The Dissertation Committee for Shounak Dhar
certifies that this is the approved version of the following dissertation:

# Modern FPGA Placement Techniques with Hardware Acceleration

Committee:

Zhigang (David) Pan, Supervisor

Mahesh A. Iyer, Co-supervisor

Nur A. Touba

Andreas Gerstlauer

Derek Chiou

Christopher Rossbach

# Modern FPGA Placement Techniques with Hardware Acceleration

**by**

**Shounak Dhar**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2019

# Acknowledgments

I would like to express my deepest appreciation to my adviser, Professor David Z. Pan at the University of Texas at Austin and my co-adviser, Dr. Mahesh A. Iyer at Intel Corporation for their fundamental role in my doctoral work. David is a great mentor who properly manages research projects and encourages me to pursue important research problems. He gave me the academic freedom to pursue many different projects while contributing valuable feedback, advice, and encouragement. Mahesh is also a great mentor who has helped me grow as a researcher. He has introduced me to the real-world problems that are most important to industry and has provided me with ample resources needed for working on those problems. This dissertation would not have been possible without their unfailing guidance, support, and encouragement.

I would like to extend my sincere thanks to Dr. Saurabh Adya and Dr. Love Singhal, who were my colleagues at Intel. I had the opportunity to work closely with them and learn about EDA tools from the industry's perspective. Their vision and experiences helped shape my ideas into actual implementations worthy of inclusion in industrial-strength EDA tools.

I would also like to extend my gratitude to my other colleagues from industry. I would like to thank Dr. Aravind Dasu from Intel who has helped me get access to the resources I needed for conducting my research.

I would also like to extend my sincere thanks to the rest of my committee members. I am grateful to Professor Nur A. Touba, Professor Derek Chiou, Professor Andreas Gerstlauer and Professor Christopher Rossbach for their feedback and support.

In addition to my advisor, committee members and colleagues from industry, I am grateful to my colleagues from UT Design Automation Laboratory for their help and feedback. I would like to thank Wuxi Li, Yibo Lin, Subhendu Roy, Xiaoqing Xu, Meng Li, Bei Yu, Jiaojiao Ou, Derong Liu, Biying Xu, Wei Ye, Zheng Zhao, Mohamed Baker Alawieh and Che-Lun Hsu.

Last but not the least, I would like to thank my family. I am deeply indebted to them for their love, support, understanding and sacrifices. Without them, this dissertation would never have been written.

I would also like to express my gratitude towards Intel Corporation for partially supporting the research work in this dissertation through Intel Strategic Research Alliance.

# Modern FPGA Placement Techniques with Hardware Acceleration

Publication No. ____________________

Shounak Dhar, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Zhigang (David) Pan
Co-supervisor: Mahesh A. Iyer

In deep sub-micron technology nodes, Application-Specific Integrated Circuits (ASICs) are becoming expensive to design and manufacture. For this reason, Field Programmable Gate Arrays (FPGAs), which are general purpose and flexible programmable hardware, are gaining more design wins in low volume and fast evolving applications. Modern FPGAs are becoming popular in high performance data analytics, search engines, autonomous cars, communication and networking applications. FPGAs are also accompanied with a complete Computer-Aided Design (CAD) toolchain, that is used to optimally map and fit the design applications or workloads onto the underlying target FPGA device. These design applications mapped onto the FPGA demand high maximum achievable clock frequency (Fmax) and low power consumption while maintaining a low compilation time, which is a major hindrance in widespread adoption of FPGAs.

The focus of this Ph.D. dissertation is the placement problem for FPGAs, which takes a major portion of the FPGA CAD tool runtime. A new algorithm for spreading cells during FPGA global placement is proposed, which achieves better wirelength and routing congestion and takes less runtime than the algorithm used in the state-of-the-art academic FPGA placer. We also propose FPGA acceleration of various subsystems of an analytic global placement algorithm, including wirelength gradient computation and spreading, which achieves significant speedup over the multi-threaded CPU version.

A new detailed placement algorithm is proposed, which offers better tradeoff between quality and runtime compared to existing methods. This algorithm is also accelerated on a GPU and an FPGA, achieving significant speedup over multi-threaded CPU implementation. Another detailed placement algorithm is also proposed which physically re-aligns timing critical paths and improves Fmax with minimal runtime overhead. Both of these algorithms for detailed placement have shown good results on industrial benchmarks and have been integrated into an industrial FPGA CAD tool flow.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In deep sub-micron technology nodes, Application-Specific Integrated Circuits (ASICs) are becoming expensive to design and manufacture. For this reason, Field Programmable Gate Arrays (FPGAs), which are general purpose and flexible programmable hardware, are gaining more design wins in low volume and fast evolving applications. Modern FPGAs are becoming popular in high performance data analytics, search engines, autonomous cars, communication and networking applications. FPGAs are also accompanied with a complete Computer-Aided Design (CAD) toolchain, that is used to optimally map and fit the design applications or workloads onto the underlying target FPGA device. The aforementioned applications mapped onto the FPGA require data processing with very low latencies, which is difficult for CPUs and GPUs. Nevertheless, a high maximum achievable clock frequency (Fmax) is required to meet the latency and throughput targets. Some of these applications also demand low power consumption as that translates to huge energy savings in data centers.

## 1.1 FPGA Architecture

FPGAs typically consist of logic array blocks (LABs), digital signal processors (DSPs), RAMs and Input-Output (IO) blocks in a rectangular grid (Figure 1.1), with interleaved routing resources. FPGAs usually have a separate clock routing network for distributing the clock signal(s) to the registers and other memory elements. Conventionally, specific columns are assigned to RAMs, DSPs, LABs and IOs. Special regular structures like carry chains are implemented using LABs placed contiguously. LABs consist of Adaptive Logic Modules (ALMs), which consist of lookup tables (LUTs), flip-flops (FFs), multiplexers (MUXes) and routing resources. Combinational logic can be implemented using LUTs. LUTs in an FPGA usually have a small number of inputs (4 to 6), so larger blocks of combinational logic are implemented using multiple LUTs. Some FPGAs allow LUTs to be configured as memory (MLABs in Intel® FPGAs). Some FPGA architectures have special fast connections from the output of a LUT to the FF located next to it.

FPGAs may also have analog components like Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) to handle applications such as signal processing. Other hard blocks include transceivers, embedded processor cores, ethernet controllers, PCI(e) controllers, etc. Some FPGAs have block RAMs which can be configured as dual port RAMs with different clocks. Modern FPGAs from leading providers like Intel and Xilinx contain millions of logic elements.

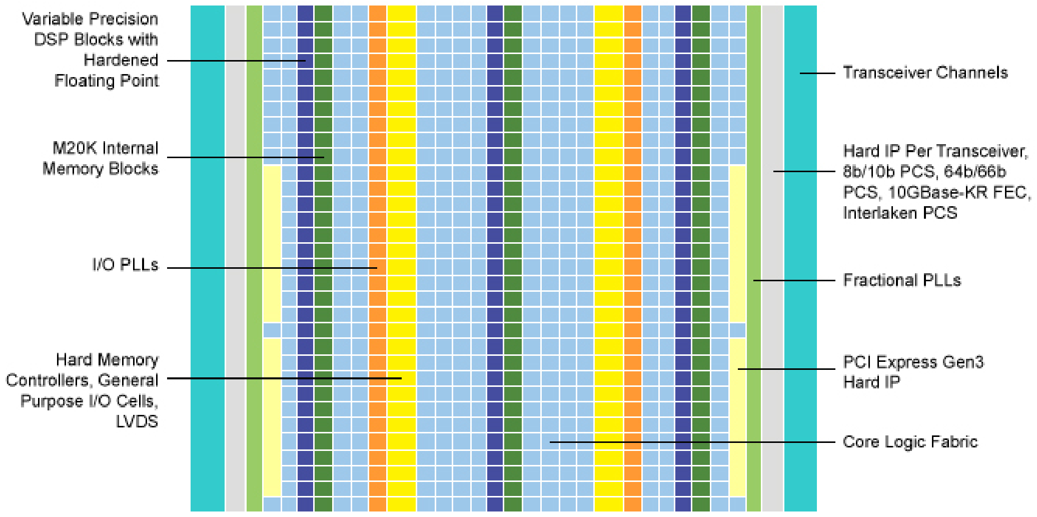

Variable Precision DSP Blocks with Hardened Floating Point
M20K Internal Memory Blocks
I/O PLLs
Hard Memory Controllers, General Purpose I/O Cells, LVDS
Transceiver Channels
Hard IP Per Transceiver, 8b/10b PCS, 64b/66b PCS, 10GBase-KR FEC, Interlaken PCS
Fractional PLLs
PCI Express Gen3 Hard IP
Core Logic Fabric

Figure 1.1: Floorplan of Intel® Arria10® FPGA. Image courtesy of Intel Corporation.

## 1.2 FPGA Design Implementation Flow

FPGA vendors usually provide their own Computer-Aided Design (CAD) tools, which take a description of the design in a high-level language like OpenCL or behavioural description like verilog or VHDL and optimally map and fit the design onto the underlying FPGA device, thereby creating a bitstream that describes the configuration of each logic and routing element in the FPGA. High level synthesis is applied to transform the design from a high level language to a Register-Transfer Level (RTL) description. Logic synthesis is then applied to transform the behavioural description into a netlist with components like LUTs, FFs, RAMs, DSPs, IOs, etc. This netlist has to honor the target FPGA's architectural constraints like number of inputs per LUT, maximum size of a RAM, etc. Then, flat global placement is performed to get locations of netlist components to aid in packing, which involves clustering LUTs and FFs into ALMs and clustering ALMs into LABs. Some LUTs inside a LAB can share inputs. FFs may have different clocks and control signals and there are restrictions on the number of different clocks and control signals and their combination within a LAB. Next, LAB-level global placement and legalization are performed to place the LABs, RAMs, DSPs and IOs on the FPGA grid. This is followed by legalization and detailed placement.

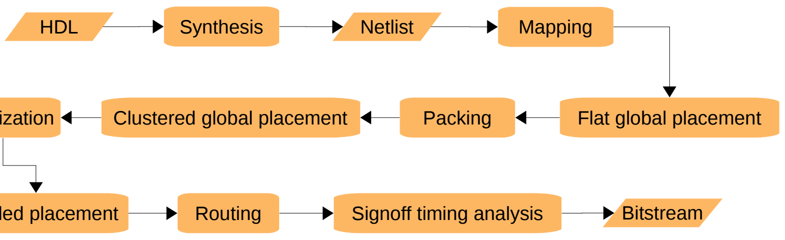


Figure 1.2: FPGA tool flow

The final steps in the flow are routing and signoff timing analysis. Physical synthesis is also performed during various stages of the placement and routing process, where parts of the netlist are re-synthesized to improve timing and resource utilization. The overall CAD tool flow is shown in Figure 1.2.

## 1.3 FPGA Placement and its Evolution

Placement is an important part of the CAD tool flow. Placement determines the physical locations of various components of the netlist and affects metrics like wirelength, timing and routing congestion. Placement can be broadly classified into three stages - global placement, legalization and detailed placement. Global placement determines rough locations of the cells on the chip with approximate models of the aforementioned metrics. Generally speaking, the objective of global placement is to distribute the cells across the floorplan such that the aforementioned metrics are globally optimized with some level of approximation. This makes it amenable and easy for detailed placement algorithms to fine-tune the placement to get the best quality of results.

Simulated-annealing-based approaches [30,53] were popular in the early days of FPGA placement. Simulated annealing involves making numerous moves (like swapping two cells or moving multiple cells in a chain-like fashion). There is a cost associated with each move and moves with high cost are accepted with some probability which depends on a parameter called temperature. The temperature is initially set to a high value, which favors locally suboptimal moves but helps in escaping local minima. The temperature is decreased gradually during the annealing process. Simulated annealing worked for small designs but the quality vs runtime tradeoff grew worse with increasing design size. Researchers subsequently investigated min-cut approaches [54] using graph and hypergraph partitioning techniques like [55], which placed netlist components by recursively partitioning the netlist. These approaches performed better than simulated annealing.

Most of the modern global placers employ an analytical technique [2,4, 18,19,21,22], where metrics like timing, wirelength and routing congestion are modeled as continuous functions of cell locations and optimize these functions using some well-known optimization algorithm. To remove cell-overlap, the placement region is divided into bins and the density of cells in each bin is constrained to some value. Analytical placement can be broadly classified as quadratic [3, 21] and nonlinear [18, 19, 64] based on the choice of objective function(s). Quadratic placement models the objectives as quadratic functions of cell locations while nonlinear placement uses more complicated but more accurate functions. The non-homogeneity in the distribution of resources on the FPGA floorplan make global placement for FPGAs more challenging than ASICs.

Global placement is followed by legalization, which places cells on legal sites while minimizing some cost like total displacement. The legality constraints depend on the FPGA architecture and the target device. Most legalization algorithms are based on network flows [23–26]. Bipartite matching based legalization is used in [21]. Other methods like dynamic programming and linear programming are also used [23].

Detailed placement is applied after legalization to reduce wirelength, fix timing errors and improve routability. Detailed placement refinement improves these metrics by accounting for irregularities and discreteness in the underlying FPGA architecture that may have caused modeling difficulties during global placement. Another objective of detailed placement is to recover from any large displacements caused during legalization. Prior work on detailed placement can be categorized into the following broad classes: 1) Greedy [32] [35] 2) Simulated Annealing [30] [31] [36] 3) Network flow/ matching [33] [34] 4)

Mixed Integer Linear programming [28] [29] 5) Interleaving or Dynamic programming [37] 6) Branch-and-bound [39]. These techniques are discussed in detail in chapter 3. Historically, variations of greedy algorithms have been the most popular methods for detailed placement.

## 1.4 Challenges of FPGA Placement

FPGA vendors are trying to enable fast development of applications from a software developer's point of view. An important drawback of using FPGAs is that it takes a long time to compile a design from RTL/high-level description to a bitstream that can be loaded onto the FPGA. FPGA CAD consists of numerous NP-hard problems which are hard to solve optimally. The best industrial FPGA CAD tools available today take several hours on average to compile a large design on state-of-the-art FPGA devices. This is extremely slow compared to compiling code for GPU or CPU. Reduction in the compilation times along with improvements to the quality of the CAD tools could encourage higher adoption of FPGAs and FPGA based solutions.

The growing size and complexity of modern FPGAs and the designs mapped onto them imposes tough requirements on the quality of the CAD tools. Global placement quality affects metrics like wiring usage, timing, power and routing congestion to a great extent. Many researchers have focused on the numerical optimization part of global placement and this is fairly mature both in ASIC and FPGA design flows. Optimizing the aforementioned metrics alone would lead to overlaps among cells, which need to be removed by spreading the cells. Spreading algorithms have significant room for improvement since research in spreading algorithms has not been as thorough. To the best of our knowledge, there is no previous work studying the relationship between the shape of placement after spreading and wirelength. Most existing works do not consider optimal placement shapes for spreading; instead they indirectly spread the cells in rectangular, diamond, or circular shapes. We need to study placement shapes in order to find the best way to spread cells. Also, the spreading algorithm should be parallelizable and should try to preserve the relative order among cells obtained from numerical optimization.

In recent times, artificial intelligence (AI) and machine learning (ML) have found various interesting applications like autonomous vehicles, internet search, etc. The scale of the algorithms in AI and ML demands extreme compute efficiency, both for training and inference, This has necessitated the use of hardware acceleration for these advanced applications using platforms like GPUs, FPGAs and custom ASICs. To enable the acceleration requirements of these advanced applications on FPGAs, we need to improve the compute efficiency and quality of results of the FPGA CAD flow. Given this, ideally, we need to design global placement algorithms for numerical optimization and spreading that are easily acceleratable on the FPGA itself.

There has been extensive research on detailed placement for ASICs. The resource constraints and the multitude of legality rules on an FPGA make the detailed placement problem harder for an FPGA than an ASIC. To cope with these difficulties, we need to develop new FPGA detailed placement algorithms. These algorithms also need to be highly parallelizable and ideally, also hardware acceleratable to meet runtime constraints.

Timing is an important metric for detailed placement because it is hard to model timing accurately during global placement. The high number of pins per LAB in an FPGA imply that a large number of timing paths can pass through one LAB. So, traditional timing driven placement techniques used in ASICs perform poorly when applied to FPGAs. Hence, we need to develop timing driven placement techniques that are targeted to FPGAs. Traditional timing driven placement approaches consist of net based and path based optimizations. Net-based algorithms optimize delays on individual nets while path-based placers optimize timing on entire paths. Both net-based and path-based optimizations are needed as they offer different benefits and complement each other.

## 1.5 Summary of Contributions

This dissertation presents new algorithms for both global placement and detailed placement, that show significant improvement over the state-of-the-art with respect to wiring usage, timing and runtime. We also propose techniques to hardware accelerate several parts of these new placement algorithms on FPGA and GPU.

Chapter 2 explores new global placement algorithms and their hardware acceleration. This chapter proposes hybrid CPU-FPGA acceleration of wirelength gradient computation, which is a major bottleneck in global placement. A study is performed on the effect of placement shapes on wirelength and a new shape-driven spreading algorithm is also proposed. The spreading problem is formulated as a min-cost flow problem and is solved using linear programming. We also propose a new fluid-flow based spreading algorithm which is highly parallelizable and accelerate it on FPGA.

Chapter 3 explores new detailed placement placement algorithms and their hardware acceleration. A dynamic-programming-based detailed placement algorithm is proposed, which has a parameter to control the tradeoff between quality and runtime. Optimal solutions to the single row placement problem can also be obtained by simply setting the parameter to the appropriate value. GPU acceleration of this detailed placement algorithm is also proposed, which achieves significant speedup compared to multi-threaded CPU versions. We compare hybrid CPU-GPU acceleration and full GPU acceleration and show that the full GPU accelerated detailed placer is faster. We also propose FPGA acceleration of the same detailed placement algorithm. A new timing-driven placement algorithm is also proposed, which focuses on critical paths and is tailored towards high connectivity netlists like those for FPGAs. Timing improvements from this algorithm complement the improvements from our dynamic-programming-based detailed placer.

Chapter 4 concludes this dissertation and discusses potential future directions for FPGA placement and its acceleration.

# Chapter 2

# Global Placement

Placement is an important part of any CAD tool flow (Figure 1.2) which consumes a large portion of the runtime, of which global placement is a major part (for example, flat and clustered global placement together take 60.1% of the runtime in [21]). Placement determines the physical locations of cells on a chip and affects important metrics like timing, power and routing congestion.

The current state-of-the-art placers take the analytic approach, where metrics like wirelength, timing etc. are modelled as continuous functions of cell locations [3,18,19,21,64] and optimized through well-known optimization algorithms. Analytical placement can be broadly classified as quadratic [3,21] and nonlinear [18, 19, 64] based on the choice of objective function(s). Quadratic placement models the objectives as quadratic functions of cell locations and

---

This chapter includes work from the following publications:
[8] S. Dhar, L. Singhal, M. A. Iyer and D. Z. Pan, *"FPGA Accelerated FPGA Placement"*, International Conference on Field-Programmable Logic and Applications, 2019.
[1] S. Dhar, L. Singhal, M. A. Iyer and D. Z. Pan, *"A Shape-Driven Spreading Algorithm using Linear Programming for Global Placement"*, Asia and South Pacific Design Automation Conference, 2019.
[9] S. Dhar, L. Singhal, M. A. Iyer and D. Z. Pan, *"FPGA Accelerated Spreading for Global Placement"*, IEEE High Performance Extreme Computing conference, 2019.
In all of these publications, the first author was responsible for developing and implementing the main ideas and conducting experiments.

the optimization process for these functions usually reduces to solving a system of linear equations. For nonlinear placement, the functions are optimized using variants of gradient descent. Nonlinear placement usually performs better than quadratic placement, although the difference in quality is small [3]. Optimizing metrics like wirelength, timing and routing congestion alone leads to overlaps between cells which need to be removed. The chip is usually divided into a grid of bins and the density of cells in each bin serves as a rough measure of cell overlap. The state-of-the-art global placement techniques can be broadly classified into two groups on the basis of techniques used to remove cell overlap: i) Continuous ii) Upper-and-lower-bound. Continuous methods like [19], [18] and [2] usually add a smooth density function to the objective and use the resultant gradients to spread the cells gradually. Upper-and-lower-bound methods like [20] and [21] find a placement which satisfies the bin density constraints after each iteration of optimizing the objective function. The cells are anchored to their new locations after each spreading step and a penalty function for displacement from these locations is added to the objective function for the next placement iteration. The penalty is increased every iteration, ensuring that the gap between the numerical optimization and spreading steps reduces at each iteration, making the placement converge. Spreading or rough legalization greatly impacts the quality of the final solution. In this dissertation, we use the upper-and-lower-bound technique.

## 2.1 FPGA Acceleration of Wirelength Gradient Computation

It's not uncommon for state-of-the-art academic and industrial placement engines to run for tens of minutes on large designs, even with multi-threading enabled. Recently, researchers have started investigating GPU acceleration of placement [11–13]. Some of these works have achieved decent speedups over multi-threaded CPU implementations.

The leading FPGA vendors have their own tools which run on CPUs. In this section, we investigate acceleration of placement on an FPGA itself, which is a more sensible option for a customer who already has FPGA(s), rather than having to use a GPU just for the purpose of programming the FPGA. Modern FPGAs are complex SoCs which have processor cores [68, 70]. Intel recently announced a CPU+FPGA on the same package [67]. We investigate the possibility of using these SoCs/hybrid systems to accelerate the CAD tools without relying on any additional hardware.

FPGAs have outperformed CPUs and GPUs on performance per watt for many important applications [16]. EDA vendors offering CAD tool services on the cloud [69] might be interested in saving power, thus making a stronger case for choosing FPGAs over GPUs for accelerating CAD algorithms.

Many of the analytical placement algorithms involve computation of expensive functions like exponents that are good candidates for FPGA acceleration. Pipelined computation of these functions gives high throughput.

Designing an application for FPGAs poses unique challenges, the most important being the switch from a control flow to a data flow paradigm. FPGAs are capable of supporting very deep pipelines, so the application has to be designed to maximize uninterrupted data flow through the pipeline, i.e, without stalls and dependencies. Applications involving random accesses over a large memory space are not particularly suitable for FPGAs, so we have to engineer our applications to avoid such access patterns.

Researchers have recently looked into acceleration of nonlinear analytical placement on GPUs [11,12]. Runtime of nonlinear analytical placement is usually dominated by gradient computation. Gradient refers to the gradient of the objective function which models metrics like wirelength and timing. To minimize overlap between cells, the placement region is usually divided into bins and a constraint is put on the maximum area of cells in each bin.

Hardware acceleration of a placement algorithm involving cell moves has been proposed in [7]. However, that is mostly applicable to detailed placement. In this section, we present a solution to accelerate analytical global placement on FPGAs [8]. To the best of our knowledge, this is the first work on acceleration of analytical global placement on FPGAs. The key contributions in this section are as follows:

- We propose hybrid CPU-FPGA acceleration of wirelength gradient computation, leveraging unique capabilities of each device and mapping the right kind of computation on them.

- We propose a stall-free pipelined hardware architecture to accelerate pin gradient computation on FPGAs. Our proposed design accesses memory in a simple streaming fashion, eliminating the overhead associated with random accesses.
- Our proposed hardware architecture also computes the objective function being optimized with no runtime overhead, unlike CPUs and GPUs.

### 2.1.1 Problem Statement for Upper-and-Lower-Bound Nonlinear Global Placement

Almost all placers incorporate wirelength as one of the optimization metrics. Half-Perimeter Wirelength (HPWL) is a commonly-used metric for wirelength. The total HPWL is the sum of HPWLs of all nets. HPWL of a net can be defined as follows:

$$xHPWL_{net} = \max_{u \in net}(x_u) - \min_{u \in net}(x_u) \tag{2.1a}$$

$$yHPWL_{net} = \max_{u \in net}(y_u) - \min_{u \in net}(y_u) \tag{2.1b}$$

$$HPWL_{net} = xHPWL_{net} + yHPWL_{net} \tag{2.1c}$$

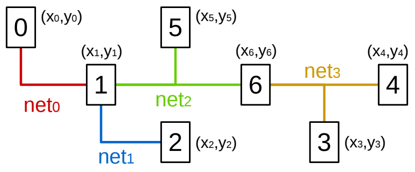


Figure 2.1: Sample netlist

Where $x_u$ and $y_u$ denote the x and y locations of pin $u$ of the net.

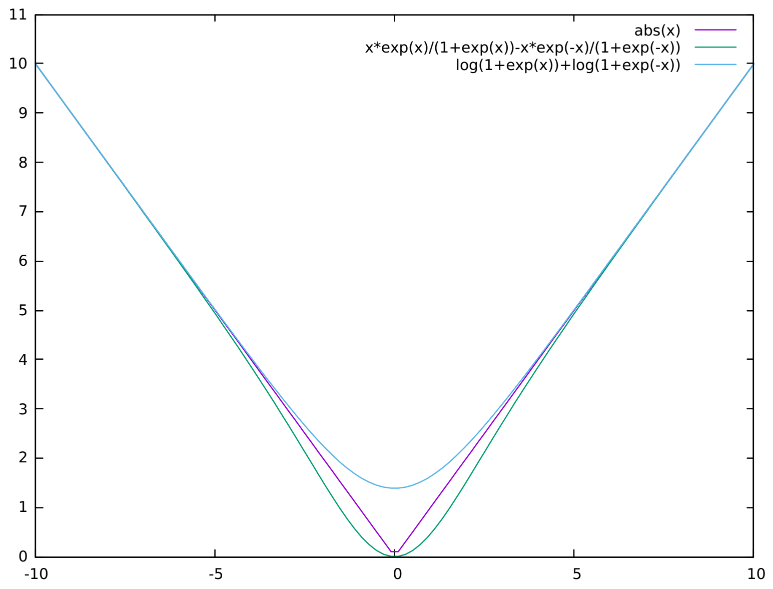


Figure 2.2: HPWL, weighted average and log-sum-exponent wirelengths in x direction for a 2-pin net with one pin fixed at 0.

Analytical placement approaches using gradient descent require that the wirelength function be differentiable, which is not the case with HPWL because of the min(.) and max(.) functions. Nonlinear analytical placers solve this problem by approximating HPWL by a differentiable function. Weighted average (WA) and log-sum-exponent (LSE) are two such mathematical approximations that previous nonlinear placers have used [18, 19, 64]. The weighted average wirelength for a net can be defined as:

$$xWA_{net} = \frac{\sum\limits_{u \in net} x_u e^{\gamma x_u}}{\sum\limits_{u \in net} e^{\gamma x_u}} - \frac{\sum\limits_{u \in net} x_u e^{-\gamma x_u}}{\sum\limits_{u \in net} e^{-\gamma x_u}} \tag{2.2a}$$

$$yWA_{net} = \frac{\sum\limits_{u \in net} y_u e^{\gamma y_u}}{\sum\limits_{u \in net} e^{\gamma y_u}} - \frac{\sum\limits_{u \in net} y_u e^{-\gamma y_u}}{\sum\limits_{u \in net} e^{-\gamma y_u}} \tag{2.2b}$$

$$WA_{net} = xWA_{net} + yWA_{net} \tag{2.2c}$$

The log-sum-exponent model is defined as:

$$xLSE_{net} = \frac{log\left(\sum\limits_{u \in net} e^{\gamma x_u}\right) + log\left(\sum\limits_{u \in net} e^{-\gamma x_u}\right)}{\gamma} \tag{2.3}$$

where $\gamma$ is a parameter used to control the smoothness of the approximation. A higher $\gamma$ means the model is less smooth but more accurate. The weighted average model has a lower error of approximation, as mentioned in [19]. Hence, we choose to use the weighted average model for our work. Also, weighted average wirelength is strictly less than HPWL, approaching HPWL asymptotically as shown in Figure 2.2. The partial derivative of the wirelength of a net for the weighted average model is given by (assuming $cell_j \in net$):

$$\begin{aligned} \frac{\partial(xWA_{net})}{\partial x_j} = {} & \frac{(1+\gamma x_j)e^{\gamma x_j}}{\sum\limits_{i \in net} e^{\gamma x_i}} - \frac{\gamma\left(\sum\limits_{i \in net} x_i e^{\gamma x_i}\right)e^{\gamma x_j}}{\left(\sum\limits_{i \in net} e^{\gamma x_i}\right)^2} \\ & - \frac{(1-\gamma x_j)e^{-\gamma x_j}}{\sum\limits_{i \in net} e^{-\gamma x_i}} - \frac{\gamma\left(\sum\limits_{i \in net} x_i e^{-\gamma x_i}\right)e^{-\gamma x_j}}{\left(\sum\limits_{i \in net} e^{-\gamma x_i}\right)^2} \end{aligned} \tag{2.4}$$

The partial derivative for LSE model is given by:

$$\frac{\partial(xLSE_{net})}{\partial x_j} = \frac{e^{\gamma x_j}}{\sum\limits_{i \in net} e^{\gamma x_i}} - \frac{e^{-\gamma x_j}}{\sum\limits_{i \in net} e^{-\gamma x_i}} \tag{2.5}$$

The total wirelength gradient can be obtained by summing up the individual gradients for each pin (connection) of each cell.

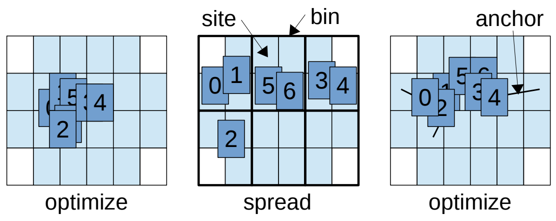


Figure 2.3: Progression of placement with alternate optimization and spreading. Each bin here is 2x2 sites and holds at most 2 cells.

The placement region is usually divided into bins and a constraint is put on the maximum area of cells in each bin. We use a rough-legalization approach like [3, 21] where cells are spread out after a phase of optimizing the objective function. For the next phase of optimization, the cells are 'anchored' to the new locations obtained after spreading and a penalty is applied for displacement from these anchor locations (Figure 2.3). Analytical placement with the weighted average model can then be formulated as an optimization problem:

$$\underset{x,y}{min}\left(\sum_{n\in nets} WA_n + \lambda \sum_{c\in cells}(x_c - x'_c)^2 + \lambda \sum_{c\in cells}(y_c - y'_c)^2\right) \tag{2.6}$$

Where $(x'_c, y'_c)$ is the anchor location of cell $c$ and $(x_c, y_c)$ is the location we want to solve for. $WA_n$ is the weighted average wirelength for net $n$. The parameter $\lambda$ controls the balance between spreading and optimization. A lower value of $\lambda$ favours less cell overflow and a higher $\lambda$ favours more wirelength optimization. We initialize it to a low value and increase it gradually for subsequent placement iterations.

### 2.1.2 Wirelength Gradient Computation

We present two different methods to accelerate wirelength gradient computation on FPGA. We first simplify our formulation to make it hardware friendly. All occurrences of $\gamma x$ in equation 2.2a can be replaced by $\hat{x}$:

$$xWA_{net} = \frac{\sum\limits_{u \in net} \hat{x}_u e^{\hat{x}_u}}{\gamma \sum\limits_{u \in net} e^{\hat{x}_u}} - \frac{\sum\limits_{u \in net} \hat{x}_u e^{-\hat{x}_u}}{\gamma \sum\limits_{u \in net} e^{-\hat{x}_u}} \tag{2.7}$$

The $\lambda$ in equation 2.6 can be scaled up by $\gamma$ so that the optimal solution remains unchanged. Thus the effect of changing $\gamma$ can be captured instead by scaling $x$ (and $y$), which eliminates many multiplication operations.

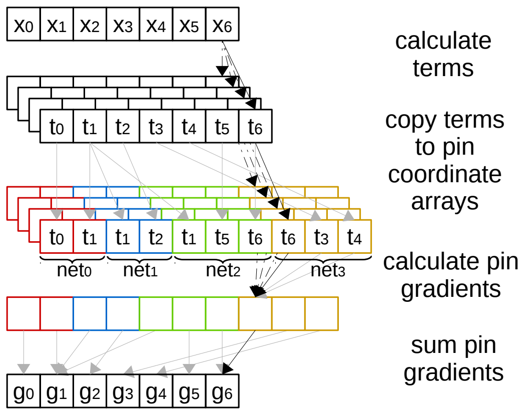


Figure 2.4: Gradient computation on CPU

We only accelerate the gradient computation for nets with $\leq$ 16 pins as high-fanout nets are harder to handle on an FPGA and the degradation in maximum operating frequency and FPGA resource usage outweigh the benefits. The number of large nets in real benchmarks is relatively small. For example, in design FPGA12 of the ISPD 2016 FPGA placement contest benchmarks [71], nets with more than 16 pins account for <0.25% of all nets. Our global placer produces slightly better result on average than [21] even though we ignore larger nets.

There are many ways of handling large nets. One way is to compute their gradients on CPU. An alternative way is to apply simple clustering and break them up into smaller nets with $\leq 16$ pins with a common driver. This is a good choice since HPWL grossly underestimates the routed wirelength for large nets. Timing is usually modelled through 2-pin nets for each driver-to-load connection, so this approach does not interfere with correct modeling of timing.

#### 2.1.2.1 CPU implementation

Before we discuss hardware acceleration, we would like to describe our baseline multi-threaded CPU implementation and identify the runtime bottlenecks. Gradient computation can be divided into 3 steps: calculating exponents, computing the gradient for each pin of each net and computing the gradient w.r.t cell locations by adding the corresponding pin gradients. Figure 2.4 shows the calculations involved in gradient computations step-by-step. First, 4 kinds of terms ($e^{x_i}$, $e^{-x_i}$, $x_ie^{x_i}$, $x_ie^{-x_i}$) are computed for each $x_i$. Then, we copy these terms into 4 expanded arrays. Each entry in an array corresponds to a pin of a net. For example, in Figure 2.1, cell 1 is connected to nets 0, 1 and 2, so $e^{x_1}$, $e^{-x_1}$, $x_1e^{x_1}$ and $x_1e^{-x_1}$ each appear at indices 1, 2 and

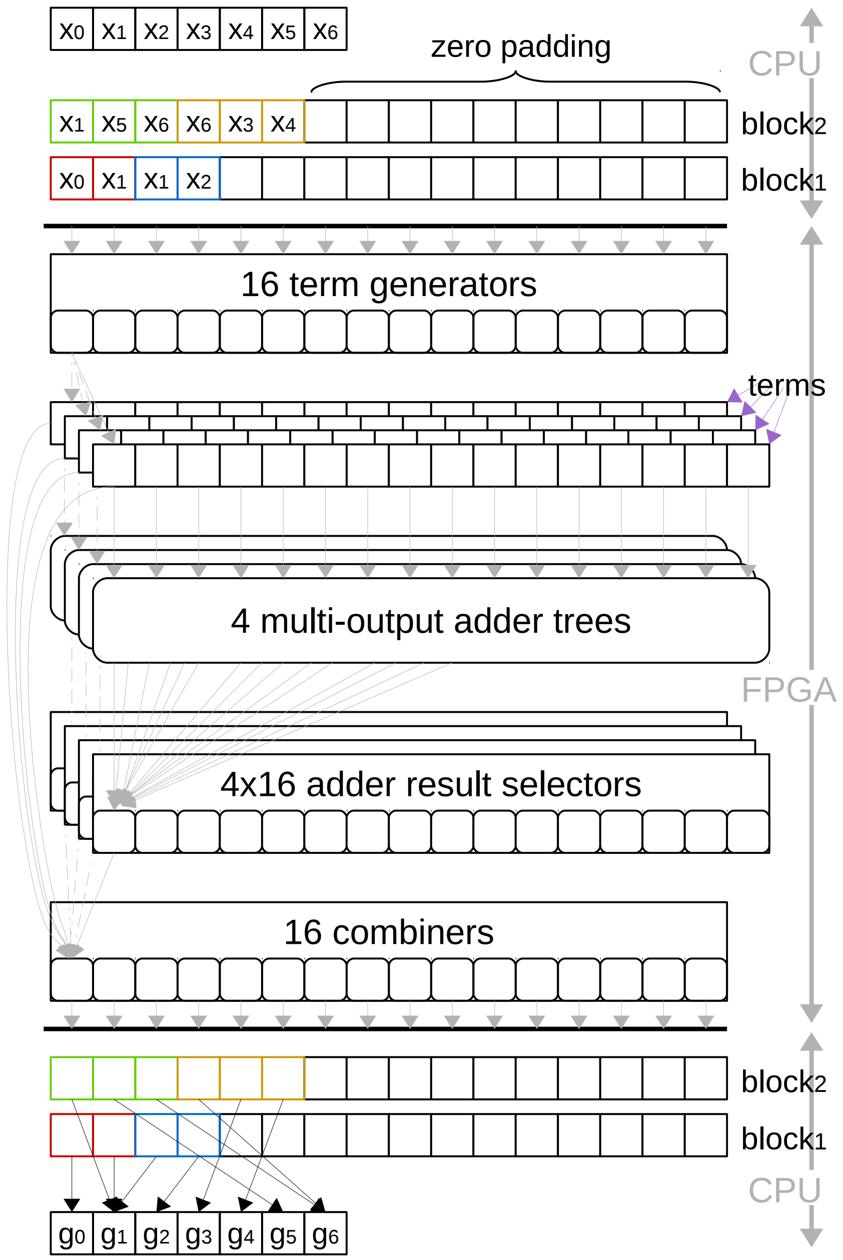


Figure 2.5: Gradient computation on CPU+FPGA. Some of the data dependencies are shown by arrows.

4 in their corresponding expanded arrays in Figure 2.4.

Computation of the terms is parallelized in a straightforward way. Pin gradient computation is parallelized by assigning groups of nets to different threads. Cell gradient computation is parallelized by assigning groups of cells to different threads, where, for each cell, a thread iterates over the corresponding entries in the pin gradient array and sums them up.

#### 2.1.2.2 FPGA Acceleration: Method 1

Iterating over pins of a net causes irregular memory accesses over a large chunk of memory, which is not suitable for FPGAs (random access over small chunks is ok as that can be mapped to on-chip RAM). Hence, we rearrange the computations and assign the parts involving random memory accesses to the CPU and the rest to the FPGA. Our hybrid CPU+FPGA implementation is a 3 step process (Figure 2.5): First, we copy the coordinates(x/y) to special arrays of pin coordinates (slightly different from our CPU implementation). Then we calculate gradient for each pin on the FPGA. Finally, we sum the gradients on CPU.

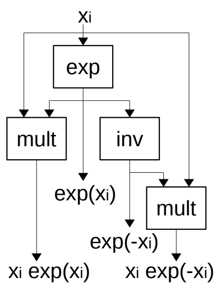


Figure 2.6: Term generator

We sort the nets by degree when we read in the netlist. This is a simple bucket sort, as the number of pins can be 2,3,...16. We use the same hardware to process nets of different degrees. We divide the nets into blocks containing 16 pins each, with zero padding, if necessary. Nets in a block are of the same degree. For example, in Figure 2.5, block 1 contains nets 0 and 1 and block 2 contains nets 2 and 3. A block can accommodate up to 8 2-pin nets or 5 3-pin nets or 2 6-pin nets, etc. Each entry in a block stores the coordinate (x or y) for the corresponding pin. Note that x and y gradients are computed separately. For each of these 16 entries, we calculate the following terms in parallel using term generators (Figure 2.6) : $e^{x_i}$, $e^{-x_i}$, $x_ie^{x_i}$ and $x_ie^{-x_i}$.

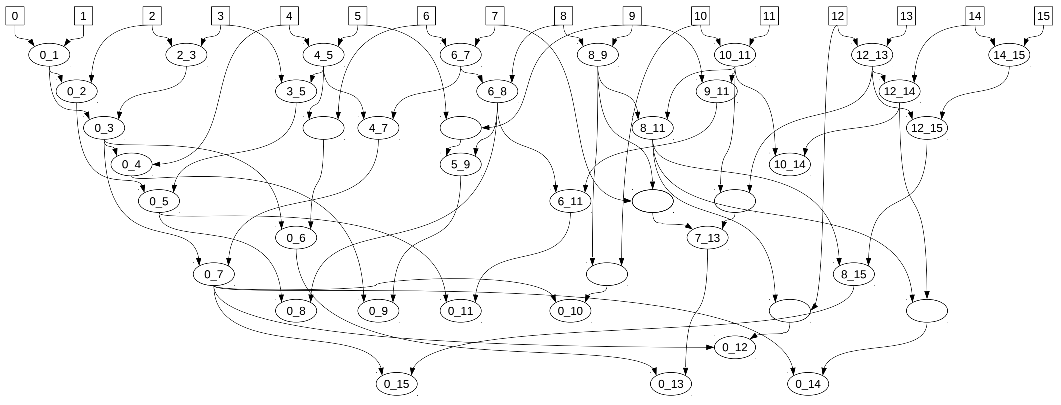


Figure 2.7: Multi-output adder tree. Each oval denotes a 2-input adder. Labels of the form a_b denote summations of inputs a,a+1,...,b. There are 34 labelled outputs, which are fed to various adder result selectors (Figure 2.8). Any path in this tree goes through at most 4 adders (depth=4).

Next, we have 4 adder trees, one for each kind of term. These adder trees calculate partial sums of terms for the 34 possible cases of contiguous net segments (8 2-pin + 5 3-pin + 4 4-pin + ... + 1 15-pin + 1 16-pin). Figure

2.7 shows a generic adder tree that can be used for any type of term. Consider the example of net 3 in Figure 2.1:

$$\begin{aligned}\frac{\partial(xWA_3)}{\partial x_6} &= \frac{(1+x_6)e^{x_6}}{e^{x_6}+e^{x_3}+e^{x_4}} - \frac{(x_6e^{x_6}+x_3e^{x_3}+x_4e^{x_4})e^{x_6}}{(e^{x_6}+e^{x_3}+e^{x_4})^2} \\ &-\frac{(1-x_6)e^{-x_6}}{e^{-x_6}+e^{-x_3}+e^{-x_4}} - \frac{(x_6e^{-x_6}+x_3e^{-x_3}+x_4e^{-x_4})e^{-x_6}}{(e^{-x_6}+e^{-x_3}+e^{-x_4})^2}\end{aligned} \tag{2.8}$$

The adder tree for $e^{x_i}$ terms has $e^{x_6}+e^{x_3}+e^{x_4}$ as one of its outputs. The adder trees enable sharing large portions of logic in the entire kernel, which leads to a smaller area than would have been possible with implementing a separate kernel for each net degree (2,3,...16).

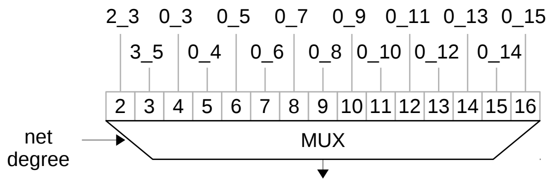


Figure 2.8: Adder result selector for index 3. Labels of the form a_b are the outputs of the corresponding adders in Figure 2.7

The correct adder output required for computing a pin's gradient is selected through an adder output selector depending on the degree of nets in the block being processed (Figure 2.8). Each such selector takes 15 of the 34 outputs from an adder tree. Selectors for different indices (pins) may take different subsets of the 34 outputs. Some of these selectors may be the same (for example, we can use the same selector for indices 0 and 1). The selector in Figure 2.8 corresponds to index 3. The possible arrangements of inputs for this selector are: (i) A 2-pin net spanning indices 2 and 3 (ii) A 3-pin net spanning indices 3, 4 and 5. (iii) A k-pin net spanning indices 0, 1, ... k-1 with 3<k≤16. Going back to Equation 2.8, we see that the selector in Figure 2.8 would select the sum of terms at indices 3, 4 and 5 in the $e^{x_i}$ terms array, which is $e^{x_6} + e^{x_3} + e^{x_4}$. The same applies for the other 3 kinds of terms.

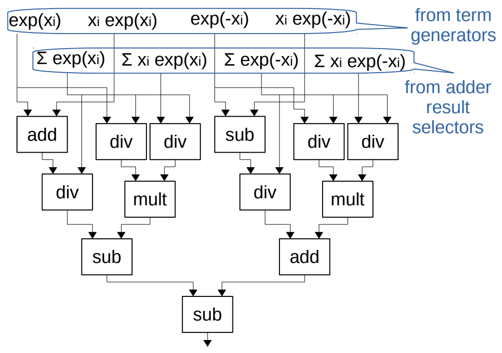


Figure 2.9: Combiner. Implements Equation 2.4

Finally, we compute the individual pin gradients using combiners (Figure 2.9). Each combiner takes as input 4 terms and 4 outputs of adder result selectors (sums of terms). We have arranged the operations to minimize floating point overflow. For example, instead of multiplying two numerically large expressions together and then dividing by some other expression, we do the division first. The same applies for small expressions in the denominator.

Since we club all the nets with the same degree together, we do not have to provide input regarding the degree for each block. We only need to change it when we start processing a block with a different degree. We can also build a max-tree structure similar to the adder tree to normalize the coordinates of pins for each net with respect to the largest coordinate. However, in practice, we found that it has very little impact on solution quality. We also calculate the wirelengths for the nets in each block using net wirelength calculators

(Figure 2.10) and add them using a simple 34-to-1 adder tree. The total wirelength is available as soon as computation on all the blocks are done (plus a few clock cycles to flush the pipeline). This is different from a CPU or GPU implementation where we need additional passes to sum the wirelength for all nets. Our implementation does not require any additional runtime.

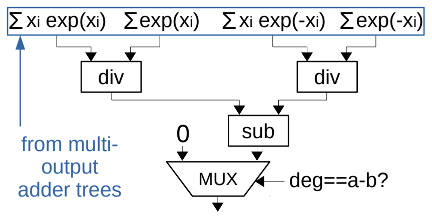


Figure 2.10: Net wirelength calculator (Equation 2.2a) for adder tree output a_b. 34 of these are instantiated, each corresponding to an adder tree output.

Note that using $\sim$ 100% of all 16 slots in a block (i.e, no zero-padding) is possible only for certain net degrees (2-pin, 4-pin, 8-pin and 16-pin), assuming there is a large number of nets for each such degree. Degrees 9, 10 and 11 lead to <75% efficiency. However, in the benchmark suite that we tested, the average efficiency was 90.6%.

#### 2.1.2.3 FPGA Acceleration: Method 2

Like method 1, this method also computes pin gradients on FPGA and computes their sums on CPU to get cell gradients. The basic building block for this implementation is a pipeline, which reads in pin coordinates sequentially and also writes pin gradients sequentially at a rate of one pin per clock cycle. We insert markers in the pin coordinate array to denote the end of a net (Figure 2.12). A marker can be a special value like -1 which the cell

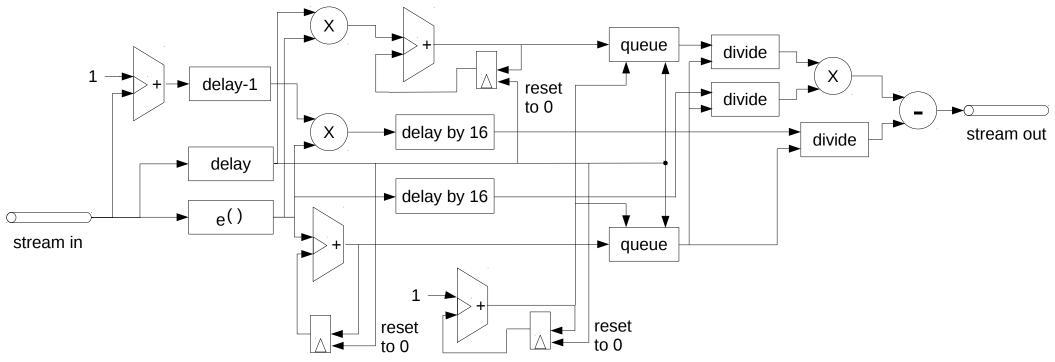


Figure 2.11: Part of a pipeline for computing gradient from equation 2.4. Computation is shown for the first two terms only to reduce clutter. The circuit for the remaining two terms is similar. We instantiate 16 such pipelines which operate in parallel.

coordinates will never take. The total number of elements in this array is number of pins + number of nets. Like method 1, we compute exponents of the incoming coordinates. Appropriate delay blocks are inserted to match the delay of the exponentiation block. The sum of the exponents is calculated and stored in a single-cycle accumulator which resets on encountering a marker. A schematic for a pipeline is shown in Figure 2.11. We instantiate 16 such pipelines on the FPGA to increase throughput.

While calculation of exponents and their sums is straightforward, it is nontrivial to divide each of the individual exponents by the calculated sum since the number of division operations depends on the number of pins of a net which can vary. We tackle this problem by using queues. The individual exponents are stored in a shift register of sufficient length ("Delay by 16" in Figure 2.11) and the calculated sum is put in a queue with a counter equal to

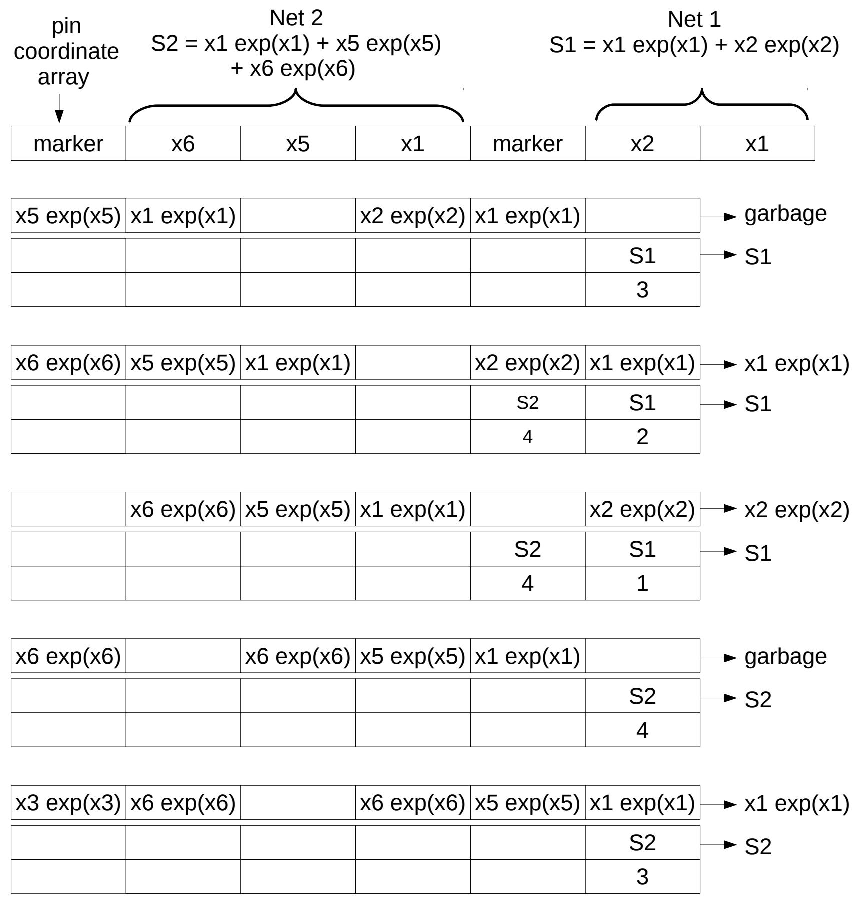


Figure 2.12: Functioning of a queue for 5 consecutive clock cycles

one plus the number of pins for that net. The queue reads the sum and count values whenever it detects that the next element in the pin coordinate array is a marker. Once a sum gets to the beginning of the queue, it stays there for the same number of cycles as the number of pins in the net. This is enforced by the counter, which decrements every cycle once the sum is at the beginning of the queue. The sum is popped from the queue when the counter reaches 0 (Figure 2.12). The pipeline needs to be flushed with a dummy 16-pin net at the beginning in order to initialize the queue(s) to the correct state(s).

Note that we need one marker per net in this implementation, so a k-pin net will be represented by k+1 entries in the input stream. The overall efficiency for a k-pin net is k/(k+1). This is inefficient for small nets (ex: 2-pin net: efficiency 2/3; 3-pin net: efficiency 3/4) but is efficient for larger nets. The maximum degree of nets this implementation can support depends on the depth of the queue. Deeper queues require more resources on the FPGA and also result in low Fmax. We have synthesized logic for up to 16 pin nets only.

Like method 1, this implementation can also calculate wirelength with no additional runtime penalty. HPWL for each net can be accumulated on encountering a marker, which denotes the end of the net.

### 2.1.3 Results

We tested our implementation on the International Symposium on Physical Design (ISPD) 2016 FPGA placement contest benchmarks [71]. The benchmark details are given in Table 2.1. We implemented a global placer

Table 2.1: Benchmark Statistics and Wirelength

| Design | $\frac{\#cells}{10^3}$ | $\frac{\#nets}{10^3}$ | Wirelength/$10^3$ | |
|---|---|---|---|---|
| | | | [21] | Ours |
| FPGA01 | 105 | 105 | 304 | 267 |
| FPGA02 | 166 | 167 | 655 | 583 |
| FPGA03 | 421 | 428 | 2654 | 3338 |
| FPGA04 | 423 | 430 | 4903 | 5351 |
| FPGA05 | 425 | 433 | 8767 | 8087 |
| FPGA06 | 704 | 713 | 4351 | 4926 |
| FPGA07 | 707 | 716 | 8040 | 7441 |
| FPGA08 | 717 | 725 | 7608 | 6672 |
| FPGA09 | 867 | 876 | 9542 | 8816 |
| FPGA10 | 952 | 961 | 4294 | 4628 |
| FPGA11 | 845 | 851 | 9260 | 8291 |
| FPGA12 | 1103 | 1111 | 4998 | 4802 |
| Ratio | | | 1 | 0.979 |

similar to the one in [21] (which is an improved version of the placer that won the contest) except that we use weighted average wirelength model instead of the quadratic wirelength model. We also implemented a parallelized spreading algorithm. We use Nesterov's accelerated gradient descent to optimize the objective function. We compared the wirelength numbers after global placement with [21]. Our placement bins are of the same dimensions as [21] and we use the same algorithm to place the cells within each bin. Table 2.7 shows that our placer produces 2.1% better wirelength on average after global placement. RAMs, DSPs and IOs are placed on legal sites in both these cases.

We ran our experiments on a machine with a 14 core, 28 thread Intel® Xeon® processor and an Intel® Arria10® FPGA on the same package. This setup allows allocation of shared memory and low latency communication between the CPU and the FPGA. All our reported runtimes include the time for moving data to and from the FPGA. Our FPGA kernels were written in OpenCL and compiled using Altera Offline Compiler version 16.0, which uses Quartus for synthesis, placement, routing and timing analysis. The compiler inserts extra pipeline stages as necessary. We use 32-bit floating point numbers for cell coordinates. The compiler for our CPU code uses SSE instructions wherever they are beneficial. We also disassembled the binary to verify SSE instruction usage.

Table 2.2: FPGA Resource Usage by Type and Fmax

| Kernel(s) | Logic | Register | RAM | DSP | Fmax |
|---|---|---|---|---|---|
| Method1 | 38% | 32% | 25% | 84% | 227 MHz |
| Method2 | 60% | 57% | 55% | 68% | 205 MHz |

The total resource utilization and maximum frequency (Fmax) are shown in Table 2.2 for the two different kernels. The queues in method 2 are responsible for the high register and RAM usage. We decided not to pursue method 2 due to the lower Fmax. Note that part of the FPGA is reserved for “blue bitstream”, which is responsible for communicating with the CPU and maintaining cache coherency.

Table 2.3 shows the runtimes for different parts of the global placement flow. Our placement algorithm has two major parts - numerical optimization and spreading. Numerical optimization is done using Nesterov’s accelerated gradient descent method, which involves computing gradient and taking gradient steps with Nesterov momentum. Wirelength gradient computation for our

Table 2.3: Efficiency and Runtimes in seconds for various parts of Global Placement

| Design | Efficiency % | | Wirelength Gradient | | | | | Global Placement | | | | | Kernel | Optimization |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | CPU | | | FPGA +CPU | Speedup | CPU | | | FPGA +CPU | Speedup | FPGA | FPGA +CPU |
| | Method1 | Method2 | 1T | 4T | 28T | | | 1T | 4T | 28T | | | | |
| FPGA01 | 90.4 | 77.6 | 65 | 23.5 | 5.9 | 2.76 | 2.14 | 102 | 37.5 | 15.3 | 12.32 | 1.24 | 1.69 | 3.11 |
| FPGA02 | 88.6 | 78.7 | 189 | 64.1 | 13.7 | 4.31 | 3.18 | 229 | 80.8 | 24.3 | 14.91 | 1.63 | 2.63 | 4.82 |
| FPGA03 | 89.6 | 78.4 | 630 | 225.7 | 41.6 | 13.45 | 3.09 | 681 | 254.6 | 56.6 | 28.36 | 2.00 | 6.13 | 15.09 |
| FPGA04 | 89.4 | 78.9 | 692 | 183.8 | 44.0 | 14.29 | 3.08 | 745 | 205.5 | 59.1 | 29.84 | 1.98 | 6.35 | 15.92 |
| FPGA05 | 90.0 | 78.9 | 704 | 188.6 | 45.5 | 14.53 | 3.13 | 758 | 210.6 | 60.7 | 30.12 | 2.02 | 6.39 | 16.38 |
| FPGA06 | 91.4 | 77.9 | 1028 | 328.2 | 71.5 | 23.49 | 3.05 | 1092 | 357.3 | 90.0 | 43.06 | 2.09 | 9.52 | 27.45 |
| FPGA07 | 91.7 | 77.9 | 1073 | 329.5 | 72.6 | 23.32 | 3.11 | 1137 | 359.2 | 91.2 | 42.10 | 2.17 | 9.53 | 26.38 |
| FPGA08 | 88.9 | 79.1 | 1141 | 339.4 | 80.0 | 26.05 | 3.07 | 1204 | 367.5 | 97.8 | 44.66 | 2.19 | 10.51 | 30.11 |
| FPGA09 | 90.0 | 78.8 | 1419 | 429.3 | 98.2 | 31.26 | 3.14 | 1489 | 460.6 | 119.2 | 52.58 | 2.27 | 12.42 | 35.33 |
| FPGA10 | 94.4 | 76.0 | 1254 | 378.4 | 89.3 | 27.45 | 3.25 | 1324 | 411.9 | 110.3 | 49.26 | 2.24 | 11.22 | 32.05 |
| FPGA11 | 90.4 | 78.6 | 1364 | 403.7 | 93.3 | 29.96 | 3.12 | 1433 | 435.2 | 113.9 | 50.96 | 2.23 | 11.89 | 34.00 |
| FPGA12 | 92.6 | 77.2 | 1380 | 457.6 | 114.4 | 35.77 | 3.20 | 1456 | 491.6 | 135.4 | 58.37 | 2.32 | 13.92 | 41.44 |
| Geomean | 90.6 | 78.2 | 708 | 220.2 | 50.3 | 16.60 | **3.03** | 786 | 251.5 | 69.3 | 34.58 | **2.00** | 7.34 | 18.88 |

CPU+FPGA implementation has two parts - computing pin gradients using the FPGA kernel and computing sums of pin gradients on the CPU to get cell gradients. The anchor gradient computation is fused with the Nesterov step. Our CPU+FPGA implementation uses all 28 threads on the CPU. We get an average speedup of 3.03x using CPU 28 threads + FPGA vs CPU 28 threads for wirelength gradient computation. The speedup for the overall global placement flow is 2x. Note that in the CPU+FPGA implementations discussed so far, only one device was being used at a time (either the CPU or the FPGA). If we use both at the same time, for example, by calculating some pin gradients on the CPU and some on the FPGA, we can achieve even more speedup.

Figure 2.13 shows the average runtime breakdown for global placement. Numerical optimization comprises the dark blue, light blue and green parts. The dark and light blue parts together represent the time spent on computing wirelength gradient (48%). Adding the green part gives the time for numerical optimization (54.6%). "Others" includes time spent on updating anchor weights, legalizing RAMs, DSPs and IOs, initializing data structures, etc.

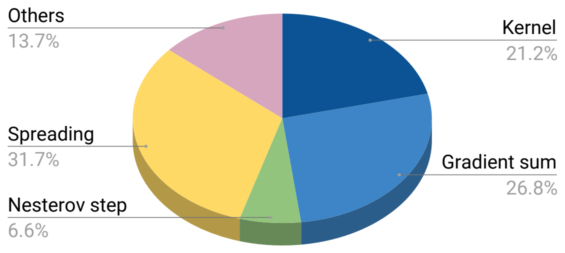


Figure 2.13: Runtime breakdown for FPGA Accelerated Global Placement

Table 2.3 also shows the time spent on the FPGA kernels. We used method 1 since it is faster in theory by a large margin. Both method 1 and method 2 access memory in a streaming fashion and are guaranteed to be stall free. Both of them read 64 bytes from memory and write 64 bytes to memory every clock cycle, which is the optimal arrangement for our Xeon+Arria10 system. With all these parameters same between the two methods, the kernel execution time is determined only by Fmax and the amount of data read from / written to memory. Table 2.3 shows the memory efficiency numbers for the two methods, which is defined as $\frac{useful\ data}{total\ data\ transferred}$. On an average, the kernel for method 2 would be $\frac{227\times0.906}{205\times0.782} = 1.28$x slower than the kernel for method 1.

Our CPU implementation is much faster than that in [11] for benchmarks of similar sizes. Note that [11] uses the LSE model, whose gradient is easier to compute than weighted average (Equations 2.5 vs 2.4). We have cross-checked the runtimes with [19], which uses the weighted average model. Following the runtime analysis in [19], one can deduce the runtime for one iteration of gradient computation for a benchmark with 1M cells on one thread to be $\sim 0.48$s. Our placer computes gradient 4800 times in total, so the time for each gradient computation can be obtained from Table 2.3 by dividing by this number. The single-threaded runtime claimed in [11] is 1.619s for benchmark mgc_superblue_11a with 925k cells and 935k nets. Our CPU+FPGA runtimes (using one device at a time) for gradient computation are comparable to the GPU runtimes in [11] for benchmarks of similar sizes. For example, mgc_superblue11_a with 925k cells and 935k nets in [11] takes 5.67ms while FPGA10 with 952k cells and 961k nets for us takes 5.7ms. Hence, even though [11] claims 93x speedup for gradient computation and we show 3.03x, the absolute runtime numbers are very close, that too with a more complicated gradient expression.

## 2.2 Spreading Cells in Global Placement using Linear Programming

Spreading or rough legalization is an important part of upper-and-lower-bound placement and greatly impacts the quality of the final solution. The most common method of spreading is bipartitioning, which has been successfully used in [20] and [21]. [20] produces results which are close in quality to [19] and [18], thus validating the effectiveness of the technique. Another method of spreading and/or legalization involves a flow formulation where cells 'flow' from overutilized to underutilized bins [17, 22–26]. The most commonly employed cost for such formulations is manhattan displacement [17,24–26]. [23] uses an approximation of total squared movement. Modeling squared movement accurately requires an edge from each overutilized bin to each underutilized bin (similar to bipartite matching), which significantly increases runtime if the number of bins is large. A method of clustering cells and solving a bipartite matching problem for assigning clusters to regions is used in [22] with the cost for moving a cluster being the change in HPWL(Half-Perimeter WireLength). This has the same scalability problem with the added runtime penalty of computing HPWL change for the assignment of each cluster to each region. Pruning possible cluster to region assignments might reduce the runtime, but limits the solution space exploration and may even fail to find a legal solution. Our work [1], on the other hand, uses a cost that is different from total or maximum displacement but can be computed as fast as these two. A key feature of our new cost is that it produces better placement shapes while maintaining relative order between cells, thereby producing better results.

In flow-based formulations, a key aspect is to realize the flow of cells across bins. Some variation of successive augmenting paths is used in [23], [24] and [25] to realize the flow. This involves moving cells from one bin to the next along a path, which requires the cells to be sorted with respect to distance from the destination bin. [24] maintains this order using a balanced binary tree, which has $\mathcal{O}(logN)$ insertion and deletion time, where $N$ is the number of cells. Although this may seem fast, it becomes slow when the number of bins is large. Our work, on the other hand, decouples the flow computation from the flow realization. We use a well-known and highly optimized solver for computing the flows. We propose a new flow realization algorithm to move cells according to the flows.

One important question that previous works overlook is what should be the best shape to spread the placement for minimizing wirelength. Changing the spreading function while keeping the same wirelength function can produce placements with different wirelengths. In this section, we address this question and also propose a linear-programming-based spreading algorithm with parameters to control the resultant placement shape while maintaining relative order among cells to a great extent. Our main contributions can be summarized as follows:

- We show that different cost functions in spreading produce different placement shapes. We analytically express the wirelengths of various shapes with some simplifying assumptions and empirically find a shape which is close to optimum.
- We propose a linear-programming-based spreading algorithm which can control the shape for spreading cells in overlap hotspots through some parameters.
- We propose a flow realization algorithm which can work on any generic flow that does not have cycles. Moreover, if the flow obeys certain conditions (as in our linear programming formulation), our algorithm preserves the relative order between cells to a great extent.

### 2.2.1 Placement Shapes

We develop an analytical expression for the total wirelength of a placement in terms of the function expressing its boundary or external shape. We make the following assumptions:

- The wires are routed horizontally or vertically or as a combination of the two
- We have a large number of cells and that the maximum allowed density is C cells per unit area of the chip

- Every cell has the same area and is connected to every other cell by a net and that all nets have exactly 2 pins

Finding the best placement shape even under these restrictive assumptions is a nontrivial problem. Intuitively, one would think that the placement should be in one connected shape and not broken into pieces since there would be many nets crossing the separation but we don't need that assumption for calculating wirelength.

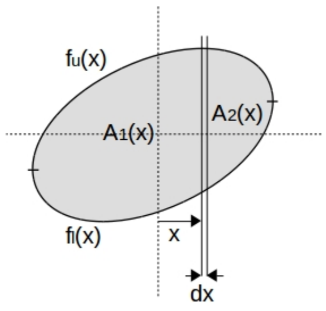


Figure 2.14: Wirelength calculation for an arbitrary shape

We can impose an x-y coordinate system on the placement shape and express the boundary of the shape as a collection of functions of x. For example, in Figure 2.14, the lower boundary is $f_l(x)$ and the upper boundary is $f_u(x)$. For non-convex shapes, we might need multiple functions as the intersection of the shape with a vertical line can be a collection of disjoint segments. Consider a narrow strip of width dx as shown in Figure 2.14. Let the area of the shape to the left of this strip be $A_1(x)$ and the area to the right be $A_2(x)$. The number of cells to the left and right are $CA_1(x)$ and $CA_2(x)$ respectively. The total number of nets crossing the strip in the x direction is $C^2A_1(x)A_2(x)$.

The contribution of all these nets to the x wirelength is $C^2 A_1(x) A_2(x) dx$. The contribution of cells within the strip are very small and that ratio approaches 0 in the limit $dx \to 0$. The total x-wirelength can be expressed as:

$$WL_x = \int C^2 A_1(x) A_2(x) dx$$

. A similar expression can be derived for the wirelength in the y-direction. The total wirelength can be expressed as the sum of x and y wirelengths.

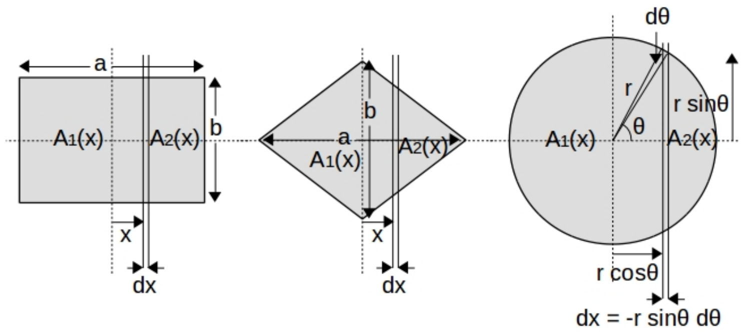


Figure 2.15: Wirelength calculation for rectangle, diamond and circle

We calculate wirelengths for some common placement shapes. We can choose appropriate units for distance such that $C = 1$. We start with a rectangle. We position the origin at the centre of the rectangle. $A_1(x) = b(\frac{a}{2} + x)$ and $A_2(x) = b(\frac{a}{2} - x)$. Figure 2.15 shows the dimensions. The x-wirelength can be obtained as follows:

$$WL_x = \int_{-\frac{a}{2}}^{+\frac{a}{2}} b\left(\frac{a}{2} + x\right) b\left(\frac{a}{2} - x\right) dx = \frac{a^3 b^2}{6}$$

Similarly, the wirelength in y direction is $\frac{a^2 b^3}{6}$. Hence, the total wirelength is $a^2 b^2 \frac{(a+b)}{6}$. If we minimize this function with respect to $a$ keeping the total area constant (thus $b$ becomes a dependent parameter), we get $a = b$, in other words, the shape is a square. The total wirelength then becomes $\frac{a^5}{3} = \frac{A^{2.5}}{3}$ where $A$ is the area of the square.

For a diamond, $A_2(x) =$ area of the triangle to the right of the dx strip $= \frac{1}{2} \times base \times height = \frac{1}{2} \times \frac{2b}{a}(\frac{a}{2} - x) \times (\frac{a}{2} - x) = \frac{b}{a}(\frac{a}{2} - x)^2$ for $x \geq 0$. $A_1(x) =$ area of the diamond$-A_2(x) = \frac{ab}{2} - \frac{b}{a}(\frac{a}{2} - x)^2$. The wirelength can be obtained as follows:

$$WL_x = 2\int_0^{\frac{a}{2}} \left(\frac{ab}{2} - \frac{b}{a}\left(\frac{a}{2} - x\right)^2\right)\frac{b}{a}\left(\frac{a}{2} - x\right)^2 dx = \frac{7a^3b^2}{240}$$

Similarly, the y-wirelength is $\frac{7a^2b^3}{240}$ and the total wirelength is $\frac{7a^2b^2(a+b)}{240}$. Minimizing this while keeping the area constant leads to $a = b$ and a total wirelength of $\frac{7a^5}{120} = \frac{7A^{2.5}}{15\sqrt{2}}$ where $A = \frac{ab}{2}$ is the area.

For a circle, we use polar coordinates. Setting $x = rcos\theta$ gives $dx = -rsin\theta d\theta$. The wirelength equation can be re-written as:

$$WL_x = -r\int_{\pi}^{0} A_1(\theta)A_2(\theta)sin\theta d\theta$$

$A_2(\theta) =$ area of a segment $=$ area of a sector $-$ area of an isosceles triangle $= \theta r^2 - \frac{1}{2} \times 2rsin\theta rcos\theta = r^2(\theta - \frac{sin2\theta}{2})$. $A_1(\theta) =$ area of the circle$-A_2(\theta) = r^2(\pi - (\theta - \frac{sin2\theta}{2}))$. The x-wirelength is:

$$r^5\int_0^{\pi} \left(\pi - \left(\theta - \frac{sin2\theta}{2}\right)\right)\left(\theta - \frac{sin2\theta}{2}\right) sin\theta d\theta = \frac{128}{45}r^5$$

The y-wirelength is same as this. The total wirelength evaluates to $\frac{256}{45\pi^{2.5}}A^{2.5} = 0.325201A^{2.5}$ where $A = \pi r^2$.

We see that for the same area, circle is better than diamond which is better than square.

**Lemma 1** *Wirelength is proportional to $A^{2.5}$ if every cell is connected to every other cell.*

**Proof:** Scaling by a factor $s$ in both x and y dimensions scales $A_1(x)$ and $A_2(x)$ by $s^2$ and the width of the strip (in Figure 2.14) scales by $s$ so the wirelength in the strip scales by $s^2 s^2 s = s^5$. Area scales as $s^2$, so wirelength scales as $A^{2.5}$. Q.E.D.

We define **normalized wirelength** $NWL$ as $\frac{WL_x + WL_y}{A^{2.5}}$.

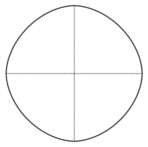


Figure 2.16: $|x|^{1.8} + |y|^{1.8} = 1$

The three shapes we considered occur in commonly used spreading functions. Minimizing manhattan displacement produces diamond shape, minimizing maximum x or y displacement produces a square and a bell-shaped potential function produces a circle.

Next, we consider the family of curves:

$$|x|^n + |y|^n = 1$$

where n is a positive real number. Note that diamond and circle are members of this family with n=1 and n=2 respectively. The shape approaches a square as n→ ∞. Closed form expressions for the area of this shape are not known, let alone the complicated wirelength function. So, we computed $A_1(x)$, $A_2(x)$ and $\int A_1(x)A_2(x)dx$ numerically. We swept n from 1 to 100. The results are shown in Table 2.4. Note that the values for circle, diamond and square(approximate, n=100) are very close to the ones obtained analytically. The best shape in this family appears to be somewhere between a circle and a diamond(Figure 2.16), with the value of n ∼1.84

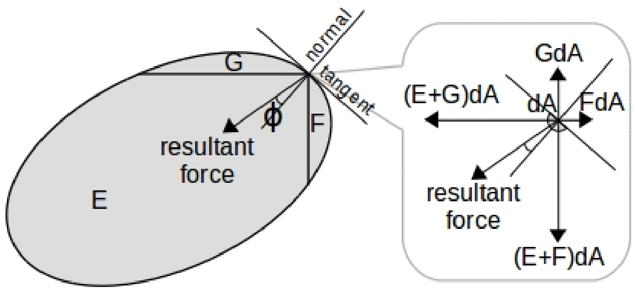


Figure 2.17: Force at the boundary and its angle with the normal

Although we are not able to find the best shape, we derive some conditions which that shape should satisfy. To do this, we assume that our placement shape is a mass of fluid in 2 dimensions. It will be in equilibrium only if forces balance out at each point on the mass. The total wirelength can be thought of as the energy of the configuration of this mass. Since work done (or energy change) is $\int force.dx$, we can differentiate the energy to obtain force on a small element dA (Figure 2.17). This evaluates to a constant amount of force per net. So, the total force in the x direction is $\#nets\_pulling\_right - \#nets\_pulling\_left$. The total force in the y direction is $\#nets\_pulling\_up - \#nets\_pulling\_down$. For the shape to be in equilibrium, the resultant force at any point on the boundary should be perpendicular to the tangent of the boundary at that point. Otherwise there is a component of force along the tangent which would make dA move in its direction. The component of force perpendicular to the tangent would be balanced by the pressure inside the mass of fluid. One can verify that circle, diamond or square (Figure 2.18) are not equilibrium shapes.

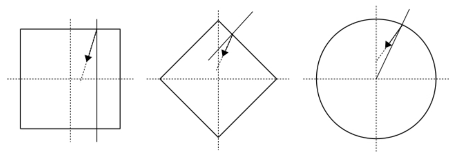


Figure 2.18: Deviation of resultant force from the normal

We numerically compute the maximum angle $\phi$ between the normal at the boundary and the resultant force direction. This is shown in Table 2.4. We see that n=1.84 gives a small deviation, hinting that this shape is "close enough" to optimal.

### 2.2.2 Min-Cost Flow based spreading

We present some min-cost flow based spreading techniques in this section. We formulate the min-cost flow problem as a linear program. Let's suppose there are N cells to be placed on a chip, which has been divided into $B$ bins. Different cells may have different areas. The maximum permissible total area of cells in bin $b_i$ is $S_i$. The initial locations of the cells are such that each bin $b_i$ has a resultant area demand $D_i$. Cells 'flow' from overutilized to

Table 2.4: Normalized wirelength and angle of resultant force

| n | $NWL$ | max $\|\phi\|$ | n | $NWL$ | max $\|\phi\|$ |
|---|---|---|---|---|---|
| 1.0 | 0.32998 | 45.00° | 2.1 | 0.32529 | 5.089° |
| 1.1 | 0.32833 | 32.11° | 2.2 | 0.32540 | 6.566° |
| 1.2 | 0.32718 | 24.08° | 2.3 | 0.32554 | 7.924° |
| 1.3 | 0.32639 | 18.08° | 2.4 | 0.32568 | 9.174° |
| 1.4 | 0.32585 | 13.35° | 2.5 | 0.32584 | 10.32° |
| 1.5 | 0.32549 | 9.500° | 3.0 | 0.32665 | 14.97° |
| 1.6 | 0.32528 | 6.302° | 4.0 | 0.32812 | 20.96° |
| 1.7 | 0.32516 | 3.640° | 5.0 | 0.32923 | 24.73° |
| 1.8 | 0.32512 | 1.523° | 10 | 0.33172 | 33.12° |
| **1.84** | **0.32512** | **0.875°** | 20 | 0.33281 | 38.14° |
| 1.9 | 0.32514 | 1.826° | 50 | 0.33323 | 41.75° |
| 2.0 | 0.32520 | 3.493° | 100 | 0.33330 | 43.18° |

underutilized bins through 'edges' among bins. More formally, the edge set $E$ is a collection of unordered pairs of the form $\{b_i, b_j\}$. This set $E$ will be different for the different formulations we present below:

#### 2.2.2.1 Formulation 1: Diagonal Flows

In this formulation, we only allow edges between adjacent and diagonally adjacent bins (Figure 2.19a). The cost of sending a unit of flow in a manhattan direction is 1 and that for a diagonal flow is $\alpha$. We impose certain restrictions on $\alpha$:

- $\alpha < 2$ otherwise there is no benefit of using a diagonal edge as it would be cheaper to send flow using two manhattan edges.
- $\alpha > 1$ as the diagonal bins are farther than the bin to the left or right

or up or down.

So, $1 < \alpha < 2$. Let $f_{ij}$ be the flow from $b_i$ to $b_j$. $f_{ji} = -f_{ij}$. Let $cost(b_i, b_j)$ be the cost of sending a unit of flow from $b_i$ to $b_j$, which is 1 if $b_i$ and $b_j$ are adjacent and $\alpha$ if they are diagonally adjacent. Let $E$ be the set of edges between adjacent and diagonally adjacent bins. The edges can be oriented arbitrarily. The formulation can be expressed as a linear program:

$$min \sum_{\{i,j\}|\{b_i,b_j\}\in E} cost(b_i, b_j)\hat{f}_{ij}$$

$$\hat{f}_{ij} \geq f_{ij} \; and \; \hat{f}_{ij} \geq -f_{ij} \; \forall\{i,j\}$$

$$\left(\sum_{j\,|\,\{b_j,b_i\}\in E} f_{ji}\right) + D_i \leq S_i \; \forall i$$

$\hat{f}_{ij} = |f_{ij}|$ is realized by the two inequalities $\hat{f}_{ij} \geq f_{ij}$ and $\hat{f}_{ij} \geq -f_{ij}$. Note that we do not need to constrain the total outflow of $b_i$ to be less than its current demand $D_i$ as the LP objective ensures that this will never happen. This formulation has $\mathcal{O}(B)$ variables.

**Lemma 2** *If all cells are initially placed in one bin, the placement after spreading will resemble an octagon (Figure 2.20a).*

**Proof:** We can analyze this by extension from the discrete to the continuous case. Suppose we have to send a unit of flow from the origin to a point $(x, y)$. The flow can be any combination of line segments, each parallel to one the lines $x = 0$, $y = 0$, $y = x$ and $y = -x$. The cost per unit of flow per unit distance(euclidean) is 1 for manhattan directions and $\frac{\alpha}{\sqrt{2}}$ for diagonal directions. Consider side 1 of the octagon in Figure 2.20a, in which $y > x$, $x \geq 0$ and $y \geq 0$. The minimum cost flow path from origin to $(x, y)$ can consist of some diagonal segments whose total length sums to $x\sqrt{2}$ and some vertical segments whose total length sums to $y - x$. Two such representative paths are shown in grey lines. The cost for sending a unit of flow through any of these paths is $\alpha x + (y - x) = y + (\alpha - 1)x$. Thus, we obtain a portion of a contour line: $y + (\alpha - 1)x = c$, where $c$ is a constant. For side 2, the minimum cost flow path consists of diagonal and horizontal segments. The equation is: $x + (\alpha - 1)y = c$. Similar analysis can be applied to the other six sides. Q.E.D.

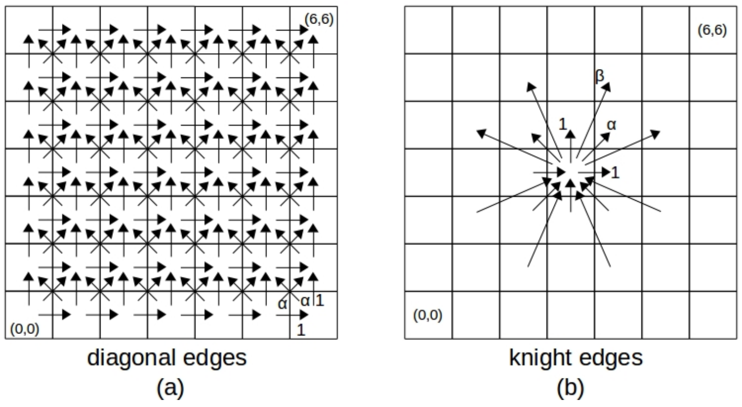


Figure 2.19: Diagonal and Knight edges. Knight edges shown for only one bin to reduce clutter.

We can change $\alpha$ to tune the angles of the octagon. When $\alpha \to 1$, this shape resembles a square. When $\alpha \to 2$, it resembles a diamond. Going back to our analysis of shapes, we set $c = 1$ and calculate the area: $A = \frac{4}{\alpha}$. The octagon intersects the line y=x at $(\frac{1}{\alpha}, \frac{1}{\alpha})$ in the first quadrant. $A_1(x)$ and

$A_2(x)$ can be defined piecewise from 0 to $\frac{1}{\alpha}$ and $\frac{1}{\alpha}$ to 1. The wirelength can be expressed by:

$$\begin{aligned}
WL = {} & 4\int_0^{\frac{1}{\alpha}} \left(\frac{2}{\alpha} + 2x - (\alpha-1)x^2\right)\left(\frac{2}{\alpha} - 2x + (\alpha-1)x^2\right) dx \\
& +4\int_{\frac{1}{\alpha}}^{1} \left(\frac{3\alpha+1}{\alpha^2} + \frac{1}{\alpha-1}\left(2x - x^2 + \frac{1-2\alpha}{\alpha^2}\right)\right) \\
& \left(\frac{\alpha-1}{\alpha^2} - \frac{1}{\alpha-1}\left(2x - x^2 + \frac{1-2\alpha}{\alpha^2}\right)\right) dx \\
= {} & \frac{68\alpha^2 + 84\alpha + 8}{15\alpha^4}
\end{aligned}$$

$NWL \left(= \frac{WL}{A^{2.5}}\right)$ for this shape is $\frac{17\sqrt{\alpha}}{120} + \frac{21}{120\sqrt{\alpha}} + \frac{1}{60\alpha\sqrt{\alpha}}$, whose minimum value is $\frac{\sqrt{5481+283\sqrt{849}}}{3} = 0.32545$ which occurs at $\alpha = \frac{21+\sqrt{849}}{34} = 1.474635$. This minimum value is less than the normalized wirelengths for square and diamond.

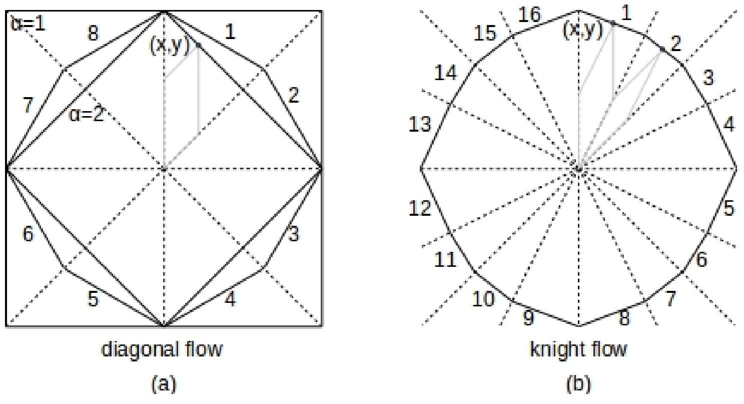


Figure 2.20: Contours for diagonal and knight flows

#### 2.2.2.2 Formulation 2: Knight's moves on a chessboard

In order to further improve our approximation of the shape, we introduce edges between bins that are situated a knight's move away from each other (Figure 2.19b). More formally, there is a 'knight' edge between bins located at $(x_1, y_1)$ and $(x_2, y_2)$ iff ($|x_1 - x_2| = 1$ and $|y_1 - y_2| = 2$) or ($|x_1 - x_2| = 2$ and $|y_1 - y_2| = 1$). We set the cost per unit of flow on these edges to $\beta$. Like $\alpha$, there should also be constraints on $\beta$ for the formulation to be reasonable (examples given w.r.t Figure 2.19b):

- $\beta > 2$ as a knight flow should be more expensive than going two bins up/down/left/right. ((3,3)→(4,5) vs (3,3)→(3,4)→(3,5))
- $\beta < 1 + \alpha$ as a knight flow should be cheaper than a diagonal + a horizontal/vertical flow. ((3,3)→(4,5) vs (3,3)→(4,4)→(4,5))
- $2\beta > 3\alpha$ as two knight flows should be more expensive than three diagonal flows. ((3,3)→(4,5)→(6,6) vs (3,3)→(4,4)→(5,5)→(6,6))
- $1 + \beta > 2\alpha$ as two diagonal flows should be cheaper than one knight + one horizontal/vertical flow. ((3,3)→(4,5)→(5,5) vs (3,3)→(4,4)→(5,5))

Note that $2\alpha - 1 < \frac{3\alpha}{2}$ for $\alpha < 2$, so the constraint $1 + \beta > 2\alpha$ is redundant. Hence, we end up with the following constraints on $\beta$: $max(2, \frac{3\alpha}{2}) < \beta < 1+\alpha$.

**Lemma 3** *If all cells are initially placed in one bin, this formulation will produce a hexadecagonal shape (Figure 2.20b).*

**Proof:** We can analyze the contours in the same manner as before. For side 1 in Figure 2.20b, $x > 0$, $y > 0$ and $y \geq 2x$. A representative least cost path from origin to $(x, y)$ consists of a knight segment and a vertical segment (shown in grey). The cost is $y + (\beta - 2)x$. For side 2, $x > 0$, $y > 0$ and $x \leq y < 2x$. A representative least cost path consists of a knight segment and a diagonal segment. The cost is $(\beta - \alpha)y + (2\alpha - \beta)x$. Hence, the contour has parts of the lines $y + (\beta - 2)x = c$ and $(\beta - \alpha)y + (2\alpha - \beta)x = c$ where $c$ is a constant. The equations for sides 3 and 4 are $(\beta - \alpha)x + (2\alpha - \beta)y = c$ and $x + (\beta - 2)y = c$ respectively. The other twelve sides can be analyzed in a similar manner. Q.E.D

The area of the shape is $\frac{4(1+\alpha)}{\alpha\beta}$. The normalized wirelength is: $\frac{WL}{A^{2.5}} = \frac{1}{(1+\alpha)^{2.5}}(\frac{\beta\sqrt{\beta}}{60\alpha\sqrt{\alpha}} + \frac{23\sqrt{\beta}}{120\sqrt{\alpha}} + \frac{41\sqrt{\alpha}}{120\sqrt{\beta}} + \frac{\sqrt{\alpha\beta}}{6} + \frac{\alpha\sqrt{\alpha}}{20\beta\sqrt{\beta}} + \frac{7\alpha\sqrt{\alpha}}{12\sqrt{\beta}} + \frac{\alpha\sqrt{\alpha\beta}}{6} + \frac{1}{20\beta\sqrt{\beta}} + \frac{7\alpha^2\sqrt{\alpha}}{24\sqrt{\beta}} + \frac{17\alpha^2\sqrt{\alpha\beta}}{120})$

whose minimum value is 0.325161 which occurs at $\alpha = 1.4933$, $\beta = 2.3092$. This is less than circle, diamond and square and is close to the best wirelength in Table 2.4.

This formulation also has $\mathcal{O}(B)$ variables.

**Definition:** A **flow path** of length n is a sequence of n edges $\{e_i\}$ from $b_{s_i}$ to $b_{t_i}$, each with some positive flow (the flow values need not be the same) such that $b_{t_i} = b_{s_{i+1}}$ except for $i = n$.

**Definition:** A flow path is said to be **monotonic** if the sequence of the coordinates $\{(x_i, y_i)\}$ of its constituent bins $b_0 \ldots b_i \ldots b_n$ is monotonic,

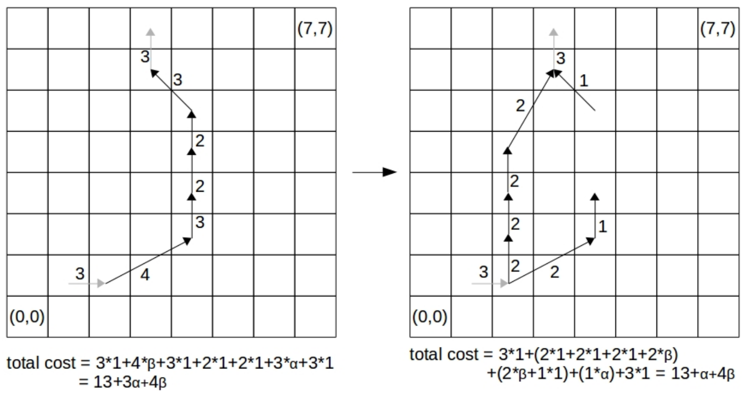

(7,7)
(0,0)
3
3
2
2
3
3
4
total cost = 3*1+4*β+3*1+2*1+2*1+3*α+3*1
= 13+3α+4β
(7,7)
(0,0)
3
1
2
2
2
1
3
2
2
total cost = 3*1+(2*1+2*1+2*1+2*β)
+(2*β+1*1)+(1*α)+3*1 = 13+α+4β

Figure 2.21: Replacing a flow path by a collection of cheaper flow paths

that is, $\{x_i\}$ is nonincreasing or nondecreasing and $\{y_i\}$ is also nonincreasing or nondecreasing.

**Lemma 4** *All flow paths in the solutions of formulations 1 and 2 are monotonic.*

**Proof:** Assume the contrary, that is, there exists a flow path which is non monotonic. Without loss of generality, we can assume that the sequence $\{x_i\}$ of x-coordinates is non monotonic (the y-coordinates can be monotonic or non-monotonic). There must exist a contiguous subsequence of $\{x_i\}$ of length $l$: $x_a, x_{a+1}, x_{a+2}, \ldots, x_{a+l-2}, x_{a+l-1}$ such that either $x_a < x_{a+1} = x_{a+2} = \cdots = x_{a+l-2} > x_{a+l-1}$ or $x_a > x_{a+1} = x_{a+2} = \cdots = x_{a+l-2} < x_{a+l-1}$ with $l \geq 3$.

Let the minimum flow among the edges in the sub-path $\{(x_a, y_a)$ to $(x_{a+l-1}, y_{a+l-1})\}$ be $f$. We can decompose the sub-path into one path $p$ with flow $f$ on its edges and another set of paths with flows $f_i - f$ where $f_i$ is the original flow on edge $e_i$ (Figure 2.21). We can replace $p$ with another path $p'$

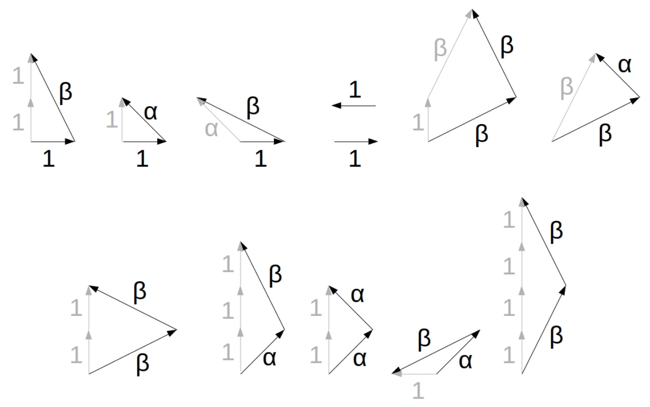


Figure 2.22: Possible non-monotonic combinations (black) and their replacement with monotonic ones (grey)

which has a strictly lower cost than $p$ without changing the resultant demand of any bin, which would contradict the optimality of the LP. We can first rearrange $p$ by moving the vertical segments in the middle to the end without changing its cost: $(x_a, y_a)$, $(x_{a+1}, y_{a+1})$, $(x_{a+l-1}, y_{a+1} + y_{a+l-1} - y_{a+l-2})$, $(x_{a+l-1}, y_{a+2} + y_{a+l-1} - y_{a+l-2})$, $\ldots$, $(x_{a+l-1}, y_{a+l-1})$. Next, we can replace the edges $\{(x_a, y_a), (x_{a+1}, y_{a+1})\}$ and $\{(x_{a+1}, y_{a+1}), (x_{a+l-1}, y_{a+1} + y_{a+l-1} - y_{a+l-2})\}$ with cheaper ones to get $p'$. Figure 2.22 shows the degenerate cases for possible combinations of the edges. All other combinations can be obtained by reflections of these edges about the lines $x = 0$, $y = 0$, $y = x$ and $y = -x$ and/or reversing the direction of flow. One can verify that the cost of the replacement edges is strictly lower than the original edges in each case due to the geometry of the edges and the constraints on $\alpha$ and $\beta$. Q.E.D.

**Corollary:** Flow paths do not form cycles.

### 2.2.3 Flow Realization

Although we add diagonal and knight edges, we do not move cells directly along those edges. Those are conceptual edges introduced for obtaining a good shape. The flow along each diagonal edge is decomposed into two manhattan paths, each with half the flow. Each knight edge can be decomposed into 7 manhattan edges. The amount of flow to send on each edge is calculated through analogy to a resistor network (Figure 2.23). We round the resulting flows as close as possible to a multiple of the most common cell area keeping the total flow same. Once we have replaced all flows by manhattan flows, we apply our flow realization algorithm described below to spread the cells.

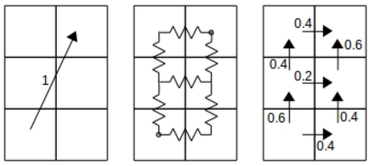


Figure 2.23: Converting non-manhattan flows into manhattan flows

We divide each bin into a grid of buckets (64x64=4096 in our case). We move buckets instead of cells. In other works, the cells in each bucket move together. We realize the flow in a manner similar to breadth first search. We can think of the bins and edges as a graph, with the bins being nodes and the edges being edges in the graph. Since there are no cycles in the graph , we can do a topological sort on the graph and assign levels to the nodes. We start from the nodes(bins) with the lowest level and push out the required number of buckets along the edges with nonzero flow. We then proceed to the next level. This process continues until all levels have been processed (Figure 2.24).

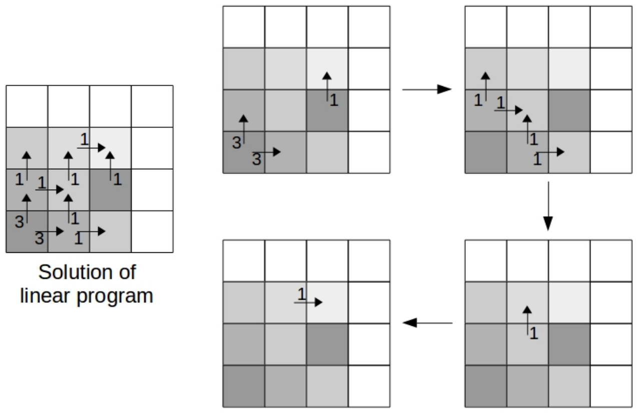


Figure 2.24: Flow realization: each shade of grey denotes a level obtained from topological sort.

Buckets with no cells are discarded. We maintain a pointer to the corresponding bin for each remaining bucket. A bucket can belong to exactly one bin at any point of time during the execution of our algorithm. Each bin maintains a vector of vectors of buckets indexed by x-coordinates of the buckets and a similar vector of vectors indexed by y-coordinates. Each bin also stores the maximum and minimum x and y coordinates for all the buckets assigned to it. For a bin, we calculate a score for pushing buckets out along each of its edges with nonzero flow. The score is defined as $\frac{distance\_to\_boundary}{flow\_on\_that\_edge}$. We then find the edge with the lowest score (Figure 2.25) and the corresponding vector of buckets which is closest to that boundary. We then push buckets out one by one from the vector to the adjacent bin. We stop if the total area of cells pushed out is about to exceed the flow on that edge. We update the score once we have finished processing a vector. We then select the vector corresponding to the lowest score and repeat the process until no more buckets can be pushed out (either all flows have been realized or pushing a bucket out from the current vector would cause excess flow).

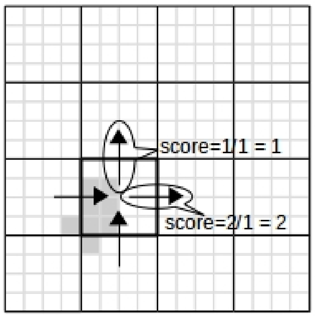


Figure 2.25: Moving buckets across bins

Note that if a bin has nonzero flow along some edge, then all the buckets assigned to that bin are on the bin's side of the corresponding bin boundary. For example, if there is a flow towards the left, then all the buckets assigned to that bin are to the right of the left edge of the bin. This is because all flow paths are monotonic. A bucket could not have come from a bin to the left of the boundary.

It may not be possible to realize the exact amount of flow in theory as the areas of buckets may not sum up to the exact value. Bipartitioning also suffers from the same drawback. However, in practice, we have seen that this algorithm is able to realize close to exact flows in most of the cases and exceeds bin capacity by 1 or 2 cells in some cases. These are handled by a chain move algorithm. Scaling the bin supply down by $\sim$1% can also mitigate this issue.

Each bin maintains a list of incoming buckets from adjacent bins. After we are done pushing flows of bins in a level, we update the following variables for the bins in the next level to which flows have been pushed out: min and max x and y coordinates of buckets, the vector of vectors for x and y directions. When we are done processing all the levels, we collect cells for each bin by iterating through the buckets assigned to them. The cells are then sorted by their original (solution of wirelength/timing/congestion optimizer) x and y coordinates and distributed evenly inside the bin along each coordinate. This procedure is same as the one used in [20] and [21].

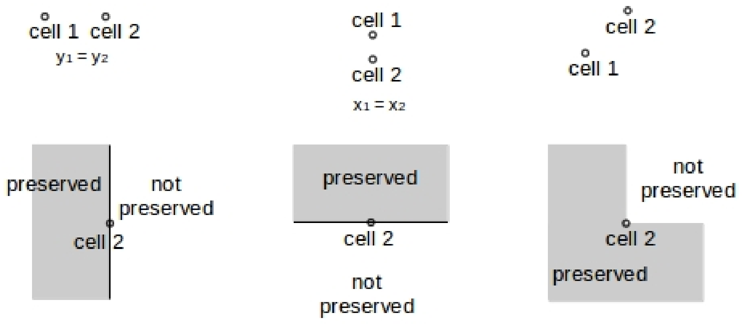


Figure 2.26: Possible locations of $\text{cell}_1$ w.r.t $\text{cell}_2$ for preserving relative order.

Let the initial coordinate of a cell $c$ before spreading be $(x_c, y_c)$ and that after spreading be $(x'_c, y'_c)$.

**Definition:** Relative order between two cells $c_1$ and $c_2$ is said to be preserved if one of the conditions hold:

i) $x_{c_1} = x_{c_2}$ and $(y'_{c_1} - y'_{c_2})(y_{c_1} - y_{c_2}) \geq 0$

ii) $y_{c_1} = y_{c_2}$ and $(x'_{c_1} - x'_{c_2})(x_{c_1} - x_{c_2}) \geq 0$

iii) $x_{c_1} \neq x_{c_2}$ and $y_{c_1} \neq y_{c_2}$ and at most one of $(y'_{c_1} - y'_{c_2})(y_{c_1} - y_{c_2})$ and

$(x'_{c_1} - x'_{c_2})(x_{c_1} - x_{c_2})$ is $\leq 0$. Figure 2.26 shows some examples.

**Lemma 5** *The following hold for two cells $c_1$ and $c_2$ if chain move is not applied:*

*i) If $c_1$ and $c_2$ are in the same bin after spreading, then relative order between $c_1$ and $c_2$ is preserved.*

*ii) If $c_1$ and $c_2$ are initially in the same bin before spreading and $x_{c_1} \neq x_{c_2}$ and $y_{c_1} \neq y_{c_2}$ and $c_1$ remains in that bin after spreading, then relative order between $c_1$ and $c_2$ is preserved.*

**Proof:** i) Easy to see since final location assignments within a bin are done by sorting with respect to initial x and y coordinates.

ii) If $x_{c_1} < x_{c_2}$ and $y_{c_1} < y_{c_2}$, $c_2$ can only move up or to the right. If it moves right, it cannot move left later as flow paths are monotonic. Similarly, if it moves up, it cannot move down later. So, $c_2$ cannot end up to the left and below $c_1$ at the same time. The other three cases with combinations of $x_{c_1} >$ or $< x_{c_2}$ and $y_{c_1} >$ or $< y_{c_2}$ can be handled in a similar fashion. Q.E.D

### 2.2.4 Results for the new Spreading Algorithm

We tested our algorithms on the ISPD 2016 FPGA placement contest [71] benchmarks. We integrated our spreading algorithm into [21]. The baseline bi-partitioning based spreading algorithm in [21] is similar to the one in [20]. Table 2.5 shows the wirelengths after LUT-and-FF-level **global**

Table 2.5: Wirelength after flat global placement in multiples of $10^3$

| Design | $\frac{\#cells}{10^3}$ | $\frac{\#nets}{10^3}$ | [21] | Diagonal ($\alpha$) $\alpha$=↓ | | | Knight ($\alpha, \beta$) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | $\alpha$=1.25; $\beta$=↓ | | | $\alpha$=1.35; $\beta$=↓ | | |
| | | | | 1.25 | 1.30 | 1.35 | 2.22 | 2.20 | 2.18 | 2.30 | 2.20 | 2.10 |
| FPGA01 | 105 | 105 | 304 | 257 | 257 | 257 | 258 | 256 | 257 | 256 | 256 | 258 |
| FPGA02 | 166 | 167 | 655 | 555 | 555 | 557 | 552 | 552 | 554 | 557 | 555 | 557 |
| FPGA03 | 421 | 428 | 2654 | 2677 | 2657 | 2671 | 2655 | 2665 | 2649 | 2666 | 2657 | 2661 |
| FPGA04 | 423 | 430 | 4903 | 4681 | 4669 | 4702 | 4659 | 4661 | 4671 | 4652 | 4633 | 4640 |
| FPGA05 | 425 | 433 | 8767 | 8850 | 8805 | 8849 | 8690 | 8702 | 8698 | 8674 | 8662 | 8701 |
| FPGA06 | 704 | 713 | 4351 | 3949 | 3933 | 3953 | 3951 | 3945 | 3971 | 3941 | 3961 | 3971 |
| FPGA07 | 707 | 716 | 8040 | 7511 | 7463 | 7477 | 7508 | 7537 | 7507 | 7515 | 7531 | 7517 |
| FPGA08 | 717 | 725 | 7608 | 7011 | 6999 | 7024 | 7011 | 7002 | 6987 | 6988 | 6984 | 6981 |
| FPGA09 | 867 | 876 | 9542 | 9318 | 9329 | 9292 | 9225 | 9231 | 9229 | 9214 | 9201 | 9221 |
| FPGA10 | 952 | 961 | 4294 | 4004 | 4020 | 3984 | 4065 | 4049 | 4066 | 4042 | 4037 | 4032 |
| FPGA11 | 845 | 851 | 9260 | 9198 | 9233 | 9265 | 9164 | 9129 | 9144 | 9123 | 9135 | 9092 |
| FPGA12 | 1103 | 1111 | 4998 | 4727 | 4734 | 4741 | 4814 | 4785 | 4785 | 4775 | 4780 | 4756 |
| Ratio | | | 1 | 0.9382 | **0.9367** | 0.9385 | 0.9370 | 0.9360 | 0.9367 | 0.9354 | **0.9349** | 0.9357 |

Table 2.6: Placement wirelength, congestion and runtime

| Design | [21] | | Ours | |
|---|---|---|---|---|
| | WL/$10^3$ | Time (s) | WL/$10^3$ | Time(s) |
| FPGA01 | 294 | 193 | 284 | 201 |
| FPGA02 | 569 | 333 | 549 | 350 |
| FPGA03 | 2700 | 944 | 2608 | 796 |
| FPGA04 | 4784 | 924 | 4716 | 894 |
| FPGA05 | 8761 | 1016 | 8633 | 1087 |
| FPGA06 | 4858 | 2115 | 4784 | 1713 |
| FPGA07 | 8227 | 2102 | 8232 | 1704 |
| FPGA08 | 7088 | 2049 | 7047 | 1727 |
| FPGA09 | 10074 | 2941 | 9811 | 2549 |
| FPGA10 | 6805 | 3889 | 6617 | 3038 |
| FPGA11 | 8688 | 2390 | 8696 | 2080 |
| FPGA12 | 5686 | 3165 | 5627 | 2834 |
| Ratio | | | **0.9821** | **0.8990** |
| Congestion | | | | |
| | Peak | #sites | Peak | #sites |
| FPGA05 | 0.8426 | 5730 | 0.8316 | 5288 |
| FPGA07 | 0.7539 | 5 | 0.6910 | 0 |
| FPGA11 | 0.7938 | 497 | 0.7773 | 221 |
| Ratio | 1 | | **0.96** | |

**placement** for some settings of $\alpha$ and $\beta$. All of these settings produce better wirelength than [21] on average. For the best setting in the table, we get **6.51**% less HPWL.

Table 2.6 shows wirelength and routing congestion (peak and number of congested sites obtained from smoothed results from a global router) after detailed placement for $\alpha$=1.35 and $\beta$=2.20. We see that some of the wire improvement after global placement carries through later stages (packing, post-packing global placement, legalization and detailed placement) to give a **final placement** wire improvement of **1.79**%. The peak routing resource utilization after detailed placement improves for all the three highly congested designs in the suite, with an average improvement of **4**%. As shown in Table 2.6, our new spreading algorithm improves the total placement runtime by 10.1% over [21].

## 2.3 FPGA Acceleration of Spreading

We proposed a spreading algorithm in the previous section which performs better than other spreading algorithms. This algorithm uses linear programming to compute flows of cells across bins. The flows are then realized by traversing a graph in a topologically sorted order. This flow realization method can yield large speedups through parallelization as all the nodes on the same topological level can be processed in parallel. The linear programming solver consumes the bulk of the runtime but is hard to parallelize. To solve this problem, we propose a new algorithm to compute flows that is highly parallelizable, coupled with a parallelized version of the flow realization algorithm described in the previous section.

As speedups obtained through multi-threaded implementations have achieved maturity, researchers have started investigating hardware acceleration of placement using GPUs [11–15] and FPGAs [7]. Acceleration of simulated-annealing-based placement is proposed in [13], [14] and [15]. A GPU-accelerated upper-and-lower-bound placer is described in [12]. This work implements the recursive partitioning and cell assignment steps for a single window on GPU. Our flow-based spreading algorithm produces better wirelength than bipartitioning. In a placement system where the numerical solver is accelerated on FPGA, the spreading part takes a significant portion of the runtime. To eliminate this runtime bottleneck, we need to accelerate the spreading algorithm. We accelerate the flow computation stage of our spreading algorithm on FPGA. The main contributions of our flow based spreading formulation [9] are:

- We propose a massively parallel algorithm for flow computation which can be used in a flow-based spreading algorithm. Our algorithm mimics the flow of a fluid across reservoirs due to pressure differences.
- We mathematically prove that the flows obtained from the solution of differential equations describing the above system do not form cycles.
- We propose a flow correction algorithm which can modify flows generated by any flow computation algorithm to produce flows that are monotonic (definition and significance discussed later). This algorithm reduces the

total cell displacement but does not change the resultant density of any bin.

- Although our continuous-time formulation guarantees acyclic flows, our discretization process may introduce some cycles. We analyze the complexity of our flow correction algorithm and show that we can use it to remove such cycles, mush faster than other cycle finding algorithms for a general graph [10].
- We accelerate our flow computation algorithm on FPGA.

### 2.3.1 Flow-based Spreading

We formulate our spreading problem in a fashion imitating the flow of a fluid across different reservoirs driven by pressure difference. The placement grid is divided into a grid of $R \times C$ bins, as shown in Figure 2.27. The bins are numbered in a row-major fashion in the 2D grid. Each bin $B_i$ has a supply value $S_i$ (maximum number of cells it can hold) and an initial demand value $D_i^0$ (the number of cells placed in that bin after an iteration of global placement). We change the supply values of all bins to the max supply value and adjust the initial bin demands by adding dummy cells. More formally, it can be described by the following two equations:

$$S' = \underset{i}{max}\ S_i \tag{2.9a}$$

$$S'_i = S'\ \forall i \tag{2.9b}$$

$$D_i'^0 = D_i^0 + S' - S_i \tag{2.9c}$$

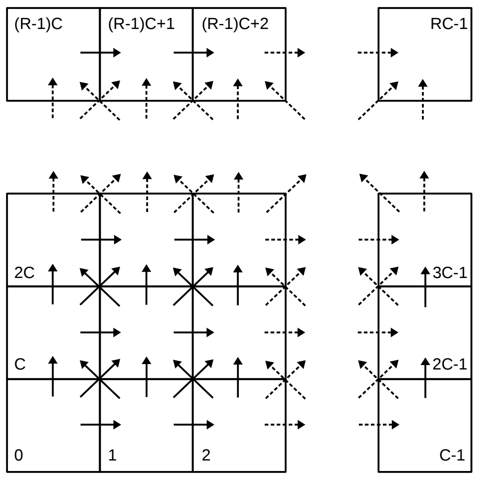


Figure 2.27: A grid containing RxC bins. Bins are numbered in a row major fashion. Arrows indicate channels between bins, which are bidirectional. The direction of an arrow denotes the direction in which flow is considered positive.

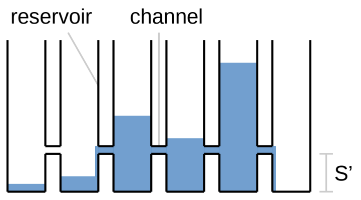


Figure 2.28: Flow of fluid across reservoirs connected by channels. The channel diameter is very small compared to the reservoir dimensions. Fluid does not flow through a channel if the fluid levels on both sides are below the level of the channel.

Each bin can be thought of as a reservoir and the demand $(D_i')$ of cells in each bin corresponds to the fluid level in that bin. There are channels between adjacent and diagonally adjacent bins which allow fluid to flow through them. More formally, the rate of flow of fluid through a channel is directly proportional to the pressure difference across it:

$$\frac{df_{i\_j}}{dt} = \sigma.(max(D'_i, S') - max(D'_j, S')) \tag{2.10}$$

Where $f_{i\_j}$ is the cumulative fluid flow through the channel between bins $B_i$ and $B_j$. $f_{i\_j} = -f_{j\_i}$. $f_{i\_j}$, $D'_i$ and $D'_j$ change over time. $\sigma$ is a constant that controls the rate of flow and we refer to it as 'conductance' of the channel. We set $\sigma < 0.125$. Figure 2.28 shows a 1-dimensional case of this model. In a realistic scenario, the fluid flow through a narrow cylindrical tube is given by:

$$\frac{df}{dt} = \frac{\pi r^4 P}{8\eta l} \tag{2.11}$$

Where $P$ is the pressure difference across the tube, $r$ is the radius of the tube, $\eta$ is the viscosity of the fluid and $l$ is the length of the tube. In our case, $\sigma$ serves the same purpose as the term $\frac{\pi r^4}{8\eta l}$. The channels are located at a height corresponding to the supply value $S'$ (which is same for all bins). There is no flow in a channel if the fluid levels on both sides are less than the supply value (Figure 2.28). The process of fluid flow through all the channels can be described by a system of first-order differential equations:

$$\frac{df_{0\_1}}{dt} = \sigma.(max(D'_0, S') - max(D'_1, S'))$$

$$\frac{df_{0\_C}}{dt} = \sigma.(max(D'_0, S') - max(D'_C, S'))$$

$$\frac{df_{0\_(C+1)}}{dt} = \sigma.(max(D'_0, S') - max(D'_{C+1}, S'))$$

$$\cdots$$

$$\frac{dD'_0}{dt} = -\frac{df_{0\_C}}{dt} - \frac{df_{0\_(C+1)}}{dt} - \frac{df_{0\_1}}{dt}$$
$$\frac{dD'_1}{dt} = \frac{df_{0\_1}}{dt} - \frac{df_{1\_C}}{dt} - \frac{df_{1\_(C+1)}}{dt} - \frac{df_{1\_(C+2)}}{dt} - \frac{df_{1\_2}}{dt}$$
$$\ldots \tag{2.12}$$

Note that a bin may have 3, 5 or 8 neighbors. For example, $B_0$ has 3 neighbors: $B_C$, $B_{C+1}$ and $B_1$. All $f_{i_j}$s are initialized to 0 and $D'_i$s are initialized to $D'^0_i$s. Eventually, for each bin $B_i$, $D'_i$ will either be $\leq S'$ or approach $S'$ in the limit $t \to \infty$ as total demand < total supply. The total flow across each channel can be obtained by simulating the above system for a sufficiently long time $t$ so that the error is low. The demand values follow an exponentially decaying function and the system convergence fast in practice. We now prove an interesting property of the flows generated by the above system of equations:

**Lemma 6** *The total flows in the limit* $t \to \infty$ *do not form cycles.*

**Proof**: Observe that once the demand $D_i$ of a bin $B_i$ reaches $S'$, it is always $\geq S'$ afterwards. Also note that if there is net a positive outflow through a channel $i\_j$ from bin $B_i$, then $D^{\infty}_i = S'$. Assume there is a cycle of length $l$: $B_{i_0} \to B_{i_1} \to \cdots \to B_{i_{l-1}} \to B_{i_0}$ in the flows. At any instant of time, if there are one or more bins with $D' < S'$, the cycle can be decomposed into a collection of paths where the beginning and ending bins of each path have $D' < S'$ and all other bins in the path have $D' \geq S'$. Consider one such path $B_{i_m} \to B_{i_{m+1}} \to \cdots \to B_{i_n}$ with $D'_m < S'$, $D'_n < S'$ and $D'_{i_k} \geq S'$ for $k$=$m+1$

to $n-1$. The flows along this path can be described by:

$$\frac{df_{i_m\text{-}i_{m+1}}}{dt} = \sigma.(S' - D'_{m+1})$$

$$\frac{df_{i_{m+1}\text{-}i_{m+2}}}{dt} = \sigma.(D'_{m+1} - D'_{m+2})$$

$$\dots$$

$$\frac{df_{i_{n-1}\text{-}i_n}}{dt} = \sigma.(D'_{n-1} - S')$$

We see that the net change in flow through this path is 0:

$$\frac{df_{i_m\text{-}i_{m+1}}}{dt} + \frac{df_{i_{m+1}\text{-}i_{m+2}}}{dt} + \cdots + \frac{df_{i_{n-1}\text{-}i_n}}{dt} = 0$$

Arguing similarly for all other paths in the collection, we conclude that the net change in flow through the cycle is 0. Since the initial flows at time $t = 0$ are 0, the sum of flows for all paths in the cycle is 0, which means that not all of those flows can be positive. This also holds if all $D'$s in the cycle are $\geq S'$. Hence, such a cycle does not exist. Q.E.D.

#### 2.3.1.1 Discrete Flow Computation

The $max(.)$ functions in the system of equations (2.12) make it hard to solve the system analytically. Hence, we simulate the system with discrete time steps. The amount of incremental flow through a channel at each time step is given by:

$$\Delta f^t_{i\text{-}j} = f^{t+1}_{i\text{-}j} - f^t_{i\text{-}j} = \sigma.(max(D'^t_i, S') - max(D'^t_j, S')) \qquad (2.14)$$

We now have a system of difference equations:

$$
\begin{gathered}
\Delta f^t_{0\_1} = \sigma.(max(D'_0, S') - max(D'_1, S')) \\
\Delta f^t_{0\_C} = \sigma.(max(D'_0, S') - max(D'_C, S')) \\
\Delta f^t_{0\_(C+1)} = \sigma.(max(D'_0, S') - max(D'_{C+1}, S')) \\
\dots \\
D'^{t+1}_0 - D'^t_0 = -\Delta f^t_{0\_C} - \Delta f^t_{0\_(C+1)} - \Delta f^t_{0\_1} \\
D'^{t+1}_1 - D'^t_1 = \Delta f^t_{0\_1} - \Delta f^t_{1\_C} - \Delta f^t_{1\_(C+1)} - \Delta f^t_{1\_(C+2)} - \Delta f^t_{1\_2} \\
\dots
\end{gathered} \tag{2.15}
$$

We start with $f^0_{i\_j} = 0$ and compute the above values for sufficient number of steps $t$ that gives a low error. We can control the rate of convergence by changing the value of $\sigma$. At each time step, we first compute the incremental flows in parallel. We then update the demands in parallel using the incremental flows.

#### 2.3.1.2 Flow Realization

We follow the flow realization procedure in [1]. We only move cells up, down left or right from a bin. We do not move cells diagonally. The flow along each diagonal channel is decomposed into two manhattan paths, each with half the flow. We round the resulting flows as close as possible to a multiple of the most common cell area keeping the total flow same. Each bin is divided into a grid of buckets and cells in each bucket move together. We construct a directed graph with bins as nodes and channels as edges. The direction of an edge corresponds to the direction in which total flow in the channel is positive. Since there are no cycles in the graph , we can do a topological sort on the graph and assign levels to the nodes. We start from the nodes(bins) with the lowest level and push out the required number of buckets along the edges with nonzero flow. We then proceed to the next level. Flows for bins at the same topological level are processed in parallel.

#### 2.3.1.3 Flow Correction Algorithm

The discrete-time simulation used to solve for the demand and flow values in every time step as modeled in equation 2.15 can lead to cycles in the flow. However, the magnitude of flow on those cycles would depend on $\sigma$ and would be small. The flows can also be made monotonic (defined below), which helps in preserving relative order between cells during flow realization. Preserving relative order leads to better wirelength.

**Definition:** A **flow path** of length n is a sequence of n+1 bins $B_{i_0}$ to $B_{i_n}$, such that there is a channel between bins $B_j$ and $B_{j+1}$ with positive flow $\forall\ 0 \leq j < n$.

**Definition:** A flow path is said to be **monotonic** if the sequence of the coordinates $\{(x_i, y_i)\}$ of its constituent bins $B_0 \ldots B_i \ldots B_n$ is monotonic, that is, $\{x_i\}$ is nonincreasing or nondecreasing and $\{y_i\}$ is also nonincreasing or nondecreasing.

Monotonicity implies absence of cycles.

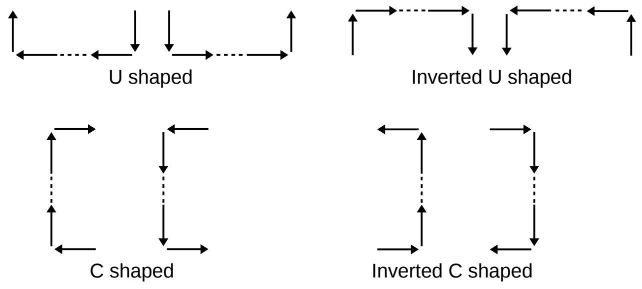


Figure 2.29: Examples of non-monotonic patterns

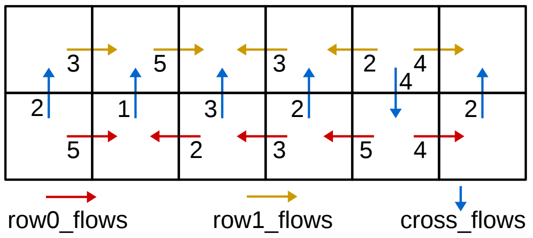


Figure 2.30: Terminology of flows used in Algorithm 1

We now present an algorithm that can take a feasible flow solution as input and produce another feasible flow solution without changing $D'$ for any bin. This algorithm also guarantees that the sum of magnitudes of all flows (after all diagonal flows have been converted to manhattan flows) would not increase.

$$cost = \sum_{i,j} |f_{i\_j}| \tag{2.16}$$

This cost serves as a proxy for total cell displacement.

Our flow correction algorithm runs in multiple iterations, each consisting four passes. Part of the pseudocode for one such pass is shown in Algorithm 1. Consider two adjacent rows of bins as shown in Figure 2.30. We call these rows $row0$ and $row1$ for simplicity. They may correspond to any two adjacent

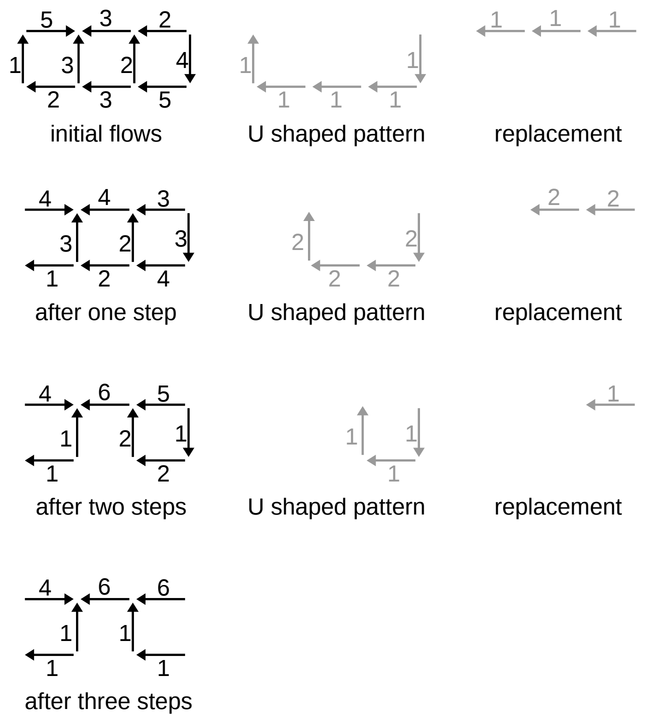


Figure 2.31: Replacing U shaped patterns for a section of the rows in Figure 2.30

rows in the bin grid. We look for U shaped patterns as shown in Figure 2.31 and replace them by the corresponding pattern, which results in a lower cost according to equation 2.16. Note that this process of replacing patterns does not change $D'$ of any bin.

We apply one pass of this algorithm starting from the bottom row and ending at the top row proceeding one row at a time to remove all U shaped flow patterns(Figure 2.29). We apply another pass from the top row to the bottom row to remove all inverted U patterns. We apply one more pass from the leftmost column to the rightmost column, replacing C shaped patterns and one more pass from the rightmost to the leftmost column replacing inverted C shaped patterns.

**Lemma 7** *A top-to-bottom pass will not create U shaped patterns if applied right after the bottom-to-top pass.*

**Proof:** Assume this is not true. Then, a new section of horizontal flow must have been created. Figure 2.32 shows the three possible cases. In case (i) we see an inverted U shaped pattern that should have been removed by Algorithm 1 first, which is a contradiction. In case (ii), we see that there is a preexisting U shaped pattern which should have been removed by the bottom-to-top pass. In case (iii) we see both types of contradictions from cases (i) and (ii). Q.E.D.

This lemma can be extended to the right-to-left pass also. Also, similar lemmas can be obtained by changing the order of the passes.

**Algorithm 1** Flow Correction for two Consecutive Rows

```
for i = 0 to C-2 do
    if row0_flows[i]*cross_flows[i]≥0 then
        continue;
    end
    continuous=true;
    continuous_min_abs=abs(row0_flows[i]);
    if row0_flows[i]==0 then
        continuous=false;
    end
    j=i+1;
    while continuous do
        if row0_flows[j-1]*cross_flows[j]>0 then
            //non-monotone path found
            min_abs=min(abs(cross_flows[i]),abs(cross_flows[j]),
            continuous_min_abs);
            decrease flows along path by min_abs;
            increase flows along substitute path by min_abs;
        end
        if cross_flows[i]==0 then
            break;
        end
        continuous_min_abs-=min_abs;
        if continuous_min_abs==0 then
            continuous=false;
        end
        j++;
        if j==C then
            break;
        end
        if row0_flows[j-2]*row0_flows[j-1]≤0 then
            continuous=false;
        end
        continuous_min_abs=min(abs(row0_flows[j-1]),continuous_min_abs);
    end
end
```

Note that a horizontal (left-to-right or right-to-left) pass can create new U or inverted U shaped patterns and a vertical (bottom-to-top or top-to-bottom) pass can create new C or inverted C shaped patterns. Even though this is the case, we can devise a scheme to apply these passes for a finite number of iterations and guarantee that the resultant flows would be monotonic. First, we can round all flow values to the nearest multiple of 0.5. Cells usually take certain discrete area values, so this is not a problem. The rounding error would be $\leq$0.25, so each bin can have an excess area of at most 1 (4 sides * 0.25) which can be handled by setting a lower supply value in the first place. Every time a pattern is replaced, it reduces the cost in equation 2.16 by 2*0.5=1. Since this cost is finite (depends on the number of cells), this process will run for a finite number of iterations. In practice, we have observed that one iteration gives sufficiently good wirelength and it is unnecessary to apply more iterations.

**Lemma 8** *Applying any one of the four passes is sufficient to remove all cycles.*

**Proof:** Consider the sequence of coordinates $(x_0, y_0)$, $(x_1, y_1)$, $\ldots$, $(x_{n-1}, y_{n-1})$, $(x_0, y_0)$ of a cycle of length n. Let $y_{min} = \min_i y_i$. There should be at least one range of indices $[p, q]$ of maximal length with $p < q$ s.t. $y_i = y_{min} \forall p \leq i \leq q$. This is the lowest part of the cycle. The sequence $(x_{p-1}, y_{q-1})$, $(x_p, y_p)$, $\ldots$, $(x_q, y_q)$, $(x_{q+1}, y_{q+1})$ is a U shaped pattern. Similarly, the rightmost part is an inverted C shaped pattern, the highest part is an inverted U shaped pattern and the leftmost part is a C shaped pattern. A cycle cannot exist if any one of these patterns does not exist. Q.E.D.

**Complexity analysis:** The bottom-to-top and the top-to-bottom passes (Algorithm 1) take $\mathcal{O}(RC^3)$ time. Similarly, the left-to-right and the right-to-left passes take $\mathcal{O}(R^3C)$ time. Hence, the total time for removing all cycles is $\mathcal{O}(min(RC^3, R^3C))$ which is $\mathcal{O}(N^2)$ where $N = RC$ is the number of bins.

Cycles in a general graph can be removed using [5] or [6]. [6] has a complexity of $\mathcal{O}((N + E)(C + 1))$ where $E$ is the number of edges and $C$ is the number of cycles. For a rectangular grid, $E$ is $\mathcal{O}(N)$ and $C$ is $\mathcal{O}(N^2)$. Our algorithm is specialized for a rectangular grid graph and is faster than [5] and [6].

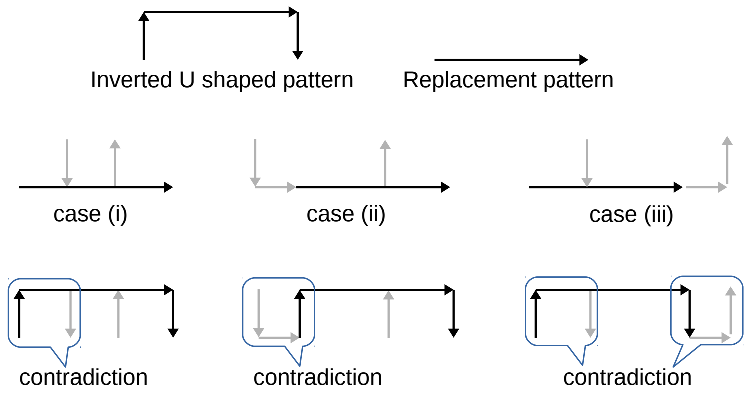


Figure 2.32: A top-to-bottom pass cannot create new U shaped patterns. Hypothetical U shaped patterns are shown in grey. The bottom row shows the flows before the inverted U shaped pattern was replaced.

### 2.3.2 FPGA Acceleration

We accelerate the flow computation part (simulation of fluid flow) on FPGA. A schematic of the hardware is shown in Figure 2.33. Our bin grid has 56x160=8960 bins, each having 3x3 sites. We first copy the bin demand values from the CPU to the FPGA and store them on the block RAMs. We have five 2-dimensional arrays implemented using the FPGA block RAMs - demand, xflows, yflows, d45flows and d135flows. xflows and yflows are flows along the x and y directions respectively. d45flows and d135 flows are flows along the $45^o$ and $135^o$ diagonal directions respectively. These arrays are stored in row major format. Each of them is padded such that each row in the array has 64 entries instead of 56.

We compute the flows using integer variables instead of floating point. This has two benefits: i) It saves resources as integer computations require less resources than floating point. ii) It reduces latency as integer computations take less number of cycles than floating point computations. This leads to less replication of block RAMs and reduces block RAM usage. Block RAM replication is necessary to ensure that the arrays can be accessed in a stall-free fashion. The actual flow values are scaled by 256 so that small flow values can be computed with negligible loss in accuracy. We have empirically determined that this produces insignificant difference in result from the original version using floating point. The memory system for each array is constrained to have one bank of width 128 bytes.

At each clock cycle, the hardware reads 2 consecutive rows from the

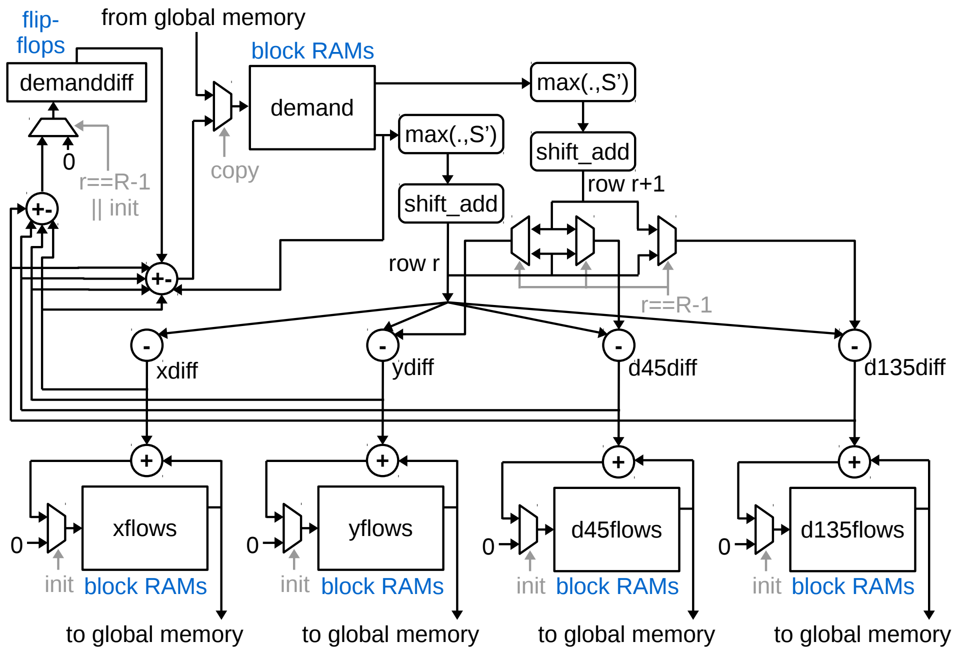


Figure 2.33: Part of the hardware for computing flows. There are 5 memory systems implemented using on-chip RAMs: demand, xflows, yflows, d45flows and d135flows. demanddiff is implemented with registers. All black connectors denote buses. Control signals for the multiplexers are shown in grey. For each time step, r goes from 0 to 159.

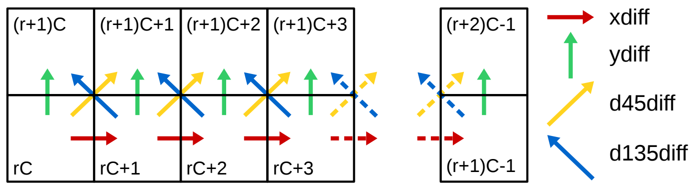


Figure 2.34: Incremental flows between rows r and r+1 calculated by the FPGA in one clock cycle.

demand array. It then computes the flow for one time step along 55 horizontal channels (xdiff), 56 vertical channels (ydiff), and 110 diagonal channels (d45diff and d135diff) (Figure 2.34). The xflows, yflows, d45flows and d135flows for the corresponding row are updated. The demand values of the current row are updated by appropriate addition/subtraction of xdiff, ydiff, d45diff and d135diff computed during the current clock cycle and adding the demand differences (demanddiff), which are flow values along the vertical and diagonal channels coming from the row below. These demand differences were calculated during the previous clock cycle and were stored in flip-flops. The demand differences for the next row are calculated by adding the corresponding vertical (ydiff) and diagonal flows (d45diff and d135diff) from the current row. On reaching the top row, the demand difference values are reset to 0. This is because flow computation for the next time step would start again from row 0 and row 0 does not have any flow coming from below. To ensure that the ydiff, d45diff and d135diff values computed for the top row (159) are 0, we have a multiplexer stage which substitutes a version of the current row for the next row. Row 160 (does not actually exist) would be the same as row 159 for ydiff. Row 160 would be row 159 offset by one place to the right for d45diff and one place to the left for d135diff.

Computing the flow for one time step along a channel requires multiplication by $\sigma$, which is $< 0.125$. To save hardware resources, we represent $\sigma$ by its 4 most significant bits and perform the multiplication using bitshifts and additions. Instead of performing this shift-add operation for every channel, we perform this when we read the two rows from the demand array. Thus, we only have to perform subtractions for computing xdiff, ydiff d45diff and d135diff. This reduces the number of shift-add operations.

### 2.3.3 Results

#### 2.3.3.1 Quality of Results

We tested our algorithm on the ISPD 2016 FPGA Placement Contest [71] benchmarks. Using the same numerical solver for global placement, we compared our fluid-flow-based spreading algorithm with the bipartitioning based spreading algorithm in [21] and the linear-programming-based spreading algorithm in [1]. The final placement wirelengths using each of these spreading techniques are shown in Table 2.7. Our algorithm produces 1% worse wirelength than the linear-programming-based algorithm in [1] but is still 0.8% better than [21]. The 1% loss is a good tradeoff for the significant gains in runtime that we get using our new spreading algorithm. Our fluid-flow based spreading algorithm produces a shape (Figure 2.35) which is slightly different from the one derived in the previous section (Figure 2.16).

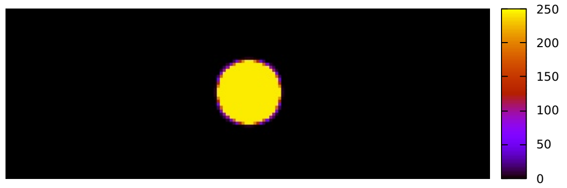


Figure 2.35: The shape of placement produced by our spreading algorithm can be inferred from the above heat map of #cells in each bin. This example has 85400 cells and no net. Maximum bin utilization is set to 84%

Table 2.7: Benchmark Statistics and Wirelength

| Design | $\frac{\#cells}{10^3}$ | $\frac{\#nets}{10^3}$ | Wirelength/$10^3$ | | |
|---|---|---|---|---|---|
| | | | [21] | [1] | Fluid |
| FPGA01 | 105 | 105 | 294 | 284 | 291 |
| FPGA02 | 166 | 167 | 569 | 549 | 555 |
| FPGA03 | 421 | 428 | 2700 | 2608 | 2615 |
| FPGA04 | 423 | 430 | 4784 | 4716 | 4711 |
| FPGA05 | 425 | 433 | 8761 | 8633 | 8635 |
| FPGA06 | 704 | 713 | 4858 | 4784 | 4840 |
| FPGA07 | 707 | 716 | 8227 | 8232 | 8421 |
| FPGA08 | 717 | 725 | 7088 | 7047 | 7059 |
| FPGA09 | 867 | 876 | 10074 | 9811 | 9994 |
| FPGA10 | 952 | 961 | 6805 | 6617 | 6616 |
| FPGA11 | 845 | 851 | 8688 | 8696 | 8887 |
| FPGA12 | 1103 | 1111 | 5686 | 5627 | 5694 |
| Ratio | | | 1 | 0.982 | 0.992 |

#### 2.3.3.2 Runtime

The runtime of most global placers is dominated by the numerical solver. So, to demonstrate the true potential of our new fluid-flow-based spreading algorithm, we need a fast numerical solver for global placement. The placer in [21] is single-threaded and uses a quadratic wirelength model. The algorithm for optimizing this model is difficult to accelerate on an FPGA. To overcome this difficulty, we use the global placer [8] described at the beginning of this chapter which has a weighted-average wirelength model and is highly parallelized and executes parts of the gradient computation step required for numerical optimization on an FPGA.

The second placer runs for 19 iterations and takes a total of 4800 gradi-

Table 2.8: Runtimes in seconds and Speedup for ISPD 2016 contest benchmarks

| | Overall Global Placement | | | | | | | Spreading Algorithm | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Gradient → | CPU | FPGA | | FPGA | | FPGA | | FPGA | FPGA | | FPGA | |
| Spreading → | LP CPU | LP CPU | | Fluid CPU | | Fluid FPGA | | LP CPU | Fluid CPU | | Fluid FPGA | |
| Design ↓ | time | time | speedup | time | speedup | time | speedup | time | time | speedup | time | speedup |
| FPGA01 | 31.03 | 27.76 | 1.12 | 12.54 | 2.47 | 9.34 | 3.32 | 22.88 | 7.76 | 2.95 | 4.36 | 5.25 |
| FPGA02 | 40.82 | 31.14 | 1.31 | 14.91 | 2.74 | 11.81 | 3.46 | 24.44 | 8.19 | 2.98 | 4.74 | 5.16 |
| FPGA03 | 77.35 | 48.36 | 1.60 | 28.24 | 2.74 | 25.71 | 3.01 | 29.66 | 9.41 | 3.15 | 6.07 | 4.89 |
| FPGA04 | 82.81 | 54.20 | 1.53 | 29.78 | 2.78 | 27.26 | 3.04 | 34.14 | 9.78 | 3.49 | 6.37 | 5.36 |
| FPGA05 | 84.63 | 53.96 | 1.57 | 30.44 | 2.78 | 27.80 | 3.04 | 32.98 | 9.69 | 3.40 | 6.33 | 5.21 |
| FPGA06 | 117.44 | 70.60 | 1.66 | 42.98 | 2.73 | 41.73 | 2.81 | 38.64 | 11.14 | 3.47 | 7.62 | 5.07 |
| FPGA07 | 122.00 | 71.41 | 1.71 | 42.38 | 2.88 | 41.03 | 2.97 | 40.52 | 11.24 | 3.61 | 8.32 | 4.87 |
| FPGA08 | 122.89 | 70.73 | 1.74 | 44.81 | 2.74 | 42.51 | 2.89 | 37.06 | 11.54 | 3.21 | 7.87 | 4.71 |
| FPGA09 | 151.02 | 85.76 | 1.76 | 52.40 | 2.88 | 51.37 | 2.94 | 45.20 | 12.51 | 3.61 | 9.20 | 4.91 |
| FPGA10 | 160.76 | 94.05 | 1.71 | 48.93 | 3.29 | 47.56 | 3.38 | 57.31 | 12.35 | 4.64 | 8.85 | 6.47 |
| FPGA11 | 143.20 | 84.04 | 1.70 | 50.91 | 2.81 | 49.52 | 2.89 | 45.52 | 12.43 | 3.66 | 9.05 | 5.03 |
| FPGA12 | 168.88 | 91.92 | 1.84 | 57.88 | 2.92 | 55.15 | 3.06 | 48.10 | 13.94 | 3.45 | 9.80 | 4.91 |
| Geomean | 97.18 | 61.12 | 1.59 | 34.61 | 2.81 | 31.73 | **3.06** | 36.79 | 10.68 | **3.44** | 7.16 | **5.15** |

Table 2.9: FPGA Resource Usage and Fmax

| Kernel(s) | Logic | Register | RAM | DSP | Fmax |
|---|---|---|---|---|---|
| gradient + spread | 48% | 47% | 37% | 67% | 201 MHz |
| gradient | 37% | 32% | 22% | 67% | 227 MHz |

ent steps for optimizing wirelength. We ran this fast placer on a machine with a 14 core, 28 thread Intel® Xeon® processor and an Intel® Arria10® FPGA on the same package [67]. The tightly coupled CPU and FPGA can share virtual memory and communicate with very low latency. The multi-threaded parts of our placer use all 28 threads. Our FPGA kernels were written in OpenCL and compiled using Altera Offline Compiler version 16.0, which uses Quartus for synthesis, placement and routing. The second placer can be configured to run the fast numerical solver from [8] with either the linear-programming-based spreading algorithm in [1] or our new fluid-flow-based spreading algorithm. The comparison of these two spreading algorithms in our accelerated global placement system is shown in Table 2.8.

Table 2.8 shows the runtimes for the two spreading algorithms as well as that for the overall global placement. For our new fluid-flow-based spreading algorithm, we show the 28-threaded CPU runtime as well as the FPGA accelerated runtime. "LP" refers to the linear-programming-based spreading algorithm in [1]. "Fluid" denotes our new fluid-flow-based spreading algorithm. Compared to the linear-programming-based spreading algorithm, our new fluid-flow-based spreading algorithm is 3.44x faster and the FPGA accelerated version is 5.15x faster. Our new global placer with FPGA-accelerated numerical solver and FPGA-accelerated fluid-flow-based spreading is 3.06x faster overall than the placer with CPU implementation of the same solver and the linear-programming-based spreading algorithm.

As shown in Table 2.8, the average runtime of the CPU implementation of our fluid-flow-based spreading algorithm is 10.68s and the corresponding overall global placement takes 34.61s. This suggests that the multi-threaded CPU implementation of our new spreading algorithm takes 31% of the global placement runtime in our accelerated global placer, which is significant. The CPU implementation of the fluid-flow-based flow computation takes 34.7% of the spreading runtime. The corresponding FPGA implementation accelerates this flow computation by 31x leading to an overall speedup of 1.5x just for the spreading algorithm.

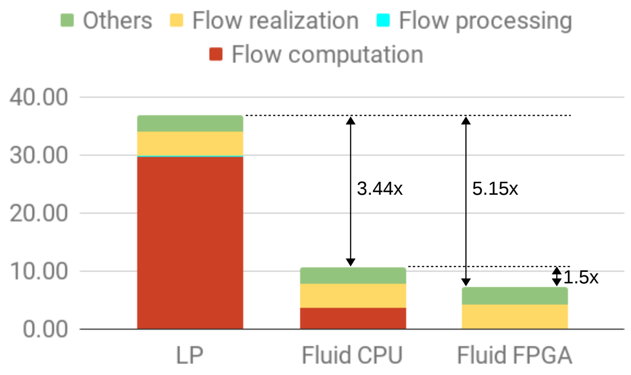


Figure 2.36: Runtime breakdown of the spreading algorithms.

Figure 2.36 shows the runtimes for different parts of the spreading algorithm(s). Computing the flows using linear program takes 80.9% of the runtime for the linear-programming-based spreading algorithm. Our multi-threaded fluid-flow-based algorithm reduces this time by 8x and FPGA acceleration reduces it further by 31x to a total of 248x. The multi-threaded flow realization algorithm is the same across all variants of the spreading algorithms shown in Figure 2.36. This step involves many memory allocation and deallocation operations and is hard to accelerate on the FPGA unless all the data structures used in spreading can fit on the on-chip RAMs. Flow processing involves conversion of flows from diagonal to manhattan and running the flow correction algorithm to remove cycles. This part consumes negligible runtime. “Others” includes time for data structure initialization and computation of anchor weights.

The FPGA resource usage and Fmax for our kernels is given in Table 2.9. We made some improvements to the gradient computation kernel in [8] to reduce resource usage. Adding the spreading kernel degrades the Fmax for both kernels and increases the runtime for the numerical optimization part by a small amount. However, accelerating the flow computation part on FPGA provides enough speedup to make up for this loss.

# Chapter 3

# Detailed Placement

Detailed placement in general (for both ASICs and FPGAs) can be categorized into the following broad classes:

- Greedy [32] [35]
- Simulated Annealing [30] [31] [36]
- Network flow/ matching [33] [34]
- Mixed Integer Linear Programming (MILP) [28] [29]
- Interleaving or Dynamic programming (DP) [37]
- Branch-and-bound [39]

---

This chapter includes work from the following publications:
[41] S. Dhar, S. Adya, L. Singhal, M. A. Iyer and D. Z. Pan, *"Detailed Placement for Modern FPGAs using 2D Dynamic Programming"*, International Conference on Computer Aided Design, 2016.
[43] S. Dhar and D. Z. Pan, *"GDP: GPU Accelerated Detailed Placement"*, IEEE High Performance Extreme Computing conference, 2018.
[42] S. Dhar, M. A. Iyer, S. Adya, L. Singhal, N. Rubanov and D. Z. Pan, *"An Effective Timing-Driven Detailed Placement Algorithm for FPGAs"*, International Symposium on Physical Design, 2017.
In all of these publications, the first author was responsible for developing and implementing the main ideas and conducting experiments.

Historically, variations of greedy algorithms have been the most popular methods for detailed placement. Some representative examples are: i)moving a cell(LAB/DSP/RAM/IO in case of FPGAs) to an empty legal site within some neighborhood of its current location ii)identifying and swapping pairs of cells that are in close proximity to each other. iii)ripple movements - shifting neighboring cells by a small amount in order to accommodate a cell at a given location. However, these greedy algorithms are susceptible to local minima. The number of nets per LAB in FPGAs is high ($\sim$50), so the optimal region for each cell is small, since moving a LAB affects many nets. Thus, the chances for finding good greedy moves decreases with increasing design size and complexity.

Simulated annealing is another important detailed placement algorithm. It is similar to greedy algorithms, except that it accepts suboptimal or hill-climbing moves with some probability depending on temperature that is determined by the annealing schedule. However, like greedy algorithms, simulated annealing has diminishing chances of finding good moves in reasonable runtime with increasing design size and complexity.

A different flavor of detailed placement algorithms involves network flows and bipartite matching. Typically, a bipartite graph is formed with cells representing one set of vertices and sites representing the other. Assigning each cell to a site incurs some cost. If one allows all cells to go to all locations, there is no good way to estimate the cost. Approximate cost functions work well when there are sufficient spaces, but they tend to perform poorly when utilization approaches 100%.

Next, we consider Linear and/or Integer Programming approaches. MILP can find the exact minimum for half-perimeter wirelength (HPWL) and some other cost functions, but it has exponential time complexity and is not scalable. This necessitates the use of smaller window sizes and more iterations, which outweighs the advantage of such algorithms.

In [37], the authors propose a dynamic programming algorithm which partitions an ASIC row into two sets of cells and optimally interleaves them, keeping the relative order of cells within each partition constant. Interleaving is done using dynamic programming, which constructs a placement solution by selecting cells from the partitions one at a time and keeping the best option at each step. This is less susceptible to local minima than greedy approaches and allows more movement than network flow/matching. However, partitioning into two sets only is quite restrictive and DP on rows only would explore limited solution space. In the next section, we propose a new DP algorithm with multiple partitions and apply it to a rectangular grid to address these concerns.

## 3.1 Detailed Placement using Dynamic Programming

We illustrate the limitations of state-of-the-art detailed placement algorithms with some examples:

- Consider the placement problem in Figure 3.1. In case of row-by-row DP,

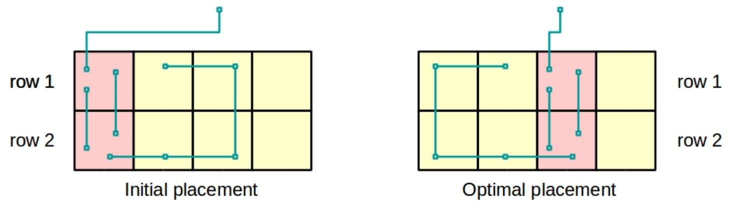


Figure 3.1: This placement would retain the initial configuration (local minima) unless the two pink cells are moved together

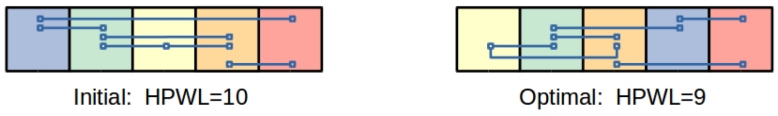


Figure 3.2: An instance which cannot be optimized by 2 partition DP

row1 and row2 are stuck in their respective local minima, as would also happen with greedy algorithms. Finding the optimal placement requires two cells in two adjacent rows to be moved simultaneously. ILP can solve this particular instance but it is not scalable. Simulated annealing might find the optimal solution, but it is not guaranteed. Network flow cannot model the net costs accurately in such a short range. Matching with independent sets is infeasible as the cells are connected.

- Consider the placement problem in Figure 3.2. It can be verified that DP with just two partitions [37] will get stuck with a solution that has HPWL of 10, independent of how the cells are partitioned. We need at least 3 partitions to solve this instance. Other approaches also suffer from similar drawbacks.

- Consider the placement problem in Figure 3.3. In many cases after global placement, just spaces need to be shifted without changing the relative order of the cells significantly. DP and network flow/matching are good choices for such placement refinements. However for dense cases like in Figure 3.4, network flow/matching perform poorly and the DP approach in [37] offers limited scope for pairwise swaps among cells.

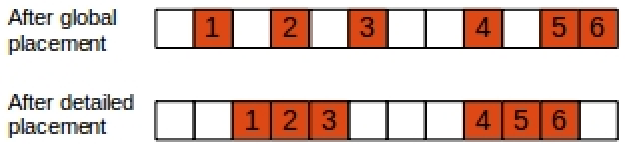


Figure 3.3: Sometimes we only need to adjust spaces during detailed placement

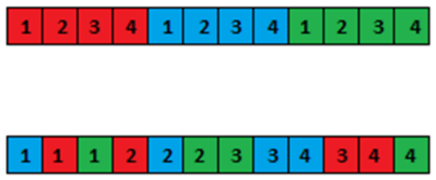


Figure 3.4: Interleaving example; The top row shows the placement before interleaving. The bottom row is the placement after interleaving

The key contributions in this section [41] are as follows:

- A dynamic programming (DP) algorithm for single row placement is proposed which has a parameter to tune quality vs runtime tradeoff. It can also find the optimum solution with the right paramemter setting. Moreover, another parameter is offered to restrict maximum displacement which improves runtime significantly.

- Our approach performs DP in a rectangular grid as opposed to the single row/column based approaches known previously. This circumvents local minima problems faced with single row/column based techniques and also allows macros (multi-row cells) to move.

- We prove that our DP algorithm runs in $\Theta(p.k.(\frac{N}{k}+1-d).((d+1)^k - d^k)+d^k)$ time, which is tractable for reasonable values of $k$, where $N$ is the number of cells, $p$ is the average degree of a net, $k$ is the number of partitions and $d$ is a parameter that controls maximum displacement of cells. As a special case, we can solve the single row placement problem optimally in $\Theta(p.N.2^N)$ time instead of the naive $\Theta(p.N!)$. (Magnitude comparison: $20.2^{20} = 20971520$ whereas $20! = 2432902008176640000$)

- We propose new parallelization schemes for our DP algorithm. In cases where we increase the complexity by increasing number of partitions, we demonstrate that a single problem instance can be solved in parallel in reasonable runtime. Our formulation also exploits the fact that x and y components of HPWL are decoupled and does 2-dimensional optimization in parallel.

### 3.1.1 Problem Statement for Dynamic Programming based Detailed Placement

Our two main objectives are maximum frequency (Fmax) and wirelength optimization. Of the many wirelength representations, Half-Perimeter WireLength (HPWL) is the most widely used and it usually correlates well with actual routed wirelength. Different weights are used for nets with different number of pins to make it correlate better with actual wirelength. Also, HPWL is simple to express in terms of the coordinates of the net end points (pins/cells). To improve route correlation, the fanout of a net is also used as a weighting factor in its HPWL calculation.

For incorporating timing information, we introduce timing nets(tnets), which are virtual 2-pin nets representing timing arcs. They connect every load to its driver. Timing weights on tnets are generated using slack information obtained from a static timing analysis tool, as in [40]. Lower slack (higher timing violation) connections get higher weights. We update timing weights at fixed intervals. Our problem formulation includes all the original nets in the design and the new virtual nets (tnets) along with all net weights. We assume the following inputs and constraints for our problem:

- Given: Hypergraph H with set of vertices V, set of nets/hyperedges E, set of sites S on which the vertices can be placed. V is the set containing all the LABs, IOs DSPs and RAMs in the current window in which DP is being applied. E contains all the nets among elements in V and also the new timing nets (tnets) that we introduce. E does not contain nets that are fully absorbed within a LAB.

- Each vertex is a unit square. Each site is also a unit square.

- Each net has a weight. Each net is connected to a vertex by a pin. Pin locations are specified by offset from the lower left corner of the vertex. Some nets have pins connecting to vertices not in V, which can be treated as fixed pins with respect to the current problem.

- Not all vertices can be placed on all sites (example: a RAM cannot be placed on a DSP site).

- Sites in S need not be contiguous.
- Number of vertices = number sites = N (blank spaces are treated as dummy vertices with no nets)

**Objective:** Assign vertices to sites such that the sum over all nets of weighted HPWL is minimized:

$$\min_{x(v),y(v)\,\forall v\in V}\{\sum_{net\in E} weight(net) \times HPWL(net)\} \tag{3.1}$$

Where HPWL is defined as:

$$\underset{of\,net}{HPWL} = \max_{v\in net} x(v) - \min_{v\in net} x(v) + \max_{v\in net} y(v) - \min_{v\in net} y(v) \tag{3.2}$$

Where $x(v), y(v)$ are coordinates of the pin on vertex $v$ connecting to the corresponding net. We can also have independent x and y weights for HPWL of each net. In that case, the objective becomes:

$$\min_{x(v),y(v)\,\forall v\in V}\{\sum_{net\in E} weighted\_HPWL(net)\} \tag{3.3}$$

Where $weighted\_HPWL$ is:

$$\begin{aligned} \underset{of\,net}{weighted\_HPWL} &= x\_weight \times \{\max_{v\in net} x(v) - \min_{v\in net} x(v)\} \\ &+ y\_weight \times \{\max_{v\in net} y(v) - \min_{v\in net} y(v)\} \end{aligned} \tag{3.4}$$

### 3.1.2 Dynamic programming in 1 dimension

For cells in a set of sites (row or a column or monotonic pattern), we partition the set of vertices (cells to be placed in those sites) into $k$ sets

$S_1, S_2, S_3, \ldots, S_k$ with $N_1, N_2, N_3, \ldots, N_k$ vertices respectively. Note that $N_1 + N_2 + N_3 + \cdots + N_k = N$, and the relative order among the vertices in each set must be preserved. Only interleaving among sets is allowed. For example, Figure 3.4 shows 3 partitions in 3 different colors, and the rearrangement of the cells in the same row maintains the relative order of cells within each partition (color).

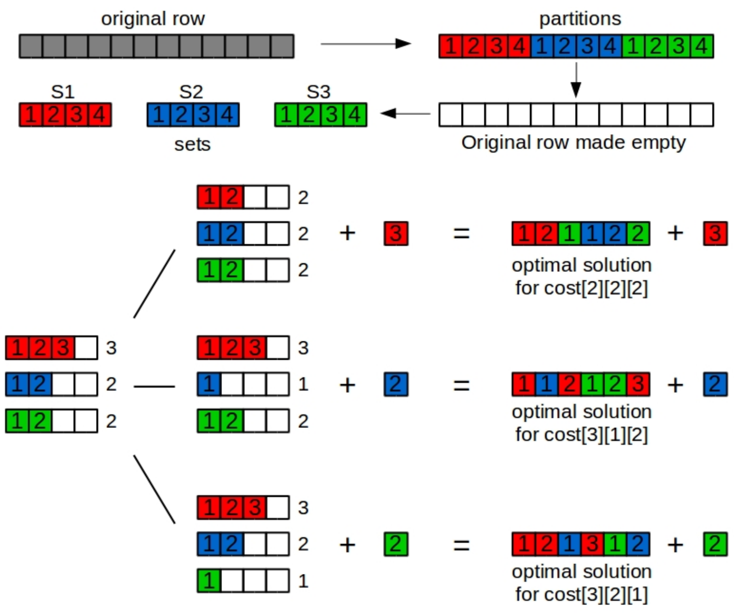

original row
partitions
S1
S2
S3
sets
Original row made empty
optimal solution for cost[2][2][2]
optimal solution for cost[3][1][2]
optimal solution for cost[3][2][1]

Figure 3.5: Computing cost[3][2][2] in the DP matrix - it depends on cost[2][2][2], cost[3][1][2] and cost[3][2][1]. Here, k=3 and N=12

### 3.1.2.1 Subproblem definition

We extend the formulation used in [37] from two partitions to $k$ partitions. We define a $k$-dimensional matrix $cost[\,][\,]\ldots[\,]$. Each dimension in the matrix corresponds to a partition and is of size $N_i + 1$, where the partition contains $N_i$ elements. Thus the cost of placing the first $i_1$ cells from $S_1$, the first $i_2$ cells from $S_2$, the first $i_3$ cells from $S_3$, $\ldots$ , the first $i_k$ cells from $S_k$ is represented in the entry of the matrix indexed as $cost[i_1][i_2][i_3]\ldots[i_k]$. This cost represents the best solution for this subproblem that essentially occupies the first $m = i_1 + i_2 + i_3 + \cdots + i_k$ sites. When this subproblem is solved, the remaining $N - m$ sites are still empty as the solution for the remaining $N - m$ cells have not yet been determined. This is required to ensure that the new placement of the cells satisfies the relative ordering of cells within each partition. We use special rules to compute the weighted HPWL cost for subproblem solutions, since all pins of the net may not have been visited in the partial solution. We discuss these rules later in this section. For nets that are affected by the cells in the subproblem, and for which not all pins have yet been considered, we assume that the nets end on the boundary of the last site in the subproblem. Figure 3.6 illustrates this situation. As the cost matrix gets incrementally computed during dynamic programming, we can conclude that the final minimum cost will be indexed as $cost[N_1][N_2]...[N_k]$.

The entries of the cost matrix are computed as follows:

$$cost[0][0]...[0] = 0;$$

$$\begin{aligned} cost[i_1][i_2]...[i_k] = min\{ & \\ & cost[i_1-1][i_2]\ldots[i_k] + \underset{at\,end}{cost\,of\,placing}\,S_1[i_1] \\ & cost[i_1][i_2-1]\ldots[i_k] + \underset{at\,end}{cost\,of\,placing}\,S_2[i_2] \\ & . \\ & . \\ & . \\ & cost[i_1][i_2]\ldots[i_k-1] + \underset{at\,end}{cost\,of\,placing}\,S_k[i_k] \\ & \} \end{aligned} \tag{3.5}$$

We illustrate this cost computation using an example. Consider a row of 12 cells with three partitions as shown in Figure 3.5. When computing the minimum cost for the subproblem indexed as $cost[3][2][2]$, we consider three cases: (i) The optimal cost of the subproblem $cost[2][2][2]$ + the cost pf placing $S_1[3]$ at the end (ii) The optimal cost of the subproblem $cost[3][1][2]$ + the cost of placing $S_2[2]$ at the end (iii) The optimal cost of the subproblem $cost[3][2][1]$ + the cost of placing $S_3[2]$ at the end. The minimum cost among all these three cases becomes $cost[3][2[2]$. It is worth noting that the costs are additive, and the cost of a subproblem depends on the pre-computed costs of smaller adjacent subproblems. If a site cannot be occupied because it is occupied by a cell whose placement is fixed, or the site is dedicated to special cells like RAMs, DSPs, etc. we set the cost of placing one of our cells in such a site as infinity. This ensures that such illegal solutions are never considered.

**Lemma 9** *This recurrence relation yields the optimal result satisfying the constraints of preserving relative order of vertices within each set. (Note that this is not the global optimum in general)*

**Proof:** When $N = 1$, we trivially obtain the optimal solution. When computing the solution for $N = 2$, we use the optimal solution from the $N = 1$ subproblem and add the minimum cost of placing the next cell at the second site location. Our costs are strictly additive, since we compute the HPWL costs only for the pins of the affected nets that are considered in any subproblem. As more pins of a net are considered in future solutions, the HPWL cost for the net may only monotonically increase. This ensures that the solution with $N = 2$ is optimal. Through induction, we can conclude that optimal solutions are computed for $N = 3, N = 4$, etc. That is, for any $N$, the placement solution computed is optimal. Q.E.D.

It can also be inferred that the optimal arrangement of the first $s$ sites is independent of the arrangements of the next $N - s$ sites for any $s \leq N$. However, the placement of a cell on the $q$-th site depends on the placements of all cells in sites $< q$.

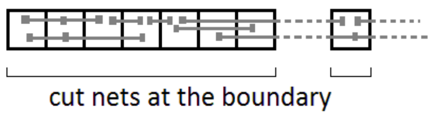


Figure 3.6: Sections of nets included in partial cost

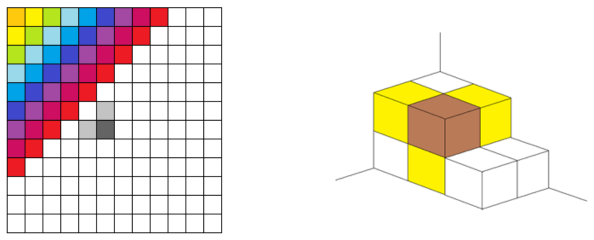


Figure 3.7: Filling the DP matrix hyperplane-by-hyperplane in 2 and 3 dimensions; Each color represents a hyperplane

#### 3.1.2.2 DP cost matrix computation

The cost matrix in the above DP formulation is a $k$-dimensional matrix, with sizes $N_1 + 1, N_2 + 1, \ldots, N_k + 1$ in the corresponding dimensions. Each entry in the matrix stores the minimum cost for the corresponding subproblem and some other details (omitted due to page limit) for tracing the optimal arrangement. This matrix can be visualized as a $k$-dimensional hypercube. Each entry in the hypercube is computed from the k entries adjacent to it in the lower dimensions. For example, in the 2-dimensional(square) matrix of Figure 3.7, the dark grey entry depends on the two light grey entries. In the 3-dimensional matrix (cube), the brown entry depends on the three yellow entries.

There are two different ways of filling the cost matrix:

1. Dimension-wise: Order the dimensions. Start filling from the lowest dimension. When it is full, move to the next dimension. This is like filling a square matrix row by row. For a cube, it is like filling plane by plane. Each plane(square matrix) is filled row by row. This approach is

the simplest to implement.

2. Hyperplane-wise: We can imagine a set of $k-1$ dimensional hyperplanes cutting through the $k$ dimensional hypercube. For a 2 dimensional case, hyperplanes are lines of the form $x + y = constant$. For 3D, hyperplanes are planes of the form $x + y + z = constant$. We can generalize for $k$D as $x_1 + x_2 + \cdots + x_k = constant$. Varying this *constant* from 0 to $N$ touches upon all the points in the hypercube. For each hyperplane, the entries in the cost matrix can be computed using the cost matrix entries computed earlier for an adjacent hyperplane. For example in Figure 3.7, the entries for the purple hyperplane can be computed using the pre-computed entries for the blue hyperplane above it. This makes the computation of all entries in a hyperplane independent of each other, thereby enabling parallelization.

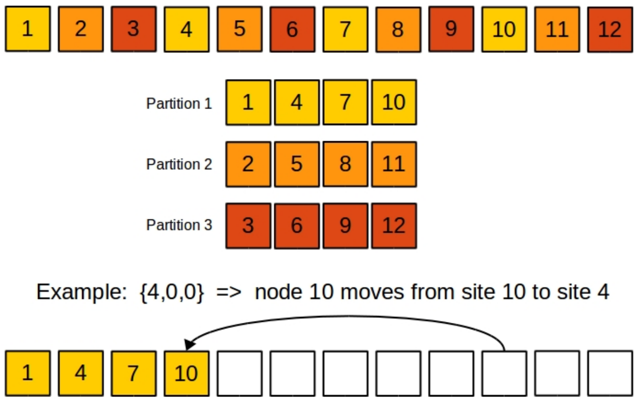


Figure 3.8: Solutions which are unlikely

We see that many of the entries in the aforementioned cost matrix correspond to solutions which are unlikely. Consider, for example, the scenario in Figure 3.8. The subproblem with 4 cells from the first partition, 0 cells from the second partition and 0 cells from the third partition has a solution where the cell at site number 10 moves to site number 4. In practical scenarios, large displacements like this are unlikely. We would like to have more control over the maximum displacement of each cell and reduce runtime by eliminating unlikely solutions. We introduce an additional constraint for this purpose:

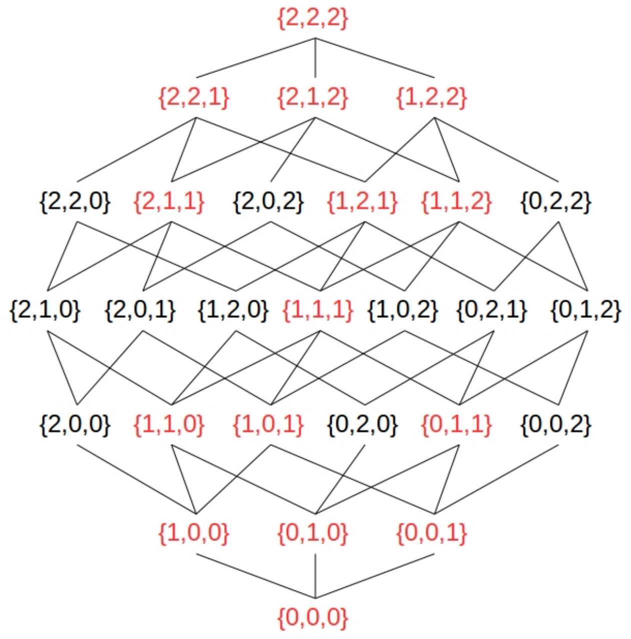
$$\max(i_1, i_2, \ldots, i_k) - \min(i_1, i_2, \ldots, i_k) \leq d \tag{3.6}$$

where $(i_1, i_2, \ldots, i_k)$ are the indices of the subproblem and $d$ is a constant parameter. This constraint greatly reduces the number of subproblems solved. Note that merely visiting all subproblems (entries in the matrix) and then deciding whether to solve or discard them is not an efficient solution as the number of matrix entries can be large for large values of $k$. Instead, we only traverse the subproblems that we intend to solve.

**Definition:** A partitioning of the set of cells in the initial placement is said to be fine if the cell at site $i$ is in partition $i \ mod \ k \ \forall i$

**Lemma 10** *The constraint in equation 3.6 limits maximum displacement for each cell to* $(k-1).(d+1)$ *in case of fine partitioning.*

**Proof:** Consider cell $i_j$ in partition $j$ and the subproblem $i_1, i_2, ..., i_j, ..., i_k$. The lowest numbered site on which this cell can be placed is $i_j - 1 + (k - 1).max(0, i_j - d) + 1 \ \geq \ k.i_j - (k-1).d$. The term $i_j - 1$ corresponds to placing all cells in partition $j$ that are before cell $i_j$ as relative order has to be maintained within each partition. The terms $max(0, i_j - d)$ are the minimum number of cells that have to be taken from each of the other $k-1$ partitions in order to satisfy the condition in equation 3.6. The highest numbered site on which cell $i_j$ in partition $j$ could have been in the initial placement is $k.i_j$. The maximum left displacement is $k.i_j - (k.i_j - (k-1).d) = (k-1).d$. The highest numbered site on which cell $i_j$ can be placed is $i_1 + i_2 + ... + i_k$. The lowest numbered site on which this cell could have initially been is $k.min(i_1, i_2, ..., i_j, ..., i_k) - k + 1$. The maximum right displacement is $i_1 + i_2 + ... + i_k - k.min(i_1, i_2, ..., i_j, ..., i_k) + k - 1 \leq k.min(i_1, i_2, ..., i_j, ..., i_k) + (k-1).d - k.min(i_1, i_2, ..., i_j, ..., i_k) + k - 1 \leq (k-1).(d+1)$. Q.E.D.



Example for *d=1*; only 15 out of 27 subproblems solved

Figure 3.9: Limiting the solution space explored

An example for the constraint in equation 3.6 is shown in Figure 3.9.

There are 6 cells and 3 partitions, each containing 2 cells. The figure shows all possible subproblems (matrix nodes) which are arranged in the form of a lattice. If we set $d = 1$ in equation 3.6, only the entries colored red are computed, which are significantly less in number than the total number of entries.

#### 3.1.2.3 Keeping track of nets

For computing the cost while placing a vertex $v$ at the $i^{th}$ site, we encounter 3 types of nets (Figure 3.10):

1. Nets which start at $v$ (i.e, no vertex of the net has been considered yet)
2. Nets which end at $v$ (i.e, remaining vertices for the net have already been considered)
3. Continuing nets. These may or may not be connected to $v$.

Finding which nets start at $v$ is easy. For each net, we know the vertices connected to it and their position in their respective sets. From the subproblem index $(i_1, i_2, \ldots, i_k)$, we check if the lowest index of any vertex connected to the net is greater than the $i's\,(i_1, i_2, \ldots, i_k)$ for the corresponding set. Similarly, we can find the nets ending at $v$. For continuing nets, we don't really need to track them individually. We just store the sum of the weights of the continuing nets, since the width of each site is 1. While calculating the cost of placing $v$ at site $i_1 + i_2 + \cdots + i_k - 1$, we first extend the nets from the previous site to the current site (take the distance between the sites and multiply by sum of weights of continuing nets). This is required because the site locations may not be contiguous due to blockages. Next, we add the costs for the starting and ending nets (nets may start/end at different points within the unit square). We also add the cost of the continuing nets (nets which started before and did not end at $v$). This incremental handling of nets ensures that the HPWL cost for a net can only remain constant or monotonically increase as more pins for an affected net get considered during the dynamic programming.

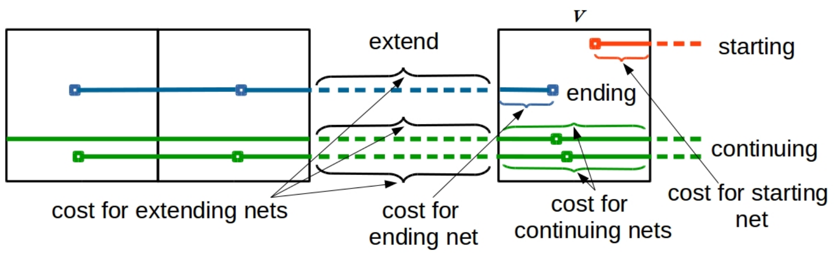


Figure 3.10: Various components in the HPWL cost: extending, starting, ending and continuing.

### 3.1.3 Complexity analysis

#### 3.1.3.1 Special case: no bound on $d$:

The $k$-dimensional cost matrix has $(N_1+1) \times (N_2+1) \times \cdots \times (N_k+1)$ entries. This is $(\frac{N}{k}+1)^k$ if the set sizes are roughly equal (This is an upper bound; this number will be lower if set sizes are unequal). For filling each entry, we look at the k entries in the dimensions immediately below. So, the complexity is lower bounded by $k.(\frac{N}{k}+1)^k$. Next, consider cost computation. For each of the $k$ choices we consider for filling an entry, we have to compute net costs. For this we have to go through all the nets connected to the vertices being placed. There is an upper bound on the nets connected to a vertex(each LAB/DSP/RAM has a fixed number of pins). Hence, this can be treated as a constant. For determining which nets start/end at $v$, one might think that the time complexity is $k$, but is is very unlikely that all nets will be connected to vertices in $k$ different sets for large $k$. If we take the sum over all matrix entries, this would lead to $\Theta(p.k.(\frac{N}{k}+1)^k)$ operations, where $p$ is the avg. number of pins per net, which can be practically bounded by a constant for realistic benchmarks. (This can be further reduced to $min(p,k).k.(\frac{N}{k}+1)^k$ operations)

#### 3.1.3.2 Exact solution

**Lemma 11** *If we set $k = N$, we will have the optimal solution.*

**Proof:** We already know that our algorithm gives optimal solution within our setting. We need to show that the setting allows exploration of the full solution space. If $k = N$, each vertex is in its own partition and preserving relative order does not apply. We will proceed by induction. For $N = 1$, it is trivial as we have only one choice in placing one vertex. Induction assumption: suppose our algorithm can arrange $M$ vertices optimally. For a problem of size $M + 1$, the last site can take any of the $M + 1$ vertices. For each choice of the last site, the previous $M$ must be arranged optimally. Our algorithm does so by the induction assumption. Since we take the optimum among all the possible $M + 1$ choices, our algorithm gives the optimal solution for $M + 1$ vertices. Q.E.D.

The complexity for the exact solution is $\Theta(p.N.(\frac{N}{N}+1)^N) = \Theta(p.N.2^N)$. One may think that this problem requires checking all the $N!$ possible enumerations($\Theta(p.N!)$), but it's actually not so. To see why, let's consider a simple case - 6 vertices $1, 2, 3, 4, 5, 6$. Suppose we already found that $3, 1, 2$ is the best arrangement for vertices $1, 2, 3$ when they are placed in the first half. Knowing this, we don't have to consider permutations $1, 2, 3, -, -, -$ , $2, 1, 3, -, -, -$ at all (Since the cost is additive; total cost = cost for $1^{st}$ half + cost of $2^{nd}$ half; the 2 halves can be optimized independently). It is worthwhile noting that $N.2^N$ is orders of magnitude less than $N!$ for even moderately large $N$. $N!$ is $\sim (N/e)^N$. For an idea of the magnitudes: $20.2^{20} = 20971520$ whereas $20! = 2432902008176640000$.

#### 3.1.3.3 General case: $d$ is finite:

We assume fine partitioning of the initial placement. We further assume that $N$ is a multiple of $k$ so that each partition has $\frac{N}{k}$ cells for the sake of simplicity. This assumption does not change the complexity bounds derived here.

The number of matrix entries computed is the number of tuples of the form $(i_1, i_2, ..., i_k)$ where $max(i_1, i_2\ , ..., i_k)\ -min(i_1, i_2, ..., i_k) \leq d$ with $0 \leq i_j \leq \frac{N}{k}\ \forall\ 0 \leq j \leq k$. This number is $(\frac{N}{k}+1-d).((d+1)^k - d^k)+d^k$, which can be calculated in the following way: The number of tuples $(i_1, i_2, ..., i_k)$ such that $0 \leq i_1, i_2, ..., i_k \leq d$ is $(d+1)^k$. The number of such tuples for $1 \leq i_1, i_2, ..., i_k \leq d+1$ is the same, and the number of tuples for $1 \leq i_1, i_2, ..., i_k \leq d$

is $d^k$. Combining these 3 values, we can get the number of tuples $(i_1, i_2, ..., i_k)$ such that $0 \leq i_1, i_2, ..., i_k \leq d+1$ and $max(i_1, i_2, ..., i_k) - min(i_1, i_2, ..., i_k) \leq d$ as $2.(d+1)^k - d^k$. (Tuples of the form $1 \leq i_1, i_2, ..., i_k \leq d$ were double-counted first and then subtracted.) We continue in this fashion for $2 \leq ... \leq d+2,\ \ 3 \leq ... \leq d+3,\ \ ...\ ,\frac{N}{k} - d \leq ... \leq \frac{N}{k}$ and sum these numbers up, which amount to $(\frac{N}{k} + 1 - d).(d+1)^k - (\frac{N}{k} - d).d^k \ = \ (\frac{N}{k} + 1 - d).((d+1)^k - d^k) + d^k$.

Each matrix node requires $\Theta(p.k)$ computation, so the overall complexity is $p.k.((\frac{N}{k} + 1 - d).((d+1)^k - d^k) + d^k)$. Sanity check: setting $d = \frac{N}{k}$ gives a complexity of $p.k.(\frac{N}{k} + 1)^k$, which was our result for the special case. $p.k.((\frac{N}{k} + 1 - d).((d+1)^k - d^k) + d^k) \ = \ p.k.((\frac{N}{k} + 1 - \frac{N}{k}).((\frac{N}{k} + 1)^k - (\frac{N}{k})^k) + (\frac{N}{k})^k) \ = \ p.k.((\frac{N}{k} + 1)^k - (\frac{N}{k})^k + (\frac{N}{k})^k) \ = \ p.k.(\frac{N}{k} + 1)^k$

If we apply DP to the whole chip in windows of size $N$, the overall runtime is $C.p.k.((\frac{1}{k} + \frac{1-d}{N}).((d+1)^k - d^k + \frac{d^k}{N}))$ where $C$ is the total number of cells/sites considered. An interesting observation is that this complexity decreases with increasing $N$ for $d = 1$.

### 3.1.4 DP in two dimensions

Interleaving within a single row/column has its own limitations - it can get stuck in a local minima due to bad ordering of cells (LABs/DSPs/RAMs/IOs etc.) in adjacent rows/columns as was shown with example in Figure 3.1. It is therefore necessary to optimize locations of cells in 2-dimensions all at once. This has been tried in the network flow / bipartite matching and Mixed Integer Linear Programming approaches as discussed before. However, in case of network flow / bipartite matching, the cost function becomes inaccurate if we try to move many cells at once. On the other hand, MILP can give good solutions but the feasible problem size is too small. Hence, we devised another algorithm to tackle this problem.

Extending our 1-dimensional DP formulation to 2 dimensions is non-trivial because the costs in the 2 dimensions are not additive. When placing a cell at a particular site, the cost of placing it cannot be directly added to the optimal solution for all cells below it, as some of the unfinished nets in the optimal solution of the subproblem may have different range of x or y coordinates. In 1D, we could say that all those nets were to the left and ended at the boundary of the last site in the subproblem, but it is not the case in 2D. We introduce additional constraints to make 2-dimensional DP formulation feasible:

1. cells in the same row will stay together
2. cells in the same column will stay together

For simplicity, we show a problem formulation with just 2 partitions for rows ($S_{r1}$,$S_{r2}$) and 2 partitions for columns ($S_{c1}$,$S_{c2}$). This easily generalizes for multiple partitions. $cost[i][j][k][l] = min\,cost$ considering $i$ rows from $S_{r1}$, $j$

rows from $S_{r2}$, $k$ columns from $S_{c1}$ and $l$ columns from $S_{c2}$:

$$
\begin{aligned}
cost[i][j][k][l] = min\{ & \\
& cost[i-1][j][k-1][l] + \underset{at\,ends}{(S_{r1}[i], S_{c1}[k])} \\
& cost[i-1][j][k][l-1] + \underset{at\,ends}{(S_{r1}[i], S_{c2}[l])} \\
& cost[i][j-1][k-1][l] + \underset{at\,ends}{(S_{r2}[j], S_{c1}[k])} \\
& cost[i][j-1][k][l-1] + \underset{at\,ends}{(S_{r2}[j], S_{c2}[l])} \\
& \}
\end{aligned}
\tag{3.7}
$$

We start from $(i, j, k, l) = (0, 0, 0, 0)$ and go till $(|S_{r1}|, |S_{r2}|, |S_{c1}|\ , |S_{c2}|)$. We can simplify our formulation with the following lemma:

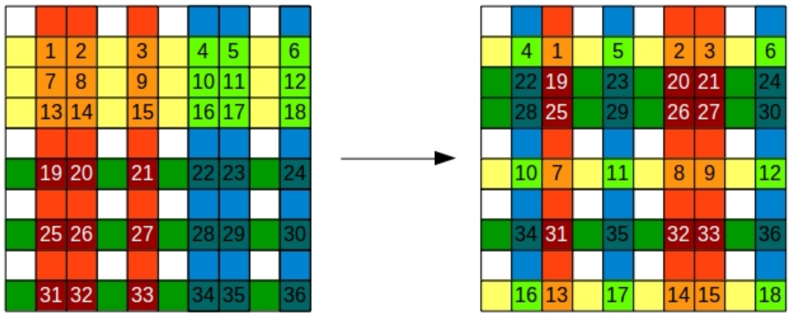


Figure 3.11: 2D DP: cells in the same row stay in one row, cells in the same column stay in one column.

**Lemma 12** *Cost of placing* $(S_{rm}, S_{cn})$ *and the ends = cost of placing* $S_{rm}$ *at row end + cost of placing* $S_{cn}$ *at column end.*

**Proof:** HPWL of a net = horizontal span + vertical span. Since all cells in the same column stay together, the $y$ components of HPWLs of all nets incident on that column will be invariant w.r.t column movement (does not change vertical span), only row movement will affect them. Similarly, since all cells in the same row stay together, the $x$ components of HPWLs of all nets incident on that row will be invariant w.r.t row movement (does not change horizontal span), only column movement will affect them. Q.E.D.

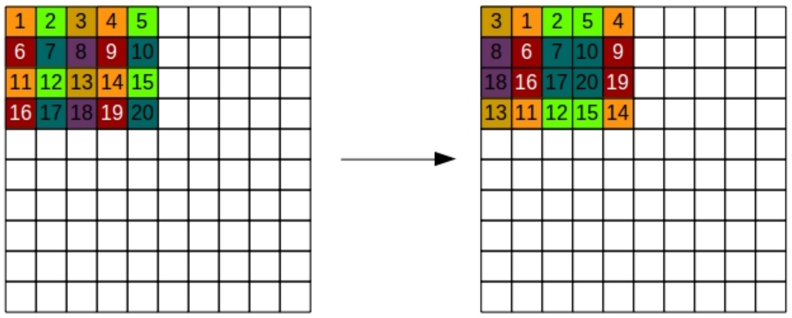


Figure 3.12: 2D DP as applied on a window; In this example, the cells in the white region are assumed to be stationary. Instead of moving a whole row or column, we move parts of rows or columns.

Figure 3.11 illustrates the main idea. Cells 1, 7 and 13 are initially in the same column, and they stay together in one column, even if they move to different rows. Similarly, cells 1, 2 and 3 stay together in one row, even if they move apart in columns. Observe that we need not move all the cells in the grid. We can choose some rows/columns for interleaving. Figure 3.12 illustrates the procedure in a small window. Here, we move sections of rows/columns instead of entire rows/columns. The cells in the white regions may be assumed to be stationary for this example. We would vary the window height and width to ensure that cells have sufficient opportunity to move with respect to each other. A whole row/column may not want to move cohesively but parts of it may be pulled in different directions. Setting width and height of window

= width and height of chip respectively enables optimization of the whole chip at once. Setting width or height to 1 would reduce this approach to the previously discussed column or row DP respectively. Observe that the x and y components of HPWL are now independent, so interleaving of rows and columns can be done in parallel. Another advantage of our 2-dimensional formulation is that it allows macros to move as demonstrated in Figure 3.13. Here, we set the window height equal to or slightly greater than the macro height. Here, we don't interleave rows(otherwise relative positions of LABs within a macro would change), only columns. For a fixed window, some macros may be protruding out and those columns are discarded from the current optimization problem. Those macros will be included when the window slides up/down.

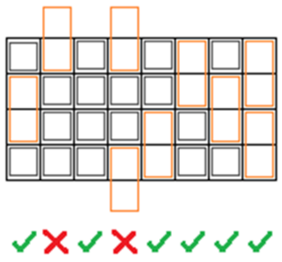


Figure 3.13: Selecting columns for 2D DP: We reject columns where macros don't fit in the window

### 3.1.5 Results

We tested our algorithm on an industrial benchmark set whose details are given in Table 3.1. We used the output of an industrial strength global placer and legalizer as starting point for all experiments in this subsection.

For 2D, we present the results for column section interleaving. We vary the window height in steps and it is same for all the experiments. We update timing weights after every four iterations. In all the data presented in this subsection, we report the geometric average across all benchmarks that have high statistical confidence.

Table 3.1: Benchmark set details

| Design size | # LABs, RAMs and DSPs |
|---|---|
| Minimum | 4156 |
| Maximum | 40889 |
| Average | 14850 |
| Number of designs | 86 |

For our experiments, we have 3 parameters: N (window length), k (number of partitions) and I (number of iterations). We set $d=\infty$ for these experiments. One iteration consists of one pass of row DP, one pass of column DP and one pass of 2D DP.

We compare our results with an implementation of the row-based DP algorithm in [37], with window size of 25 and 16 iterations. Each DP iteration for this implementation has 3 rounds of row optimization to be comparable in terms of number of moves attempted. On the average, our algorithm improves wirelength by 3.46%, and the maximum clock frequency (Fmax) by 0.45%. Since our algorithm is designed with parallelism in mind, we observe that the parallel runtime of our algorithm is 5.66x lower than the serial runtime of [37]. For the same number of partitions, our results indicate that applying DP on rows, columns, and rectangular grids improves wirelength by 2.6%, while improving Fmax by 0.2%.

Table 3.2: Comparison with [37]

| Parameters | $\Delta$Wirelength(%) | $\Delta$Fmax(%) |
|---|---|---|
| [37], N=25, I=16 | -1.11 | 1.14 |
| ours, N=25, k=3, I=16 row DP only | -1.97 | 1.39 |
| ours, N=25, k=3, I=16 row + column + 2D DP | -4.57 | 1.59 |

We run separate experiments by varying N, k and I individually to see their effect on wirelength and Fmax. The results are shown in Tables 3.3, 3.4 and 3.5.

Table 3.3: Effect of changing window length for k=3 and I=16

| Parameter | $\Delta$Wirelength(%) | $\Delta$Fmax(%) |
|---|---|---|
| N=10 | -4.04 | 1.48 |
| N=25 | -4.57 | 1.59 |
| N=50 | -4.98 | 2.24 |
| N=100 | -5.11 | 1.89 |

From Table 3.3, we see that increasing window length yields better improvement in wire and Fmax on average. A longer window allows larger cell displacement. Since we use tnets and weights on nets, it is important that they actually correlate with the net criticality in order to model timing correctly. If we move a cell too far in one step, some other nets may become critical. This can explain the slight dip in Fmax improvement for N=100. The optimum N for Fmax improvement appears to be between 50 and 100.

Table 3.4 shows an interesting result. Increasing number of partitions

Table 3.4: Effect of changing number of partitions for N=25 and I=16

| Parameter | ΔWirelength(%) | ΔFmax(%) |
|---|---|---|
| k=3 | -4.57 | 1.59 |
| k=5 | -4.93 | 1.28 |
| k=7 | -5.02 | 0.71 |

improves wire but decreases Fmax improvement. Wire improvement is related to cell displacement, and bigger k allows more displacement (less number of relative order constraints). However, large displacement steps are not good for Fmax, for the same reason as stated before.

Table 3.5: Effect of changing number of iterations for N=25 and k=16

| Parameter | ΔWirelength(%) | ΔFmax(%) |
|---|---|---|
| I=10 | -4.38 | 0.98 |
| I=16 | -4.57 | 1.59 |
| I=25 | -4.70 | 2.28 |
| I=40 | -4.81 | 3.23 |

Table 3.6: Runtimes

| | Experiment | Runtime(seconds) |
|---|---|---|
| 1 | Serial, N=25, k=3, I=16 | 113.12 |
| 2 | Parallel(16 threads), N=25, k=3, I=16 | 14.28 |
| Parallel speedup = 7.92 | | |
| 3 | [37] serial, N=25, I=16 | 80.89 |

Table 3.5 shows that iterating more with same N and k consistently improves both wire and Fmax. From these experiments, we learn that making many small moves is better than making a few abrupt moves for increasing Fmax. In general, running more iterations also allows our algorithm to work

Table 3.7: DP with new timing cost and selective LAB optimization

| Parameter | Δ Fmax(%) | Δ Wirelength(%) |
|---|---|---|
| geomean | 7.352 | 0.594 |
| confidence | 13.534 | 2.245 |

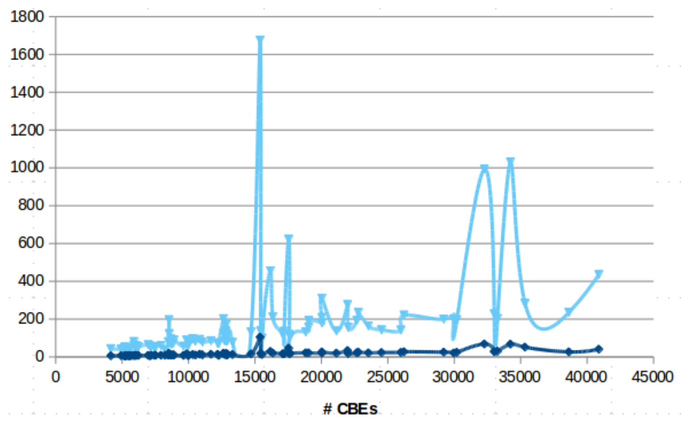


Figure 3.14: Serial(light blue) and parallel(dark blue) runtimes(in seconds) vs design size; #CBEs = #LABs + #DSPs + #RAMs

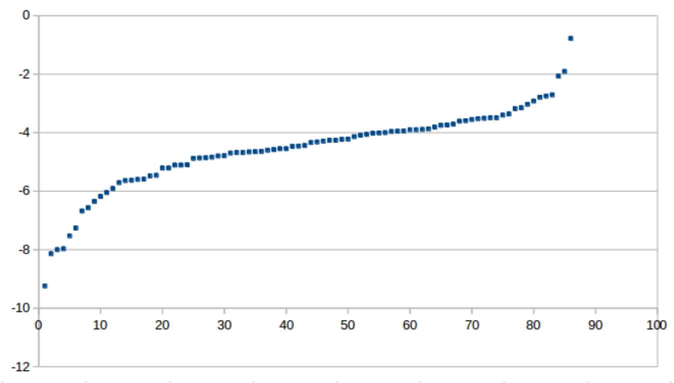


Figure 3.15: % Wire change (sorted from smallest to largest) for all designs

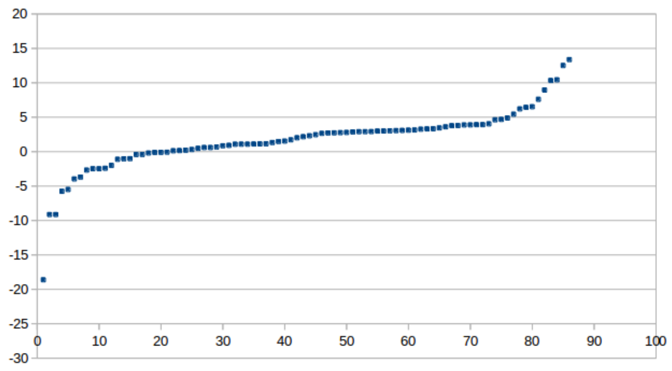


Figure 3.16: % Fmax change (sorted from smallest to largest) for all designs

with more accurate timing information since the timing weights are updated after every four iterations.

Table 3.6 shows the runtime improvement our algorithm gets by parallelizing. Experiments were run on 2.7 GHz, Intel® Xeon® 2680, 16 core machines with 16 threads. Runtime vs. design size is shown in Figure 3.14. Sorted %Fmax and %wirelength changes are shown in Figures 3.15 and 3.16. As we discussed before, our linear timing cost with tnets and net weights may not be accurate for very large displacements. However, we can always cache the initial placement and discard our changes if Fmax degrades. By doing this for N=25, k=3 and I=16, we get 2.51% better Fmax and 3.60% better wirelength over our starting point (legalized global placement). Running more iterations will in general cost more runtime, but can improve wirelength and Fmax as shown in Table 3.5.

We also implemented an enhanced timing cost and ran DP selectively on LABs in critical paths and observed significant improvement in Fmax (as shown in Table 3.7) with placer worst case time increase of 13.428% and externally measured total time increase of 3.096%

## 3.2 GPU Acceleration of Dynamic Programming

Detailed placement takes a significant part of total placement runtime, especially after global placement has been accelerated, as described in chapter 2. Reducing detailed placement runtime through parallelization and acceleration can yield considerable runtime improvements. However, not all detailed placement algorithms are amenable to parallelization / acceleration. It is important to devise the right algorithm that can run fast and yet produce good quality results.

Prior work on detailed placement can be classified into the following broad classes:

- Greedy [32] [35]
- Simulated Annealing [30] [31] [36]
- Network flow/ matching [33] [34]
- Mixed Integer Linear Programming (MILP) [28] [29]
- Interleaving or Dynamic programming (DP) [37]
- Branch-and-bound [39]

Among these, greedy / simulated annealing techniques are the easiest to accelerate on a GPU as demonstrated in [14] and [15]. They involve evaluating and performing numerous moves and can be parallelized in terms of move calculations but they require synchronization steps to find the best move and resolve conflicting moves. In some cases, good moves are uncovered only when certain other moves have been performed that interfere with each other and need to be serialized.

Network flow/ matching, mixed integer-linear programming and branch-and-bound techniques typically solve many small instances of the respective problems and can be parallelized by assigning different problems to different threads or workers. However, parallelization within the problem itself is limited for these techniques. Moreover, the number of variables and constraints needed for linear / integer programs increases rapidly with increasing problem size, leading to unfavorable runtime. Interleaving or dynamic programming, on the other hand, offers fine-grained parallelism within the problem. The core of the algorithm involves creating a dense solution matrix, which can be easily accelerated on a GPU.

The key contributions of our work towards accelerating detailed placement on a GPU [43] are as follows:

- We propose several optimizations to the dynamic programming algorithm described in the previous section to enable it to run fast on a GPU, which include restructuring the algorithm to improve memory access patterns, grouping similar work together, managing threads, etc.
- We propose a flow which performs the entire detailed placement on a

GPU, thus eliminating the memory transfer overhead between CPU and GPU.

- We analyze the complexity for the slowest component of our algorithm and show that the speedup of our proposed method is linear in the number of workers or threads available.

### 3.2.1 Overall Flow

The dynamic programming algorithm is applied by dividing the chip floorplan into non-overlapping windows which are sufficiently far apart and solving an independent DP problem in each window. Each set of windows generates a batch of DP problems ($\sim$126 in our case) for the GPU and multiple such sets are required for covering the entire chip.

We have 2 different flows with GPU acceleration: hybrid and full GPU (Figure 3.17). Recall that DP involves filling a matrix (Equation 3.5). In the hybrid flow, the DP problems are first formed using multi-threaded CPU code. We then offload the work of filling the matrix entries (Equation 3.5) to the GPU. Once the matrix is filled, we copy the solution back from the GPU and update the object (cell) locations using CPU. In the full GPU flow, we transfer the entire netlist and floorplan data to the GPU memory at the beginning of detailed placement stage. All subsequent iterations of detailed placement take place on the GPU. We copy the location data back from the GPU at the end of detailed placement.

We discuss the details of the GPU implementation of the dynamic programming algorithm for $k$=3, which has the best QoR/runtime tradeoff according to the results published in [41]. Deciding the value of $k$ at runtime on a GPU would incur significant performance impact as it would add an extra level of iteration. It is better to have a separate implementation for each desired value of $k$ as it allows us to manually unroll loops and optimize our code. It is also possible to experiment with multiple implementations for different values of $k$ on several GPUs and dynamically pick the best solution,

### 3.2.2 CUDA basics

We briefly discuss some relevant details of CUDA as described (subject to change) in [65]. Computations in CUDA are organized into kernels. The workload in each kernel is distributed into a number of blocks (can be specified at kernel launch time). Each block consists of a number of threads (can also be specified). CUDA blocks run on streaming multiprocessors in the GPU. The blocks are scheduled independently depending on the availability of streaming multiprocessors. Threads within a block are bundled into groups of 32 called warps. Threads in a half-warp run in lockstep with each other. Any divergence of control flow in the half-warp (ex: some threads executing an if condition while others executing the corresponding else condition) results in sequential execution of the two divergent paths. Warps themselves may not run in lockstep with each other. Hence, it is sometimes necessary to explicitly synchronize threads in a block.

There are multiple levels of memory with caches. There is a maximum

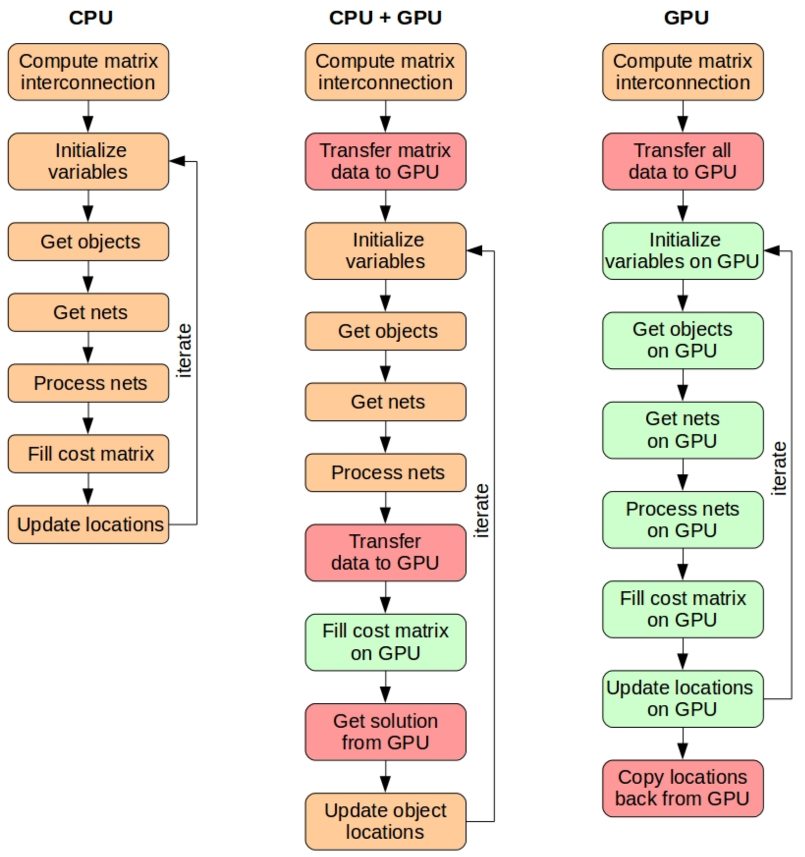


Figure 3.17: CPU, hybrid (CPU+GPU) and GPU flows. Orange parts execute on CPU, green parts execute on GPU and memory transfers between CPU and GPU are shown in red.

of 64kB shared memory per block (at least for our device, TITAN Xp). Memory accesses should be coalesced for best performance. Unaligned accesses and memory bank conflicts cause multiple reads. One exception for bank conflicts is when all threads in a block access the same address in a bank, in which case the data is broadcast to all threads. These features also depend on the compute capability of the device.

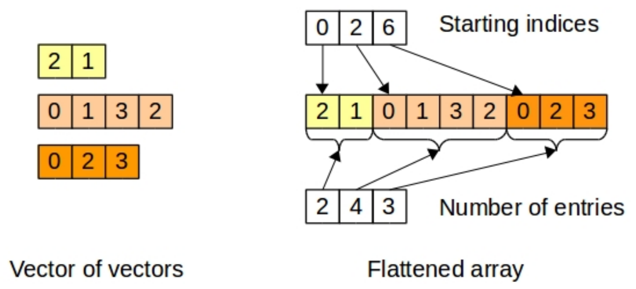


Figure 3.18: Flattened data structures.

#### 3.2.2.1 Data structures

We use flattened data structures in the form of arrays of fields as they lead to better coalescing of memory accesses. For example, the cells for each net and nets for each cell are stored in compressed sparse row format (Figure 3.18). The data for all the DP problems are stored sequentially in arrays.

Since we cannot use data structures similar to stl (C++ standard template library) vectors, sets or maps due to the inefficiency of dynamic memory allocation, we adapted the algorithms to use arrays. This introduced some complications like extra memory requirement. Example: For collecting nets (without duplicates) connected to a set of objects in a DP problem, we maintain an array of size number of nets × number of problems resembling an adjacency matrix between nets and problems.

Table 3.8: Kernels for various tasks

| Task | Kernel(s) |
|---|---|
| Initialize variables | Init Cluster Variables, Init Cluster Net Variables, Init Net Variables |
| Get objects | Init Objects |
| Get nets | Parallel scan to calculate cumulative #nets, Get Nets |
| Process nets | Parallel scan to get net starting locations, Preprocess Nets, Preprocess Sets, Init Nets |
| Fill cost matrix | Fill Cost Matrix |
| Update locations | Trace Solution, Update Locations |

#### 3.2.2.2 Tasks

Our GPU flow can be loosely grouped into a set of tasks (Figure 3.17). Each task (Table 3.8) calculates some variables of interest using the kernels that we developed and/or existing libraries like Thrust [66]. We list the tasks below:

- **Initialize variables:** All variables used by all kernels are initialized for each iteration of detailed placement.
- **Get objects:** The chip floorplan is divided into windows on which DP problems are solved. We need to collect the movable objects in each window and the sites on which those objects can be placed.

- **Get nets:** All the nets connected to the objects in each DP problem are collected and duplicates are removed. A net in the netlist may belong to multiple DP problems.

- **Process nets:** We have to calculate data for each net in each problem like first and last objects connected to it in each set and the bounding box of the external pins (pins which are not in the current problem).

- **Fill cost matrix:** Cost matrices for all DP problems are filled with data such as partial minimum cost, partial solution for the minimum cost, and partial sum of net weights for each matrix entry.

- **Update locations:** The solution to each DP problem is constructed by traversing the cost matrix from end to beginning and the locations of the objects are updated.

Using too many threads on one CUDA block can sometimes slow the program down due to various factors like irregular memory access, scarcity of registers, etc which also depend on the architecture of the GPU being used. So we decided to test two versions of our matrix filling kernel: one which separates out different DP problems into different CUDA blocks (we call it **independent sub-flow**) and one in which all the blocks (except the last one) have the same number of threads and may solve parts of multiple DP problems at any given time (we call it **combined sub-flow**).

### 3.2.3 Kernels

We describe some of the important kernels in this section. For the sake of simplicity, we only list the important variables and operations within the kernels and omit all other details. The number of threads per block (*blockDim*) and number of blocks are specified at the beginning of kernel descriptions.

#### 3.2.3.1 Fill Cost Matrix

***Independent:*** *blockDim = 1024, #blocks = #problems*

***Combined:*** *blockDim = 1024, #blocks =* $\lceil \frac{\#nodes\, on\, hyperplane \times \#problems}{1024} \rceil$

The main workload of the DP algorithm is filling the cost matrix. We fill the matrix hyperplane-by-hyperplane as this offers the most amount of parallelism. Each thread in a block fills at most one entry in each hyperplane.

For our implementation, we chose N=84 which is more than sufficient as wirelength improvement saturates at higher N. We also require that all partitions be of the same size, which is the case for N=84. This ensures that we have to compute which nodes belong to which hyperplane and the interdependence between hyperplane nodes only once, which saves a lot of time as this kind of computation is hard to accelerate on a GPU.

Since we have a fixed problem size (84), we introduce dummy objects and sites at the end if our actual problem is smaller. Note that this does not alter the solution space explored as the since we ensure that the dummy objects are placed only at the dummy sites at the periphery of the window.

**Algorithm 2** Fill Cost Matrix (independent sub-flow)

```
for $i$ = 1 to $N$ do
    //shared variables: matrix_offset, sitegap
    if threadId == 0 then
        calculate matrix_offset and sitegap
    end
    syncthreads;
    //per-thread variables: cost1, cost2, cost3
    cost1=$\infty$; cost2=$\infty$; cost3=$\infty$;
    if threadId < # nodes on hyperplane then
        determine id using matrix_offset, blockId and threadId;
        if # objects taken from set 1 > 0 & site is legal then
            for net $\in$ last object taken from set 1 do
                determine if net starts, ends or continues;
                cost1 = cost1 + cost of net;
            end
        end
        if # objects taken from set 2 > 0 & site is legal then
            for net $\in$ last object taken from set 2 do
                determine if net starts, ends or continues;
                cost2 = cost2 + cost of net;
            end
        end
        if # objects taken from set 3 > 0 & site is legal then
            for net $\in$ last object taken from set 3 do
                determine if net starts, ends or continues;
                cost3 = cost3 + cost of net;
            end
        end
        matrix_cost[id]=min(cost1,cost2,cost3);
        matrix_argmin[id]=argmin(cost1,cost2,cost3);
    end
    syncthreads;
end
```

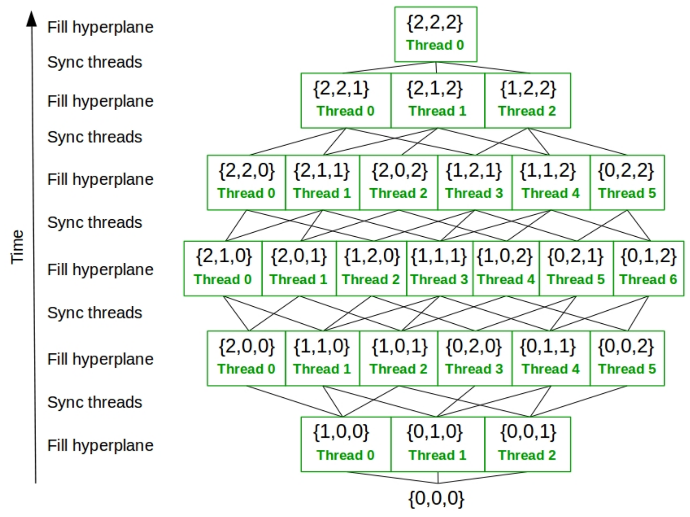


Figure 3.19: Filling the cost matrix for a single problem. This operation is performed on one block. Different nodes in a hyperplane are assigned to different threads in the block. Threads are synchronized after each hyperplane is filled.

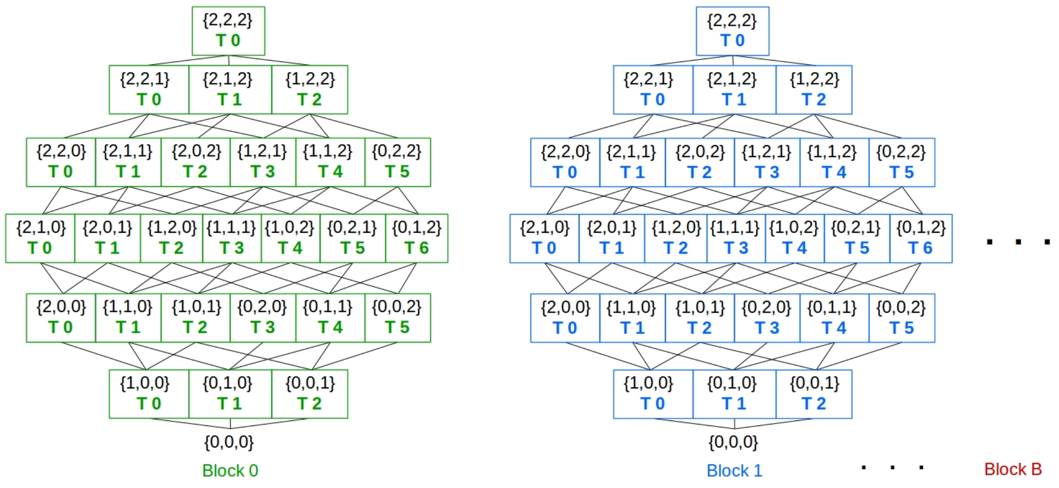


Figure 3.20: Independent sub-flow: Different problems are assigned to different blocks which run independently.

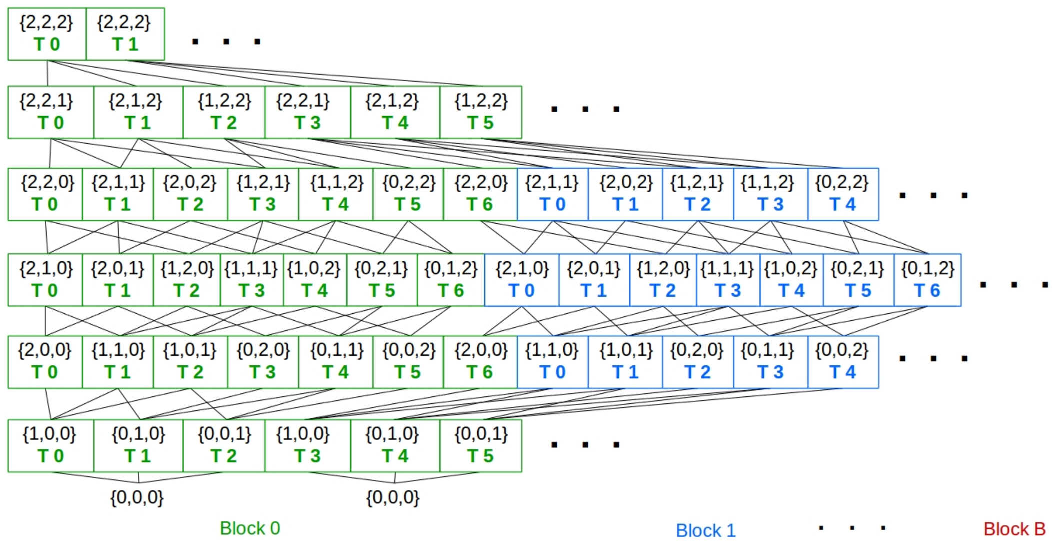


Figure 3.21: Combined sub-flow: Hyperplanes from different problems are grouped together. This example has blockDim=7

**Independent sub-flow:** We assign each DP problem to a block (Figure 3.20) to avoid synchronization between blocks. This assignment implies that all the threads within a block are not fully utilized. The sizes of different hyperplanes are different, with the max size near the middle of the hypercube and minimum near the corners. For DP with three partitions, the average size of a hyperplane is $\Theta(N^2)$. In general, for a k-partition DP, it is $\Theta(N^{k-1})$. Note that our choice of mapping one DP problem to one block also limits the problem size as each block can have at most 1024 threads. For $k = 3$, the maximum hyperplane size is $\sim \frac{N^2}{4\sqrt{3}} \leq 1024$. Hence, $N \leq 84$, which is more than sufficient.

Algorithm 2 describes the process of filling matrix entries for a DP problem for the independent sub-flow. Each block (DP problem) iterates over hyperplanes indexed from 1 to N. Each iteration of the loop proceeds as follows: First, shared variables like $matrix\_offset$ and $sitegap$ (distance of the current site from the previous site) are calculated by thread0. We synchronize all threads in the block after this to ensure that they all receive the right value for these shared variables. Each thread then independently checks if its id is less than the number of nodes on the current hyperplane. If this condition is satisfied, then it evaluates three different cases of appending an object from a set to the end of the current placement and selects the best one. All the threads in the block are then synchronized so that computation on the current hyperplane is completed. Figure 3.19 shows the execution of the algorithm for one block (problem).

**Combined sub-flow:** We utilize all the threads in each CUDA block (except the last one) in this sub-flow. This approach also fills the matrix hyperplane-by-hyperplane, except that we club all the matrix nodes for the corresponding problems into one combined hyperplane and fill that with one kernel call (Figure 3.21). Each thread has to determine which DP problem it is working on and also the corresponding matrix node. One disadvantage of this approach is that we have to iterate over the hyperplanes outside of the kernel. We do it using a loop in the CPU code. We maintain a pointer to the current hyperplane on the GPU memory which is passed as a variable to the kernel and also increment it in the loop. The rest of the kernel is similar to the kernel in the independent sub-flow.

#### 3.2.3.2 Get Nets

*blockDim = 1, #blocks = #problems*

This kernel (Algorithm 3) finds the ids for nets for each DP problem. Each block maps to a DP problem. Each block iterates over all objects in the problem and all nets for each object (nested iteration) and assigns a serial number to a net if it has not been visited before. To do this, we need to maintain an array of the size number of nets × number of problems. Even though this may seem large, it is manageable (400000 nets and 126 problems: ∼200MB of memory). We only use one thread per problem to do all the work as using more threads can increase runtime due to irregular memory accesses from the nested iteration.

**Algorithm 3** Get Nets

```
count = 0;
for $i$ = 1 to $N$ do
    for net ∈ $object_i$ do
        if net not marked for this problem then
            mark net for this problem;
            id of net in this problem = count;
            count++;
        end
    end
end
# nets in this problem = count;
```

#### 3.2.3.3 Preprocess Nets

*blockDim = 1024, #blocks = #problems*

This kernel (Algorithm 4) collects all the nets incident on objects in a DP problem and stores them in an array without duplicates. Each DP problem is mapped to a block. Each block iterates over all the nets in the netlist and checks if a net is incident on some object in the problem (using id of a net in a problem computed by Get Nets kernel). Each thread iterates on a subset of nets. We have observed that the runtime of this kernel depends heavily on the policy of allocating nets to threads. A stride of 1024 runs in 1ms whereas a stride of 1 runs in 36ms for one set of problems in one test case (Figure 3.22).

**Algorithm 4** Preprocess Nets

```
if threadId==0 then
    calculate offset for this problem in expanded net array;
end
syncthreads;
//offset used in constructing subsets of nets
for net $\in$ subset of nets do
    if net $\in$ this DP problem then
        add net to this problem using id computed in kernel Get Nets;
        set problem id for this instance of net;
    end
end
```

#### 3.2.3.4 Preprocess Sets

*blockDim = N, #blocks = #problems*

This kernel (Algorithm 5) finds the nets connected to objects in each of the three sets for each problem. Each DP problem maps to a block and each thread in the block maps to an object in the problem. Each thread iterates over all nets connected to the object and fills net ids for the set to which the

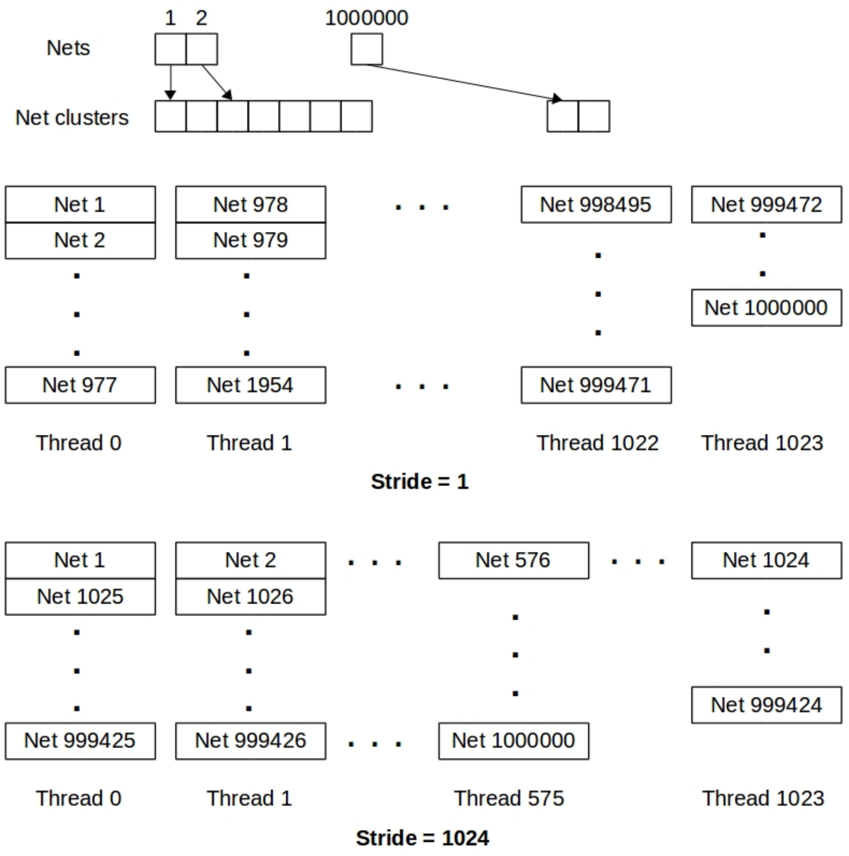


Figure 3.22: Two different memory access strides for processing nets on the same block for an example netlist with 1000000 nets.

object belongs.

**Algorithm 5** Preprocess Sets

```
for $net \in object_{threadId}$ in current problem do
    if $object_{threadId} \in set1$ then
        add $net$ to set1 net ids;
    end
    if $object_{threadId} \in set2$ then
        add $net$ to set2 net ids;
    end
    if $object_{threadId} \in set3$ then
        add $net$ to set3 net ids;
    end
end
```

#### 3.2.3.5 Initialize Nets

$blockDim = 64$, $\#blocks = \lceil \frac{total\,\#nets\,for\,all\,problems}{64} \rceil$

This kernel (Algorithm 6) finds the first and last objects in each set and external bounding box for all the nets collected for all the DP problems. All the collected nets are split among blocks and threads. We have empirically determined the best block dimension to be 64. Each thread maps to a collected net in a problem. The same net can belong to multiple problems so there can be multiple instances of the same net in the array. Hence, we need an array to store which problem that particular instance of a net belongs to. Each thread iterates over all cells connected to a net.

#### 3.2.3.6 Trace Solution

*blockDim = 1, #blocks = #problems*

**Algorithm 6** Initialize Nets

```
calculate $net\_id$ using $blockId$ and $threadId$;
if $net\_id$ < $total$ #nets for all problems then
    for $object \in net_{net\_id}$ do
        find the $problem$ to which $object$ belongs;
        find the $set$ to which $object$ belongs;
        update first and last objects for $net_{net\_id}$ in $set$;
        update external bounding box of $net_{net\_id}$ using x/y location of $object$;
    end
end
```

This kernel traverses a filled cost matrix from the last entry to the first and finds the set from which the best solution for each site has been taken (Figure 3.23). Each DP problem maps to a block. Each block has to iterate over N matrix entries only, so 1 thread per block is a reasonable choice.

### 3.2.4 Complexity analysis

As stated in the previous section, filling the cost matrix (which is the main bottleneck) involves $\Theta(p.k.M.(\frac{N}{k}+1)^k)$ operations, where $p$ is the average number of pins per net and M is the number of problems.

#### 3.2.4.1 Independent sub-flow:

Assume that we have T threads. The total amount of work for filling the cost matrix is divided among those T threads roughly evenly (there is an upper bound on work per thread which varies linearly with k). However, waiting for threads to synchronize within blocks leads to some under-utilization of threads. For our implementation with k=3, this contributes a constant factor

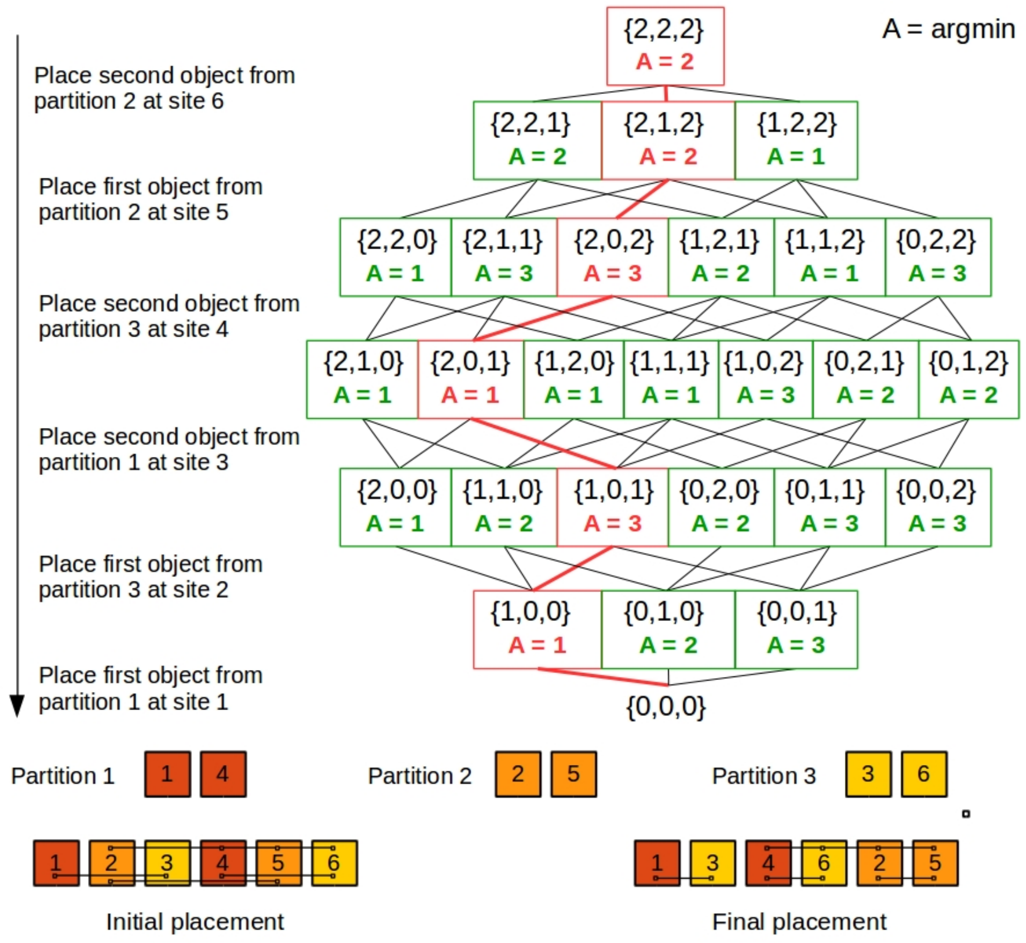


Figure 3.23: Constructing the placement solution after the cost matrix is filled.

to the runtime. We can roughly calculate this factor by extension from discrete to continuous domain. The number of matrix entries to be filled can be thought of as the volume ($\frac{N^3}{27}$) of a cube with side $\frac{N}{3}$. In our GPU implementation, the number of threads that we require per block is proportional to the maximum cross section area of the cube perpendicular to the main diagonal, which corresponds to a hexagon of side $\frac{N}{3\sqrt{2}}$. This hexagon is swept over the diagonal of the cube whose length is $\frac{N}{\sqrt{3}}$. This leads to a total volume of $\frac{N^3}{12}$. The constant factor in our runtime is the ratio of the two volumes which is $\frac{27}{12} = 2.25$. Hence, $runtime = p \times 3 \times max(\frac{2.25 \times M.(\frac{N}{3}+1)^3}{T}, N)$. There is a lower bound on the parallel runtime because in the best case we have to fill N hyperplanes one after another but each of them would take constant time.

#### 3.2.4.2 Combined sub-flow:

In this case, all the blocks are fully utilized except the last block in each hyperplane. The number of blocks grows linearly with problem size as the maximum number of threads per block is a constant. The effect of the last block can be accounted for by pessimistically adding one extra block for each hyperplane which adds up to a constant multiple of N. $runtime = p \times 3 \times max(\frac{M.(\frac{N}{3}+1)^3+c.N}{T}, N)$, where c is a small constant.

### 3.2.5 Results

We tested our algorithm on the ISPD 2016 FPGA Placement Contest [71] benchmarks. We used the packing, global placement and legalization flow of UTPlaceF [21], which is in improved version of the placer that secured the first place in the contest. Table 3.9 shows some benchmark statistics (post packing).

The output of our implementation of the DP algorithm is same for CPU, CPU+GPU and GPU flows. We ran our experiments on a machine with an Intel® i9-7900X CPU @ 3.3GHz, 10 cores and 20 threads and an NVIDIA® TITAN Xp GPU. Table 3.10 shows the runtimes for multi-threaded CPU, hybrid (CPU+GPU) and GPU flows. The CPU part in our hybrid implementation uses 20 threads. The runtimes are reported for one iteration of DP over the whole chip. We observe that the GPU flow provides speedups in the range of 3-5x over 20-threaded CPU implementation for large designs.

Table 3.9: Benchmark Statistics

| Design | #cells | #nets |
|---|---|---|
| FPGA01 | 8315 | 44933 |
| FPGA02 | 13504 | 67118 |
| FPGA03 | 42523 | 180499 |
| FPGA04 | 43041 | 209852 |
| FPGA05 | 43038 | 244844 |
| FPGA06 | 65626 | 310637 |
| FPGA07 | 65530 | 360837 |
| FPGA08 | 64164 | 319900 |
| FPGA09 | 67023 | 389237 |
| FPGA10 | 66129 | 449126 |
| FPGA11 | 66489 | 372202 |
| FPGA12 | 68183 | 405376 |

Table 3.12 shows the runtime breakdown for various tasks. As expected, a major portion of the runtime is spent in filling the cost matrix. Table 3.13

Table 3.10: Runtimes in milliseconds for ISPD 2016 contest benchmarks

| Design | Multi-threaded CPU; T=threads | | | | | | | | | | | Hybrid | | GPU | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1T | 2T | 4T | 6T | 8T | 10T | 12T | 14T | 16T | 18T | 20T | indep. | comb. | indep. | comb. |
| FPGA01 | 2467 | 2120 | 1382 | 1375 | 1139 | 938 | 854 | 775 | 722 | 693 | 625 | 394 | 424 | 276 | 324 |
| FPGA02 | 3498 | 3152 | 1903 | 1932 | 1637 | 1208 | 1213 | 1103 | 990 | 939 | 816 | 437 | 466 | 282 | 327 |
| FPGA03 | 10776 | 8039 | 4462 | 3160 | 2443 | 1995 | 1676 | 1567 | 1459 | 1505 | 1471 | 727 | 699 | 395 | 385 |
| FPGA04 | 11858 | 7892 | 5097 | 3521 | 2779 | 2317 | 1936 | 1716 | 1765 | 1762 | 1671 | 766 | 774 | 431 | 422 |
| FPGA05 | 14366 | 10837 | 5839 | 4101 | 3176 | 2588 | 2235 | 2019 | 2055 | 2070 | 1978 | 855 | 879 | 463 | 458 |
| FPGA06 | 17714 | 9673 | 5134 | 3659 | 2760 | 2334 | 2602 | 2335 | 2239 | 2091 | 1941 | 959 | 936 | 499 | 501 |
| FPGA07 | 20537 | 11129 | 6167 | 4304 | 3350 | 2703 | 2961 | 2726 | 2555 | 2427 | 2229 | 1031 | 1041 | 545 | 558 |
| FPGA08 | 18795 | 10981 | 5806 | 4010 | 3092 | 2489 | 2740 | 2557 | 2389 | 2198 | 2059 | 915 | 910 | 484 | 497 |
| FPGA09 | 22489 | 13186 | 7010 | 4862 | 3787 | 3151 | 3388 | 3119 | 2924 | 2762 | 2554 | 1107 | 1158 | 580 | 629 |
| FPGA10 | 19035 | 10041 | 6271 | 4207 | 3434 | 2773 | 2635 | 2683 | 2564 | 2440 | 2287 | 1028 | 1024 | 560 | 567 |
| FPGA11 | 20418 | 10932 | 6015 | 4228 | 3231 | 2660 | 3016 | 2771 | 2625 | 2487 | 2270 | 977 | 1011 | 555 | 581 |
| FPGA12 | 18571 | 9437 | 5296 | 3703 | 2928 | 2450 | 2585 | 2377 | 2292 | 2134 | 1935 | 981 | 958 | 540 | 556 |

Table 3.11: Runtimes in milliseconds spent on computation for ISPD 2016 contest benchmarks

| Design | Multi-threaded CPU; T=threads | | | | | | | | | | | Hybrid | | GPU | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1T | 2T | 4T | 6T | 8T | 10T | 12T | 14T | 16T | 18T | 20T | indep. | comb. | indep. | comb. |
| FPGA01 | 2448 | 2100 | 1363 | 1355 | 1119 | 918 | 834 | 755 | 702 | 674 | 606 | 229 | 251 | 111 | 163 |
| FPGA02 | 3479 | 3133 | 1884 | 1912 | 1618 | 1118 | 1193 | 1083 | 970 | 919 | 797 | 269 | 290 | 114 | 165 |
| FPGA03 | 10757 | 8020 | 4443 | 3140 | 2423 | 1976 | 1657 | 1547 | 1439 | 1485 | 1452 | 555 | 519 | 215 | 204 |
| FPGA04 | 11839 | 7872 | 5077 | 3502 | 2760 | 2298 | 1917 | 1697 | 1745 | 1743 | 1651 | 600 | 592 | 246 | 242 |
| FPGA05 | 14346 | 10818 | 5820 | 4081 | 3157 | 2568 | 2215 | 2000 | 2035 | 2050 | 1958 | 688 | 703 | 278 | 275 |
| FPGA06 | 17694 | 9653 | 5115 | 3640 | 2741 | 2314 | 2583 | 2316 | 2219 | 2072 | 1921 | 783 | 761 | 307 | 310 |
| FPGA07 | 20518 | 11109 | 6148 | 4285 | 3330 | 2683 | 2941 | 2707 | 2536 | 2408 | 2210 | 864 | 864 | 348 | 362 |
| FPGA08 | 18776 | 10962 | 5787 | 3990 | 3073 | 2470 | 2721 | 2538 | 2370 | 2179 | 2040 | 746 | 729 | 291 | 302 |
| FPGA09 | 22471 | 13167 | 6991 | 4843 | 3767 | 3132 | 3369 | 3100 | 2905 | 2743 | 2535 | 935 | 979 | 380 | 428 |
| FPGA10 | 19016 | 10022 | 6252 | 4188 | 3415 | 2754 | 2616 | 2663 | 2545 | 2421 | 2268 | 857 | 846 | 361 | 367 |
| FPGA11 | 20399 | 10913 | 5996 | 4209 | 3211 | 2641 | 2997 | 2752 | 2607 | 2469 | 2251 | 804 | 831 | 356 | 382 |
| FPGA12 | 18554 | 9419 | 5279 | 3686 | 2910 | 2433 | 2568 | 2360 | 2274 | 2116 | 1919 | 743 | 776 | 338 | 354 |

Table 3.12: Runtime breakdown for various tasks for full GPU flow

| Task | %Runtime | |
|---|---|---|
| | indep. | comb. |
| Initialize variables | 2.63 | 2.39 |
| Get objects | 0.13 | 0.12 |
| Get nets | 1.39 | 1.26 |
| Process nets | 4.48 | 4.07 |
| Fill cost matrix | 91.26 | 92.07 |
| Update locations | 0.07 | 0.06 |

shows variation of runtimes for various kernels with respect to blockDim. The runtimes are reported for one set of DP problems (∼126). The kernel 'Initialize Nets' takes exceptionally high runtime for blockDim = 1024. We suspect that this is due to high register demand for the kernel as it computes many variables.

Table 3.13: Runtimes of Kernels in milliseconds vs blockDim

| blockDim | Initialize Nets | Fill Cost Matrix (comb.) |
|---|---|---|
| 1 | 19.0 | 126.1 |
| 2 | 15.0 | 150.3 |
| 4 | 15.2 | 98.6 |
| 8 | 15.5 | 70.7 |
| 16 | 15.2 | 53.0 |
| 32 | 14.5 | 41.3 |
| 64 | 14.5 | 40.3 |
| 128 | 14.5 | 39.8 |
| 256 | 14.5 | 38.6 |
| 512 | 14.7 | 41.1 |
| 1024 | 53.5 | 42.6 |

We see that the runtimes for the combined sub-flow are slightly higher than those for the independent sub-flow. This confirms our hypothesis that utilizing all threads in each block creates other problems like irregular memory access, unavailability of registers, etc. which offset the benefits of parallelization. The fact that reducing blockDim from 1024 to 256 reduces the time taken to fill the cost matrix further supports our conclusion.

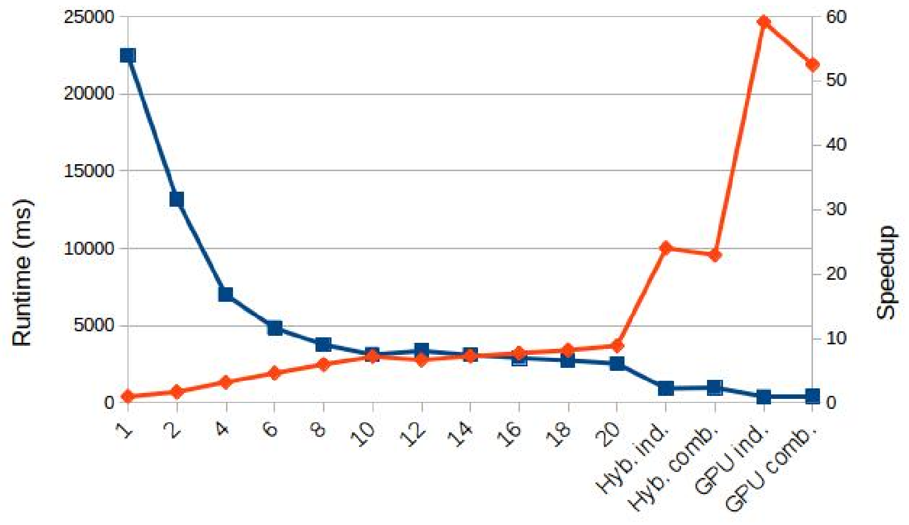


Figure 3.24: Runtime for computation (blue) in milliseconds on left axis and speedup (red) over single-threaded CPU implementation on right axis for the design FPGA09 for multi-threaded CPU, hybrid and GPU implementations

Since there is significant runtime penalty for transferring data between CPU and GPU, it is reasonable to move major parts of the tool flow like global placement (as demonstrated in [11]), packing and legalization to GPU. In such a case, the memory allocation overhead can be hidden by previous operations So, we report runtimes without the data transfer part for GPU in Table 3.11. We have also excluded memory allocation times for the CPU part. We see that we can get up to 7.01x speedup for large benchmarks like FPGA08.

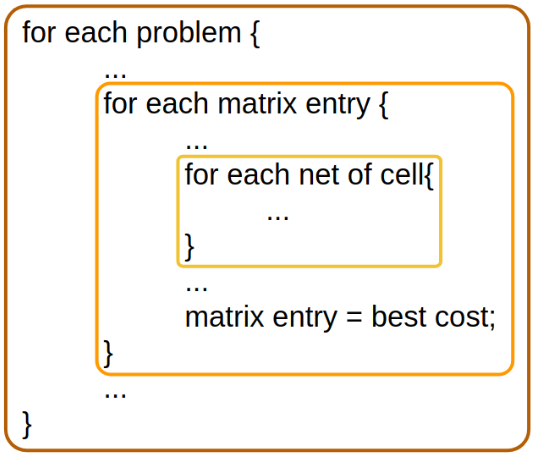
Note that in both Tables 3.10 and 3.11, data transfer from CPU to GPU happens only once in the GPU flow and the data is reported for one iteration. If we increase the number of iterations (for example, to 16, as in [41]), the ratios between Table 3.10 and Table 3.11 values would asymptotically converge to 1. Hence, speedups of ~6x seem plausible.

## 3.3 FPGA Acceleration of Dynamic Programming

Detailed placement takes a significant portion of the overall placement runtime, even when it is parallelized and run on multi-threaded CPUs. Hence it is desirable to accelerate detailed placement on hardware. Section 3.2 describes GPU acceleration of the dynamic-programming-based detailed placement algorithm in section 3.1. In this section, we describe FPGA acceleration of the same dynamic programming algorithm. This is an attractive option to customers who already have FPGAs because it would not be necessary to buy additional devices like GPUs just to help program the FPGAs.

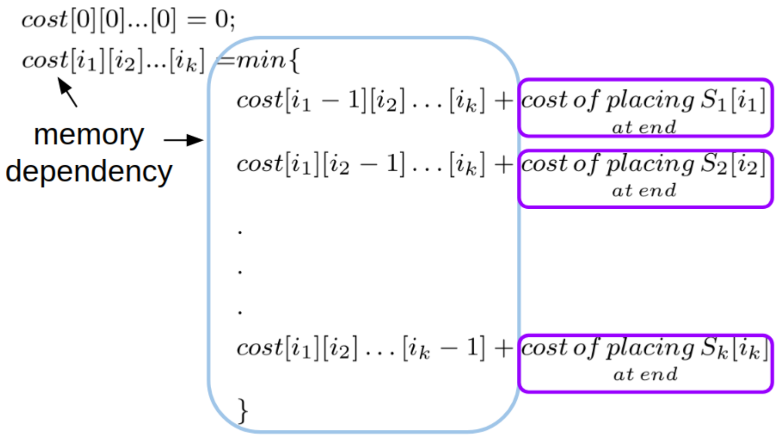
We briefly summarize the tasks involved in our dynamic-programming-based detailed placement algorithm:

- Initialize DP problems. This step involves finding movable objects in each window, assigning sets to those objects, collecting nets connected to those objects and finding endpoints of all such nets.
- Compute matrix entries. This step involves computing solutions to subproblems and saving the best solution for each matrix entry.
- Trace best solution through the matrix and update cell locations.



```
for each problem {
        ...
        for each matrix entry {
                ...
                for each net of cell{
                        ...
                }
                ...
                matrix entry = best cost;
        }
        ...
}
```

Figure 3.25: Various levels of parallelization within the dynamic programming algorithm. Each level is highlighted by a rectangle.



$$cost[0][0]...[0] = 0;$$

$$cost[i_1][i_2]...[i_k] = min\{$$
$$cost[i_1-1][i_2]\ldots[i_k] + \boxed{cost\,of\,placing\,S_1[i_1]\,\underset{at\,end}{}}$$
$$cost[i_1][i_2-1]\ldots[i_k] + \boxed{cost\,of\,placing\,S_2[i_2]\,\underset{at\,end}{}}$$
$$\vdots$$
$$cost[i_1][i_2]\ldots[i_k-1] + \boxed{cost\,of\,placing\,S_k[i_k]\,\underset{at\,end}{}}$$
$$\}$$

Figure 3.26: Dynamic programming recurrence and memory dependency.

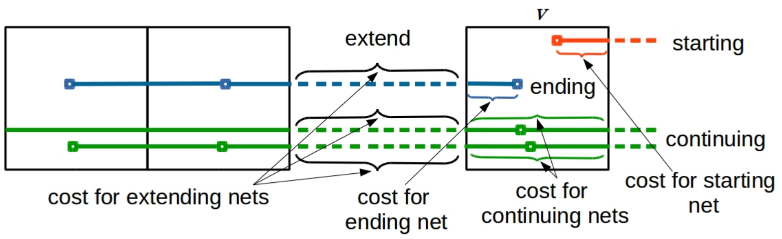
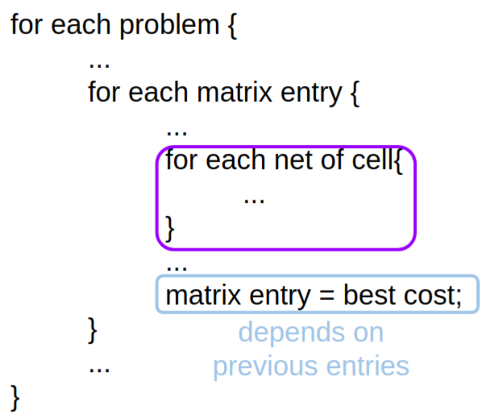
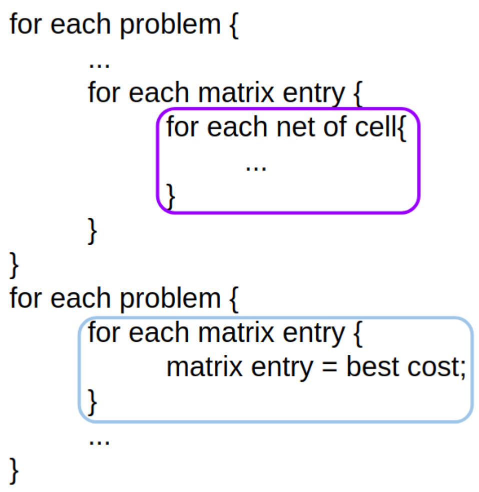


Figure 3.27: Various types of nets encountered while solving a subproblem



```
for each problem {
        ...
        for each matrix entry {
                ...
                for each net of cell{
                        ...
                }
                ...
                matrix entry = best cost;
        }
        ...
}
```

Figure 3.28: Computations involved in filling a matrix entry highlighted by rectangles.



```
for each problem {
        ...
        for each matrix entry {
                for each net of cell{
                        ...
                }
        }
}
for each problem {
        for each matrix entry {
                matrix entry = best cost;
        }
        ...
}
```

Figure 3.29: Separating out computations involved in filling a matrix entry.

Filling the matrix entry takes majority of the runtime, as discussed in the previous section. Each matrix entry represents the best solution to a subproblem in which a certain number of the cells are placed and the rest are not yet placed. For computing the cost while placing a cell $v$ at the $i^{th}$ site, we encounter 3 types of nets, as shown in Figure 3.27:

1. Nets which start at $v$ (i.e, no cell of the net has been considered yet)
2. Nets which end at $v$ (i.e, remaining cells for the net have already been considered)
3. Continuing nets. These may or may not be connected to $v$.

For each net, we know the vertices connected to it and their position in their respective sets. From the subproblem index $(i_1, i_2, \ldots, i_k)$, we can check if lowest index of any vertex connected to the net is greater than the $i's$ $(i_1, i_2, \ldots, i_k)$ for the corresponding set. This tells us whether the net starts at $v$. Similarly, we can find the nets ending at v. For continuing nets, we don't really need to track them individually. We just store the sum of the weights of the continuing nets. While calculating the cost of placing $v$ at site $i_1+i_2+\cdots+i_k-1$, we first extend the continuing nets from the previous site to the current site(take the distance between the sites and multiply by sum of weights of continuing nets). Next, we add the costs for the starting and ending nets(nets may start/end at different points within the unit square). We also add the cost of the intermediate continuing nets (nets which started before and did not end at $v$). We subtract the ending nets' weights from the previous sum of continuing weights. To this, we add the weights of starting nets to get the new continuing nets' weight's sum.

Since FPGAs and GPUs have different strengths and weaknesses, the FPGA implementation of dynamic programming is very different from the corresponding GPU implementation. There are three main levels of parallelization in our DP algorithm:

- Solving different problems in parallel (brown rectangle in Figure 3.25).
- Computing different matrix entries in parallel (orange rectangle in Figure 3.25).
- Parallelizing iteration over nets for each cell (yellow rectangle in Figure 3.25).

Solving an entire DP problem on FPGA requires that all the data associated with that problem to be present on the on-chip RAMs (M20ks or MLABs in Intel FPGAs). Random accesses to the main memory is slow and would otherwise become the runtime bottleneck. The device that we have access to has limited on-chip memory, hence solving an entire problem on the FPGA is infeasible. We can fill multiple matrix entries in parallel as we did for GPU acceleration in Section 3.2. However, the matrix is large (24389 entries for a window of size 84). Multiple concurrent read and write accesses lead to replication of the memory by the compiler which leads to high block

RAM usage. Hence, parallelizing iteration over nets is the best option for FPGA acceleration. This would require replication of the nets array but that is smaller compared to the matrix for a problem.

### 3.3.1 Hybrid CPU-FPGA Implementation

Our flow initializes the DP problems on CPU, fills the matrix entries using both CPU and FPGA and updates the placement using CPU. We accelerate parts of the matrix entry computation on FPGA. Computing a matrix entry can be decomposed into two tasks, as highlighted by the purple and blue rectangles in figures 3.28 and 3.26. The purple rectangle denotes iteration over all the nets connected to a cell and determining if a net starts, ends or continues. This step also computes the sum of weights of the continuing nets and the incremental cost for placing the cell under consideration at the current site. This step is independent of other matrix entries. The blue rectangle denotes iteration over possible solutions the subproblem (at most $k$) corresponding to the current matrix entry and selecting the best solution. This step depends on other matrix entries, as shown in Figure 3.26.

We separate the computation of each matrix entry into two different loops as shown in Figure 3.29. The first loop executes on FPGA and the second one executes on CPU. The memory dependency on the second loop causes it to be launched every 13 cycles if implemented on the FPGA, hence we chose to run it on the CPU.

For our FPGA implementation, we limit the maximum number of nets per cell to 64. If a cell in a DP problem has more than 64 nets, we can either solve that problem on the CPU or run the kernel multiple times on the FPGA and pass a different set of nets each time. We also insert dummy nets if the actual number of nets connected to a cell is less than 64.

We use 8 and 16-bit integers wherever possible to reduce memory usage. For example, the starting and ending cells for a net for each set can be represented by 8 bits as our window size is 84. The external endpoints (no connected to any cell in the window) can be represented by 16-bit integers as our floorplan measures 168x480 sites.

We fully unroll the loop highlighted by the blue rectangle in Figure 3.29. We choose $k = 3$, as that gives the best tradeoff for QoR vs. runtime. Incremental costs for 3 possible subproblems are calculated at each clock cycle. Thus 64x3=192 entries are read from the nets array at each clock cycle. This implementation allows us to process one matrix entry at each clock cycle.

### 3.3.2 Results

We tested our algorithm on the ISPD 2016 FPGA Placement Contest [71] benchmarks. We used the packing, global placement and legalization flow of UTPlaceF [21], which is in improved version of the placer that secured the first place in the contest. We implemented a row based dynamic programming algorithm as a proof of concept.

We ran our experiments on a machine with a 14 core, 28 thread Intel® Xeon® processor and an Intel® Arria10® FPGA on the same package. This setup allows allocation of shared memory and low latency communication between the CPU and the FPGA. All our reported runtimes include the time for moving data to and from the FPGA.

Table 3.14: Runtimes in milliseconds and Speedup

| Design | CPU | CPU+FPGA | speedup |
|---|---|---|---|
| FPGA01 | 3475 | 1142 | 3.04 |
| FPGA02 | 4256 | 1592 | 2.67 |
| FPGA03 | 6105 | 3304 | 1.85 |
| FPGA04 | 7142 | 3465 | 2.06 |
| FPGA05 | 9689 | 4378 | 2.21 |
| FPGA06 | 8500 | 5001 | 1.70 |
| FPGA07 | 9723 | 5227 | 1.86 |
| FPGA08 | 8157 | 4522 | 1.80 |
| FPGA09 | 11536 | 5391 | 2.14 |
| FPGA10 | 9060 | 5861 | 1.55 |
| FPGA11 | 9986 | 5258 | 1.90 |
| FPGA12 | 9628 | 4898 | 1.97 |
| Geomean | | | 2.03 |

Table 3.15: FPGA Resource Usage and Fmax for DP kernel

| Logic | Register | RAM | DSP | Fmax |
|---|---|---|---|---|
| 51% | 47% | 66% | <1% | 225 MHz |

Table 3.14 shows the runtimes and speedups for individual designs. We achieve a speedup of 2.03x on average using our hybrid CPU-FPGA implementation vs. the CPU implementation. Portions of our that run on CPU code are multi-threaded and vectorized wherever possible. We use all 28 threads on the CPU. Also, in the CPU-FPGA implementation, only one device (either the CPU or the FPGA) is actively involved in computations at any given time.

We can achieve even more speedup by using both devices simultaneously.

Table 3.15 shows the resource usage and Fmax for our DP kernel. We are limited by the amount of available RAMs. We can implement multiple kernels on an FPGA with more RAMs. We use very few DSPs as most of our operations are on integers and those are implemented using carry chains instead of DSPs.

## 3.4 Timing-Driven Detailed Placement

The high flexibility of FPGAs comes at the cost of performance. Designs typically run much slower on FPGAs compared to ASICs. Improving the maximum frequency (Fmax) for a design on an FPGA can have a profound impact in certain markets. Detailed placers can provide significant improvements in timing as they are able to model delays more accurately compared to global placers. Timing-driven detailed placement for ASICs is relatively mature. There are a few important differences for timing-driven detailed placement between FPGAs and ASICs:

- LABs in FPGAs have many more pins ($\sim$50) compared to standard cells in ASICs (2-5)
- LABs in FPGAs have multiple output pins, hence can be start-points of multiple timing paths whereas most standard cells have one output, and generally have fewer number of different output paths

- Routing resources in an FPGA are fixed. Hence, wirelength and delay estimation for a net cannot be done by simple steiner routes but have to take routing resources in the underlying FPGA target device into account

Timing-driven detailed placement has two aspects - (i) the objective function or 'metric' that we are directly trying to optimize (ii) how we explore our solution space. The objective function can be loosely classified as net-based [44, 46–48, 59], path-based [49, 52, 56, 57] or a hybrid of the two [45, 51]. The general theme of net-based objective functions is to run timing analysis, generate slacks and criticalities for nets and use those values to generate net weights (more critical nets get higher weights). Then, placement is performed to minimize weighted wirelength. They do not optimize critical paths explicitly. In a linear weighted model, nets with higher weights dominate nets with lower weights. This necessitates the use of constraints on length or delay, or slack for nets ( [61], [62]). Some algorithms count the number of critical paths passing through a net and/or the number of end-points affected by a net, and use this information for generating weights [46]. Generally speaking, net-based approaches work well in a global perspective. They tend to converge when the placement is close enough to optimal from the global perspective. While these approaches optimize for total negative slack, they leave significant room for improvement as they do not optimize the most critical paths, and may create new critical paths while trying to reduce delays of other nets.

Path-based optimization algorithms try to model exact delays for the most critical paths and optimize them. Many of them use linear programming or lagrangian relaxation formulations. Some approaches use simulated annealing. Linear programs scale poorly, especially for FPGAs where LABs can have $\sim$50 pins and moving one LAB can affect a large number of paths. Simulated annealing also has scalability problems and it cannot maintain the same solution quality with similar runtime for increasingly larger modern designs.

A variety of ideas have been proposed for solution space exploration or the actual 'placement'. The most common ones are greedy swaps or moves or shifting of cells [44, 49]. Some works extend the greedy approaches to tunnel through barriers or use hill-climbing moves like simulated annealing [46, 48, 51]. Many of the techniques prevalent in popular literature concentrate on minimizing their objective function first to generate a placement that can have possible overlaps and legalize afterwards [45, 52]. Some approaches which use linear or integer programming also incorporate the legalization into the LP or IP. A discrete optimization technique based on choosing candidate locations is proposed in [57] but the authors try to address all affected critical paths together which is infeasible for FPGAs. Also, they choose disjoint sets of candidate locations for different nodes on a critical path, which restricts the solution space.

Below are the limitations and areas of improvement for state-of-the-art timing-driven detailed placement techniques for FPGAs:

- Traditional net-based timing optimization tends to saturate at some distance from the global optimum. Further, they tend to oscillate. The output of net-based detailed placement has a large scope for improvement.

- Linear programming (LP)-based critical path optimizations are not suitable for FPGAs since LABs in FPGAs have a large number of pins and moving one LAB affects many paths leading to a large number of constraints for LP.
- The discrete optimization of critical paths in [57] attempts to minimize the maximum delay of all the critical paths incident on a set of nodes. This is infeasible for FPGAs due to the large number of paths per node (LAB)
- Critical path optimization techniques which move one path node at a time are highly susceptible to getting stuck in local minima. Therefore, we need to optimize all the critical path nodes concurrently.

To alleviate these limitiations, we propose a new timing-driven detailed placement algorithm for FPGAs. The main contributions of our work [42] are as follows:

- Our new timing-driven placement algorithm is tailored towards high connectivity netlists like those for FPGAs
- Our algorithm to optimize critical paths where the path nodes are allowed to move to a set of candidate locations which may overlap with

candidate locations of other path nodes. This gives more freedom for movement than [57], while ensuring that the final solution is overlap-free.

- We formulate our optimization problem as a shortest path problem on a layered network of candidate locations for each path node
- We use hard delay limits for nets which prevents degradation in the worst slack. This is an effective way of controlling side paths rather than minimizing the maximum delay for a set of paths.
- Our formulation enables us to use dynamic programming to solve for the shortest path, which is faster than the branch-and-bound algorithm in [57]
- Timing improvements from our algorithm are complementary to those achieved using conventional net-based detailed placement algorithms, thus augmenting their capabilities.
- Our algorithm has negligible effect on wirelength and congestion and has a small runtime overhead

### 3.4.1 Problem Formulation for Timing-Driven Detailed Placement

#### 3.4.1.1 Timing Model

We introduce virtual 2-pin nets called tnets for each source-sink pair in each net. Tnets represent timing arcs. They capture routing information of the corresponding net segments and hence provide accurate information for timing calculation. Delay between any two locations on the FPGA grid is modeled in a lookup-table fashion for fast access. The lookup tables are sufficiently small as the regular routing architecture in FPGAs leads to uniform delays. This delay depends on current cell placement and can be easily modified for incremental changes. We expect the routing information and congestion maps to be practically undisturbed during the course of our algorithm as we would be moving a very small fraction of the cells (and therefore, nets).

#### 3.4.1.2 Setting up the Optimization Problem for a Critical Path

Let's consider the example shown in Figure 3.30. It shows a portion of the FPGA grid with different types of sites. In this grid, A-B-C-D-E is a critical path that we expect to optimize. We pick some candidate locations for each of the nodes A, B, C, D, E that are in close proximity to the path (shown in Figure 3.31). For example B can move to B1, B2, etc. and C can move to C1, etc. B and C can also move to BC1, BC2, etc. with the constraint that both of them should not end up in the same location. Legality is also taken into account while choosing candidate locations. The set of these candidate locations is called 'neighborhood' of the path. (Details on how the neighborhood is selected is discussed later). The set of candidate locations for a single path node is called a 'sub-neighborhood'. Candidate locations for two consecutive path nodes may overlap (ex: B and C can go to BC1, BC2, etc and D and E can go to DE1, DE2, etc.) but candidate locations for two nodes that are not adjacent in our chosen path may not overlap (ex: AC, AD, BD etc. are not allowed). We stress the importance of our 'chosen' path. There could be another net (which may branch into or out of the current path) from A to C making A and C adjacent, but we only have the edges A-B, B-C, C-D, D-E in our chosen path. We will discuss how we tackle side paths like A-C shortly. We ensure that original locations of the path nodes are also in the candidate location set.

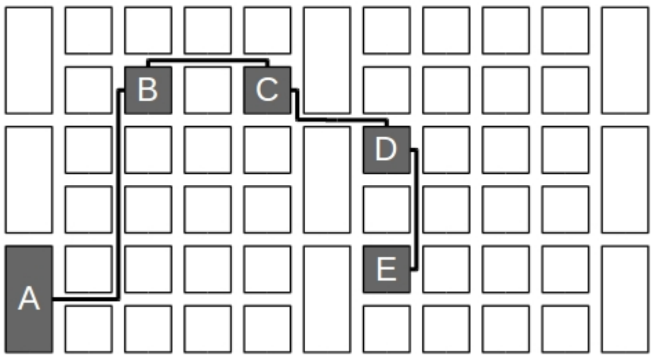


Figure 3.30: FPGA grid with a critical path

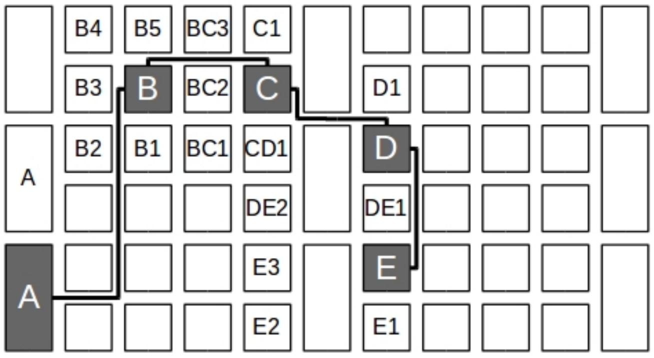


Figure 3.31: Neighborhood chosen around a critical path

Candidate locations for path nodes can be empty or occupied by some other object (LAB, RAM, DSP, etc.). If a candidate location is empty, we may allow the corresponding path node to move there. If they are occupied by some other object, we may swap the object with the corresponding path node. For example, in Figure 3.32, assume that B1 is an empty site and B4 is occupied. In this case, B could move to B1 or B4. If B moves to B4, the cell that is currently at B4 must move to B's original site.

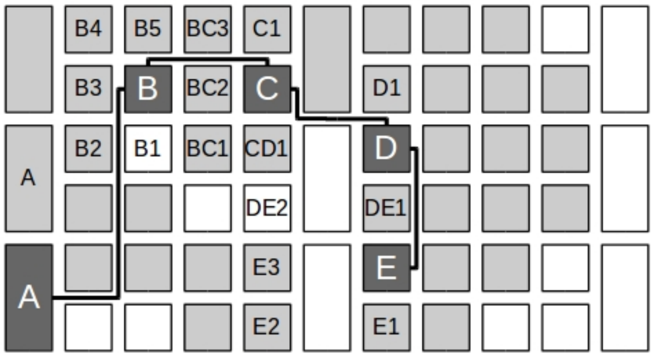


Figure 3.32: Placement of other cells in the neighborhood

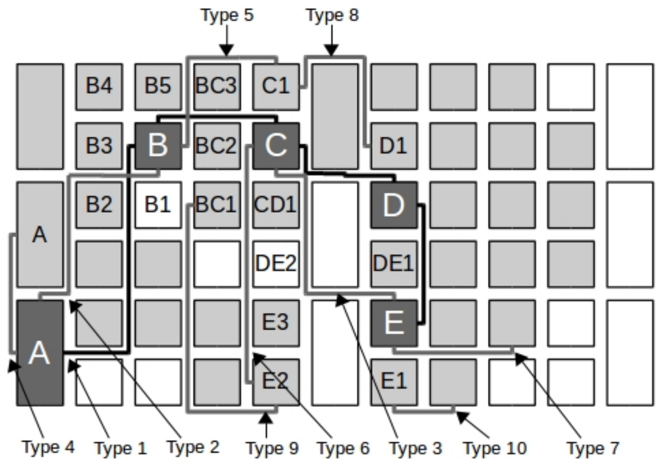


Figure 3.33: Classification of tnets

#### 3.4.1.3 Classification of Tnets

We now consider the set of all tnets connected to the critical path nodes and the neighborhood nodes. They can be classified into the following 10 types (illustrated in Figure 3.33):

- **Type 1:** tnets in the critical path (one path node to the next or previous node)
- **Type 2:** tnets between consecutive path nodes that are not in the current critical path
- **Type 3:** tnets from one path node to another path node at distance 2 or more in the critical path
- **Type 4:** tnets from a path node to its neighbor
- **Type 5:** tnets from one path node to the neighbors assigned to the next or previous path node
- **Type 6:** tnets from one path node to neighbors assigned to path nodes at distance 2 or more in the critical path
- **Type 7:** tnets from a path node to outside the neighborhood
- **Type 8:** tnets between neighbors assigned to consecutive path nodes
- **Type 9:** tnets between neighbors assigned to path nodes at a distance 2 or more apart in the critical path

- **Type 10:** tnets from a neighborhood node to a node outside the neighborhood

When a neighbor is assigned to 2 path nodes like BC1, DE1, etc. the types of some tnets may vary depending on the context. For example, when we are finding new locations of tnet pins by swapping BC1 with B, we will treat BC1 as B's neighbor and not C's neighbor. Similarly, when we consider swapping BC1 with C, we will treat BC1 as C's neighbor and not B's neighbor.

#### 3.4.1.4 Shortest Path Problem

Our objective is to achieve minimum delay for the path A-B-C-D-E while ensuring that other paths do not become more critical than the one which is currently most critical. To achieve this, we formulate a shortest path problem with certain constraints on tnet delays. The maximum delay that can be allowed on a tnet is denoted by $delay\_limit_{tnet}$. These delay limits are calculated by a slack allocation algorithm right after each timing analysis (discussed later).

Let there be N nodes on the critical path. This implies there are N-1 tnets on the critical path. Each path node has a choice of some candidate locations. We construct a graph as follows: The graph has N layers, one for each node in the critical path. Each layer has nodes corresponding to the candidate locations for that path node. For example, in Figure 3.34, the layer for B has nodes B1 to B5 and BC1 to BC3. We add an edge for each feasible pair of locations of adjacent nodes in the critical path. For example, two adjacent nodes, B and C have a feasible pair of locations B5, C1 if B can move to B5 and C can move to C1. The edge represents the delay between B and C after the movement. Also, observe that all BCs in B's layer have outgoing edges to all Cs, BCs and CDs in C's layer except the corresponding BC. This exclusion is necessary to prevent nodes from overlapping. BC2 in B's layer does not have an edge to BC2 in C's layer as that could potentially lead us to choose both BC2s implying that B and C both go to site BC2. The edges essentially model the delays of the type 1 tnets defined above. For example, the edge from B1 to C1 in the graph represents the delay of the tnet B-C when B is moved to B1 and C is moved to C1.

We want to find locations for the path nodes such that the delay of the critical path (which is the sum of the delays represented by these edges) corresponding to the node locations is minimized.

When we move or swap nodes, the delays of tnets connected to the nodes being moved will change. These tnets can be classified into the following types:

- Case (i): delay independent of any other move or swap
- Case (ii): delay dependent on move or swap of adjacent path node
- Case (iii): delay dependent on move or swap of a path node at a distance of 2 or more in the critical path

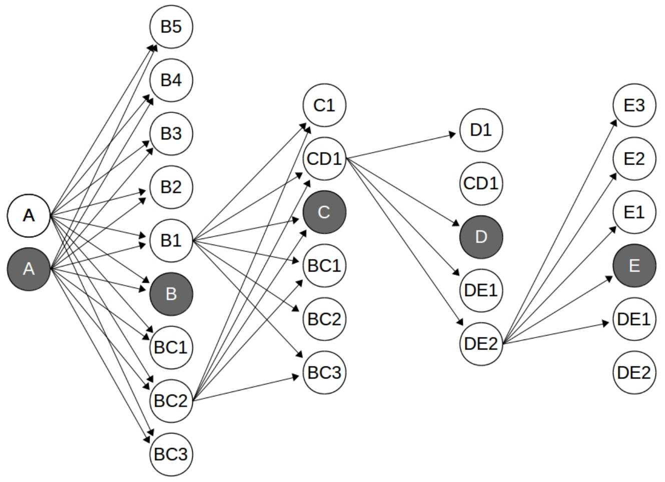


Figure 3.34: Shortest path problem; All outgoing edges for only some of the nodes are shown. Note that BC2 in B's layer does not have an edge to BC2 in C's layer. This is necessary to prevent overlaps. Similar case with CD1 and DE2

Case (i) consists of tnet types 4, 7 and 10. Case (ii) consists of tnet types 1, 2, 5 and 8. Case (iii) consists of types 3, 6 and 9.

As stated earlier, each tnet has a delay limit. Some placements in the chosen candidate locations may violate the delay limits of some tnet connected to the nodes being moved. If such a case occurs, we remove that candidate location from our graph.

Tnet delays in case (i) can be computed for each candidate location with the current placement information of the other nodes in the netlist. If we find a candidate location that violates the delay limit of some tnet, we remove that location from our graph. Tnet delays for case (ii) are computed by considering pairs of location assignments for consecutive path nodes. If any pair of location assignments causes a tnet delay limit violation, we remove the corresponding edge from the graph. For case (iii), we compute net delays based on the current placement of nodes and we update the delays when we reach the corresponding path node downstream while finding the shortest path. We remove the edge to the corresponding node from the graph if there is a delay limit violation.

### 3.4.2 Components of our Timing-Driven Detailed Placement Algorithm

#### 3.4.2.1 Selecting a Critical Path

We store the delay and slack values obtained from timing analysis in the tnets. For each tnet, we compute a parameter called *criticality* ($\in [0,1]$), according to [48]:

$$criticality_{tnet} = 1 - \frac{slack_{tnet} - worst\_slack}{D_{max}} \tag{3.8}$$

Where $D_{max}$ is the critical path delay (maximum of arrival times of all sinks for the corresponding clock) and $slack_{tnet}$ is the difference between the required and arrival times of the tnet's load pin.

We pick all the nets with criticality greater than a certain threshold $c$. We have empirically determined the best value of $c$ to be 0.98. We extract critical paths from these selected tnets based on connectivity information from the netlist. Note that a tnet can belong to more than one critical path.

Critical paths are extracted by the following algorithm: Initialize a critical path consisting of only one tnet. The path is grown by successively adding tnets to the front and back of our current critical path. For the starting node of the critical path, we go through all the tnets that drive the tnet connected to this node and find the one with the highest criticality (this criticality value will be same as the criticality of all the tnets in the current critical path) and add that tnet to critical path. Ties in criticality value are broken arbitrarily, but such cases are highly unlikely. For the ending node of the critical path, we similarly go through all the tnets that are driven by the tnet connected to this node and find the one with the highest criticality and add it to the critical path. Propagation stops when we reach timing start/end points.

The *criticality* metric normalizes the slack of a tnet to the longest path delay for the corresponding clock. This allows us to distinguish between similar slack tnets, weighting ones with a higher longest path delay to be more critical.

#### 3.4.2.2 Slack Allocation

The simplest way of allocating slack while preserving the worst slack is to assign the minimum possible marginal delay increase for each tnet. We get slack values for each tnet from timing analysis. Assuming there are no combinational cycles in the logic, we can count the number of distinct timing paths passing through each tnet. These are paths with respect to different timing end points. We also compute the length of the longest timing path (number of tnets on that path) passing through each tnet by forward propagation. This can be done only once as the netlist is not being changed. Now, we can set delay_limit for a tnet as follows (extending the concepts from [46] and [62]):

$$delay\_limit_{tnet} = delay_{tnet} + \frac{slack_{tnet} - worst\_slack}{longest\_path\_length_{tnet}} \tag{3.9}$$

This slack allocation scheme ensures that even if all tnets increase in delay to be at their upper bound limits, the total delay of the worst path through these tnets would not be any worse than that of the original worst critical path. However, note that this is not the optimal slack allocation. We have pessimistically limited the maximum delay for some tnets but they could go even higher without affecting the worst slack. [58] discusses the slack allocation problem in detail. Optimal slack allocation is generally achieved by solving linear programs, but that would be too slow for our purpose. In our work, we use a simple slack allocation algorithm similar to the idea described above.

#### 3.4.2.3 Neighborhood Extraction

We extract candidate locations for each node in the critical path from within a square of size $d$ centered at that path node. For example, Figure 3.35 shows a critical path A-B-C and three squares of side length 5 centered at A, B and C respectively. It is highly likely that these squares would overlap, and we have to decide which location to assign to which node or pair of nodes adjacent in the critical path. For this, we first check the legality of placing a critical path node in all the locations lying within its square. Illegal locations would not be considered henceforth.

After this, we compute the distances of each of the locations lying within some square from the corresponding critical path nodes (shown in Figure 3.35). We assign each location to the critical path node which is closest to it (Figure 3.36). We can also add a second node that is adjacent to the chosen node in the critical path. Consider the example in Figure 3.36. The black location AB is closest to A. So, we assign it to A first. The next closest path node is C, but C is not adjacent to A in the critical path. So, we assign it to B. The case with the black location(s) C is similar. They are closest to C, so we assign them to C first. The next closest path node is A, but A is not adjacent to C in the critical path, so we cannot assign it to A. They are not in B's box, se we cannot assign them to B either. We are left with C only.

It may so happen that some path nodes in the middle of the path are assigned too few sites due to conflict with other path nodes. In such cases, we adjust the site assignment by borrowing sites from adjacent path nodes so that each node has sufficient chance to move. If we want to assign more locations to a particular critical path node, we traverse the locations within its box that are assigned to some other node(s) one by one and keep assigning them to this node subject to the condition that the resultant number of locations assigned to the node from which we are borrowing should not be less than that for the current node. If we assign a location to 2 nodes, we ensure that they are adjacent in the critical path.

The nodes in the middle of the critical path are connected to two tnets which are likely to be in different directions. However, the starting and ending nodes have only one tnet each from the critical path. Hence we give a higher priority to the starting and ending nodes in the critical path in case of ties as these nodes have a definite direction of movement which could shorten the path.

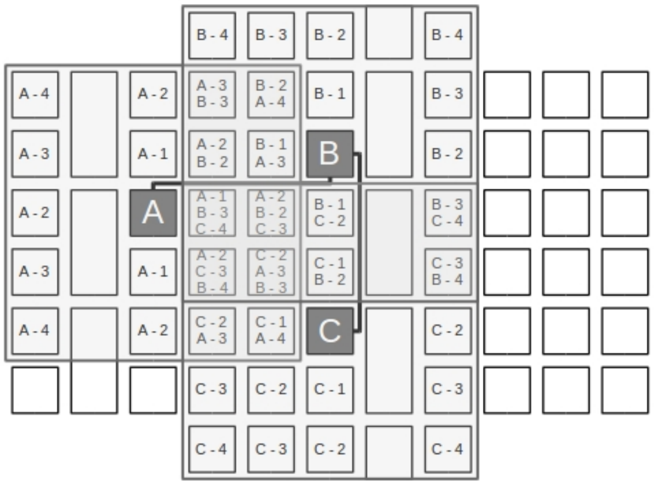


Figure 3.35: Extracting neighborhood around a critical path

### 3.4.2.4 Finding the Shortest Path

Once we have built the graph, we can find the shortest path from any node in the first layer to any node in the last layer. We do this using breadth-first traversal on layers which runs in $\Theta(E)$ time on a layered graph like ours, where $E$ is the number of edges. We do not need an elaborate algorithm like Dijkstra's due to the layered nature of our network. The delay for a node in layer $i$ can be calculated from the delays for layer $i - 1$ and the delays of the edges between the two layers.

We initialize delays of all nodes in the graph except the first layer to infinity. The nodes in the first layer are assigned delay value 0. We proceed

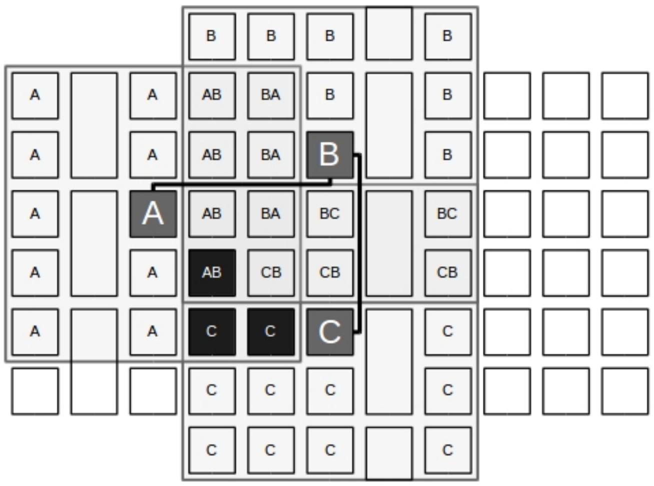


Figure 3.36: Assignment of locations to critical path nodes

layer by layer. At step $i$, we compute the outgoing delays for each node in layer $i-1$ by adding the previously computed delay for that node to the delay of the outgoing tnet. Thus, we get a set of delay values for each node in layer $i$ corresponding to the incoming tnets for that node. We set the delay for that node to the minimum of all its incoming delays. We also keep a pointer to the incoming tnet which led to the minimum delay for each node. This is useful for tracing the optimal location assignment for the critical path nodes.

The cost(cumulative delay) for a node $v$ in the graph is given by:

$$cost(v) = \min_{u \in input(v)} \{cost(u) + edge\_cost(u,v)\} \tag{3.10}$$

When we have chosen a tnet with minimum cumulative incoming delay for a node in level $i$, we also store the locations of the nodes in levels before $i$ that affect the case (iii) tnets. Thus we will have accurate placement

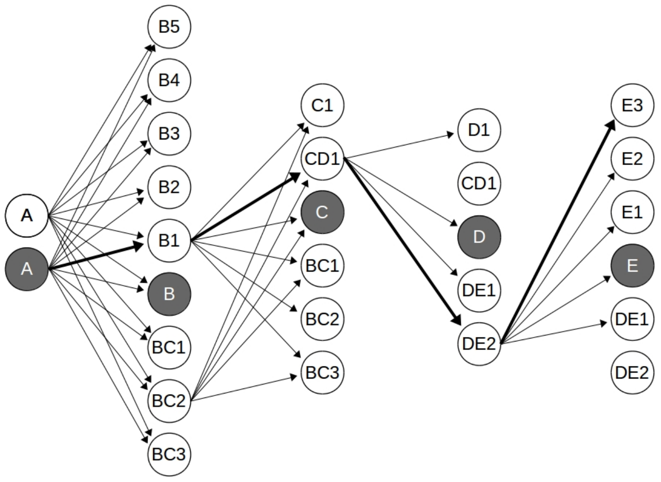


Figure 3.37: A solution to the shortest path problem

information when computing tnet delays for layer $i + 1$.

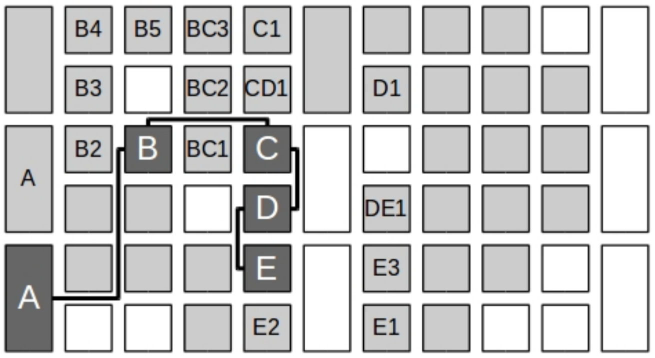


Figure 3.38: Changing placement to reflect the shortest path

Once we have found the shortest path, we change the node locations to reflect the same. Figure 3.37 shows a possible shortest path. Here, the shortest path goes through A's original location, B1, CD1, DE2 and E3. So, we choose A's original location for A, B1 for B, CD1 for C (and move the object previously at CD1 to C's original location), DE2 for D and E3 for E (and move the object previously at E3 to E's original location), as shown in Figure 3.38.

### 3.4.3 Complexity Analysis

We assume that the average length of a critical path is $N$, the average size of sub-neighborhood for each path node is $M$ ($=d^2$) and the average no. of pins per node (LAB or DSP or RAM) is $p$.

#### 3.4.3.1 Extracting the critical path from a tnet

Path extraction involves forward and backward propagation for the seed tnet. At each step, we go through all the incoming or outgoing tnets for a node that share a combinational path with the seed tnet and choose the one with the highest criticality. The amortized no. of tnets that we go through per node is $p$. We do this for at most $N$ nodes. Hence, the complexity for extracting a path is $pN$.

#### 3.4.3.2 Extracting the neighborhood from a path

The average number of sites that we consider for each path node is $M$. We have to compute distances from each path node to all sites within its box. This will require a total of $MN$ operations. Assigning the sites to nodes will take a constant multiple of $MN$ time.

#### 3.4.3.3 Generating the graph given the neighborhood

Time complexity here is dominated by edge costs. There are $(N-1)M^2$ edges in the graph. We have to iterate over at most $2p$ tnets for each edge. Hence, edge cost computation requires $2p(N-1)M^2$ time. Cost computation for case (i) tnets takes an additional $pNM$ time.

#### 3.4.3.4 Solving for shortest path

We iterate over the incoming edges for each node at each level and store the minimum cost. We have to go through at most $M$ incoming edges for each node starting from the second layer. There are a total of $M(N-1)$ such nodes. Hence the total time taken is $(N-1)M^2$.

We see that the overall complexity is dominated by complexity of graph generation, which is $\mathcal{O}(pNM^2)$

### 3.4.4 Parallelization Schemes

The most widely used method of speeding up an optimization procedure is to divide the problem into subproblems with little or no interaction and solve them in parallel. In out context, this would mean optimizing different critical paths in parallel. We are thus forced to ensure that the neighborhoods that we choose for different paths are disjoint and that there is no tnet connecting these neighborhoods. Also, critical paths are not spread uniformly over the chip but tend to form clusters at a few spots. Many different critical paths can share a LAB. Therefore, these paths cannot be optimized in parallel. Instead, we look at ways to speed up our algorithm for a single critical path.

Consider our shortest path problem. While computing the cost for each edge in the graph, we have to iterate over all the tnets under case (ii) incident on the two path nodes corresponding to that edge. We have already seen that the complexity for computing the edge costs is $\mathcal{O}(pNM^2)$, which is high. Hence we would like to speed up this part of our algorithm.

**Observation:** The cost of each edge in the graph that we form is independent of the cost of other edges.

Using this observation, we can compute all the edge costs in parallel. A similar observation shows that delays for tnets under case (i) can also be computed in parallel.

**Observation:** Each node within a single layer of our graph (for finding shortest path) is independent of the other nodes in the same layer.

Each node only depends on the nodes on the previous layer which have outgoing edges to that node. Since we solve for shortest path dynamically layer-by-layer, we can parallelize the computation at each layer. This is similar to parallelization of timing analysis where the computations for different timing end-points are independent.

None of the above parallelization schemes affect the placement or Fmax results. They only change runtime.

### 3.4.5 Results

We integrated our algorithm in an industrial FPGA design implementation flow and tested the algorithm on an industrial benchmark set.

Table 3.16: Benchmark set details

| Design size | # LABs, RAMs and DSPs |
|---|---|
| Minimum | 4156 |
| Maximum | 40889 |
| Average | 14850 |
| Number of designs | 86 |

The industrial benchmark set details are given in Table 3.16. Logic utilizations for all designs are shown in Figure 3.39. Our base flow consists of an industrial strength timing-driven global placer followed by a legalizer followed by the net-based timing-driven detailed placement from [41]. In our new flow, we run our critical path based detailed placer after the net based detailed placer. We set the value of $d$ to 5. In all the data presented in this subsection, we report the geometric average across all benchmarks that have high statistical confidence.

We compare our results with the net-based detailed placement algorithm in [41]. On the average, our algorithm improves the maximum clock frequency (Fmax) at placement stage by 4.5% on top of the net-based placer in [41], while degrading wirelength by only 0.2%. Our average runtime overhead is 7.5% of placement and packing runtime.

Our results indicate that path-based algorithms for detailed placement

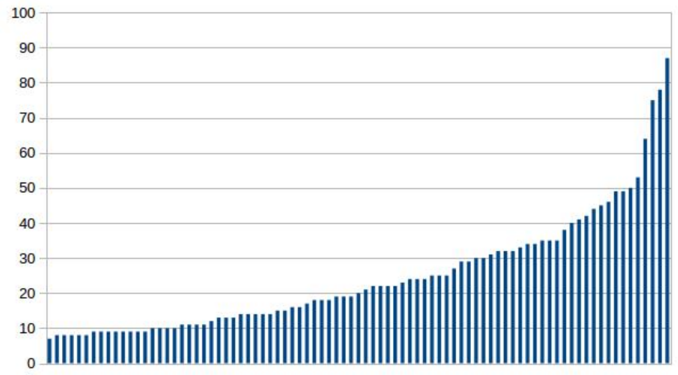


Figure 3.39: % Logic utilization (y-axis) for all designs

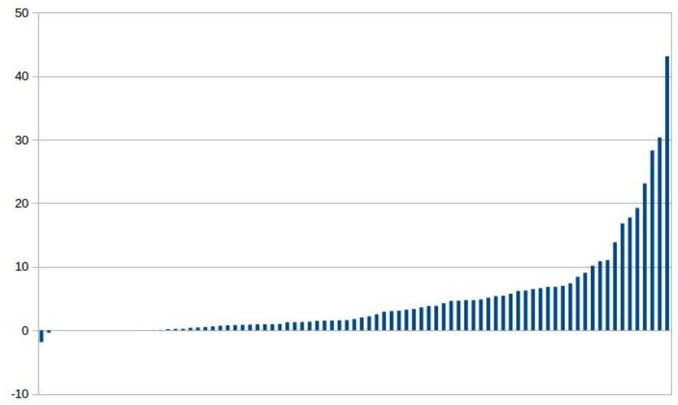


Figure 3.40: % Fmax change (y-axis) for all designs

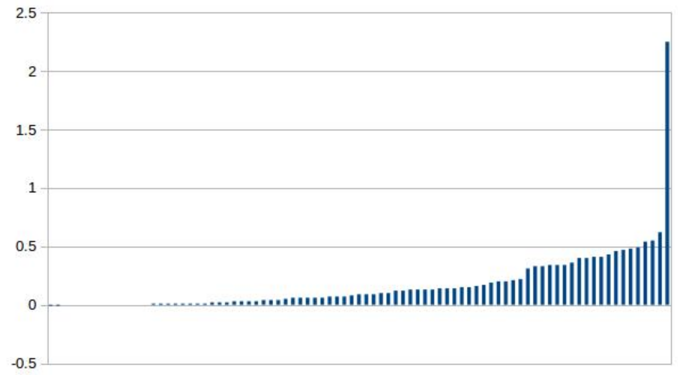


Figure 3.41: % Wirelength change (y-axis) for all designs

are complementary to net-based algorithms, especially considering that we are interested in optimizing timing as well as wirelength.

Table 3.17: Results for our Algorithm

| ΔFmax(%) | ΔWirelength(%) | ΔRuntime(%) |
|---|---|---|
| 4.5 | 0.2 | 7.5 |

The Fmax and wirelength histograms for all designs are shown in Figures 3.40 and 3.41 respectively. We observe that the majority of the Fmax changes are within 10% but there are some extremely good outliers. The variance in the Fmax changes are due to factors like the structure of the design, utilization of the design, modeling errors in earlier stages of the flow, varying resource usages by type, etc. Two designs have a negative Fmax change, which may be due to our relaxation of delay limits slightly beyond the worst slack. Most of the wirelength changes are within 0.5%. The negligible impact on wirelength is expected as our algorithm only works on a few critical paths and leaves most of the nets undisturbed.

# Chapter 4

# Conclusion

This dissertation proposes new algorithms for FPGA placement as well as hardware acceleration techniques for some of those algorithms. Both global and detailed placement algorithms have been investigated and some of these algorithms have also been integrated into industrial design implementation tools for state-of-the-art FPGAs.

Global placement is an important technique used in design implementation tools for modern FPGAs. Global placement significantly affects the quality of results with respect to wiring usage, timing and routability. Global placement techniques are numerical in nature and consume a significant part of the overall placement runtime. In chapter 2, we presented some new algorithms for improving the quality and runtime for FPGA global placement. A theoretical analysis of how placement shapes affect wirelength is provided with certain assumptions and a near-optimal shape is determined empirically. A min-cost-flow-based and shape-driven spreading algorithm is proposed and is implemented using linear programming. A flow realization algorithm is also proposed which can work on any given flow satisfying certain conditions and preserves the relative order among cells to a great extent. Our experimental results demonstrate that our new spreading algorithm achieves 1.79% better wirelength compared to an improved version [21] of the first place placer in the ISPD 2016 FPGA placement contest [71]. Routing congestion and runtime are also better compared to the same placer.

Chapter 2 also proposes FPGA acceleration of wirelength gradient computation and spreading, which are the two main components of global placement. Hybrid CPU-FPGA acceleration of wirelength gradient computation for global placement has been proposed which achieves an average speedup of 3.03x for wirelength gradient computation and 2x for the entire global placement flow using only one device at a time (either CPU or FPGA). We organize the gradient computation process into different tasks and map each task to the device which is best suited to it (CPU or FPGA). Our runtimes for wirelength gradient computation are comparable to the best known GPU implementation for benchmarks of similar sizes, even though our gradient function is more complicated. Our results show that global placement for each design in the ISPD 2016 FPGA placement contest [71] benchmark suite can be completed in less than a minute.

Finally, we propose a new fluid-flow based spreading algorithm for use in an analytical global placement system. Our new algorithm is massively parallel and we also accelerate the algorithm on an FPGA. While maintaining the quality of results, our new spreading algorithm achieves a speedup of up to 6.47x when compared to a linear-programming-based spreading algorithm [1]. By combining our FPGA-accelerated spreading algorithm with our FPGA-accelerated numerical solver, we achieve an overall global placement average speedup of 3.06x when compared to a CPU implementation of the same solver with the linear programming based spreading algorithm in [1].

Global placement is accompanied by detailed placement techniques to help correct modeling errors during global placement (that occur due to approximations in the numerical formulation of global placement). Detailed placement helps to account for the fine-grained architectural constraints of modern FPGAs to further improve the quality of results. In chapter 3, we discuss challenges in detailed placement for modern FPGAs, and propose new detailed placement algorithms to address them. A new dynamic-programming-based detailed placement algorithm is proposed, which improves both wirelength and the maximum clock frequency obtained (Fmax). Unlike previous works, our dynamic programming formulation for detailed placement can be applied to multiple partitions as well as to a rectangular grid of placeable objects. We also proposed parallelization schemes for our algorithm. We integrated our new dynamic programming based detailed placement algorithm in an industrial FPGA design implementation flow. Experimental results on industrial benchmarks demonstrate that our DP algorithm achieves good improvements in wirelength and Fmax, with minimal runtime overhead, when compared to existing DP approaches and the output of industrial-strength global placement and legalization engines.

We also discussed the challenges in timing-driven detailed placement for modern FPGAs and proposed a new critical path optimization technique to address them. We propose a new timing-driven detailed placement algorithm for FPGAs, which uses a shortest path delay formulation combined with delay budgets for each net to prevent timing degradation in all the affected paths. We also proposed parallelization schemes for our algorithm. Experimental results on industrial benchmarks demonstrate that our algorithm achieves 4.5% improvement in Fmax on average with negligible wirelength penalty and minimal runtime overhead. We also demonstrated that our algorithm is complementary to net-based detailed placement techniques.

Chapter 3 also demonstrates GPU acceleration of our dynamic-programming-based detailed placement algorithm. Certain modifications are proposed to the dynamic-programming-based detailed placement algorithm to enable it to run faster on a GPU. We propose two serially-equivalent flows, one of which uses both CPU and GPU and one which runs entirely on the GPU. We further propose two sub-flows for each of these flows to explore the potential architectural limitations of GPUs. All of our flows are equivalent to the serial CPU version, ensuring that there is no loss of QoR. Experimental results show that we can achieve 5.5 to 7x speedup over the multi-threaded CPU version. We also demonstrate FPGA acceleration of the same dynamic-programming-based detailed placement algorithm, which achieves 2.03x speedup on average.

FPGA CAD tools face the challenge to improve performance and maintain fit-ability at the same pace as the growth in size and complexity of modern FPGAs, as well as the workloads that are run on the FPGAs. As multi-threaded CPU implementations give diminishing returns, hardware acceleration techniques have the potential to provide scalable runtime improvements. Our work shows the advantage of using FPGA and GPU acceleration for traditional EDA problems in placement technology. We plan to extend some of these concepts to other parts of the design implementation flow, like clustering and routing. It is possible to get even more speedup than we have demonstrated by using larger FPGAs with higher memory bandwidth. For memory-intensive compute applications, the ability to fit the entire compute as well as the data required for the compute within the same FPGA would help alleviate the bottlenecks encountered during memory access. Hence, larger FPGAs with higher memory capacities and bandwidths would yield superlinear runtime improvements. Beyond hardware acceleration, further improvement to placement algorithms may be possible. Enhancements to global placement to enable faster convergence is a promising area for continuing research. Numerical formulations for hardware-accelerated global placement with reduced-precision numbers is another promising area to improve the compile time without trading off the quality of results. This approach can significantly reduce the amount of data movement required, and hence speed up global placement by orders of magnitude even on devices with limited memory bandwidth.

# Bibliography

[1] S. Dhar, L. Singhal, M. A. Iyer and D. Z. Pan, *"A Shape-Driven Spreading Algorithm using Linear Programming for Global Placement"*, Asia and South Pacific Design Automation Conference, 2019.

[2] P. Spindler and U. Schlichtmann, *"Kraftwerk2-A Fast Force-Directed Quadratic Placement Approach Using an Accurate Net Model"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:27, Issue: 8), 2008

[3] T. Lin and C. Chu, *"POLAR 2.0: An effective routability-driven placer"*, Design Automation Conference, 2014

[4] T. Lin, C. Chu and G. Wu, *"POLAR 3.0: An Ultrafast Global Placement Engine"*, International Conference on Computer-Aided Design, 2015

[5] J. C. Tiernan, *"An Efficient Search Algorithm to find the Elementary Circuits of a Graph"*, Communications of the ACM (Volume: 13, Issue: 12), 1970

[6] D. B. Johnson, *"Finding all the Elementary Circuits of a Directed Graph"*, SIAM Journal on Computing, (Volume: 4, Issue: 1), 1975

[7] C. Fobel, G. Grewal and A. Morton, *"Using Hardware Acceleration to Reduce FPGA Placement Times"*, Canadian Conference on Electrical and Computer Engineering, 2007

[8] S. Dhar, L. Singhal, M. A. Iyer and D. Z. Pan, *"FPGA Accelerated FPGA Placement"*, International Conference on Field-Programmable Logic and Applications, 2019.

[9] S. Dhar, L. Singhal, M. A. Iyer and D. Z. Pan, *"FPGA Accelerated Spreading for Global Placement"*, IEEE High Performance Extreme Computing conference, 2019.

[10] P. Mateti and N. Deo, *"On Algorithms for Enumerating All Circuits of a Graph"*, SIAM Journal on Computing, (Volume: 5, Issue: 1), 2012

[11] C. Lin and M. D. F. Wong, *"Accelerate Analytical Placement with GPU: A Generic Approach"*, Design Automation and Test in Europe, 2018

[12] R. Pattison, C. Fobel, G. Grewal and S. Areibi, *"Scalable Analytic Placement for FPGA on GPGPU"*, International Conference on ReConFigurable Computing and FPGAs, 2015

[13] A. Al-Kawam and H. M. Harmanani, *"A parallel GPU implementation of the TimberWolf placement algorithm"*, International Conference on Information Technology, 2015

[14] A. Choong, R. Beidas and J. Zhu, *"Parallelizing Simulated Annealing-Based Placement using GPGPU"*, International Conference on Field Programmable Logic and Applications, 2010

[15] C. Fobel, G. Grewal and D. Stacey, *"A Scalable, Serially-Equivalent, High-Quality Parallel Placement Methodology Suitable for Modern Multicore and GPU Architectures"*, International Conference on Field Programmable Logic and Applications, 2014

[16] J. Cong, Z. Fang, M. Lo, H. Wang, Jingxian Xu nad Shaochong Zhang, *"Understanding Performance Differences of FPGAs and GPUs"*, International Symposium on Field-Programmable Custom Computing Machines, 2018

[17] Ulrich Brenner and Markus Struzyna, *"Faster and Better Global Placement by a New Transportation Algorithm"*, Design Automation Conference, 2005

[18] Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang, *"NTUplace3: An Analytical Placer for Large-Scale Mixed-Size Designs With Preplaced Blocks and Density Constraints"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume: 27, Issue: 7), 2008

[19] Jingwei Lu, Pengwen Chen 2, Chin-Chih Chang, Lu Sha, Dennis J-.H. Huang, Chin-Chi Teng and Chung-Kuan Cheng, *"ePlace: Electrostatics*

*Based Placement Using Nesterovs Method"*, Design Automation Conference, 2014

[20] Tao Lin, Chris Chu, Joseph R. Shinnerl, Ismail Bustany and Ivailo Nedelchev, *"POLAR: A High Performance Mixed-Size Wirelengh-Driven Placer With Density Constraints"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:34, Issue: 3), 2015

[21] Wuxi Li, Shounak Dhar and David Z. Pan, *"UTPlaceF: A Routability-Driven FPGA Placer with Physical and Congestion Aware Packing"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:37, Issue:4), 2018

[22] Ameya R. Agnihotri and Patrick H. Madden, *"Fast Analytic Placement using Minimum Cost Flow"*, Asia and South Pacific Design Automation Conference, 2007

[23] Ulrich Brenner, Anna Pauli and Jens Vygen, *"Almost Optimum Placement Legalization by Minimum Cost Flow and Dynamic Programming"*, International Symposium on Physical Design, 2004

[24] Nima Karimpour Darav, Ismail S. Bustany, Andrew Kennings, David Westwick and Laleh Behjat, *"Eh?Legalizer: A High Performance Standard-Cell Legalizer Observing Technology Constraints"*, ACM Transactions on Design Automation of Electronic Systems, (Volume: 23, Issue: 4), May 2018

[25] Ulrich Brenner, *"BonnPlace Legalization: Minimizing Movement by Iterative Augmentation"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume: 32, Issue: 8), 2013

[26] Minsik Cho, Haoxing Ren, Hua Xiang and Ruchir Puri, *"History-based VLSI legalization using network flow"*, Design Automation Conference, 2010

[27] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao and D. Burger, *"A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services"*, International Symposium on Computer Architecture, 2014

[28] Shuai Li, Cheng-Kok Koh, *"Mixed integer programming models for detailed placement"*, Proceedings of the International Symposium on Physical Design, 2012, pp 87-94

[29] Shuai Li, Cheng-Kok Koh, *"MIP-based detailed placer for mixed-size circuits"*, Proceedings of the International Symposium on Physical Design, 2014, pp11-18

[30] V. Betz and J. Rose, *"VPR: A new Packing, Placement and Routing Tool for FPGA Research"*, Proceedings of the 7th Int. Workshop on Field-Programmable Logic and Applications, 1997, pp 213-222

[31] Ednaldo Mariano Vasconcelos de Lima, Dr. Antnio Carlos Cavalcanti and Dr. Lucdio dos Anjos Formiga Cabral, *"A New Approach to VPR Tool's FPGA Placement"*, World Congress on Engineering and Computer Science, 2007

[32] Min Pan, Natarajan Viswanathan and Chris Chu, *"An efficient and effective detailed placement algorithm"*, IEEE/ACM International Conference on Computer-Aided Design, 2005

[33] H. Bian, A.C. Ling, A. Choong, J. Zhu, *"Towards scalable placement for FPGAs"*, Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2010

[34] Konrad Doll, Frank M. Johannes, and Kurt J. Antreich, *"Iterative placement improvement by network flow methods"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:13 , Issue: 10 ), 1994, pp 1189-1200

[35] Myung-Chul Kim, Jin Hu, Dong-Jin Lee and Igor L. Markov, *"A SimPLR Method for Routability-driven Placement"*, Proceedings of the International Conference on Computer-Aided Design, 2011, pp 67-73

[36] Ken Eguro, Scott Hauck and Akshay Sharma, *"Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement"*, Proceedings of the 42nd annual Design Automation Conference, 2005

[37] Sung-Woo Hur and John Lillis, *“Mongrel: Hybrid Techniques for Standard Cell Placement*, Proceedings of the International Conference on Computer-Aided Design, 2000.

[38] Devang Jariwala, John Lillis, *“On Interactions Between Routing and Detailed Placement”*, Proceedings of the International Conference on Computer-Aided Design, 2004, pp387-393

[39] A. E. Caldwell, A. B. Kahng and I. L. Markov, *“Optimal End-Case Partitioners and Placers for Standard-Cell Layout”*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume:19, No. 11), November 2000

[40] David Z. Pan, Bill Halpin and Haoxing Ren, *“Timing Driven Placement”*, Chapter 21, Handbook of Algorithms for Physical Design Automation, 2008

[41] S. Dhar, S. Adya, L. Singhal, M. A. Iyer and D. Z. Pan, *“Detailed Placement for Modern FPGAs using 2D Dynamic Programming”*, International Conference on Computer Aided Design, 2016

[42] S. Dhar, M. A. Iyer, S. Adya, L. Singhal, N. Rubanov and D. Z. Pan, *“An Effective Timing-Driven Detailed Placement Algorithm for FPGAs”*, International Symposium on Physical Design, 2017

[43] S. Dhar and D. Z. Pan, *“GDP: GPU Accelerated Detailed Placement”*, IEEE High Performance Extreme Computing conference, 2018

[44] Chrystian Guth, Vinicius Livramento, Renan Netto, Renan Fonseca, Jose Luis Guntzel, Luiz Santos, *"Timing-Driven Placement Based on Dynamic Net-Weighting for Efficient Slack Histogram Compression"*, International Symposium on Physical Design, 2015

[45] Amit Chowdhary, Karthik Rajagopal, Satish Venkatesan, Tung Cao, Vladimir Tiourin, Yegna Parasuram, Bill Halpin, *"How Accurately Can We Model Timing In A Placement Engine?"*, Design Automation Conference, 2005

[46] Tim Kong, *"A novel net weighting algorithm for timing-driven placement"*, International Conference on Computer Aided Design, 2002.

[47] Haoxing Ren, David Z. Pan, David S. Kung, *"Sensitivity guided net weighting for placement-driven synthesis"*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 2005.

[48] Alexander Marquardt, Vaughn Betz, Jonathan Rose, *"Timing-Driven Placement for FPGAs"*, International Symposium on Field Programmable Gate Arrays, 2000

[49] Haoxing Ren, David Z. Pan, Charles J. Alpert, Gi-Joon Nam, Paul Villarrubia, *"Hippocrates: First-Do-No-Harm Detailed Placement"*, Asia and South Pacific Design Automation Conference, 2007

[50] Huimin Bian, Andrew C. Ling, Alexander Choong, Jianwen Zhu, *"Towards scalable placement for FPGAs"*, International Symposium on Field

Programmable Gate Arrays, 2010

[51] Natarajan Viswanathan, Gi-Joon Nam, Jarrod A. Roy, Zhuo Li, Charles J. Alpert, Shyam Ramji, Chris Chu, *"ITOP: Integrating Timing Optimization within Placement"*, International Symposium on Physical Design 2010

[52] Tao Luo, David Newmark, David Z. Pan, *"A New LP Based Incremental Timing Driven Placement for High Performance Designs"*, Design Automation Conference, 2006

[53] Gang Chen and Jason Cong, *"Simultaneous placement with clustering and duplication"*, ACM Transactions on Design Automation of Electronic Systems, (Volume: 11, Issue:3), 2006.

[54] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan, *"Timing-driven partitioning-based placement for island style FPGAs"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), (Volume:24, Issue:3), 2005.

[55] Charles M. Fiduccia and Robert M. Mattheyses, *"A linear-time heuristic for improving network partitions"*, ACM/IEEE Design Automation Conference, 1982.

[56] Andrew B. Kahng , Stefanus Mantik, Igor L. Markov, *"Min-Max Placement for Large-Scale Timing Optimization"*, International Symposium on Physical Design, 2002

[57] Michael D. Moffitt, David A. Papa, Zhuo Li, Charles J. Alpert, *"Path Smoothing via Discrete Optimization"*, Design Automation Conference, 2008

[58] Siddharth Joshi, Stephen Boyd, *"An Efficient Method for Large-Scale Slack Allocation"*, 2008

[59] Ken Eguro, Scott Hauck, *"Enhancing Timing-Driven FPGA Placement for Pipelined Netlists"*, Design Automation Conference, 2008

[60] Chao Chris Wang, Guy G. F. Lemieux, *"Scalable and Deterministic Timing-Driven Parallel Placement for FPGAs"*, International Symposium on Field Programmable Gate Arrays, 2011

[61] Mei-Fang Chiang, Takumi Okamoto, Takeshi Yoshimura, *"Register Placement for High-performance Circuits"*, Design Automation and Test in Europe, 2009

[62] Bill Halpin, C. Y. Roger Chen, Naresh Sehgal, *"Detailed Placement with Net Length Constraints"*, International Workshop on System On Chip, 2003

[63] Igor L. Markov, Jin Hu, Myung-Chul Kim, *"Progress and Challenges in VLSI Placement Research"*, International Conference on Computer Aided Design, 2012

[64] Meng-Kai Hsu, Valeriy Balabanov and Yao-Wen Chang, *"TSV-Aware Analytical Placement for 3-D IC Designs Based on a Novel Weighted-Average*

*Wirelength Model"*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (Volume:32, Issue:4), 2013

[65] https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[66] https://docs.nvidia.com/cuda/thrust/index.html

[67] https://software.intel.com/en-us/hardware-accelerator-research-program

[68] https://www.intel.com/content/www/us/en/
products/programmable/soc/stratix-10.html

[69] https://www.cadence.com/solutions/cadence-cloud

[70] https://www.xilinx.com/products/silicon-devices/soc.html

[71] http://www.ispd.cc/contests/16/

# Vita

Shounak Dhar received a B.Tech degree in Electrical Engineering with minor in Mechanical Engineering from Indian Institute of Technology, Bombay, in 2014. He started his Ph.D. program at the University of Texas at Austin in 2014, with research advisor David Z. Pan and co-advisor Mahesh A. Iyer from Intel Corporation. He has interned at Altera, San Jose in summer of 2015, and at Intel Corporation, San Jose during the summers of 2016 and 2017.

Shounak Dhar's research interests include physical design for VLSI, hardware acceleration and optical computing.

Permanent address: shounak.dhar@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.