using a weighted sum of the timing connection costs and our wirelength cost differences. We ran VPR [4] in timing-driven mode on a workstation CPU to place all of the netlists in the IWLS-based benchmark set. Then we ran our proposed T-CAMIP on a commodity GPU using the same temperature schedule and targeting the same number of moves per temperature as VPR. Our experimental results show absolute runtime improvements between $24-51\times$ (mean $31\times$) for our parallel GPU T-CAMIP implementation over timing-driven VPR running on a workstation CPU. To establish the effect of adding a connection-based timing objective on quality, we ran our W-CAMIP and T-CAMIP across netlists from the MCNC (Table C.1) benchmark set. Our results show a mean 20% improvement of estimated critical-path delay compared to our wirelength-driven placer, while only increasing wirelength by 3.9%.

7.1 Future work

There are several direct branches of research extending from the contributions presented in this thesis.

Support for heterogeneous blocks

The implementations of our wirelength-driven and timing-driven *concurrent associated-moves iterative placement* algorithms presented in Chapters 5-6 only support netlists containing input blocks, output blocks, and logic blocks. However, it is straight-forward to extend our CAMIP methodology to handle any number of types of blocks, *e.g.*, multipliers, digital-signal processing blocks, block memory, etc. Updating the wirelength-driven and timing-driven CAMIP implementations to handle these types of blocks will make it possible to place a much larger set of benchmark netlists, since the latest VTR package is capable of taking modern designs including heterogeneous elements all the way from Verilog[1] through to placement and on to routing. Since there are many reference Verilog

[1] A general purpose hardware description language.

designs available, this opens the door to much larger benchmark sets to tune and evaluate our CAMIP model against.

Alternative move pair patterns

Although the method that we proposed for creating *Associated Move-Pair Patterns (AMPPs)* effectively generated large numbers of non-conflicting swaps, our CAMIP model is compatible with any mechanism that generates associated-moves, where associated-moves are those that must be applied together in an "all-or-nothing" fashion for a placement to remain valid. While most iterative placement methods proposed in literature only consider single moves or swaps, some proposed work has shown promising results for alternative move selection. For instance, in [27], the authors propose several strategies for what they refer to as "directed moves". While the presented results show significant improvements in quality by using the proposed directed moves strategies, the added runtime limited the utility of their approach. However, it may be possible to integrate the directed moves strategy to generate moves in a pattern-based approach similar to our associated move-pair patterns. This could lead to improved quality in our CAMIP implementations by simply using a different strategy for move selection, while the rest of the implementation would stay the same including objective-cost calculation.

Potential timing improvements

In Chapter 6, we presented a parallel method for calculating the longest incoming path and outgoing path for each block in the placement, using structured parallel patterns which exhibit weak-scaling. We also presented a method to compute connection-based costs equivalent to those computed by VPR [4]'s timing-driven mode. However, since operations such as sort are much less expensive when implemented in parallel, new parallel timing-based strategies may be investigated to potentially improve solution quality. For example, while computing the longest incoming path and the longest outgoing path of each block, it would be trivial to keep track of the driver and sink of the associated synchronous path, along with the closest block on each path. Based on this information, move selections could be guided based on, *e.g.*, the frequency with which each block falls on any of k longest paths in the circuit. Such calculations would likely prove too expensive to warrant implementing serially. However, in parallel, these calculations are of the same complexity

as the rest of the algorithm, and thus, may provide improvements in quality while not impacting overall scalability.

Target other parallel platforms

While we presented implementations of our CAMIP model targeting the NVIDIA CUDA-GPU platform, our key contribution is the mapping of iterative placement onto well-defined, best-known, patterns of parallel computation. One of the benefits of using such patterns is that they may be implemented using one of many parallel platforms. Therefore, our proposed algorithms for wirelength-driven and timing-driven placement could be mapped to other parallel frameworks such as Intel Threading Building Blocks [76] or Cilk Plus [75] to target multicore architectures or Intel's MIC manycore architecture. Also, specialized libraries specifically targeting NVIDIA GPUs are known to provide higher performance implementations of certain parallel patterns than are available in the Thrust library [89, 90]. By porting our implementations to use these high-performance GPU-specific libraries, there may be room for even more significant improvements in absolute runtime compared to VPR than we report in our results in this thesis. Note, however, that improvements are not expected to increase by more than a constant factor since the scalability of our proposed CAMIP algorithms with respect to the isoefficiency function will remain unchanged.

Apply variation of CAMIP to other problem domains

While the problem that we targeted in this thesis was FPGA placement, our CAMIP model could easily be modified to optimize for other combinatorial optimization problems that may be encoded as a permutation. Other design automation problems may benefit from a similar approach, such as FPGA routing or ASIC placement and routing. Furthermore, many important operational research problems may be modelled as a variation of the *Travelling Salesman Problem (TSP)*. By modifying our CAMIP model, we could use a simulated annealing approach to optimize for the TSP. If this proves to be effective, it would provide a scalable algorithm for many important real-world problems.

Appendix A

Author publications

The following is a list of publications relating to FPGA placement and the work in this thesis.

Christian Fobel and Gary Grewal. "A parallel Steiner tree heuristic for macro cell routing". In: *IEEE International Conference on Computer Design*. 2008, pp. 27–33. DOI: 10.1109/ICCD.2008.4751836.

Christian Fobel, Gary Grewal, and Andrew Morton. "Hardware accelerated FPGA placement". In: *Microelectronics Journal* 40.11 (2009), pp. 1667–1671. DOI: 10.1016/j.mejo.2008.09.008.

Christian Fobel, Gary Grewal, and Deborah Stacey. "Using GPUs to accelerate FPGA wire-length estimate for use with complex search operators". In: *24th Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2011, pp. 001129–001134. DOI: 10.1109/CCECE.2011.6030638.

Christian Fobel, Gary Grewal, and Deborah Stacey. "GPU-accelerated wire-length estimation for FPGA placement". In: *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 14–23. DOI: 10.1109/SAAHPC.2011.16.

Robert Collier, Christian Fobel, Gary Grewal, and Mark Wineberg. "Depictions of genotypic space for evaluating the suitability of different recombination operators". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 609–616. DOI: 10.1145/2330163.2330250.

Robert Collier, Christian Fobel, Laura Richards, and Gary Grewal. "A formal and empirical analysis of recombination for genetic algorithm-based approaches to the FPGA placement problem". In: *25th IEEE Canadian Conference on Electrical and Computer Engineering*. IEEE, 2012, pp. 1–6. DOI: 10.1109/CCECE.2012.6334856.

Christian Fobel, Gary Grewal, Robert Collier, and Deborah Stacey. "GPU approach to FPGA placement based on Star+". In: *2012 IEEE 10th International New Circuits and Systems Conference (NEWCAS)*. 2012, pp. 229–232. DOI: 10.1109/NEWCAS.2012.6328998.

Robert Collier, Christian Fobel, Ryan Pattison, Gary Grewal, Shawki Areibi, and Peter Jamieson. "Advancing genetic algorithm approaches to field programmable gate array placement with enhanced recombination operators". In: *Evolutionary Intelligence* (2014), pp. 1–18. DOI: 10.1007/s12065-014-0114-6.

Christian Fobel, Gary William Grewal, and Deborah Stacey. "Forward-scaling, serially equivalent parallelism for FPGA placement". In: *Great Lakes Symposium on VLSI 2014*. Ed. by Joseph R. Cavallaro, Tong Zhang, Alex K. Jones, and Hai Helen Li. ACM, 2014, pp. 293–298. DOI: 10.1145/2591513.2591543.

Christian Fobel, Gary Grewal, and Deborah Stacey. "A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and GPU architectures". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927481.

Appendix B

Tuning timing trade-off, ω

In this chapter, we show the effect of adjusting the timing trade-off parameter on *a)* the estimated wirelength, and *b)* the estimated critical-path delay, for each of the MCNC benchmark netlists (see Appendix C) placed using our T-CAMIP implementation proposed in Chapter 6. Each box-plot corresponds to 10 placement runs using our T-CAMIP implementation, each starting with a different random seed. As discussed in Section 6.3.2 and Section 6.5 , as ω increases, more emphasis is placed on the wirelength objective during placement, resulting in a typical reduction in the final estimated wirelength. In contrast, as ω decreases, more emphasis is instead placed on the timing delay objective during placement, typically resulting in a lower final estimated critical-path delay.

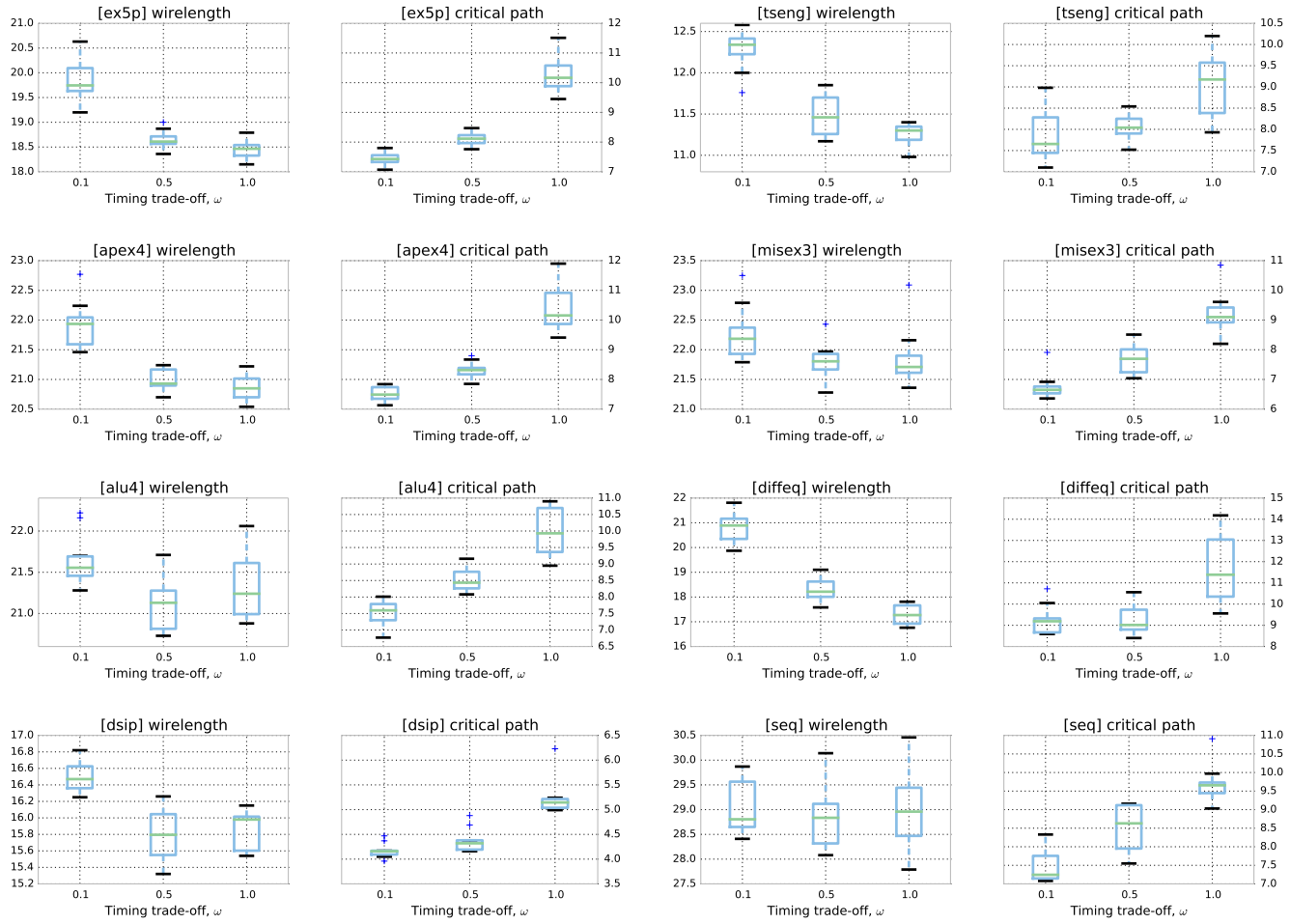


Figure B.1: W-CAMIP versus T-CAMIP: Estimated wirelength and critical-path delay.

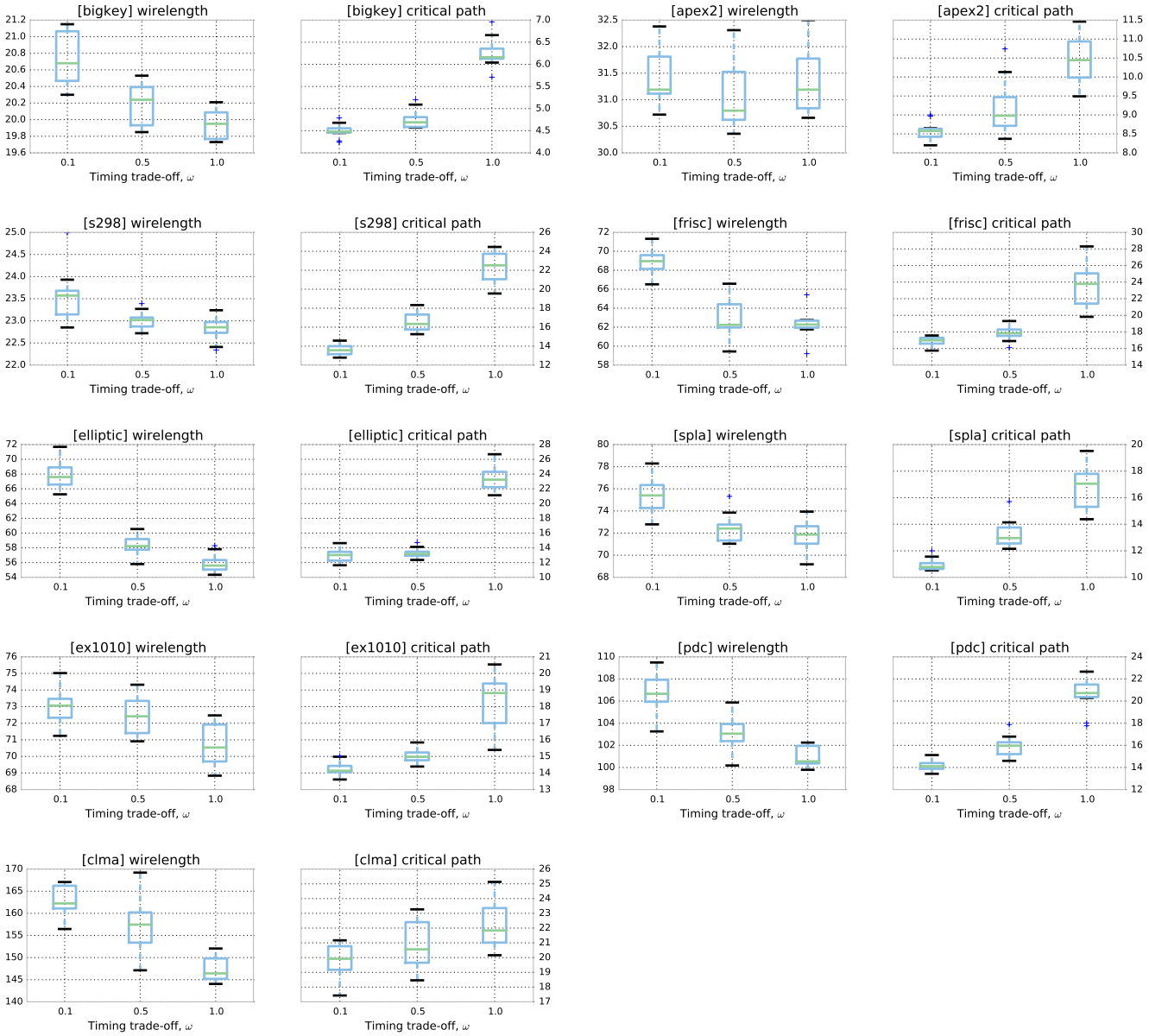


Figure B.2: W-CAMIP versus T-CAMIP: Estimated wirelength and critical-path delay (continued).

Appendix C

MCNC benchmarks

In Table C.1 below, we list details of the MCNC [55] benchmark netlists used in the experiments discussed in Section 6.5.2.

Netlist	Connections	Inputs	Outputs	Logic blocks	Delay levels	Mean fanout
alu4	6944	14	8	1522	8	4.5
apex2	8608	38	3	1878	9	4.5
apex4	5750	9	19	1262	7	4.5
bigkey	8473	229	197	1707	4	4.4
clma	38940	62	82	8383	16	4.6
des	7957	256	245	1591	7	4.3
diffeq	7234	64	39	1497	14	4.6
dsip	7468	229	197	1370	4	4.7
elliptic	17491	131	114	3604	18	4.7
ex1010	20686	10	10	4598	9	4.5
ex5p	5074	8	63	1064	8	4.7
frisc	17234	20	116	3556	23	4.8
misex3	6379	14	14	1397	8	4.5
pdc	21784	16	40	4575	10	4.7
s298	8894	4	6	1931	15	4.6
s38417	29242	29	106	6406	11	4.5

Netlist	Connections	Inputs	Outputs	Logic blocks	Delay levels	Mean fanout
s38584.1	28585	38	304	6447	10	4.4
seq	7984	41	35	1750	8	4.5
spla	17514	16	46	3690	9	4.7
tseng	5244	52	122	1047	13	4.8

Table C.1: MCNC benchmark netlists [55].

Appendix D

Implementation details

In this chapter, we describe pertinent details of the implementation used to conduct the experiments discussed in Section 5.4 and Section 6.5. Note that both W-CAMIP and T-CAMIP are under active development, where the latest source code and documentation (including input and output file format descriptions) can be found at: `https://github.com/cfobel/camip`. All parallel operations in W-CAMIP and T-CAMIP are written in C++ using the Thrust library[72]. The high-level serial execution flow expressed in the pseudocode in Section D.2 is written in Python, where Cython[91] is used to call the underlying C++ code to carry out parallel calculations on the GPU. The project can be installed using the included `setup.py`[92] file, which also compiles the C++ source code as Cython extension modules.

The remainder of this chapter is laid out as follows. Section D.1 describes the data structures used in the implementations of W-CAMIP and T-CAMIP associated with our results presented in Chapter 5 and Chapter 6, respectively. In Section D.1.3, we show empirical evidence demonstrating that the space complexity of both our W-CAMIP implementation and T-CAMIP implementation is $\Theta(C)$, where C is the number of connections in the netlist. In Section D.2, we present pseudocode for our implementation of the CAMIP algorithm we proposed in Chapter 4. We describe criteria for a valid placement in Section D.3 and present pseudocode to verify that a placement is valid. In Section D.4, we present C++ source code for an example program using the Thrust library[72], including compilation instructions for targeting a host CPU only (no GPU used) or an NVIDIA GPU. We would like to stress again that although we have used the Thrust library[72] to code our W-CAMIP and T-CAMIP implementations, the

base parallel patterns are not restricted to any particular architecture or implementation. In other words, our CAMIP algorthm could be ported to any parallel architecture or library, as long as the parallel patterns listed in the following sections are available: Section 4.4, Section 5.3, and Section 6.4.

D.1 Data structures

Each data structure described in Section D.1.1 to Section D.1.2 contains a set of arrays. Within each structure, unless otherwise noted, all arrays are of the same length. Each structure groups together arrays that are related to a particular group of elements from the netlist, i.e., blocks, nets, connections. We list the space complexity for each structure, and provide evidence in Section D.1.3 that both our W-CAMIP implementation and T-CAMIP implementation have $\Theta(C)$ space complexity, where C is the number of connections in the netlist.

There are a few points of interest regarding the data structures described in this section. All floating point arrays in our CAMIP implementations use a 32-bit floating point data type, which is the same as in VPR 5.0[4]. Note that 32-bit data types were selected throughout because, a) larger ranges are not needed to represent any present FPGA technologies, b) GPUs are optimized for single precision floating point calculations, and c) greater precision is not needed for this application. Moreove, each array in the data structures listed in this section corresponds to a `thrust::device_vector<...>`[72] instance. Perhaps most importantly, array data transfer between CPU memory and GPU memory only happens **twice** in our CAMIP implementation. The first transfer occurs before the first set of moves is evaluated and copies the following data from the CPU memory to the GPU memory:

1. Netlist information, including block-to-net connections, connection types, block types, etc. (space complexity $\Theta(C)$, where C is the number of connections in the netlist).

2. Initial position of each block (space complexity $\Theta(B)$, where B is the number of blocks in the netlist).

The second transfer occurs after the annealing termination condition has been met (i.e., placement is finished), where the final position of each block is copied from the GPU memory

to the CPU memory. The space complexity of the final block positions is $\Theta(B)$. All array calculations described in the pseudocode in Section D.2, operate only on data in GPU memory. Only scalar reduction results (e.g., total placement wirelength cost, maximum cost of a single group of associated moves, etc.) are transferred from the GPU to the CPU during placement, with a corresponding space complexity of $O(1)$, which does not grow with netlist size. Limiting transfers between CPU memory and GPU memory avoids data transfer bottlenecks between moves during placement like those found in the GPU placement implementation proposed by Choong et al.[77].

D.1.1 Base data structures

The data structures described in this section are used in both the W-CAMIP and T-CAMIP implementations.

`[block]` structure

Table D.1 lists the arrays contained within the `[block]` structure, where the length of each array is equal to the number of blocks in the netlist, B (i.e., space complexity of $\Theta(B)$). For each block:

- `slot_key`: The assigned permutation slot.
- `slot_key_prime`: The proposed permutation slot based on the current associated move-pairs pattern.
- `p_x` and `p_y`: The `x` and `y` position associated with the permutation slot `slot_key`.
- `p_x_prime` and `p_y_prime`: The `x` and `y` position associated with the proposed permutation slot `slot_key_prime`.
- `delta_cost`: The difference in cost due to moving from `(x, y)` to `(x_prime, y_prime)`.
- `group_key`: The unique identifier assigned to the associated move pair the block belongs to.

array	dtype
slot_key	uint32
slot_key_prime	uint32
p_x	int32
p_y	int32
p_x_prime	int32
p_y_prime	int32
delta_cost	float32
group_key	int32
sorted_group_key	int32
packed_group_key	int32
rejected_block_key	int32

Table D.1: `block`: Block structure data types

The `[block]` structure also contains the following intermediate arrays where the upper bound on length is equal to the number of blocks:

- `sorted_group_key`: Used when sorting block keys according to associated-moves group key.
- `packed_group_key`: Map of group keys to the contiguous range $(0, \texttt{group\_count}]$.
- `rejected_block_key`: List of keys for blocks that shall not be moved during the current round of moves.

`[block_link]` structure

Table D.2 lists the arrays contained within the `[block_link]` structure. The `[block_link]` structure contains net membership information for each block. Note that we use the terms *"link"* and *"connection"* interchangeably in the data structures to shorten variable names in the source code. The length of each array in the `[block_link]` structure is equal to the number of

connections in the netlist, C, resulting in space complexity $\Theta(C)$. For each connection between a block and a net, the `[block_link]` structure contains:

- `block_key`: The key of the block.
- `net_key`: The key of the block.
- `cost`: The estimated wirelength cost of the net.
- `cost`: The estimated wirelength cost of the net based on the assigned position of the block.
- `cost_prime`: The estimated wirelength cost of the net based on the *proposed* position of the block.

array	dtype
`block_key`	`int32`
`net_key`	`int32`
`cost`	`float32`
`cost_prime`	`float32`

Table D.2: `block_link`: Block connection structure data types

`[net]` structure

Table D.3 lists the array contained within the `[net]` structure, where the length is equal to the number of nets in the netlist, N (i.e., space complexity of $\Theta(N)$). For each net:

array	dtype
`r_inv`	`float32`

Table D.3: `net`: Net structure data types

- `r_inv`: The reciprocal of the rank of the net. Note that the reciprocal is stored since the rank is only used in a denominator and multiplication is computationally cheaper than

division.

`[net_link]` structure

Table D.4 lists the arrays contained within the `[net_link]` structure. Similar to the `[block_link]` structure, the `[net_link]` structure contains the net membership membership information for each block, i.e., each connection between a net and a block. Note that the entries `[net_link]` structure are sorted by net key to support calls to Thrust `reduce_by_key` (which requires keys to be sorted). Recall that we use the terms *"link"* and *"connection"* interchangeably in the data structures to shorten variable names in the source code. As in the case of the `[block_link]` structure, the length of each array in the `[net_link]` structure is equal to the number of connections in the netlist, C, resulting in space complexity $\Theta(C)$. For each connection between a block and a net, the `[net_link]` structure contains:

array	dtype
`net_key`	`int32`
`block_key`	`int32`
`x`	`float32`
`x2`	`float32`
`y`	`float32`
`y2`	`float32`
`cost`	`float32`
`reduced_keys`	`int32`

Table D.4: `net_link`: Net connection structure data types

- `net_key`: The key of the net.
- `block_key`: The key of the block.
- `x`: The `x`-coordinate of the block.
- `x2`: The square of the `x`-coordinate of the block.

- `y`: The `y`-coordinate of the block.
- `y2`: The square of the `y`-coordinate of the block.

In addition, the `[net_link]` structure contains the following intermediate arrays where upper bound on length is equal to the number of connections between nets and blocks:

- `cost`: After a category reduction, the estimated wirelength cost of each net.
- `reduced_keys`: After a category reduction, the net key associated with each cost in `cost`.

`[group]` structure

Table D.5 lists the arrays contained within the `[group]` structure, where the length of each array is equal to the number of blocks in the netlist, B (i.e., space complexity of $\Theta(B)$). The `[group]` structure contains membership information for each group of associated moves, i.e., each block belonging to every group. The `[group]` structure consists of the following arrays:

array	dtype
`block_key`	`uint32`
`delta_cost`	`float32`

Table D.5: `group`: Group structure data types

`block_key`

For each block, the key of the block with the lowest permutation slot key within the corresponding group of associated moves. Since permutation slot keys are unique, this ensures a single unique value is assigned to each group of associated moves.

`delta_cost`

The difference in cost due to moving the associated block. After a category reduction by `block_key`, the `delta_cost` array contains the total difference in cost for each group of associated moves.

D.1.2 Timing data structures

The data structures described in this section are only used in the T-CAMIP implementation. Recall from Section 6.1 that the timing analysis used in our proposed T-CAMIP placement method combines the following delays for each block in the netlist based on a given placement, a) the longest *incoming* path delay, and b) the longest *outgoing* path delay. In our implementation, we use the label "**arrival**" to refer to data structures that deal with *incoming* delays. Along similar lines, we use the label "**departure**" to refer to data structures that deal with *outgoing* delays. Note that most data structures in this section are designated as either "**arrival**" or "**departure**" structures.

`[timing.arch]` structure

Table D.6 lists the array contained within the `[timing.arch]` structure. The `[timing.arch]` structure contains the point-to-point delays for the architecture, as generated by VPR v5.0[4]. Note that the *absolute position* of each point does not affect delay. These delays are based only on:

- Distance between two points.
- Type of block (e.g., input, output, logic, etc.) at each point.

The space complexity for the `[timing.arch]` structure is $\Theta(w+h)$, where w is the width of the placement grid and h is the height of the placement grid. For more information, see VPR v5.0 source code[4].

The only array contained in the `[timing.arch]` structure is:

- `delays`: The point-to-point delays between each block type in the architecture.

array	dtype
`delays`	`float32`

Table D.6: `timing.arch`: Architecture delay structure data types

`[timing.arrival.block]` structure

Table D.7 lists the arrays contained within the `[timing.arrival.block]` structure, where the length of each array is equal to the number of blocks in the netlist, B (i.e., space complexity of $\Theta(B)$). The `[timing.arrival.block]` structure contains the following information for each block, listed according to array name:

- `longest_paths`: The delay of the longest incoming path to the block.
- `min_source_longest_path`: The minimum delay of the longest *incoming* paths to any driver of an incoming connection to the block.

array	dtype
`longest_paths`	`float32`
`min_source_longest_path`	`float32`

Table D.7: `timing.arrival.block`: Incoming block delay structure data types

`[timing.arrival.connections]` structure

Table D.8 lists the arrays contained within the `[timing.arrival.connections]` structure. The `[timing.arrival.connections]` structure is the largest data structure in the T-CAMIP implementation (requiring the same space as the `[timing.departure.connections]` structure), consisting of 15 arrays, with overall space complexity of $\Theta(C)$, where C is the number of netlist connections. The `[timing.arrival.connections]` structure contains the following information for each driver to sink connection:

- `source_key`: The key of the driver block.
- `sync_source`: `1` if the driver is synchronous, `0` otherwise.
- `target_key`: The key of the sink block.
- `delay`: The delay (looked up in `timing.arch.delays`) between the source (*driver*) block and the target (*sink*) block.

- `source_longest_path`: The delay along the longest incoming path to the source (*driver*) block.
- `target_longest_path`: The delay along the longest incoming path to the source block plus the `delay` between the source block and the target block.
- `max_target_longest_path`: After a category reduction of the `target_longest_path` array, the delay along the longest incoming path to the target (*sink*) block.
- `reduced_keys`: After a category reduction of the `target_longest_path` array, the key of the target block for each corresponding maximum incoming path delay.
- `delay_type`: An enumerated type based on the type of driver block and the type of sink block (e.g., driver is an input, sink is a logic block).
- `sink_type`: An enumerated type indicating the type of sink block.
- `driver_type`: An enumerated type indicating the type of driver block.
- `cost`: Criticality cost of connection between driver and sink.
- `cost_prime`: Criticality cost of connection between driver and sink based on the *proposed* position of the *sink* block (assuming *driver* stays in current position).
- `delay_prime`: The delay (looked up in `timing.arch.delays`) between the source (*driver*) block and the target (*sink*) block based on the *proposed* position of the *sink* block (assuming *driver* stays in current position).

array	dtype
`source_key`	`int32`
`sync_source`	`uint8`
`target_key`	`int32`
`delay`	`float32`
`source_longest_path`	`float32`
`target_longest_path`	`float32`
`max_target_longest_path`	`float32`

array	dtype
reduced_keys	int32
delay_type	uint8
sink_type	uint8
driver_type	uint8
cost	float32
cost_prime	float32
reduced_target_cost	float32
delay_prime	float32

Table D.8: `timing.arrival.connections`: Incoming connection timing structure data types

`[timing.arrival.special_blocks]` structure

Table D.9 lists the arrays contained within the `[timing.arrival.special_blocks]` structure. The `[timing.arrival.special_blocks]` structure contains three arrays with independent lengths, where each array contains the keys of blocks belonging to the corresponding group of special blocks:

array	dtype
external_block_keys	int32
sync_logic_block_keys	int32
single_connection_blocks	int32

Table D.9: `timing.arrival.special_blocks`: Special incoming delay block data structures

- `external_block_keys`: Keys of blocks that are external *inputs*.
- `sync_logic_block_keys`: Keys of logic blocks that are synchronous (i.e., connected to a clock).
- `single_connection_blocks`: Keys of logic blocks that have a single connection, either

incoming or outgoing. Such blocks generate warnings in VPR, e.g., "constant generator".

`[timing.departure.block]` structure

Note that outgoing path delays of each block are evaluated by considering connections in the reverse direction compared to the `[timing.arrival]` structures. Therefore, instead of considering connections such that the **driver** is the source and the **sink** is the target, for the `[timing.departure]` structures we consider connections such that the **sink** is the source and the **driver** is the target. Table D.10 lists the arrays contained within the `[timing.departure.block]` structure, where the length of each array is equal to the number of blocks in the netlist, B (i.e., space complexity of $\Theta(B)$). The `[timing.departure.block]` structure contains the following information for each block, listed according to array name:

- `longest_paths`: The delay of the longest *outgoing* path from the block.
- `min_source_longest_path`: The minimum delay of the longest *outgoing* paths from any sink of an outgoing connection to the block.

array	dtype
`longest_paths`	`float32`
`min_source_longest_path`	`float32`

Table D.10: `timing.departure.block`: Outgoing block delay structure data types

`[timing.departure.connections]` structure

Instead of considering connections such that the **driver** is the source and the **sink** is the target, for the `[timing.departure.connections]` structure we consider connections such that the **sink** is the source and the **driver** is the target. Table D.11 lists the arrays contained within the `[timing.departure.connections]` structure. The `[timing.departure.connections]` structure has the same space complexity as the `[timing.arrival.connections]` structure, which is of $\Theta(C)$, where C is the number of netlist connections. The `[timing.departure.connections]`

structure contains the following information for each driver to sink connection:

- `source_key`: The key of the **sink** block (again, note that the sink is the source for outgoing paths).
- `sync_source`: `1` if the **sink** is synchronous, `0` otherwise.
- `target_key`: The key of the **driver** block.
- `delay`: The delay (looked up in `timing.arch.delays`) between the source (**sink**) block and the target (**driver**) block.
- `source_longest_path`: The delay along the longest **outgoing** path from the source (**sink**) block.
- `target_longest_path`: The delay along the longest **outgoing** path from the source block *plus* the `delay` between the source block and the target block.
- `max_target_longest_path`: After a category reduction of the `target_longest_path` array, the delay along the longest outgoing path from the target (**driver**) block.
- `reduced_keys`: After a category reduction of the `target_longest_path` array, the key of the target block for each corresponding maximum outgoing path delay.
- `delay_type`: An enumerated type based on the type of driver block and the type of sink block (e.g., driver is an input, sink is a logic block).
- `sink_type`: An enumerated type indicating the type of sink block.
- `driver_type`: An enumerated type indicating the type of driver block.
- `cost`: Criticality cost of connection between driver and sink.
- `cost_prime`: Criticality cost of connection between driver and sink based on the *proposed* position of the **driver** block (assuming **sink** stays in current position).
- `delay_prime`: The delay (looked up in `timing.arch.delays`) between the source (**sink**) block and the target (**driver**) block based on the *proposed* position of the **driver** block (assuming **sink** stays in current position).

array	dtype
source_key	int32
sync_source	uint8
target_key	int32
delay	float32
source_longest_path	float32
target_longest_path	float32
max_target_longest_path	float32
reduced_keys	int32
delay_type	uint8
sink_type	uint8
driver_type	uint8
cost	float32
cost_prime	float32
reduced_target_cost	float32
delay_prime	float32

Table D.11: `timing.departure.connections`: Outgoing connection timing structure data types

`[timing.departure.special_blocks]` structure

Table D.12 lists the arrays contained within the `[timing.departure.special_blocks]` structure. The `[timing.departure.special_blocks]` structure contains three arrays with independent lengths, where each array contains the keys of blocks belonging to the corresponding group of special blocks:

- `external_block_keys`: Keys of blocks that are external *outputs*.
- `sync_logic_block_keys`: Keys of logic blocks that are synchronous (i.e., connected to a clock).

- `single_connection_blocks`: Keys of logic blocks that have a single connection, either incoming or outgoing. Such blocks generate warnings in VPR, e.g., "constant generator".

Note that although the length of arrays in the `[timing.departure.special_blocks]` structure are independent from one another, the maximum length of any array in the structure is equal to the number of blocks in the netlist, B. Therefore, the space complexity of the `[timing.departure.special_blocks]` structure is $O(B)$.

array	dtype
`external_block_keys`	`int32`
`sync_logic_block_keys`	`int32`
`single_connection_blocks`	`int32`

Table D.12: `timing.departure.special_blocks`: Special outgoing delay block data structures

D.1.3 Space complexity

As shown in Fig. D.1, across all benchmarks in the MCNC and IWLS sets, WCAMIP requires 57-65 bytes/connection and TCAMIP requires 123-142 bytes/connection. Note that the variations are due to structures where size is based on block count or net count, rather than connection count. However, since the number of connections is higher than block count or net count by definition, the growth rate of the total size of all data structures is in $\Theta(C)$, where C is the number of connections.

D.2 Pseudocode

In this section, we present high-level pseudocode outlining the key operations performed by the CAMIP algorithm we proposed in Chapter 4[1]. Listing D.1 shows pseudocode for the main loop of the CAMIP algorithm described in Section 4.1. Note that each line 12–16 corresponds to a) a box in the flow chart in Fig. 4.1, and b) one or more parallel patterns. Furthermore, all

[1]As previously mentioned, full source code can be found at `https://github.com/cfobel/camip`.

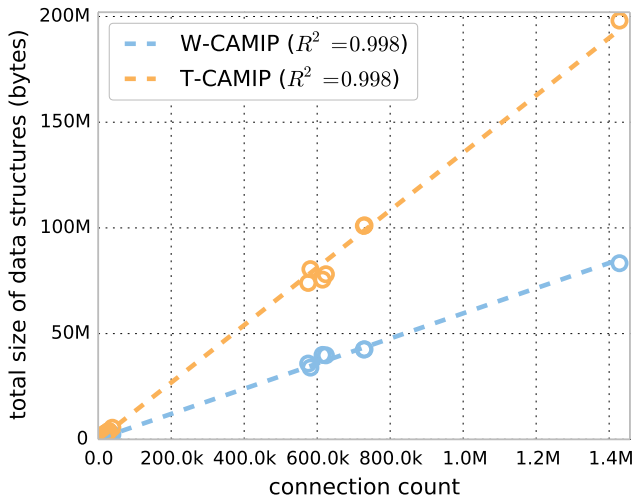


Figure D.1: Total size of data structures for W-CAMIP and T-CAMIP.

operations outside lines 12–16 have time and space complexity $O(1)$. Listing D.2 to Listing D.6 in Section D.2.1 to Section D.2.5 contain the pseudocode corresponding to each function listed on lines 12–16 of Listing D.1.

It should be noted that all runtime results reported in Section 5.4 and Section 6.5 correspond to the wall-clock time for the steps in Listing D.1. The runtime of the steps in Listing D.1 do not include the initial loading of netlist and architecture data from input files into data structures, or the generation of the initial random placement. However, unlike previous work in the literature[33, 34], we include all time spent performing critical-path delay calculations in the reported runtime results when timing calculations are enabled. Our W-CAMIP and T-CAMIP implementations print the combined wall-clock time for the steps in Listing D.1 upon placement completion. In addition, the wall-clock time of each iteration of Listing D.1 is recorded along with other statistics for each iteration (e.g., temperature, fraction of accepted moves) in a data file. For details on the statistics data file, see `https://github.com/cfobel/camip`.

```
1 # Outer loop
2 while not exit(total_cost, temperature):
3     total_moves = 0
4     rejected_moves = 0
5
```

```
    # Inner loop
    while total_moves < moves_per_temperature:
        max_move_d = max(anneal_schedule.rlim, 1)
        seed = randint()

        ## Start parallel ##
        propose_moves(seed, max_move_d)
        evaluate_moves()
        move_count, rejected_moves = assess_groups(temperature)
        apply_groups(rejected_moves)
        evaluate_placement()
        ## Stop parallel ##

        total_moves += move_count
        rejected_moves += rejected

    success_ratio = (total_moves - rejected_moves) / total_moves
    anneal_schedule.update_state(success_ratio)
```

Listing D.1: `main_loop` pseudocode.

In a similar manner, the absolute runtime results reported for VPR[4] in Section 5.4 and Section 6.5 only include wall-clock time of the main loop (i.e., not the initial population of data structures or generation of initial random placement).

D.2.1 `propose_moves`

Listing D.2 shows the pseudocode for the `propose_moves` function in Listing D.1. Based on the provided seed and maximum move distance, `max_move_d`:

- Generate a random move pattern (constant time).
- Compute the new permutation slot for each block in the placement, based on the generated moves pattern.

- Compute the `(x, y)` corresponding to the proposed slot for each block.

```
def propose_moves(seed, max_move_d):
    srand(seed)  # Serial, O(1)

    # Generate a random associated move-pair pattern based on the current
    # maximum move distance.
    move_pattern = random_vpr_pattern(max_move_d)  # Serial, O(1)

    # Thrust 'transform'.
    block.slot_key_prime = slot_moves(block.slot_keys,
                                      move_pattern)

    # Extract positions into p_x and p_x based on permutation
    # slot assignments using Thrust 'transform'.
    block.{p_x_prime, p_y_prime} = extract_positions(block.slot_key_prime)
```

Listing D.2: `propose_moves` pseudocode.

Note that all operations that are not constant-time are implemented using Thrust `transform`[72] operations (i.e., map of sequence[1]).

D.2.2 `evaluate_moves`

Listing D.3 shows the pseudocode for the `evaluate_moves` function in Listing D.1.

- Compute the total cost per block, based on the current set of proposed moves.
- For each block, compute the difference in cost between the current position and the newly proposed position.

```
def evaluate_moves():
    # Calculate difference in cost for each netlist connection,
    # sorted by block key using Thrust 'transform' and
    # 'reduce_by_key'.
```

```
5     block_link.cost_prime = calc_cost_prime(block_link.block_key,
6                                             block_link.net_key,
7                                             block.p_x,
8                                             block.p_x_prime,
9                                             net_link.x,
10                                            net_link.x2,
11                                            block.p_y,
12                                            block.p_y_prime,
13                                            net_link.y,
14                                            net_link.y2,
15                                            net.r_inv, 1.59)
16
17    # Compute move-deltas using a Thrust 'reduce_by_key' call
18    # over a 'transform' iterator.
19    block.delta_cost = minus_float(block_link.cost_prime,
20                                   block_link.cost)
```

Listing D.3: `evaluate_moves` pseudocode.

Note that all operations are implemented using one of the following Thrust operations:

- `reduce_by_key`[72] (i.e., category reduction[1]).
- `transform`[72] (i.e., map[1]).

D.2.3 `assess_groups`

Listing D.4 shows the pseudocode for the `assess_groups` function in Listing D.1. The operations performed by the `assess_groups` function include:

- For each block, compute the key of the group of concurrent-associated moves the block belongs to.
  - For a swap, which corresponds to a group of two concurrent-associated moves, the group key is equal to the minimum corresponding permutation slot key.

- For each group of concurrent-associated moves, compute the difference in cost due to applying all moves in the group.
- For each group of concurrent-associated moves, assess if *all* moves in the group should be *accepted* or *rejected*.

```
def assess_groups():
    # Compute the key of the group that each block belongs to
    # using Thrust `transform`.
    # **Note that group keys are not necessarily contiguous.**
    block.group_key = compute_block_group_keys(block.slot_key,
                                               block.slot_key_prime,
                                               block.group_key,
                                               total_slot_count)

    # Use Thrust `reduce` to count the number of blocks where the proposed
    # permutation slot is the same as the current permutation slot, i.e.,
    # the number of unmoved blocks in the current set of proposed moves.
    unmoved_count = count_equal(block.slot_key,
                                block.slot_key_prime)
    evaluated_count = total_block_count - unmoved_count

    # No blocks were assigned a non-zero move this iteration.
    if unmoved_count == total_block_count:
        # Since no blocks are moved, no moves are evaluated and, thus,
        # there are no moves to reject.
        rejected_count = 0
        return evaluated_count, rejected_count

    # Use Thrust `sequence`. to fill `group.block_key` with the range
    # `(0, total_block_count]`, a list of contiguous block keys.
    group.block_key = range(0, total_block_count)

    # Copy the group key for each block into a new array using Thrust
```

```
29      # `copy`. The new array will be modified by the following sort
30      # operation.
31      block.sorted_group_key = block.group_key[:]
32
33      # Sort block keys by according to group key using Thrust
34      # `sort_by_key`.  This brings together keys of blocks belonging to the
35      # same group of associated moves.
36      sort_by_key(block.sorted_group_key, group.block_key)
37
38      # Map the group key of each block to a key in the contiguous range
39      # `(0, group_count]` using Thrust `inclusive_scan`. We refer to the
40      # mapped group keys as the "packed" group keys.
41      block.packed_group_key = packed_group_keys(block.group_key,
42                                                 group.block_key)
43
44      # Compute total difference in cost per *group* (i.e., sum of
45      # costs for all moves in group) using Thrust `reduce_by_key`
46      # over a `permutation` iterator.
47      group.delta_cost = compute_group_deltas(block.packed_group_key,
48                                              block.delta_cost,
49                                              group.block_key)
50
51      # Using Thrust `copy_if`, collect the keys of blocks that
52      # do not belong to a group that meets either of the following
53      # criteria:
54      #
55      #  - Delta cost less than or equal to 0.
56      #  - Anneal acceptance function passes (for non-improving
57      #    moves).
58      block.rejected_block_key, rejected_count =
59          assess_group_deltas(temperature, group.block_key,
60                              block.packed_group_key,
61                              group.delta_cost)
```

```
62
63      return evaluated_count, rejected_count
```

Listing D.4: `assess_groups` pseudocode.

Note that all operations are implemented using one of the following Thrust operations:

- `reduce_by_key`[72] (i.e., category reduction[1]).
- `transform`[72] (i.e., map[1]).
- `sequence`[72] *(i.e., map[1] to store a range of values).*
- `inclusive_scan`[72] (i.e., fused map and prefix sum[1]).
- `copy_if`[72] (i.e., fused map and pack[1]).

D.2.4 `apply_groups`

Listing D.5 shows the pseudocode for the `apply_groups` function in Listing D.1. The purpose of the `apply_groups` function is to update the placement according to accepted moves during the current iteration of the inner loop in Listing D.1.

```
1  def apply_groups(rejected_count):
2      # Using Thrust 'transform', restore original slot key for each block
3      # where the corresponding move has been marked as rejected.
4      block.slot_key_prime = copy_if_rejected(block.slot_key,
5                                              block.rejected_block_key)
6
7      # Swap updated block permutation slot keys to use as current
8      # slot keys.  This operation is O(1), since we're just
9      # swapping array references (no copy is performed).
10     block.slot_key, block.slot_key_prime = (block.slot_key_prime,
11                                             block.slot_key)
```

Listing D.5: `apply_groups` pseudocode.

Note that all operations that are not constant-time are implemented using Thrust `transform`[72] operations (i.e., map of sequence[1]).

D.2.5 `evaluate_placement`

Listing D.6 shows the pseudocode for the `evaluate_placement` function in Listing D.1. The `evaluate_placement` function computes the cost of: a) each net, and b) the complete placement.

Note that all operations are implemented using one of the following Thrust operations:

- `transform`[72] (i.e., map[1]).
- `reduce_by_key`[72] (i.e., category reduction[1]).
- `reduce`[72] (i.e., reduction[1]).

```
1  def evaluate_placement(self):
2      # Extract positions into p_x and p_x based on permutation
3      # slot assignments using Thrust `transform`.
4      block.p_x, block.p_y = extract_positions(block.slot_key)
5
6      # Compute intermediate vectors needed to compute Star+ using Thrust
7      # `reduce_by_key`.
8      net_link.x, net_link.x2, net_link.y, net_link.y2 =
9          sum_xy_vectors(net_link.block_key, net_link.net_key,
10                        block.p_x, block.p_y)
11
12     # Compute the Star+ per net along with the total placement cost using
13     # intermediate vectors along with Thrust `transform` and Thrust
14     # `reduce`.
15     net_link.cost, total_cost = star_plus_2d(net_link.x, net_link.x2,
16                                              net_link.y, net_link.y2,
17                                              net.r_inv, 1.59,
18                                              net_link.cost)
19
```

```
    # For each block, compute the sum of Star+ costs for all nets
    # connected to the block using Thrust 'reduce_by_key'.
    block_link.cost = block_star_plus_2d(block_link.block_key,
                                         net_link.cost,
                                         block_link.net_key)
    return total_cost
```

Listing D.6: `evaluate_placement` pseudocode.

D.3 Placement verification

In the context of placement, a correct solution is a mapping of each block in a netlist to a position on a target architecture. Using a permutation encoding to represent the block key occupying each available position on the target architecture (where empty locations are permitted), a correct placement ensures that:

- Each block in the netlist is represented *exactly once* in the permutation.
- Each block occupies a position of the appropriate type (i.e., inputs/outputs are mapped to I/O-pads, logic blocks are mapped to configurable logic positions).
- All permutation locations that are not occupied by a netlist block are *empty*, *i.e.*, only blocks in the netlist are present in the placement permutation.

Listing D.7 shows pseudocode that may be used to verify that a placement meets the criteria listed above. Note that placement verification may optionally be enabled when running our proposed CAMIP implementations (W-CAMIP and T-CAMIP), though is not required.

```
## Architecture ##

# Assume 4 input/output locations/slots on target architecture.
io_slot_count = 4
# Assume 6 input/output locations/slots on target architecture.
logic_slot_count = 6
```

```
7
8  # Create empty permutation representing available positions on target
9  # architecture, where slots 0-3 correspond to input/output locations and
10 # slots 4-9 correspond to logic locations.  All positions are marked as
11 # "empty" using a value of -1.
12 #
13 # Initial contents of permutation:
14 #
15 #                              configurable logic blocks
16 #                               <-------------------->
17 permutation = [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
18 #              <---------->
19 #               inputs/outputs
20
21
22 # ## Netlist ##
23
24 # Assume netlist to be placed has:
25 #  - 2 input blocks
26 #  - 1 output block
27 #  - 4 logic blocks
28 input_count = 2
29 output_count = 1
30 logic_count = 4
31 block_count = input_count + output_count + logic_count
32
33 # Each block is assigned a key, according to the following map:
34 #
35 #                  output
36 #                    |
37 block_keys = [0, 1, 2, 3, 4, 5, 6]
38 #             <-->    <------->
39 #            inputs   logic blocks
```

```
#
# The above keys are used to mark the position of the corresponding
# block in the permutation.


## Assign a position to each block ##
io_count = input_count + output_count
# Contents of permutation after populating with initial placement:
#
#                               configurable logic blocks
#                               <--------------------->
permutation = [ 0,  1,  2, -1,  3,  4,  5,  6, -1, -1]
#             <---------->
#              inputs/outputs

# Extract position for each block from permutation.
#                          output
#                            |
block_positions = [-1, -1, -1, -1, -1, -1, -1]
#                  <-->     <---------->
#                   inputs      logic blocks

for position_index, block_key in enumerate(permutation):
    if block_key >= 0:
        # Placement position is occupied.
        block_positions[block_key] = position_index
# Contents of `block_positions`: [0, 1, 2, 4, 5, 6, 7]


## Verify placement is correct ##

# Start count for each block to zero.
#
```

```
#                          output
#                            |
block_occurrences = [0, 0, 0, 0, 0, 0, 0]
#                     <-->     <-------->
#                   inputs    logic blocks
empty_count = 0

# Count the number of times each block occurs in permutation.
for block_key in permutation:
    if block_key >= 0:
        # Placement position is occupied.
        block_occurrences[block_key] += 1
    else:
        # Placement position is unoccupied.
        empty_count += 1

# Verify each block occurs *exactly once* in the placement.
assert((min(block_occurrences) == 1) and (max(block_occurrences) == 1))

# Verify all positions not occupied by blocks are marked as empty.
assert(empty_count == (io_slot_count + logic_slot_count) - block_count)

# Verify all input/output blocks are assigned to input/output permutation
# slots.
assert(min(block_positions[:io_count]) >= 0)
assert(max(block_positions[:io_count]) < io_slot_count)

# Verify all logic blocks are assigned to logic permutation slots.
assert(min(block_positions[io_count:]) >= io_slot_count)
assert(max(block_positions[io_count:]) < io_slot_count + logic_slot_count)
```

Listing D.7: Placement verification pseudocode

D.4 Thrust example

Listing D.8 contains an example program demonstrating how `thrust::reduce_by_key` can be used to sum the `x`-positions and `y`-positions of all blocks connected to each net, as described in Equation 5.12 and Equation 5.13 on page 108.

```
1  #include <thrust/device_vector.h>
2  #include <thrust/reduce.h>
3  #include <thrust/copy.h>
4  #include <thrust/iterator/permutation_iterator.h>
5  #include <stdint.h>
6  #include <iterator>
7  #include <iostream>
8  #include <iomanip>
9
10 #define DUMPN(d_vector,N,dtype,width) \
11   std::cout << std::setw(width) << #d_vector << ": "; \
12   thrust::copy_n(d_vector.begin(), N, \
13                  std::ostream_iterator<dtype>(std::cout, ", ")); \
14   std::cout << std::endl;
15
16 #define DUMP(d_vector,dtype,width) DUMPN(d_vector,(d_vector.end() - \
17                                          d_vector.begin()),dtype,width)
18
19
20 int main(void) {
21   /* # Block positions # */
22   // `(x, y)` positions for 4 blocks (blocks 0-3 from Fig. 4.13).
23   uint32_t p_x[] = {2, 3, 3, 0};
24   uint32_t p_y[] = {0, 0, 3, 2};
25   size_t block_count = sizeof(p_x) / sizeof(uint32_t);
26
27   /* # Nets #
28    *
```

```
29     * Assume each block drives a net (i.e., there are 4 nets in the
30     * netlist):
31     *
32     *  - Net 0: connects blocks `0, 1, 2, 3`.
33     *  - Net 1: connects blocks `1, 0`.
34     *  - Net 2: connects blocks `2, 1`.
35     *  - Net 3: connects blocks `3, 0, 1`. */
36    uint32_t net_keys[] =   {0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3};
37    uint32_t block_keys[] = {0, 2, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 3};
38    size_t connection_count = sizeof(net_keys) / sizeof(uint32_t);
39
40    // Create device vectors to store position of each block.
41    thrust::device_vector<uint32_t> d_p_x(block_count);
42    thrust::device_vector<uint32_t> d_p_y(block_count);
43
44    // Create device vectors to store netlist connection info.
45    thrust::device_vector<uint32_t> d_net_keys(connection_count);
46    thrust::device_vector<uint32_t> d_block_keys(connection_count);
47    thrust::device_vector<uint32_t> d_block_x(connection_count);
48    thrust::device_vector<uint32_t> d_block_y(connection_count);
49
50    // Create device vectors to store reduced keys/values.
51    thrust::device_vector<uint32_t> d_reduce_keys(connection_count);
52    thrust::device_vector<uint32_t> d_net_x_sums(connection_count);
53    thrust::device_vector<uint32_t> d_net_y_sums(connection_count);
54
55    typedef thrust::device_vector<uint32_t>::iterator dev_iterator;
56
57    // Copy the initial position of each block to device memory.
58    // N.B., the following copy operations are only performed
59    // *once* in the CAMP implementation at the start of placement.
60    thrust::copy_n(&p_x[0], block_count, d_p_x.begin());
61    thrust::copy_n(&p_y[0], block_count, d_p_y.begin());
```

```
62  thrust::copy_n(&block_keys[0], connection_count,
63                 d_block_keys.begin());
64  thrust::copy_n(&net_keys[0], connection_count,
65                 d_net_keys.begin());
66
67  /* # Sum x-position of all blocks by net #
68   *
69   * For each net, compute the sum of the x-positions of all
70   * connected blocks. */
71
72  // Look-up the x-position for the block associated with each
73  // netlist connection.
74  thrust::copy_n(
75    thrust::make_permutation_iterator(
76        d_p_x.begin(), d_block_keys.begin()),
77    d_block_keys.size(), d_block_x.begin());
78
79  // Look-up the y-position for the block associated with each
80  // netlist connection.
81  thrust::copy_n(
82    thrust::make_permutation_iterator(
83      d_p_y.begin(), d_block_keys.begin()),
84    d_block_keys.size(), d_block_y.begin());
85
86  thrust::pair<dev_iterator, dev_iterator> new_end;
87
88  // Reduce (sum) connection x-positions by net key.
89  new_end = thrust::reduce_by_key(
90    d_net_keys.begin(), d_net_keys.end(), d_block_x.begin(),
91    d_reduce_keys.begin(), d_net_x_sums.begin());
92
93  // Reduce (sum) connection y-positions by net key.
94  new_end = thrust::reduce_by_key(
```

```
95      d__net_keys.begin(), d__net_keys.end(), d__block_y.begin(),
96      d__reduce_keys.begin(), d__net_y_sums.begin());
97
98    size_t net_count = new_end.first - d__reduce_keys.begin();
99
100   /* # Results #
101    *
102    * ## Summary ##
103    *
104    *  - `net_count` is equal to 4 (i.e., the number of nets in the
105    *    netlist).
106    *  - The number of nets is less than the number of connections.  After
107    *    reduction, the per-net results are packed to the beginning of the
108    *    following vectors:
109    *    * `d__reduce_keys`
110    *    * `d__net_x_sums`
111    *    * `d__net_y_sums`
112    *
113    * ## Array contents ##
114    *
115    * The final array contents are as follows:
116    *
117    *              d__net_keys: 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3,
118    *            d__block_keys: 0, 2, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 3,
119    *               d__block_x: 2, 3, 2, 3, 3, 0, 2, 3, 3, 0, 2, 3, 0,
120    *               d__block_y: 0, 3, 0, 0, 3, 2, 0, 0, 3, 2, 0, 0, 2,
121    *
122    *                             net_count
123    *                          <---------->
124    *           d__reduce_keys: 0, 1, 2, 3, (remaining elements unused...)
125    *            d__net_x_sums: 5, 8, 8, 5, (remaining elements unused...)
126    *            d__net_y_sums: 3, 5, 5, 2, (remaining elements unused...)
127    */
```

```
128   DUMP(d__net_keys, uint32_t, 20);
129   DUMP(d__block_keys, uint32_t, 20);
130   DUMP(d__block_x, uint32_t, 20);
131   DUMP(d__block_y, uint32_t, 20);
132   std::cout << std::endl;
133   DUMPN(d__reduce_keys, net_count, uint32_t, 20);
134   DUMPN(d__net_x_sums, net_count, uint32_t, 20);
135   DUMPN(d__net_y_sums, net_count, uint32_t, 20);
136   return 0;
137 }
```

Listing D.8: `netlist_example.cu`

Lines 60-65 copy the static netlist connection data and the initial block positions from CPU (i.e., host) memory to GPU (i.e., device) memory. In our W-CAMIP implementation and T-CAMIP implementation, this transfer from host memory to device memory is performed *only once* (at the start of placement). Lines 74–84 correspond to populating matrix X and matrix Y, as defined in Equation 5.4 and Equation 5.5, respectively, on page 105. Note that in Lines 74–84 all data transfers operate completely within device memory (i.e., no memory transfer between CPU and GPU). Lines 89–91 compute the vector $\vec{n_x}$ as defined Equation 5.12, while Lines 94–98 compute the vector $\vec{n_y}$ as defined Equation 5.13 (see page 108).

More usage details regarding the `thrust` library can be found in the `thrust` quick start guide[93] and in the numerous examples included in the `thrust` source code (available on the `thrust` project page[72]).

Compilation

A program that uses `thrust`[72] device vectors may be compiled to target one of several available device backends[94]. Note that the `thrust` library (header-only) is included in the NVIDIA CUDA Toolkit[95].

The target backend may be specified during compilation by setting the `THRUST_DEVICE_SYSTEM` define as described on the Thrust wiki[94]. The `netlist_example` program may be compiled to

target CUDA-capable GPU devices using ‘nvcc‘ (included in the NVIDIA CUDA Toolkit[95] as of version 5.5) using:

```
nvcc netlist_example.cu -o netlist_example-cuda \
  -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CUDA
```

Alternatively, the `netlist_example` program may be compiled to run entirely on the CPU by setting `THRUST_DEVICE_SYSTEM` to `THRUST_DEVICE_SYSTEM_CPP`, utilizing only host memory, by using the following command:

```
nvcc netlist_example.cu -o netlist_example-cpp \
  -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CPP
```

Note that the `netlist_example` program may be compiled to run entirely on the CPU without installing the CUDA Toolkit. This can be helpful when developing in an environment without a CUDA GPU installed. To compile the `netlist_example` code using `g++` without installing the CUDA Toolkit:

1. Download the `thrust` header library from the `thrust` project page[72].
2. Rename the `netlist_example.cu` file to `netlist_example.cpp`, since `g++` does not recognize files with the `.cu` extension.
3. Run the following command to compile using the host C++ backend:

   ```
   g++ netlist_example.cpp -o netlist_example-cpp \
     -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CPP \
     -I<path containing ‘thrust‘ header directory>
   ```

Bibliography

[1] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming*. Boston: Morgan Kaufmann, 2012.

[2] Ananth Y. Grama, A. Gupta, and V. Kumar. "Isoefficiency: measuring the scalability of parallel algorithms and architectures". In: *Parallel Distributed Technology: Systems Applications, IEEE* 1.3 (1993), pp. 12–21. DOI: 10.1109/88.242438.

[3] Donald E Knuth. "Big omicron and big omega and big theta". In: *ACM SIGACT News* 8.2 (1976), pp. 18–24.

[4] Jason Luu et al. "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 4.4 (Dec. 2011), 32:1–32:23. DOI: 10.1145/2068716.2068718.

[5] Christian Fobel and Gary Grewal. "A parallel Steiner tree heuristic for macro cell routing". In: *IEEE International Conference on Computer Design*. 2008, pp. 27–33. DOI: 10.1109/ICCD.2008.4751836.

[6] Christian Fobel, Gary Grewal, and Andrew Morton. "Hardware accelerated FPGA placement". In: *Microelectronics Journal* 40.11 (2009), pp. 1667–1671. DOI: 10.1016/j.mejo.2008.09.008.

[7] Christian Fobel, Gary Grewal, and Deborah Stacey. "Using GPUs to accelerate FPGA wirelength estimate for use with complex search operators". In: *24th Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2011, pp. 001129–001134. DOI: 10.1109/CCECE.2011.6030638.

[8] Christian Fobel, Gary Grewal, and Deborah Stacey. "GPU-accelerated wire-length estimation for FPGA placement". In: *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 14–23. DOI: 10.1109/SAAHPC.2011.16.

[9] Robert Collier et al. "Depictions of genotypic space for evaluating the suitability of different recombination operators". In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. Philadelphia, Pennsylvania, USA: ACM, 2012, pp. 609–616. DOI: 10.1145/2330163.2330250.

[10] Robert Collier et al. "A formal and empirical analysis of recombination for genetic algorithm-based approaches to the FPGA placement problem". In: *25th IEEE Canadian Conference on Electrical and Computer Engineering*. IEEE, 2012, pp. 1–6. DOI: 10.1109/CCECE.2012.6334856.

[11] Christian Fobel et al. "GPU approach to FPGA placement based on Star+". In: *2012 IEEE 10th International New Circuits and Systems Conference (NEWCAS)*. 2012, pp. 229–232. DOI: 10.1109/NEWCAS.2012.6328998.

[12] Robert Collier et al. "Advancing genetic algorithm approaches to field programmable gate array placement with enhanced recombination operators". In: *Evolutionary Intelligence* (2014), pp. 1–18. DOI: 10.1007/s12065-014-0114-6.

[13] Christian Fobel, Gary William Grewal, and Deborah Stacey. "Forward-scaling, serially equivalent parallelism for FPGA placement". In: *Great Lakes Symposium on VLSI 2014*. Ed. by Joseph R. Cavallaro et al. ACM, 2014, pp. 293–298. DOI: 10.1145/2591513.2591543.

[14] Christian Fobel, Gary Grewal, and Deborah Stacey. "A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and GPU architectures". In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927481.

[15] Xilinx. *7 series FPGAs overview*. Tech. rep. Xilinx. URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (visited on 02/15/2015).

[16] Altera Corporation. *Stratix V FPGA family overview*. Tech. rep. Altera Corporation. URL: http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/overview/stxv-overview.html (visited on 02/15/2015).

[17] Bo Hu and Malgorzata Marek-Sadowska. "FAR: fixed-points addition & relaxation based placement". In: *Proceedings of the 2002 international symposium on Physical design*. San Diego, CA, USA: ACM, 2002, pp. 161–166. DOI: 10.1145/505388.505426.

[18] Kristofer Vorwerk and Andrew Kennings. "An improved multi-level framework for force-directed placement". In: *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*. IEEE Computer Society, 2005, pp. 902–907. DOI: 10.1109/DATE.2005.59.

[19] Yonghong Xu and Mohammed A. S. Khalid. "QPF: efficient quadratic placement for FPGAs". In: *International Conference on Field Programmable Logic and Applications*. Vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 555–558. DOI: 10.1109/FPL.2005.1515784.

[20] Padmini Gopalakrishnan, Xin Li, and Lawrence Pileggi. "Architecture-aware FPGA placement using metric embedding". In: *Proceedings of the 43rd annual Design Automation Conference*. San Francisco, CA, USA: ACM, 2006, pp. 460–465. DOI: 10.1145/1146909.1147033.

[21] Natarajan Viswanathan, Min Pan, and Chris Chu. "FastPlace: An efficient multilevel force-directed placement algorithm". In: *Modern Circuit Placement*. Ed. by Gi-Joon Nam and Jason Cong. Series on Integrated Circuits and Systems. Springer US, 2007, pp. 193–228. DOI: 10.1007/978-0-387-68739-1_8.

[22] Chandra Mulpuri and Scott Hauck. "Runtime and quality tradeoffs in FPGA placement and routing". In: *Proceedings of the 2001 ACM/SIGDA ninth International Symposium on Field Programmable Gate Arrays*. Monterey, California, United States: ACM, 2001, pp. 29–36. DOI: 10.1145/360276.360294.

[23] Michael Hutton, Khosrow Adibsamii, and Andrew Leaver. "Timing-driven placement for hierarchical programmable logic devices". In: *Proceedings of the 2001 ACM/SIGDA ninth International Symposium on Field Programmable Gate Arrays*. Monterey, California, United States: ACM Press, 2001, 311. DOI: 10.1145/360276.360286.

[24] P. Banerjee and S. Sur-Kolay. "Faster placer for island-style FPGAs". In: *International Conference on Computing: Theory and Applications*. 2007, pp. 117–121. DOI: 10.1109/ICCTA.2007.62.

[25] Manuel Rubio-Solar et al. "A FPGA optimization tool based on a multi-island genetic algorithm distributed over grid environments". In: *IEEE International Symposium on Cluster Computing and the Grid*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 65–72. DOI: 10.1109/CCGRID.2008.96.

[26] Meng Yang, A. Almaini, and Pengjun Wang. "FPGA placement optimization by two-step unified genetic algorithm and simulated annealing algorithm". In: *Journal of Electronics (China)* 23.4 (July 2006), pp. 632–636. DOI: 10.1007/s11767-005-0198-3.

[27] K. Vorwerk, A. Kennings, and J.W. Greene. "Improving simulated annealing-based FPGA placement with directed moves". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.2 (2009), pp. 179–192. DOI: 10.1109/TCAD.2008.2009167.

[28] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. "High-quality, deterministic parallel placement for FPGAs on commodity hardware". In: *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*. Monterey, California, USA: ACM, 2008, pp. 14–23. DOI: 10.1145/1344671.1344676.

[29] Cristinel Ababei. "Speeding up FPGA placement via partitioning and multithreading". In: *International Journal of Reconfigurable Computing* 2009 (2009), 6:1–6:9. DOI: 10.1155/2009/514754.

[30] Hao Li and Yue Zhuo. "Criticality history guided FPGA placement algorithm for timing optimization". In: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. Orlando, Florida, USA: ACM, 2008, pp. 267–272. DOI: 10.1145/1366110.1366175.

[31] M. Xu et al. "Near-linear wirelength estimation for FPGA placement". In: *Canadian Conference on Electrical and Computer Engineering*. 2009, pp. 1198–1203. DOI: 10.1109/CJECE.2009.5443860.

[32] Vaughn Betz and Jonathan Rose. "VPR: a new packing, placement and routing tool for FPGA research". In: *Field-Programmable Logic and Applications*. Ed. by Wayne Luk, PeterY.K. Cheung, and Manfred Glesner. Vol. 1304. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 213–222. DOI: 10.1007/3-540-63465-7_226.

[33] Matthew An, J. Gregory Steffan, and Vaughn Betz. "Speeding up FPGA placement: Parallel algorithms and methods". In: *Proceedings of the 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines*. FCCM '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–185. DOI: 10.1109/.58.

[34] Adrian Ludwin and Vaughn Betz. "Efficient and deterministic parallel placement for FPGAs". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 16.3 (June 2011), 22:1–22:23. DOI: 10.1145/1970353.1970355.

[35] William Swartz and Carl Sechen. "Timing driven placement for large standard cell circuits". In: *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. San Francisco, California, United States: ACM Press, 1995, 211215. DOI: 10.1145/217474.217531.

[36] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. "Timing-driven placement for FPGAs". In: *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*. Monterey, California, United States: ACM, 2000, pp. 203–213. DOI: 10.1145/329166.329208.

[37] K. Shahookar and P. Mazumder. "VLSI cell placement techniques". In: *ACM Computer Survey* 23.2 (1991), 143220. DOI: 10.1145/103724.103725.

[38] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman San Francisco, 1979.

[39] Peter Jamieson. "Revisiting genetic algorithms for the FPGA placement problem". In: *GEM*. 2010, pp. 16–22.

[40] J.M. Kleinhans et al. "GORDIAN: VLSI placement by quadratic programming and slicing optimization". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 10.3 (1991), pp. 356–365. DOI: 10.1109/43.67789.

[41] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". In: *Bell System Technical Journal* 49.2 (1970), pp. 291–307. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[42] Jens Vygen. "Algorithms for large-scale flat placement". In: *Proceedings of the 34th annual Design Automation Conference*. Anaheim, California, United States: ACM, 1997, pp. 746–751. DOI: 10.1145/266021.266360.

[43] Hans Eisenmann and Frank M. Johannes. "Generic global placement and floorplanning". In: *Proceedings of the 35th annual Design Automation Conference*. San Francisco, California, United States: ACM, 1998, pp. 269–274. DOI: 10.1145/277044.277119.

[44] Hussein Etawil, Shawki Areibi, and Anthony Vannelli. "Attractor-repeller approach for global placement". In: *Proceedings of the 1999 IEEE/ACM international conference on Computer-Aided Design*. 1999, pp. 20–24. DOI: 10.1109/ICCAD.1999.810613.

[45] Georg Sigl, Konrad Doll, and Frank M. Johannes. "Analytical placement: A linear or a quadratic objective function?" In: *Proceedings of the 28th ACM/IEEE Design Automation Conference*. San Francisco, California, United States: ACM, 1991, pp. 427–432. DOI: 10.1145/127601.127707.

[46] R. Barrett et al. *Templates for the solution of linear systems*. Philadelphia, PA: Society for Industrial Mathematics, 1994.

[47] J. R. Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain*. Technical Report UMI Order Number: CS-94-125. Carnegie Mellon University, 1994. URL: http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf (visited on 02/15/2015).

[48] David M. Young. "Iterative methods for solving partial difference equations of elliptical type". PhD Thesis. Harvard University, May 1950. URL: http://www.ma.utexas.edu/CNA/DMY/david_young_thesis.pdf (visited on 02/15/2015).

[49] Ming Xu. "Analytic methods for the FPGA placement problem". PhD. Guelph, Ontario, Canada: Department of Computing and Information Science. University of Guelph, May 2009, p. 192.

[50] P. Maidee, C. Ababei, and K. Bazargan. "Timing-driven partitioning-based placement for island style FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.3 (2005), pp. 395–406. DOI: 10.1109/TCAD.2004.842812.

[51] George Karypis et al. "Multilevel hypergraph partitioning: application in VLSI domain". In: *Proceedings of the 34th annual Design Automation Conference*. Anaheim, California, United States: ACM, 1997, pp. 526–529. DOI: 10.1145/266021.266273.

[52] Peng Du. "Fast heuristic techniques for FPGA placement based on multilevel clustering". Masters of Science. Guelph, Ontario, Canada: University of Guelph, 2003, p. 147.

[53] S. Areibi et al. "Hierarchical FPGA placement". In: *Canadian Journal of Electrical and Computer Engineering* 32.1 (2007), pp. 53–64. DOI: 10.1109/CJECE.2007.364333.

[54] CAD Group at Politecnico di Torino. *CAD Group: ITC'99 Benchmarks (2nd release)*. Sept. 2007. URL: http://www.cad.polito.it/tools/itc99.html (visited on 02/15/2015).

[55] S. Yang. "Logic synthesis and optimization benchmarks user guide version 3.0". In: *MCNC, Jan* (1991).

[56] Chih-Liang Eric Cheng. "RISA: accurate and efficient placement routability modeling". In: San Jose, California, United States: IEEE Computer Society Press, 1994, 690695.

[57] Kalliopi Tsota, Cheng-Kok Koh, and Venkataramanan Balakrishnan. "Guiding global placement with wire density". In: *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. San Jose, California: IEEE Press, 2008, pp. 212–217. DOI: 10.1109/ICCAD.2008.4681576.

[58] Murray Cole. *Algorithmic skeletons: Structured management of parallel computation*. Cambridge, MA, USA: MIT Press, 1991.

[59] Marco Aldinucci and Marco Danelutto. "Skeleton-based parallel programming: Functional and parallel semantics in a single shot". In: *Computer Languages, Systems & Structures* 33.3–4 (2007), pp. 179–192. DOI: 10.1016/j.cl.2006.07.004.

[60] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. DOI: 10.1145/1186736.1186737.

[61] Aashish Phansalkar et al. *Four generations of SPEC CPU benchmarks: what has changed and what has not*. Tech. rep. 2004.

[62] John Hennessy and David Patterson. *Computer architecture – A quantitative approach*. Ed. by D. Penrose. Morgan Kaufmann, 2003.

[63] *CUDA GPUs*. URL: `https://developer.nvidia.com/cuda-gpus` (visited on 02/15/2015).

[64] Allan Gottlieb and Clyde P. Kruskal. "Complexity results for permuting data and other computations on parallel processors". In: *Journal of the ACM (JACM)* 31.2 (1984), pp. 193–209. DOI: `10.1145/62.322423`.

[65] Charles E. Leiserson et al. "The network architecture of the Connection Machine CM-5 (Extended Abstract)". In: *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '92. San Diego, California, USA: ACM, 1992, pp. 272–285. DOI: `10.1145/140901.141883`.

[66] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. "A complexity theory of efficient parallel algorithms". In: *Theoretical Computer Science* 71.1 (1990), pp. 95–132. DOI: `10.1016/0304-3975(90)90192-K`.

[67] Charles E. Leiserson. "The Cilk++ concurrency platform". In: *The Journal of Supercomputing* 51.3 (2010), pp. 244–257. DOI: `10.1007/s11227-010-0405-3`.

[68] R. Behrends et al. *Whitepaper – NVIDIA NVLink High-Speed Interconnect: Application Performance*. Tech. rep. NVIDIA Corporation, 2014, p. 19.

[69] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: `10.1145/1465482.1465560`.

[70] John L Gustafson, Gary R Montry, and Robert E Benner. "Development of parallel methods for a 1024-processor hypercube". In: *SIAM journal on Scientific and Statistical Computing* 9.4 (1988), pp. 609–638. DOI: `10.1137/0909041`.

[71] Guy E Blelloch. "Scans as primitive parallel operations". In: *IEEE Transactions on Computers* 38.11 (1989), pp. 1526–1538. DOI: `10.1109/12.42122`.

[72] Jared Hoberock and Nathan Bell. *Thrust: A parallel template library*. Version 1.7.0. 2010. URL: http://thrust.github.io/ (visited on 02/15/2015).

[73] Edward A. Lee. "The problem with threads". In: *Computer* 39.5 (May 2006), pp. 33–42. DOI: 10.1109/MC.2006.180.

[74] W Daniel Hillis and Guy L Steele Jr. "Data parallel algorithms". In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183. DOI: 10.1145/7902.7903.

[75] James Reinders. *Intel Threading Building Blocks*. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[76] Arch D. Robison. "Composable parallel patterns with Intel Cilk Plus". In: *Computing in Science & Engineering* 15.2 (Mar. 2013), pp. 66–71. DOI: 10.1109/MCSE.2013.21.

[77] Alexander Choong, Rami Beidas, and Jianwen Zhu. "Parallelizing simulated annealing-based placement using GPGPU". In: *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 31–34. DOI: 10.1109/FPL.2010.17.

[78] Christoph Albrecht. *IWLS 2005 Benchmarks*. 2005. URL: http://iwls.org/iwls2005/benchmarks.html (visited on 02/15/2015).

[79] J.B. Goeders, G.G.F. Lemieux, and S.J.E. Wilton. "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition". In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 2011, pp. 41–48. DOI: 10.1109/ReConFig.2011.27.

[80] Chris C. Wang and Guy G.F. Lemieux. "Scalable and deterministic timing-driven parallel placement for FPGAs". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: ACM, 2011, pp. 153–162. DOI: 10.1145/1950413.1950445.

[81] Cristinel Ababei. "Parallel placement for FPGAs revisited". In: *Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA: ACM, 2009, pp. 280–280. DOI: 10.1145/1508128.1508186.

[82] M Xu, Gary Gréwal, and Shawki Areibi. "StarPlace: A new analytic method for FPGA placement". In: *Integration, the VLSI Journal* 44.3 (2011), pp. 192–204. DOI: 10.1016/j.vlsi.2011.02.001.

[83] R.A. Horn and C.R. Johnson. *Topics in matrix analysis*. Cambridge University Press, 1994.

[84] *GeForce 8800 GT specifications*. URL: http://www.nvidia.ca/object/product_geforce_8800gt_for_mac_us.html (visited on 02/15/2015).

[85] *GeForce GT 430 specifications*. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gt-430/specifications (visited on 02/15/2015).

[86] *GeForce GTX 480 specifications*. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications (visited on 02/15/2015).

[87] Michael Frigge, David C. Hoaglin, and Boris Iglewicz. "Some implementations of the boxplot". In: *The American Statistician* 43.1 (1989), pp. 50–54. URL: http://www.jstor.org/stable/2685173 (visited on 02/15/2015).

[88] *GeForce GTX 980 specifications*. URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications (visited on 02/15/2015).

[89] Duane Merrill. *NVIDIA Labs CUB*. Version 1.3.2. 2014. URL: http://nvlabs.github.io/cub/ (visited on 02/15/2015).

[90] Justin Foley. *CUB in Action – some simple examples using the CUB template library*. 2014. URL: http://www.microway.com/hpc-tech-tips/cub-action-simple-examples-using-cub-template-library/ (visited on 02/15/2015).

[91] *Cython: C-Extensions for Python*. URL: http://cython.org (visited on 02/15/2015).

[92] *Installing Python Modules*. URL: https://docs.python.org/2/install (visited on 02/15/2015).

[93] *Quick start guide*. URL: https://github.com/thrust/thrust/wiki/Quick-Start-Guide (visited on 02/15/2015).

[94] *Device backends*. URL: https://github.com/thrust/thrust/wiki/Device-Backends (visited on 02/15/2015).

[95] *NVIDIA CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-downloads (visited on 02/15/2015).