UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

JULIA CASARIN PUGET

UFRGSPlace: A Wirelength Driven FPGA Placement Algorithm

Thesis presented in partial fulfillment of the requirements for the degree of Master of Microeletronics

Advisor: Prof. Dr. Ricardo Reis

Porto Alegre
July 2018

CIP — CATALOGING-IN-PUBLICATION

Puget, Julia Casarin

UFRGSPlace: A Wirelength Driven FPGA Placement Algorithm / Julia Casarin Puget. – Porto Alegre: PGMICRO da UFRGS, 2018.

58 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2018. Advisor: Ricardo Reis.

I. Reis, Ricardo. II. Título.

ABSTRACT

FPGAs are semiconductor devices that can be reprogrammed to reach different application requirements after manufacturing. The architecture of an FPGA can be homogeneous, containing only standard blocks of an FPGA, IOs and CLBs, or heterogeneous, being able to include also other types of blocks such as memory blocks and DSPs.

One of the steps required in the design flow of an FPGA is placement, in which positions for the inner components of the FPGA are selected, and it is highly dependent upon its architecture, varying dramatically from one family of FPGA devices to another.

In this work, we present an EDA tool that was developed for placing heterogeneous FPGAs aiming routability. It is split into two flows, one that was implemented for the ISPD 2016 Placement Contest (ISPD..., 2016), in which, the tool got the 4th place, and one with the partitioning algorithm hMETIS (KARYPIS et al., 1999) in the FPGA placement flow stage.

Keywords: Routability. Optimization. Placement. FPGA. Partitioning. Microelectronics.

UFRGSPlace: Um Algoritmo para posicionamento de FPGAs Visando ao Comprimento de Fio

RESUMO

FPGAs são dispositivos semicondutores que podem ser reprogramados para atender a diferentes requisitos de aplicações após a sua fabricação. A arquitetura de uma FPGA pode ser homogênea, contendo apenas blocos padrão de uma FPGA, IOs e CLBs, ou heterogênea, podendo conter outros tipos de blocos, como de memória ou DSPs. Um dos passos necessários no fluxo de projeto numa FPGA é o posicionemnto, em que posições para os componentes internos da FPGA são selecionados, e esse passo é altamente dependente da sua arquitetura, variando dramaticamente de uma família de dispositivos FPGA para outra. Neste trabalho, é apresentada uma ferramenta de EDA que foi desenvolvida para posicionamento de FPGAs heterogêneas, visando a roteabilidade. Ele é dividido em dois fluxos, um que foi implementado para a competição ISPD 2016 Contest (ISPD. . . , 2016), na qual, a ferramenta ficou na quarta colocação, e outro com o algoritmo de particionamento hMETIS (KARYPIS et al., 1999) adicionado ao fluxo de posicionamento.

Palavras-chave: Routing, Optimization, Placement, FPGA, Microelectronics, Partitioning.

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CLB	Configurable Logic Block
DSP	Digital Signal Processor
EDA	Electronic Design Automation
FF	Flip-flop
FM	Fiduccia-Mattheyses
FPGA	Field Programmable Gate Array
HPWL	Half Perimeter Wire-length
IC	Integrated Circuit
ICCAD	International Conference on Computer Aided Design
IO	In/Out
ISPD	International Symposium on Circuit Design
LUT	Lookup Table
RAM	Random Access Memory
RTL	Register Transfer Level
UFRGS	Universidade Federal do Rio Grande do Sul
VLSI	Very Large Scale Integration
ULSI	Ultra Large Scale Integration
VPR	Versatile Place and Route

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

1.1 IC Design and Automation

The whole process of designing integrated circuits had been done mainly by hand until the middle of the 1970s, when CAD software began to exist, for automation and optimization of design processes. CAD software that aids in the steps of chip design is, nowadays, referred to as electronic design automation (EDA) software.

In the years following the publication of "Introduction to VLSI Systems" (MEAD; CONWAY, 1980), which promoted the concept of chip design beginning with a specification of its behavior made in a programming language, the complexity of the chips has increased significantly. This increase was also because designers began to have more access to verification and simulation tools, so it was easier to check if the design would function accordingly to their expectations before its construction.

The design flow of a VLSI circuit consists of many steps, as we can see in Figure 1.1. Physical design is a complex process, as noted by (SHERWANI, 1993), even when it is broken into steps.

The main focus of this work is on one of the steps into which the physical design flow is divided, the placement step, when component positions are selected, which is approached in more detail in Section 1.2. Another important step comes afterward - routing - which is when we create the wire connections between the logic gates.

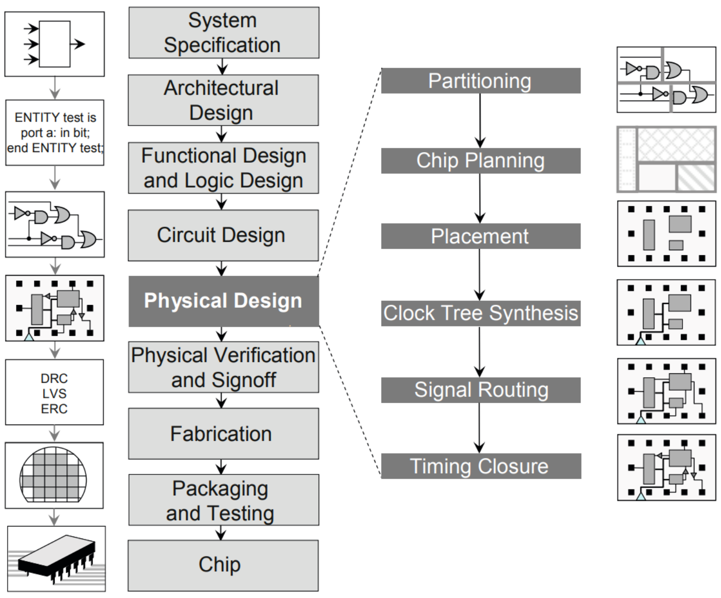


Figure 1.1: Typical VLSI design flow (KAHNG et al.,).

1.2 Placement

Placement is one of the stages of the VLSI circuit physical design flow. In this stage, valid positions for the minor logic components that comprise the circuit are selected, generally with the goal of minimizing a cost function. The placement problem is said to be NP-Complete (SHAHOOKAR; MAZUMDER, 1991), which is typically addressed by using heuristics and approximation algorithms to acquire solutions in reasonable runtime and providing scalability when it comes to large circuits.

Traditionally, in this stage, there is an objective to minimize the wirelength of the connections between the components and also try to avoid congested areas - where the number of connections outcomes the available space for routing -, but other properties of the circuit can be addressed, such as minimizing the area occupied by the components, or minimizing the propagation delay of a signal. The resulting placement must have all the components occupying valid positions, meaning that there must be no overlaps between any of them.

The placement step implemented for the tool covered in this work follows an analytical approach, which is divided into three phases: global placement, legalization and detailed placement (PAUL; DAS; BANERJEE, 2015).

During global placement, some algorithms relax the overlapping restriction, although it is sometimes interesting for the algorithm to reduce overlaps. A position is chosen for the components whilst minimizing wirelength. As said in (BRENNER; VYGEN, 2008), analytical placement works with the relaxation of allowing overlaps at first, minimizing a function that estimates wirelength, and then removing the overlaps.

Legalization is the step in which the remaining overlapping is removed, and components are moved to valid positions. In our case, the legalization step is more complex, because there are rules only present in the scope of FPGAs: instead of just having to eliminate overlaps and align the components to rows, we also have to assure that each component is inserted in its right slot. This scenario will be explained in Chapter 3, Section 3.2.

Detailed placement makes fine adjustments to positions by performing fine-grain optimizations that are hard to be seen during the global placement step, aiming to reduce wirelength after legalization is done. It can also be implemented to reach a certain goal, for example, minimize area.

The general physical design flow of an FPGA can be seen in Figure 1.2. It also compares it with a generic ASIC flow by outlining the steps that are not needed for FPGAs. As seen in Figure 1.2, FPGAs do not need the steps of scan insertion, clock tree synthesis and physical design verification, because the FPGA has already been manufactured and, although reconfigurable, it already has its resources defined.

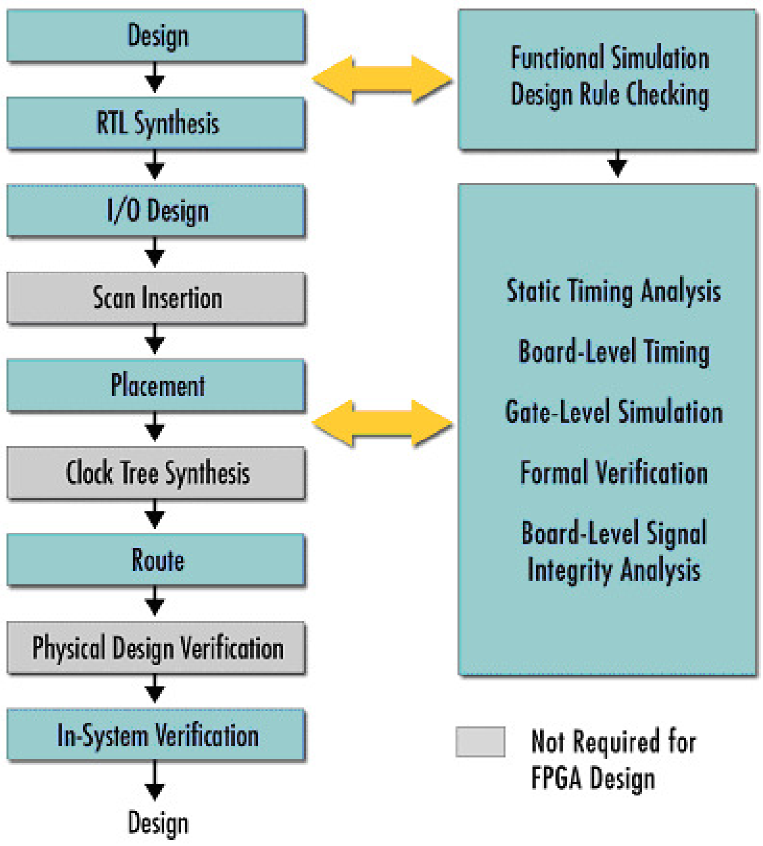


Figure 1.2: Generic FPGA physical design flow. (FPGA..., 2016a)

Currently, the development of physical synthesis algorithms for FPGAs is in a less mature stage than for ASICs, mostly due to the complexity in terms of the legalization rules mentioned above and the fact that the FPGAs have already been manufactured, with all its resources already defined. For example, when it comes to ASICs, when trying to meet timing closure in a circuit's synthesis, one can apply gate sizing, insertion of buffers to the circuit, or even rely on the placement algorithm with timing constraints. In the case of FPGAs, the methodology of gate sizing does not apply anymore, as the FPGA's hardware has already been manufactured, so, the developer has fewer degrees of freedom to work with to reach the timing closure objective. With this kind of situation in mind, the development of algorithms for the synthesis of FPGAs can be more difficult.

1.3 FPGAs

FPGAs are board circuits that can contain from thousands to more than a million logic gates within them, and their interconnections and cells are programmable after manufacturing. Since the introduction of the first FPGAs by Xilinx in the 1980s, FPGAs have grown in capacity by more than a factor of 10000 and performance by a factor of 100. Cost and energy per operation have decreased by more than a factor of 1000. Figure 1.3 shows this advancement. Capacity is the number of logic cells that the FPGA lodges, speed is same-function performance in programmable fabric. Price and power are per logic cell and are scaled up by 10000x.

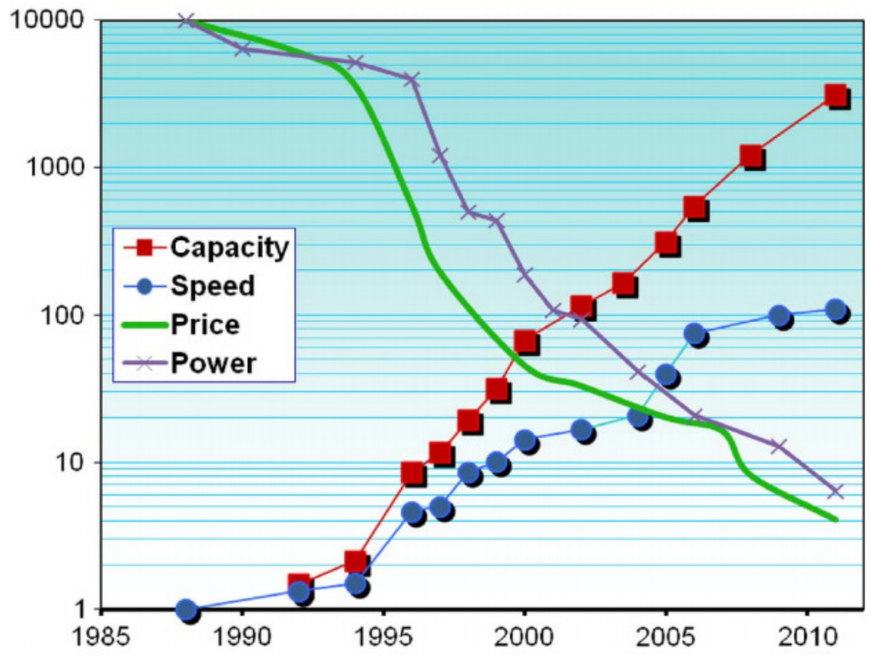


Figure 1.3: Xilinx FPGA attribute changes through the decades. (TRIMBERGER, 2015)

The architecture of an FPGA can be homogeneous, meaning it contains only the standard blocks for an FPGA (logic blocks and input and output), or heterogeneous, meaning it can contain other types of blocks, such as memory (RAMs) and signal processing (DSPs). The FPGA architecture targeted for placement in this work was from the Ultrascale (XILINX. . . , 2016) family.

On the next page, in Figure 1.4, we can see what a typical homogeneous FPGA's layout would look like. A generic homogeneous FPGA contains a matrix of configurable logic blocks (CLBs) that, altogether, perform the current function the FPGA was programmed for executing, and IO blocks, for input and output data. Also, there are switching blocks for selecting the routes the wires will follow (as they are programmable, the routes are not fixed).

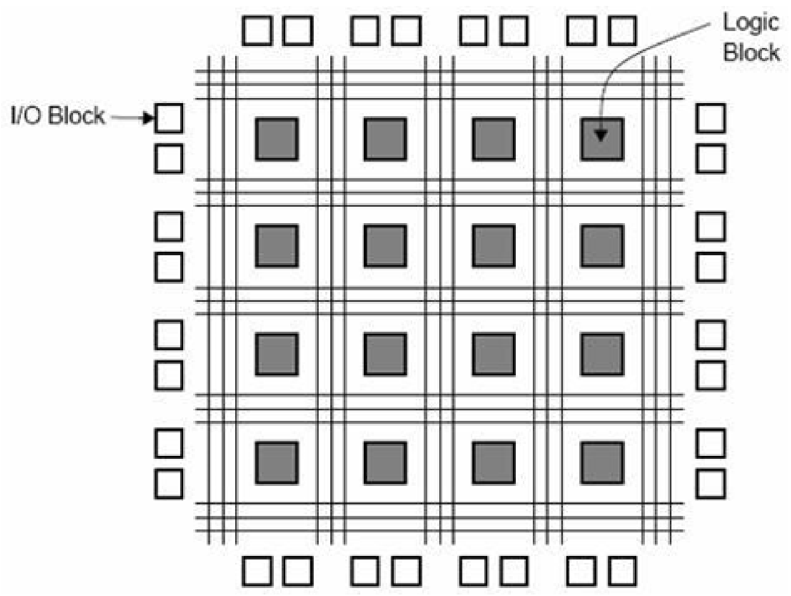


Figure 1.4: An example of a simple homogeneous FPGA layout. (FPGA..., 2016b)

FPGAs differ from standard-cell (XIU, 2008) methodology in ASIC circuits in many aspects, but this one especially influences on the placement stage: they already have fixed slots for each logic component; hence, we must only select which component goes where, not having to worry about them overlapping one another at some point. With that in mind, legalization regarding FPGA placement becomes more about following manufacturer rules and constraints regarding shared signals and inputs than actually removing overlaps, as mentioned in Section 1.2.

The following sections display the contributions of this work and the scope of this dissertation.

1.4 Contributions of This Work

The main contributions of this work can be summarized as:

1. An EDA tool for routability driven placement of homogeneous and heterogeneous FPGAs that supports inputs in the bookshelf format.
2. A set of algorithms capable of generating solutions with completely legal packing. All of the solutions provided total accordance with the set of packing constraints, which is addressed in Chapter 3. The main contribution of this tool concerning robustness and coherence with the design constraints is the packing stage of FPGA placement, which is the most prone to failures in rule checking because of the complexity of the constraints.
3. The tool implements a novelty packing refinement algorithm that improves the usage of

resources.

4. Another version of the FPGA placement with the same flow as the one above, except for utilizing graph partitioning before the packing stage.

1.5 Outline of this dissertation

This work is organized in 5 chapters. Chapter 2 describes the state-of-the-art that we have in FPGA placement as well as one algorithm that is still a foundation for some placement algorithms. Chapter 3 explains what was the ISPD 2016 Competition, its input files, and rules that would apply for the packing state (one of the placement stages that are required for an FPGA architecture). Chapter 4 describes our flow steps, and Chapter 4 explains in detail UFRGSPlace, which is the main contribution of this work. Chapter 5 finishes with a conclusion and work that we plan on performing in the future for improving the tool.

2 RELATED WORK

In this chapter, relevant works with respect to the subject of placement for FPGAs are summarized. The algorithms that will be cited are based on analytical placement, clustering FPGA blocks and placing with hierarchical simulated annealing.

2.1 Analytical Placement

Analytical placement, as mentioned in (MONTEIRO et al., 2015) is a technique that maps the placement problem to another formulation that can be numerically analyzed with the objective of minimizing a certain parameter. An analytical cost function for the placement problem is defined and then minimized with the use of numerical optimization methods. Based on the cost function, analytical placers are classified in either non-linear or quadratic.

2.1.1 HeAP: Heterogeneous Analytical Placer

HeAP (GORT; ANDERSON, 2012) is an analytical placer for heterogeneous FPGAs comprised of the same type of cells there are in the benchmarks used for the ISPD contest: DSPs, RAM, LUTs. They adapted a placer that was mainly built for state-of-the-art ASICs to the heterogeneous FPGA environment. One main difference between ASICs and FPGAs is that FPGAs contain discrete positions for each type of cell, the slots, so each type must go inside its own type of slot. It also deals with macros of LUT-based arrangements, such as carry chains.

Instead of counting the runtime based on parallelizing existing algorithms, they focused on runtime from an algorithmic perspective, presenting a fast approach for placing FPGAs. The algorithm they adapted is Simpl (KIM; LEE; MARKOV, 2010), and the connections between blocks are made based on the *bound2bound* model. They adapt Simpl by extending the concept of target locations for inserting cells to different block types.

Macros consist of multiple unit-sized blocks having a fixed placement with respect to one another (e.g. carry chains), and are given a single *(x,y)* location in the formulation. For modeling connections to macro blocks properly, they take into account the position of the unit-sized block within the macro to which the connection is actually made.

Figure 2.1 displays a connection between a unit-sized block and a macro block. It is made to the middle block within the macro, which has a x offset of 1, relative to the macro origin. Such offsets can be modeled by incorporating them into the vector representing the connections between movable and fixed blocks, allowing the actual connection length to be minimized, rather than a connection to the origin of the macro.

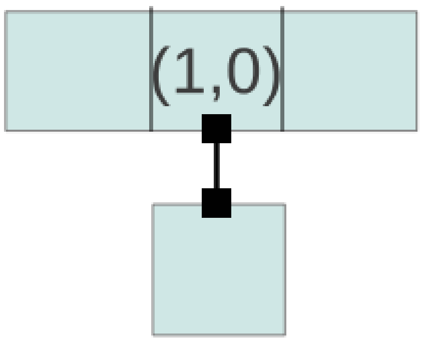


Figure 2.1: Connection between unit-sized block and macro block. (GORT; ANDERSON, 2012)

As said previously, legalization for FPGAs is more complicated because there are discrete positions, as there are certain kinds of slots for certain kinds of cells. Here, it is done essentially by recursive partitioning for finding valid positions.

First, they find a subarea of the FPGA that is over-utilized (i.e. illegal), which will mean that the blocks within this area must be spread to a region with more room. To find this over-utilized area, adjacent locations occupied by more than one block are repeatedly clustered together, until all clusters are bordered on all sides by non-over-utilized locations.

Next, the area is expanded in both vertical and horizontal axes until it is enough to accommodate all blocks that are retained there. The area is expanded as said until its occupancy (OA) divided by its capacity (CA) is less than a maximum utilization factor β, as in Equation 2.1.

$$OA/CA < \beta, where\beta <= 1. \tag{2.1}$$

The value β used in their experiments was 0.9.

Afterwards, yet still in the legalization process, they make two cuts to the design: a source cut and a target cut. The source cut refers to to the blocks being placed, and the target cut refers to the area into which the blocks are placed.

The source cut splits the blocks into two partitions, whilst the target cut splits the area into two subareas, and the blocks within each partition are spread to these subareas.

That is made to minimize the imbalance between the number of blocks in each partition and make the utilization of the subareas closer to equivalent between one another, minimizing the difference between them.

The term "utilization" (U) is defined as the occupancy (O) divided by the capacity (C) of the subarea (Equation 2.2).

$$U_{subarea} = O_{subarea}/C_{subarea} \tag{2.2}$$

.

For generating the source cut, cells are first sorted by location in x or y axis, depending on the orientation of the desired cut. After sorting the cells, a pivot is chosen in a sorted list, where all blocks to the left of it are assigned to the left/bottom partition, and all blocks to the right of the pivot are assigned to the right/top.

In Figure 2.2, the blocks on the right side of the image are to be placed inside the region to the left side. There's a source cut that splits blocks between the partitions as evenly as possible, and it's labeled "BAD". It is not possible to make a corresponding target cut so that $U_{subarea} <= 1$.

It is also possible to make bad cuts if choosing the target cut first (label "BAD" on the left. which does not leave enough room in the upper partition for the tall block on the right). They mitigate possible issues by allowing the target cut generation phase to perturb the source cut by moving blocks between partitions.

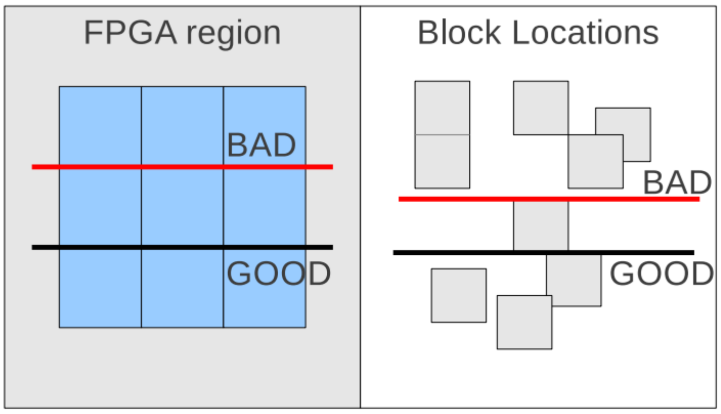


Figure 2.2: Source and target cuts. (GORT; ANDERSON, 2012)

The target cut labeled "GOOD" would be chosen because it minimizes the difference between the utilization of the subareas.

The disturbance to the source cut would be made by moving a block from the overutilized partition closest to the cutline to the underutilized partition, which would lead to a legal source/target cut combination.

Next, cells in subareas are spread to be distributed evenly. The spreading method used is proposed in SimPL, which splits the subarea into 10 same sized target bins and splits the cells into 10 same capacity source bins. Mapping cells from their original locations in their source bins to new locations after spreading in the target bins is acquired by using linear interpolation.

The processes mentioned above, both cut generation and spreading, are done recursively for each left and right subareas, alternating between x and y cut directions, until a single block remains in each subarea.

In case of not finding a legal cut, the recursion jumps up one level and switches to a largest-first greedy packing algorithm, placing blocks in non-increasing order into positions as close as possible to the ones given by the solver results.

One of two conditions must be met as a stopping criterion: HPWL for the legalized placement has stalled for a number $stall_{max}$ of iterations or the quality of the solved solution is close enough to the quality of the legalized solution, such that there's no need for further improvements.

They compare themselves to a tool called QUARTUS II from Altera (QUARTUS...,) for placing and routing Altera Cyclone II 90nm FPGA family (ALTERA...,). Table 2.1 shows the results of the comparison between the two placers when the flow is not timing-driven. "WL" columns report postrouted wirelength in micrometers (divided by 1000), "MHz" columns give the circuit speed in MHz, and "RT" columns report placement run-time in seconds.

	Quartus II			HeAP			Ratios		
	WL	MHz	RT	WL	MHz	RT	WL	MHz	RT
adpcm	145	24.8	21.0	180	23.3	1.8	0.80	1.06	11.7x
aes	133	24.6	8.0	138	22.9	2.5	0.96	1.07	3.2x
bf	96	34.7	3.0	99	36.2	1.2	0.96	0.96	2.5x
dfdiv	109	36.1	7.0	121	37.3	1.6	0.90	0.97	4.3x
dfsin	242	11.9	18.0	271	11.8	3.6	0.89	1.01	5.0x
gsm	110	14.5	4.0	110	15.9	1.9	1.00	0.92	2.1x
jpeg	321	9.1	25.0	366	8.4	3.4	0.88	1.08	7.4x
oc_des	111	110.0	4.0	110	114.6	1.4	1.01	0.96	2.9x
oc_video	85	71.3	3.0	75	73.2	0.7	1.14	0.97	4.4x
sha	110	53.4	6.0	114	49.7	2.0	0.96	1.08	3.1x
geomean	133	29.5	18.1	141	29.4	1.8	0.95	1.01	4.1x

Table 2.1: Comparison between HeAP and non timing-driven Quartus II flow. (GORT; ANDERSON, 2012)

The last three columns give the ratios of Quartus II to HeAP for wirelength and maximum frequency, and the speedup in runtime achieved by HeAP over Quartus II. The last row of the tables gives the geometric mean across all circuits. We see that HeAP offers a 4.1 speedup comparing to Quartus II's non timing-driven flow, at a 5% cost to wirelength and roughly flat maximum frequency.

2.1.2 StarPlace

StarPlace (XU; GRÉWAL; AREIBI, 2011) uses a new net model called *star+*, an extension of the star model. The new model is continuously differentiable, meaning minima can be found analytically, and incremental changes caused by block movement are computed in O(1). Star+ model addresses weaknesses of the more common routing cost approximations. Compares itself with HPWL and quadratic wirelength approximation. HPWL lacks accuracy on nets with more than three terminals and is not minimizable. With quadratic approximation, minimization is possible, but large nets are favoured.

The *star+* model is a linear-like objective function. It takes the distance from all blocks of the net to the net's center of mass, squares it, adds a positive number *phi*, takes the square root and multiplies by a factor *gamma*, which is a factor to compensate underestimation of wirelength. When phi is positive, the model is always differentiable. Benefits of this model are constant complexity update caused by block swapping and linearity, which addresses the problem of bias towards big nets. Using HPWL with this model, when compared to VPR, the current state-of-the-art placer based on simulated-annealing that will be mentioned next, resulted in lower critical-path delays and an average 9% runtime improvement.

An analytic approach is then presented using the star+ model. It iteratively solves a system of non-linear equations and partitions solution to converge towards a legal placement. Conjugate Gradient (CG) (HESTENES; STIEFEL, 1952) and successive over-relaxation (SOR) (HADJIDIMOS, 2000), two solvers for non-linear equations, are compared alongside VPR's simulated annealing.

The two solvers are outperformed by VPR in regards of wirelength, but they are much faster and reduced critical path delay when compared to VPR. In particular, the SOR method is 78% faster than VPR, has an increased wirelength of 1.1% and reduces critical path delay by 8.7%. The authors also highlight that wire-length is usually less important than critical path delay.

2.2 Stochastic: Simulated-Annealing Based

2.2.1 VPR: Versatile Place and Route

One of the most well known state of the art FPGA placement algorithms is VPR (BETZ; ROSE, 1997). The algorithm used is based on simulated annealing. The cost function is a linear congestion cost function, which is a sum of the horizontal and vertical spans of the nets' bounding boxes. The span is then divided by the average channel capacity and the result multiplied by a factor to prevent underestimation of net sizes.

The annealing schedule starts with a random placement of the circuit. Defining *N* as the number of blocks in the circuit, the algorithm performs *N* pairwise swaps of logic blocks, and computes the standard deviation of the cost over the *N* movements. To ensure any movement is possible at the start of the annealing, the initial temperature is set to 20 times the deviation.

Figure 2.3 shows superficially what VPR does in its flow.

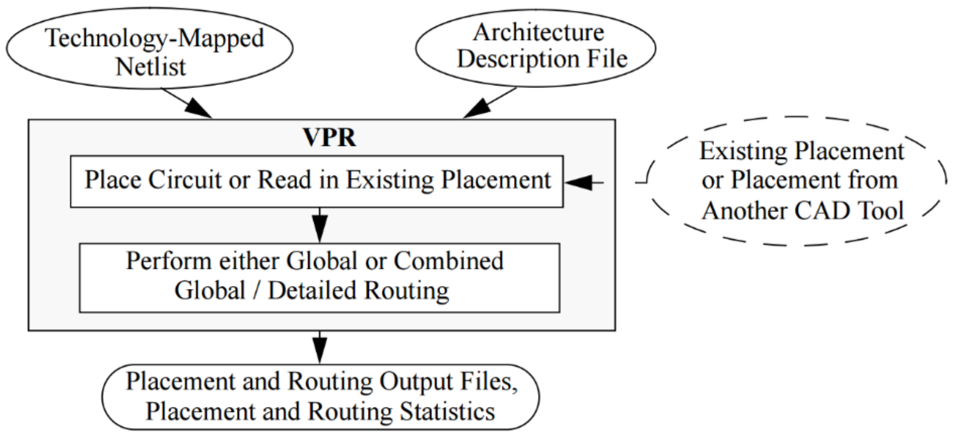

Technology-Mapped Netlist
Architecture Description File
VPR
Place Circuit or Read in Existing Placement
Existing Placement or Placement from Another CAD Tool
Perform either Global or Combined Global / Detailed Routing
Placement and Routing Output Files, Placement and Routing Statistics

Figure 2.3: VPR CAD flow. (BETZ; ROSE, 1997)

The number of moves for each temperature is 10 times N. The temperature is updated according to the rate of acceptance of the movements. A factor α multiplies the temperature after every 10*N moves, and it has a value that depends on the rate of acceptance of the movements performed, trying to keep the rate of acceptance close to 0.44 for as long as possible (because, according to (LAM; DELOSME, 1988), who developed a theoretically derived statistical annealing schedule, the optimum acceptance rate of proposed new configurations is 44%).

Also, a range limiter is present. Blocks can only be swapped with blocks closer than a distance D. This distance is updated alongside the temperature, and also depends on the rate of acceptance. The new limit is the old limit multiplied by a factor of $(1 - 0.44 + R)$, being R the rate of acceptance of the movements. With many movements being accepted, R is closer to 1, and the distance limit increases, and when it is closer to 0 the window becomes smaller.

VPR outperformed every comparable algorithm back in the time of its launching (1997) in number of tracks required for routing. It requires 10% less tracks when compared to TRACER (LEE; WU, 1997), the second best on the comparisons, and 13% less than the third runner up, which is VPR combined with SEGA (LEMIEUX; BROWN; VRANESIC, 1997) detailed placer.

It is worth mentioning that the benchmarks used for the comparisons are no bigger than 8383 logic blocks, which is too small for modern FGPAs.

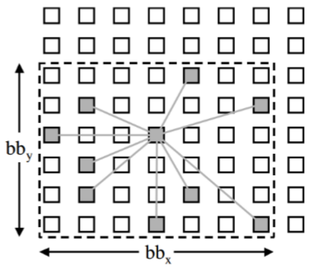
Although the classic version being too simple (only reaches homogeneous FPGAs) and being too slow for large circuits (runtime bottleneck of simulated annealing), VPR is important because it is one of the foundations of modern FPGA placement and because it still manages to outperform modern day algorithms in wirelength (even though more wirelength may not be a big issue if the signal that passes through the wires has a large frequency).

Many algorithms of nowadays consist partially of making some changes to VPR's structure, such as the next algorithm in 2.2.2.

2.2.2 Congestion Driven Placement

The work "A Congestion Driven Algorithm Placement For FPGA Synthesis" (ZHUO; LI; MOHANTY, 2006) aims to improve the standard wire cost function. The authors argue that nets that overlap cause congestion, so there must be a trade-off between net size and overlap. The new factor CC (congestion cost) penalizes logic block slots that are inside the bounding boxes of multiple nets. The introduction of such factor reduces the channel width by an average of 7.1% and critical path delay by 0.7% when compared to the default VPR. Figure 2.4 displays the cost function that was used formerly by VPR and Figure 2.5 displays the congestion factor they made for overlapping nets to join the former equation that is shown in Figure 2.4, which uses sempiperimeter-based logic.

The algorithm in Figure 2.6 is simulated-annealing based and uses the wiring cost function of VPR, with another factor that takes into account the congestion that happens when there are overlaps. There is a multiplication factor called CC, it has an exponent that is dynamically decreased as the temperature cools down.



$$WireCost = \sum_{i=1}^{N} q(i) * (bb_x(i) + bb_y(i))$$

where *bbx (i)* = the horizontal span of net ‘i’,
bby (i) = vertical span,
q(i) = a correction factor

because doesn't precisely estimate the cost for wiring inside the box, and for nets with more than 3 terminals

Figure 2.4: Semiperimeter-based logic cost function applied to VPR.

$$CC = \left(\frac{\sum_{i,j} U_{i,j}^2}{nx \cdot ny} \Big/ \left(\frac{\sum_{i,j} U_{i,j}}{nx \cdot ny} \right)^2 \right)^k ,$$
$$1 \le i \le nx, 1 \le j \le ny,$$

$$\sum_{i=1}^{n} a_i^2/n \ge \left(\sum_{i=1}^{n} a_i/n\right)^2$$

Provided that this inequality is always true

CC >= 1 always

where *Ui,j*= the number of bounding boxes that share *CLBi,j*
k = small positive integer,

Figure 2.5: Congestion factor that multiplies the wire cost now

Algorithm 1: Computing Bounding Box Cost

```
procedure COMPBBCOST($r$)
  CLEARBLKUSAGE($U$)
  $cost \leftarrow 0$
  for $n \leftarrow 0$ to $num\_nets$
    GETBOUNDINGBOX($bb[n]$)
    for $i \leftarrow bb[n].xMin$ to $bb[n].xMax$
      for $j \leftarrow bb[n].yMin$ to $bb[n].yMax$
        $U[i,j] \leftarrow U[i,j] + 1$
    $cost \leftarrow cost +$ GETNETCOST($n$)
  $congestion \leftarrow$ CONGESTIONFUNC($r, U$)
  return $(cost * congestion)$

procedure CONGESTIONFUNC($r, U$)
  $sum \leftarrow 0$
  $sos \leftarrow 0$
  $levels \leftarrow maxExp - minExp + 1$
  for $i \leftarrow 1$ to $nx$
    for $j \leftarrow 1$ to $ny$
      $sos \leftarrow sos + U[i,j] * U[i,j]$
      $sum \leftarrow sum + U[i,j]$
  $base \leftarrow sos * nx * ny / sum^2$
  $k \leftarrow minExp + (int)((r-1) * levels/rMax)$
  return $(base^k)$
```

Figure 2.6: Simulated annealing based algorithm (ZHUO; LI; MOHANTY, 2006).

Table 2.2 was extracted from (ZHUO; LI; MOHANTY, 2006), and it shows the results comparing this algorithm ("ours" in the table) with VPR and VPRb, which is a version of VPR that is run without the timing-driven constraints in the cost function, so it just focuses on congestion and is similar to the algorithm in Figure 2.6. CP is critical path delay, to show that this algorithm doesn't make the original solution worse, and CW is channel width for routing.

Circuit	VPR		VPRb				Ours					
	CP	CW	CP	Ratio	CW	Ratio	CP	Ratio	CW	Ratio	minExp	maxExp
tseng	55.62	8	71.56	1.2866	7	0.875	53.65	0.9646	7	0.875	1	2
apex4	93.25	14	131	1.4048	13	0.9286	92.75	0.9946	13	0.9286	1	3
misex3	95.74	12	107.7	1.1249	11	0.9167	82	0.8565	11	0.9167	1	3
dsip	70.79	7	82.14	1.1603	6	0.8571	66.62	0.9411	6	0.8571	1	4
ex1010	195.3	12	206.1	1.0553	10	0.8333	182.6	0.935	11	0.9167	2	4
clma	228.4	14	243.7	1.067	13	0.9286	213.7	0.9356	13	0.9286	3	5
diffeq	101.5	8	93.38	0.92	7	0.875	63.44	0.625	8	1	1	3
spla	203.1	15	181	0.8912	14	0.9333	158.9	0.7824	15	1	1	4
seq	95.72	12	109.7	1.1461	12	1	81.65	0.853	12	1	1	3
elliptic	137.2	12	187.1	1.3637	11	0.9167	126.8	0.9242	11	0.9167	1	4
pdc	193.5	19	211.3	1.092	17	0.8947	198.9	1.028	17	0.8947	2	4
frisc	135	14	176	1.3037	12	0.8571	142.7	1.057	13	0.9286	1	4
bigkey	78.56	7	88.7	1.1291	7	1	99.46	1.266	6	0.8571	1	4
des	121.2	8	114.7	0.9464	8	1	124.2	1.0248	7	0.875	1	4
alu4	82.1	11	121.7	1.4823	10	0.9091	98.93	1.205	10	0.9091	1	3
apex2	90.02	12	134.8	1.4974	11	0.9167	111.8	1.242	11	0.9167	1	3
ex5p	84.06	15	129.4	1.5394	13	0.8667	87.1	1.0362	13	0.8667	1	1
s298	135.7	8	204	1.5033	8	1	148.5	1.0943	8	1	1	4
s38417	103.3	8	156.9	1.5189	7	0.874	112.3	1.087	8	1	2	5
s38584.1	97.92	8	126.7	1.2939	8	1	98.21	1.003	8	1	2	5
Ave.				1.2363		0.9192		0.9928		0.9294		

Table 2.2: Results of the comparison between Congestion Driven Placement, VPR and VPRb (ZHUO; LI; MOHANTY, 2006).

2.3 Simultaneous Timing Driven Clustering and Placement (KARYPIS; KUMAR, 2000) for FPGAs

The authors explain that an efficient clustering requires placement information, which is possible if you optimize both simultaneously. It achieves better results than VPR alone in wirelength as well as critical path delay.

Table 2.3 displays the results within a table with respect to wirelength, which is the main goal of the work proposed here, explained in more detail in Chapter 4.

Circuit	VPR	SCPlace	Improvement
ex5p	112.47	98.91	13.71%
apex4	113.639	101.45	12.02%
misex3	123.616	105.63	17.02%
tseng	94.9456	70.57	34.55%
alu4	123.03	104.68	17.53%
dsip	195.544	138.69	41.00%
seq	173.641	152.99	13.50%
diffeq	132.271	107.2	23.39%
apex2	190.324	165.73	14.84%
s298	166.899	164.96	1.17%
des	278.122	257.286	8.10%
bigkey	171.986	196.81	-12.61%
spla	426.227	352.635	20.87%
elliptic	359.011	284.821	26.05%
ex1010	463.618	364.774	27.10%
pdc	704.286	580.969	21.23%
frisc	584.732	482.289	21.24%
s38584.1	576.457	354.476	62.62%
s38417	696.701	494.657	40.85%
clma	1701.02	1271.88	33.74%
Ave.			21.89%

Table 2.3: Results of the comparison between Simultaneous Timing Driven Clustering and Placement (SCPlace) and VPR in terms of wirelength, using the total weighted bounding box wirelengths as the only optimization objective, in which both algorithms are given initial clustering solutions. They outpreform VPR in terms of wirelength improvement by 22% on average.(CHEN; CONG, 2004).

2.4 Summary

Table 2.4, below, summarizes the main characteristics of the works that were mentioned previously in Section 2.1, 2.2 and 2.3.

Parameter	VPR	HeAP	Congestion Driven	Simultaneous Timing Driven
Packing Technique	V-Pack (clustering BLEs)	Omitted, but invoked in few cases	V-Pack	During placement
Placement Technique	Simulated Annealing (stochastic)	Based on Simpl, bipartitions	VPR with cost insertion	Quadratic programming

Table 2.4: Summary of the main characteristics of the works that were cited previously.

2.5 HMETIS (KARYPIS et al., 1999) and (KARYPIS; KUMAR, 2000)

HMetis is a software package for partitioning large hypergraphs that implements the concept of multilevel partitioning based on the partitioning described in (KARYPIS et al., 1999) and (KARYPIS; KUMAR, 2000). Figure 2.7 displays the difference between a traditional partitioning approach and a multilevel one. It is said to deliver partition solutions that are consistently better than KL (KERNIGHAN; LIN, 1970), FM (FIDUCCIA; MATTHEYSES, 1988) and other popular partition algorithms would.

HMETIS's step by step flow follows what is mentioned above, that is, a phase of coarsening for producing a sequence of successive smaller graphs, an initial partition solution for the coarsened graph (hMETIS uses the algorithm of multiple random bisections followed by the Fiduccia-Mattheyses(FM) refinement algorithm in this step), successive uncoarsening and partition refinement of the graph, and, finally, a refinement step which they call "V-cycle".

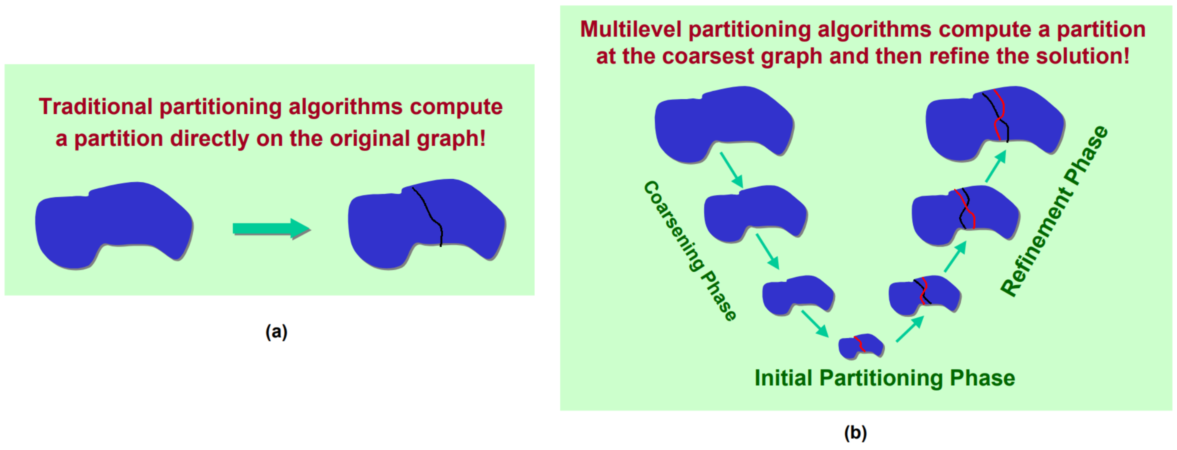

Traditional partitioning algorithms compute a partition directly on the original graph!
(a)
Multilevel partitioning algorithms compute a partition at the coarsest graph and then refine the solution!
Coarsening Phase
Refinement Phase
Initial Partitioning Phase
(b)

Figure 2.7: (a) Traditional partitioning algorithms and (b) multilevel partitioning algorithms. (HMETIS. . . ,)

The version of HMetis utilized in this work is the last version, published in 2007, for which the developers did not provide full disclosure regarding the changes made in the "V-cycle" refinement algorithm, but the first version of this algorithm is implemented as follows: it consists of two phases, the coarsening and the uncoarsening phase. The coarsening phase preserves the initial partitioning that is input to this algorithm. the groups of vertices that are combined to form the vertices of the coarse graphs correspond to vertices that belong only to one of the two partitions. As a result, the original bisection is preserved through out the coarsening process, and becomes the initial partition from which we start performing refinement during the uncoarsening phase. The uncoarsening phase is identical to the uncoarsening phase of the multilevel hypergraph partitioning algorithm described earlier. It moves vertices between partitions as long as such moves improve the quality of the bisection.

The 2007 version also carries these changes: a new set of coarsening schemes that lead to better and more stable cuts, better refinement routines for circuits with non-unit cell weights, floating point numbers for cell and net weights, support for 64 bit architectures and KPM-based k-way refinement routines.

2.6 Top 3 Placement Algorithms of ISPD 2016 Competition

2.6.1 GPlace

Gplace (PATTISON et al., 2016) consists of two tools altogether: GPlace-pack and GPlace-flat. I was based on the tool StarPlace, mentioned in Subsection 2.1.2, developed by the same authors. GPlace-pack is an adaptation of StarPlace for the Ultrascale $^{\text{TM}}$. In this algorithm, before packing stage, cells are spreaded randomly, and a simple optimization aiming wirelength is applied, in a similar fashion to Star+, but trying to maintain the cells near the regions they belong to, for example, LUTs near slice slots, and RAMs near RAM slots.

After that the circuit region is split into bins. Inside each bean, a cell is selected to be the seed of a slice. The other cells in the bin are added to the slice in case they do not violate any legalization constraint and do not contribute to the slice with many external pins (as the cost for connections external to the slice is not zero as it is inside the slice). The candidates for being selected to be placed within the slice are ordered by affinity to the slice, meaning, in this case, the number of edges (external connections) between the candidate and the slice. From the cells of the bins that were not clusterized into a slice, another cell is selected to be the seed of a new slice, and so on, until there are no more unclusterized slice-type cells.

GPlace-pack estimates routing cost using wirelength per area, in which, the wire length model used is the weighted half-perimeter wirelength of VPR, the algorithm mentioned in Subsection 2.2.1.

After the Contest, they developed GPlace-Flat, which has no explicit packing stage.

2.6.2 UTPlaceF

UTPlaceF (LI; DHAR; PAN, 2018) was the algorithm that took the first place at the ISPD Contest. It has, as main contributions, a packing algorithm that uses placement information, congestion control techniques, and a detailed placement that improves wirelength without worsening routability.

The flow starts with a quadratic placement, an approximate legalization, then, optimizes wirelength and improves density. After a satisfactory solution is reached, one in which wirelength in limited by a certain threshold, detailed placement begins, running three iterations of it, each one beginning with a congestion estimate. Based on the estimate, cells that occupy congested regions are swollen, occupying a bigger area. With this new area, another iteration of quadratic placement is done, and the solution goes under approximate legalization. At this moment, cells that are non slice-type, meaning RAMs, DSPs and IOs are legalized and moved with the objective of preserving the global density.

After three iterations, if HPWL is within a certain threshold, packing stage begins. If not, three more iterations are performed, until HPWL is, finally, limited by the threshold. In packing, each LUT whose fan out is one, a connection with an FF, makes a LUT-FF pair with it. The pairs formed are combined when they respect the legalization rules, and then slices that are disjoint are merged. If packing extrapolates a maximum amount of slices, it is remade with the relaxation of some criteria, until the amount of slices is under the limit.

After packing another quadratic placement is performed, this time, considering slices as cells, abstracting LUTs and FFs, and considering the placement initial information acquired at the beginning of the flow. Finally, the placement final solution is legalized, giving priority to congested regions.

2.6.3 RippleFPGA

RippleFPGA (PUI et al., 2016) consists of partitioning, packing, global placement with congestion estimation, window-based legalization and routing resource-aware detailed placement, in a fashion similar to ASIC placement. It starts with partitioning, considering the cells as a set of vertices V and the connections among them as a set of edges E of the graph $G = (V, E)$, setting the weights of the edges $w_{u,v}$ to be the number of nets that connect vertices u and v.

They partition the graph G into two clusters recursively until the sizes of the clusters are too small (set as 25% of the total number of vertices of G, and the minimum cut should be, at most, 5% of the total number of nets). The resulting clusters are stored as leaves of a tree called clusterTree.

In global placement, the initial position of each cluster is determined by the connections to IO and the connections between the clusters. Since the connections to IO have small contribution in the routed wirelength, their reallocation algorithm only minimizes the disturbance with respect to the connections among clusters. A few iterations of global placement (quadratic) are performed with legalization for RAM and DSP cells in between the process, until finding the physical location of each cell considering their connections globally. The global placement phase is also congestion driven, utilizing a congestion map for the circuit, build according to the net bounding box, estimating routing congestion by how many nets may use the routing resources of that site, by accumulating the number of bounding boxes overlapping a site.

In the stage of packing, they group LUTs and FFs into BLEs (basic logic element), each one consisting of one single LUT and one or more FFs that are connected to it. After that, they try to merge two BLEs into one if they respect certain constraints.

After this stage, they perform the legalization only of the BLEs into the slices, deciding when to place the whole BLE into a slice as a single block and when to split it into its basic components, LUT and FF cells, and place them by themselves within slices. If they decide to keep the BLEs as blocks, they will mat them into their positions in the next stage, detailed placement. The final stage is detailed placement, they perform BLE assignment to slices and slice swapping.

2.6.4 Summary

Table 2.5, below, summarizes the main characteristics of the works from the Top 3 that were mentioned previously in this section and also UFRGSPlace, which took the fourth place.

Parameter	UTPlaceF	RippleFPGA	GPlace	UFRGSPlace
Packing Technique	Position based merges	Position based merges	Bin based	Net based
Placement Technique	POLAR (quadratic)	Ripple (quadratic)	StarPlace (quadratic)	Quadratic
Congestion Estimation	Density based cell bloating	Weighted bounding box per GCell	Wire-length per area	HPWL

Table 2.5: Summary of the main characteristics of the works that were cited previously.

3 PROJECT RULES AND EVALUATION METHOD

In this chapter, the project rules and evaluation method for the ISPD Contest are explained. The tool that was developed in this work was tested using the benchmark suite provided by this contest, therefore, supports all the design constraints that the contest proposed.

The ISPD 2016 Contest (ISPD. . . , 2016) was about routability-driven FPGA placement for Xilinx's Ultrascale architecture. The contenders should provide a placement solution concerning routability, which not only can consist of minimizing wirelength, but also optimizing the use of routing resources, especially having in mind that some congested areas may appear after routing. The benchmark suite used to obtain all the results shown in this work was the one provided for this contest.

Regarding routability, CAD tools for physical design must perform with balanced use of routing resources to be up to date with competitive high-performance architectures of nowadays. While the wider bus implementation in FPGAs that aid in applications that demand large data throughput might lead to the need for a lower system clock frequency, significant timing-closure challenges now arise due to a lack of routing resources required to support systems with wide buses. With this in mind, the bottleneck of the routing problem regarding throughput becomes the lack of routing resources (especially, wider bus wire resources in antiquated FPGAs) other than the clock frequency.

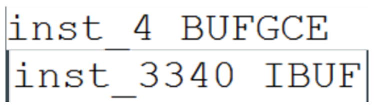
Some FPGA vendors use algorithms based on simulated annealing, which are blind to global design metrics (otherwise, the algorithms would be way too slow), hence, these place-and-route algorithms contribute to aggravate this situation of being up to date with the demands of the applications.

When modern day architectures such as Ultrascale are taken into consideration and heterogeneity becomes present, not only there might be a need for wider buses as mentioned, but also the number of cells may increase dramatically. The concern about the scalability of the flow is, as well, a sign of the upcoming obsolescence of a pure simulated annealing approach (KIRKPATRICK; GELATT; VECCHI, 1983). The top five tools on ISPD 2016 Contest (ISPD. . . , 2016) all implemented quadratic flows for placement.

3.1 Inputs, Outputs and Metrics

As input, the organizers provided a benchmark suite using enhanced bookshelf format. Each design in the benchmark suite had the following files:

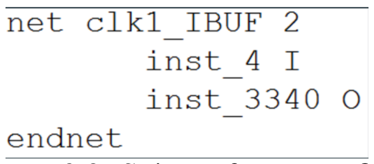
- design.aux: auxiliary file listing the files as the input of the placer;
- design.nodes: as shown in Figure 3.1, a snippet from the file, specifies instance/object attributes (name, master cell and dimensions etc.);



```
inst_4 BUFGCE
inst_3340 IBUF
```

Figure 3.1: Snippet from nodes file.

- design.nets: as shown in Figure 3.2, a snippet from the file, specifies the set of nets in the design;



```
net clk1_IBUF 2
       inst_4 I
       inst_3340 O
endnet
```

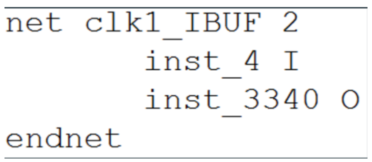
Figure 3.2: Snippet from nets file.

- design.pl: specifies the location of the instances/objects including I/O and fixed instance locations;
- design.scl: as shown in Figure 3.3 and Figure 3.4, (enhanced) extended from the original bookshelf format to represent FPGA placeable area and architecture specific placement information;
- design.wts: currently unused as all instances/objects and nets have the same weight;
- design.shapes: (new to the bookshelf format, not actually provided in the benchmarks) specifies the shapes/groups of the instances/objects that are placed together.

The placement result for each benchmark was then evaluated by the routing tool Vivado Router (VIVADO. . . , 2016) by Xilinx Inc.

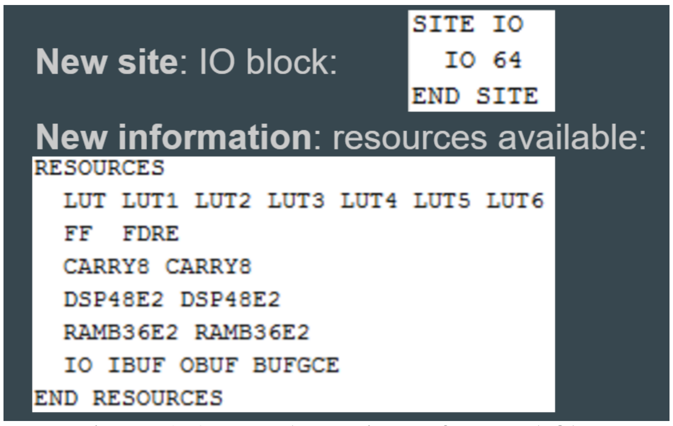
These were the evaluating criteria:

1. Legality of the placement
2. Routability of the placement (routable or not within a time-out limit, 12 hours)
3. Routed total wire-length
4. Placement runtime



```
net clk1_IBUF 2
        inst_4 I
        inst_3340 O
endnet
```

Figure 3.3: Snippet from scl file.



New site: IO block:

```
SITE IO
  IO 64
END SITE
```

New information: resources available:

```
RESOURCES
  LUT LUT1 LUT2 LUT3 LUT4 LUT5 LUT6
  FF  FDRE
  CARRY8 CARRY8
  DSP48E2 DSP48E2
  RAMB36E2 RAMB36E2
  IO IBUF OBUF BUFGCE
END RESOURCES
```

Figure 3.4: Another snippet from scl file.

Metric is in Equations 3.1 and 3.2, where *RuntimeFactor* is between -0.1 and 0.1, there is 1% scaling factor for every runtime reduction or addition of 10% against the median runtime of all place and route solutions. Hence, the runtime factor is a means of penalizing or rewarding a solution by a character of up to 10% regarding its runtime. Routed wirelength is the base of the score.

$$Score = RoutedWirelength * (1 + RuntimeFactor) \tag{3.1}$$

$$RuntimeFactor = -(Runtime - MedianRuntime)/10.0 \tag{3.2}$$

3.2 Rules and Main Information

As previously mentioned, placement must be implemented for a heterogeneous architecture, where the space for each CLB is called "SLICE", and "SITE" for BRAM and DSP blocks. The dimensions for each FPGA that was benchmarked are of 168 rows and 480 columns.

Aside from having to assign row-aligned positions to the components, they must as well be inserted in the right places for their type, for instance, a LUT cell must be inserted in a slot called "SLICE", a BRAM cell must be inserted in a slot for BRAMs, not in a "SLICE". That's why the legalization step is more complex than it would be for ASICs (just the general concept of legalization, as, of course, there are different constraints that may be applied for each, which would increase the difficulty). There is one specific kind of slot for each type of cell, so there is not as much freedom to move as it would be in a simple case scenario legalization for an ASIC.

As for the FPGA components, DSP cells will go inside the DSP site, IO cells will go inside the IO site, and so on, except for the "SLICE", where CLB components will fit: there are three different types of cell that go there, which are FLIP-FLOPs, LUTs and CARRY8s.

The step in which we assign "SITE" positions for FLIP-FLOPs, LUTs and CARRY8s, clustering them, is called "PACKING". This step must be performed in agreement with "SLICE" legality, referring to internal "SLICE" connectivity rules.

Each CLB "SLICE" contains 33 different slots that cells should be assigned to: 16 for LUTs, 16 for FFs and 1 for CARRY8. Below, the packing rules:

- **LUT Packing**
  - If implementing a 6-input LUT with one output, one can only use LUT 1 (leaving LUT 0 unused) or LUT 3 (leaving LUT 2 unused) or ... or LUT 15 (leaving LUT 14 unused).
  - If implementing two 5-input LUTs with separate outputs but common inputs, one can use LUT 0, LUT 1 or LUT 2, LUT 3 or ... or LUT 14, LUT 15.
  - The above rule of coming LUTs with separate outputs but common inputs, holds good for 5-input LUTs (as mentioned above) or fewer input LUTs as well.
  - If implementing two 3-input (or fewer input) LUTs together (irrespective of common inputs), one can use LUT 0, LUT 1 or LUT 2, LUT 3 or ... or LUT 14, LUT 15.
- **FF Packing** , with distribution shown in Figure 3.6
  - All FFs can take independent inputs from outside the SLICE, or outputs of their corresponding LUT pair (FF 0 can take LUT 0 or LUT 1 output as input, ..., FF 15 can take LUT 14 or LUT 15 output as input).
  - All can be configured as either edge-triggered D-type flip-flops or level-sensitive latches. The latch option is by top or bottom half of the SLICE (0 to 7, and 8 to 15). If the latch option is selected on an FF, all eight FFs in that half must be either used as latches or left unused. When configured as a latch, the latch is transparent when the clock input (CLK) is High.
  - There are two clock inputs (CLK) and two set/reset inputs (SR) to every SLICE for the FFs. Each clock or set/reset input is dedicated to eight of the sixteen FFs, split by top and bottom halves (0 to 7, and 8 to 15). FF pairs (0,1 or 2,3 or ... or 14,15) share the same clock and set/reset signals. The clock and set/reset signals have pro-

grammable polarity at their slice inputs, allowing any inversion to be automatically absorbed into the CLB. Figure 3.5 displays the distribution of signals.

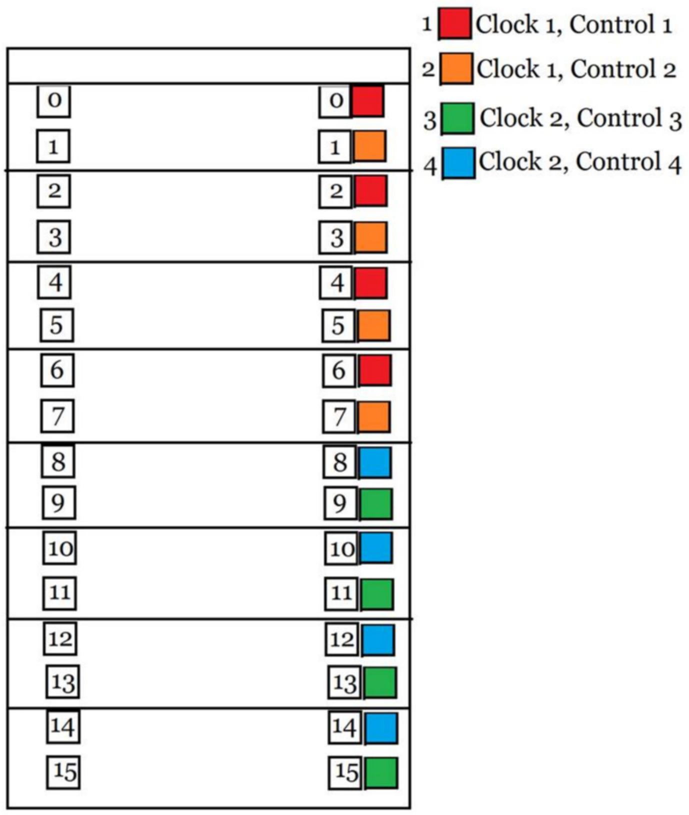


Figure 3.5: FFs clock and control distribution.

- There are four clock enables (CE) per SLICE. The clock enables are split both by top and bottom halves, and by the two FFs per LUT-pair. Thus, the CEs are independent for: FF 0, FF 2, FF 4, FF 6 , FF 1, FF 3, FF 5, FF 7 , FF 8, FF 10, FF 12, FF 14 , FF 9, FF 11, FF 13, FF 15.
  When one storage element has CE enabled, the other three storage elements in the group must also have CE enabled. The CE is always active High at the slice, but can be inverted in the source logic. For FFs without any net connections to CE pin, Vivado ^{TM} software automatically inserts a constant 1, so they are not to be combined up with other FFs that have their CE pins connected
- The two SR set/reset inputs to a SLICE can be programmed to be synchronous or asynchronous. The set/reset signal can be programmed to be a set or reset, but not both, for any individual FF. The configuration options for the SR set and reset functionality of a register or latch are: No set or reset, Synchronous set (FDSE primitive), Synchronous reset (FDRE primitive), Asynchronous set (preset) (FDPE

primitive), Asynchronous reset (clear) (FDCE primitive). The SR set/reset input can be ignored for groups of four flip-flops (the same groups as controlled by the CE inputs). When one FF has SR enabled, the other three FFs in the group must also have SR enabled.

- The choice of set or reset can be controlled individually for each FF in a SLICE. The selection of synchronous (SYNC) or asynchronous (ASYNC) set/reset (SYNC_ATTR) is controlled in groups of eight FFs, individually for the two separate SR inputs.

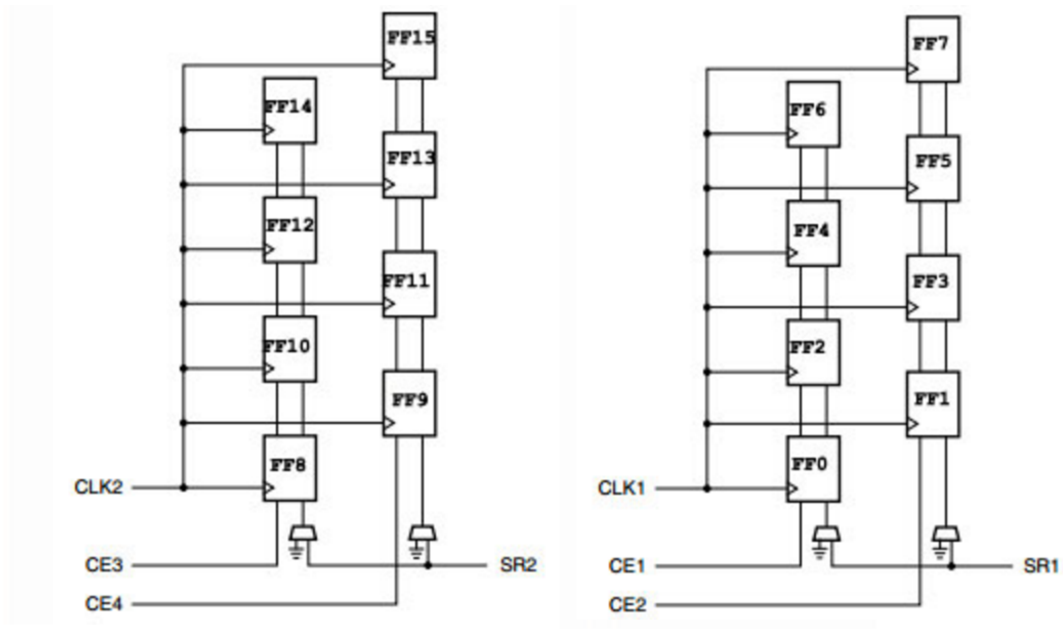


Figure 3.6: FF Packing (ISPD…, 2016).

Figures 3.7 and 3.8 illustrate a bit more how this process applies. It is important to pack the slices efficiently, because the routing cost inside each slice is zero. Figure 3.7, associates the occupation of the slices with the wirelength, being the right side's packing distribution denser, having more than one slice fully packed, against the more sparse packing from the left side, and, assumingly, more wirelength.

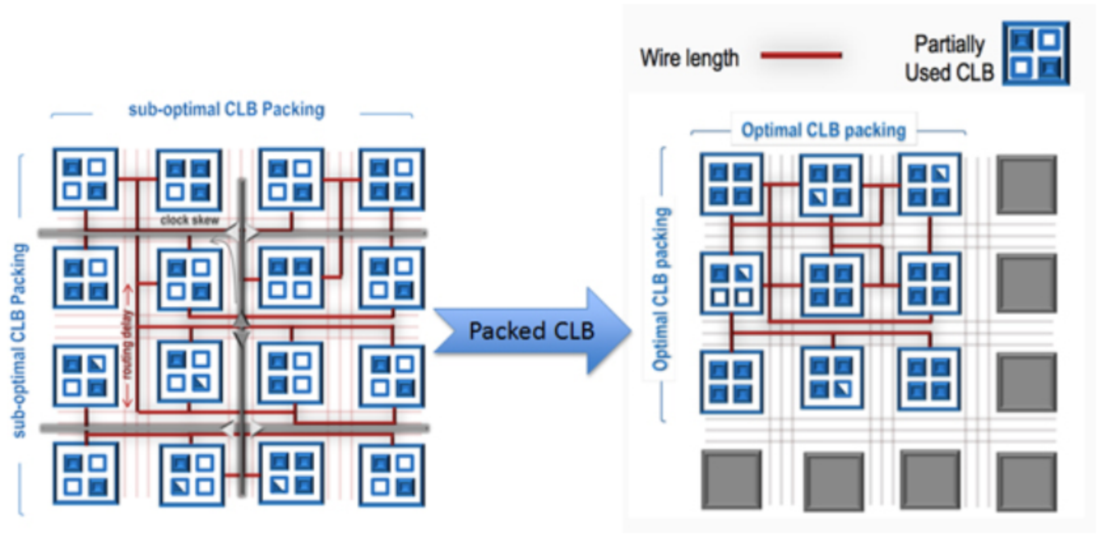


Figure 3.7: Packing (XILINX..., 2014).

In Figure 3.8, a LUT6 is in the slot 0. As it is a LUT6 and, therefore, uses 6 inputs, the other LUT in the pair (slot 1) must be left unused. Also, LUT3 must only drive up to two flip-flops inside the same slice, and they must be accommodated in the same LUT-FF pair that LUT3 is at.

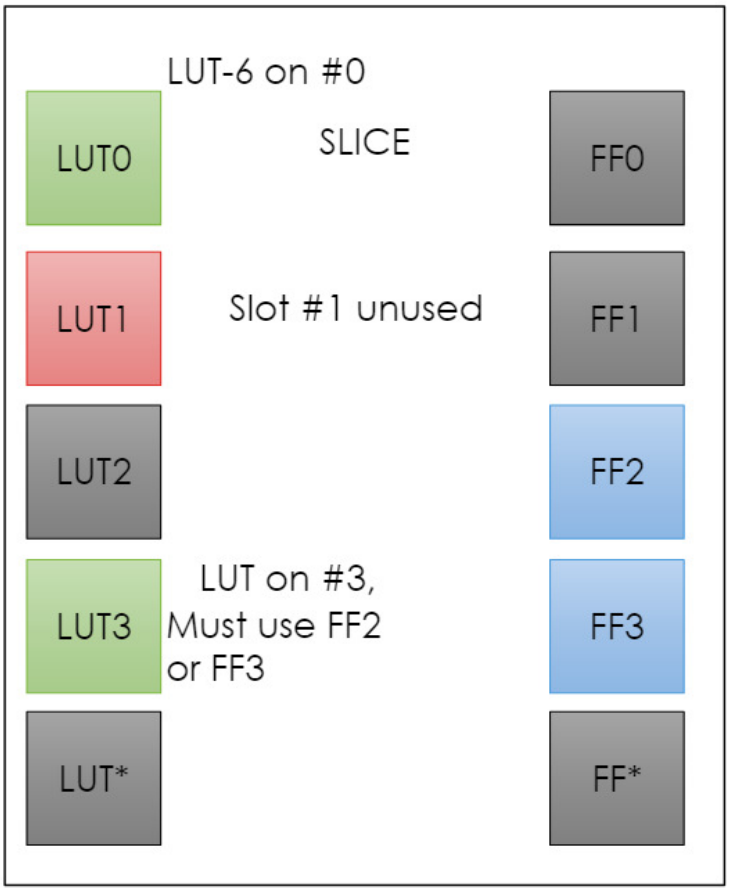


Figure 3.8: Some of the LUT rules.

3.3 Benchmark Information

Table 3.1 show how many cells of each type there are in the benchmarks of the ISPD 2016 Contest. For example, in benchmark 2, there are 100 DSP cells (DSP48E2), 100 RAM cells (RAMB36E2), 98K LUTs, 74K flip-flops (FDRE) and 150 I/O cells (IBUF and OBUF). The benchmarks have a much more significant number of "SLICE" type cells, which means that, as all "SLICE" cells need to be packed, the packing stage has a high impact on the placement solution.

Benchmark	#LUT	#FF	#RAM	#DSP	#IO
FPGA01	49K	55K	0	0	150
FPGA02	98K	74K	100	100	150
FPGA03	245K	170K	600	500	400
FPGA04	245K	172K	600	500	400
FPGA05	246K	174K	600	500	400
FPGA06	345K	352K	1000	600	600
FPGA07	344K	357K	1000	600	600
FPGA08	485K	216K	600	500	400
FPGA09	486K	366K	1000	600	600
FPGA10	346K	600K	1000	400	600
FPGA11	467K	363K	1000	400	600
FPGA12	488K	602K	600	500	400

Table 3.1: Xilinx benchmark statistics.

4 UFRGSPLACE PLACEMENT ALGORITHM

This algorithm is divided into two flows, almost identical to one another. Both have the same stages of placement and packing explained in detailed below, with the difference that the 2018 version, outlined in Section 4.6, implements the partitioning algorithm HMetis for delivering partitions of cells to the packing stage instead of nets, as in the tool developed for the ISPD 2016.

4.1 Packing

Upon parsing the input files, the tool has access to all cells and nets. However, for most of the cases, cells aren't directly placeable into the circuit. The complex cells - that is, RAM and DSP blocks - can be directly placed, but all LUTs, FFs and IO cells must be packed first. The actual structures that can be placed for these cases are slices and IO blocks. The IO block is elementary: 64 IO cells are allowed inside it, with no legalization issues at all. The slices, on the other hand, are very intricate and troublesome when it comes to legalization. There are 16 slots for LUTs, another 16 slots for FFs and one slot for a CARRY8.

The good aspect of packing is that routing inside slices is entirely free when it comes to resource usage. That means an algorithm that can effectively deal with legality and generate an outstanding packing result reduces the difficulty of the placement dramatically, creating slices with reasonable connectivity and a small number of placeable objects. After all cells are packed, the actual placement can take place.

4.2 Placement

Given a set of blocks - which can be RAM, DSP, IO blocks or slices -, a placement algorithm is used to determine in which slot every block will be placed. There are specific slots for each type of block. This could effectively make legalization heavily interfere with the initial placement it was given, which is very undesirable. An example of a situation that could potentially happen when this isn't taken into account is a DSP block that happened to be placed in the middle of a "slice-only" region. It will suffer a rather large displacement towards a valid position, increasing dramatically the routing resources necessary to route its nets - especially when one considers that it was initially placed in that specific region because it is connected to cells there.

With this in mind, we developed a placement flow that consists of packing, placement and legalization.

4.3 2016 Contest Placement Flow

Initially, without taking into account the actual architecture and clusters, we perform a quadratic placement. Given the result of this initial placement (seen in Figure 4.1), we legalize all complex cells (RAM, IO and DSP), trying to move all of them to positions that are nearest to the ones they had in the initial placement solution, as seen in Figure 4.2.

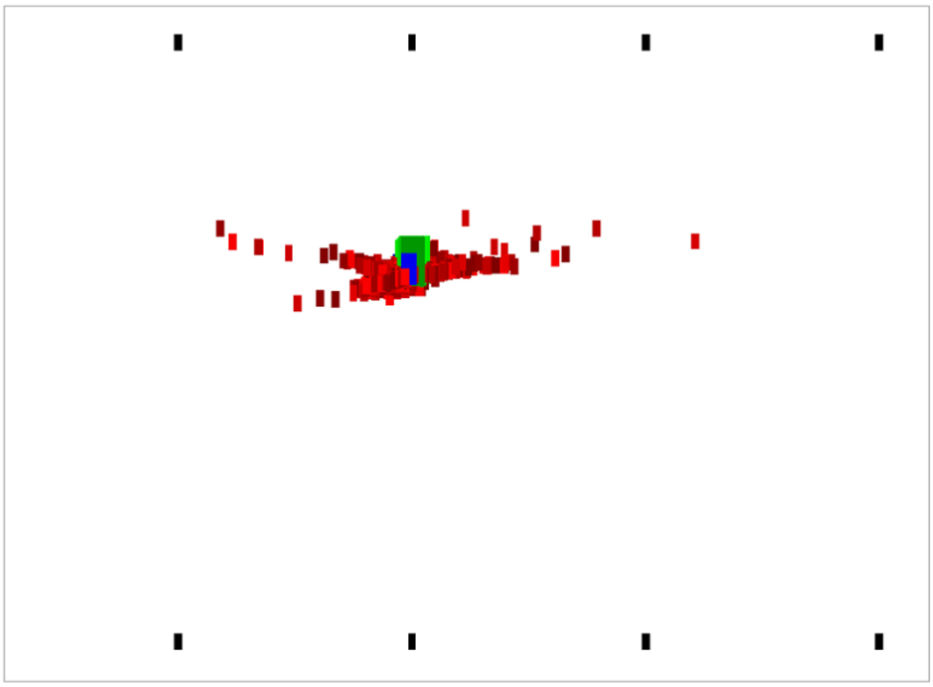


Figure 4.1: Initial quadratic placement. Rectangles in black are fixed IO cells, rectangles in red are FF and LUT cells, blue rectangles are DSP and green rectangles are RAM.

The objective with this sequence of steps is making sure the lack of freedom of the position of these cells is compensated by placing them first, so that the other cells - especially slices, which have much more freedom - can be placed closer to the complex cells with which they share connections. We also spread the cells that were attracted to a single region after quadratic placement. We tried to spread the cells radially so that they would move equivalently, and we used the clique model for nets with up to 4 nodes and the star model for bigger nets (Figure 4.3).

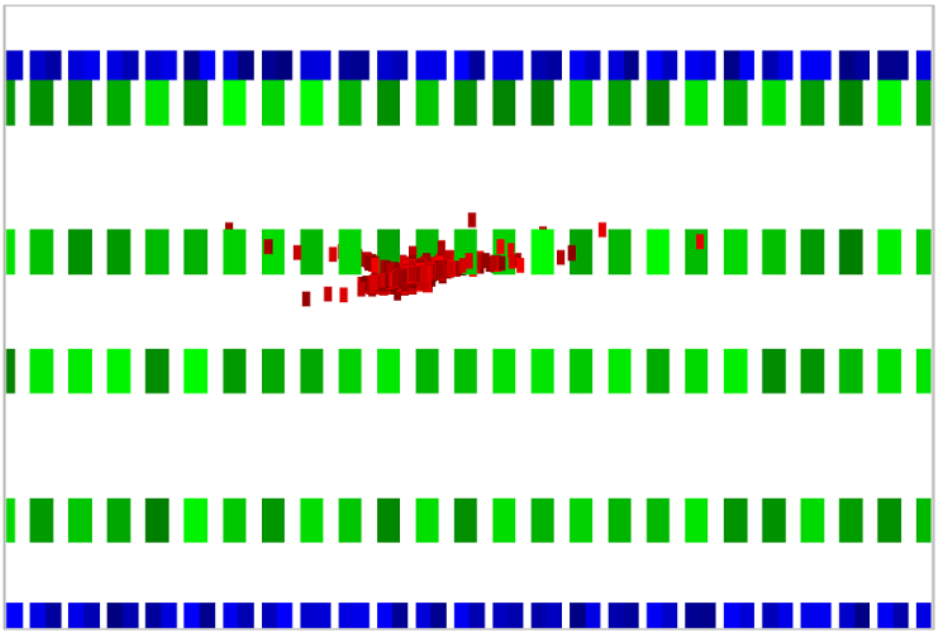


Figure 4.2: Legalization of DSP and RAM..

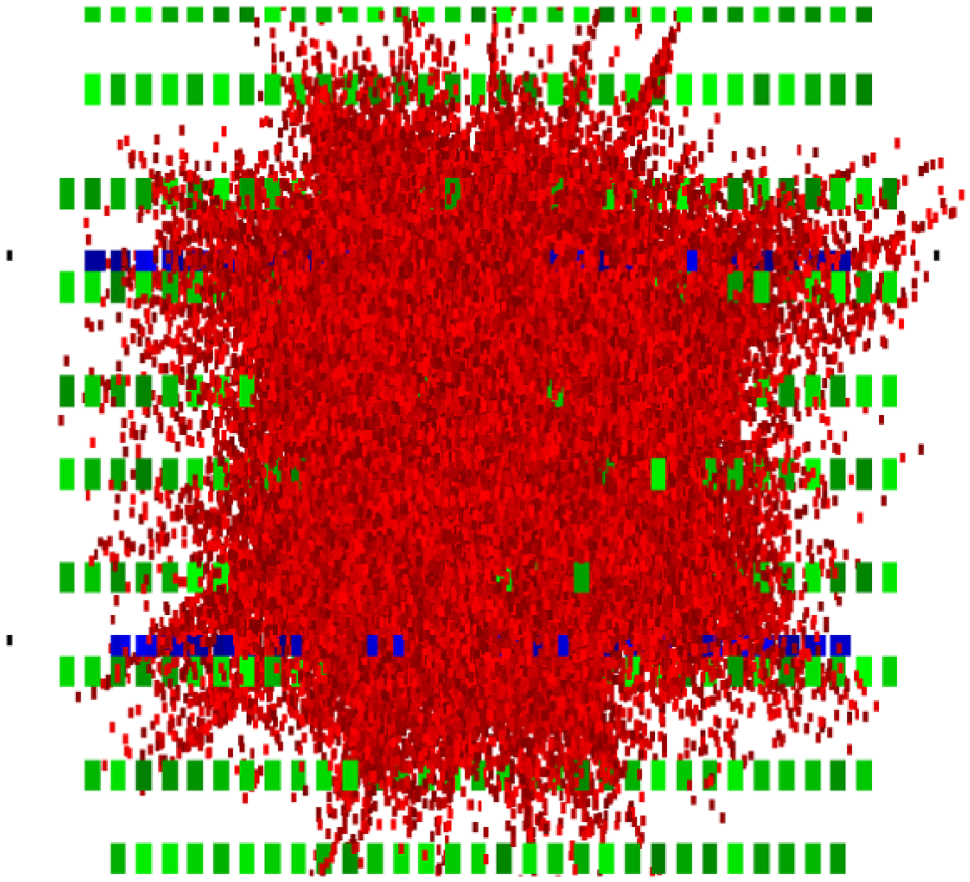


Figure 4.3: Radial spreading of slice cells during a quadratic placement flow.

After this initial step, a new quadratic placement is performed (similar to the first one, but only with slice cells) and a packing flow. The structure of a slice is fundamental to understand the legality constraints. The idea of our particular flow is removing whole nets from the circuit when possible. To accomplish this, the whole net must be inside a single slice, which is difficult because of legality constraints.

The nets are sorted in decreasing size because we want to fit a whole net into as few slices as possible (remembering that the routing cost inside a slice is zero). Starting with the smallest nets will allow this to occur more often than not. We used two pointers to manage the list of slices we are using when packing a net. These two pointers allow for quick access to a slice that has a good chance of being suitable to fit the cell. Not needing checking on every slice is important because it would be too slow for the bigger nets. When the cell could not be packed, due to the legality or the slice being full, the pointer points to the next slice, be it an already partially used slice or a brand new one.

For nets that start with a large number of LUTs followed by a large number of FFs, the pointer for a slice with free FFs will remain untouched until FFs start getting placed, while the pointer for the slice with free LUTs will go further on its own.

The function packCell is optimized in the sense of using the slots entirely. There is a check for signal sharing that would make it legal for it to occupy a position, and the rule for FFs placed together with sink LUTs is checked as well (rules were mentioned in Chapter 3, Section 3.2). Figures 4.4, 4.5 and 4.6 show what it would be like to perform the packing functions. Packing poses as a challenge because of legality constraints. However, it is beneficial to routability, because connections inside a slice don‘t use any routing resources, therefore, have no cost.

Our algorithm keeps track of the last slice with free space for FFs and the last slice with free space for LUTs. So we can always know which ones are available for LUTs and FFs. This way, when we are packing a large set of cells, we won‘t have a massive overhead looking through many slices for a free and legal spot. Finding slices with room for FFs is of much need, because, as in Figure 4.4, LUT 1 is the driver of FF 2 and also of FFs from 5 to 10, and one LUT may only be in the same slice as the maximum of two of its FF sinks. The other FF sinks must, then, be packed somewhere else.

Figure 4.5 shows how the net with blue and orange cells would be clustered. Its LUTs can occupy slice 1, but only two of the sink FFs can fill the same slice as the driver LUT 1. The others are clustered in slice 2, respecting the signal rules. Figure 4.6 shows an example of a merging of two slices into one. To respect legality, however, many slices are left with a single type of cell packed in them. Another step (Figure 4.6) on the packing flow will address this problem.

Our algorithm tries to pack a whole net in the least amount of slices possible. If a net happens to be packed entirely in a single slice, we effectively save routing resources.

The pointers to the last slice with free space are updated when the packing fails. There is no attempt of trying to place a cell on a previous slice, nor trying to see which legality rule would have been broken. This is because packing is already a prolonged process, and adding a considerable overhead must carry its own weight;

The problem with this algorithm is that it generates too many slices, making it impossible to place. This is due to the large number of small nets - with 2 or 3 cells. These tiny nets are packed using a single slice, because it would require too much overhead of management to try fitting them into some other slice previously used by another net. This leaves most slices in the circuit with a huge waste, with nearly 90% of the slice being left unused.

To address this issue, we chose to apply a merging flow after all cells were packed (which is also a means of optimizing the wirelength), with an example shown in Figure 4.6.

There are four types of merges we perform:

- **LUT only:** merges slices with only LUTs in them;
- **FF only:** merges slices with only FFs in them;
- **FF and LUT merge:** merges slices with only LUTs with slices with only FFs;
- **Mixed merge:** merges slices with both LUTs and FFs freely.

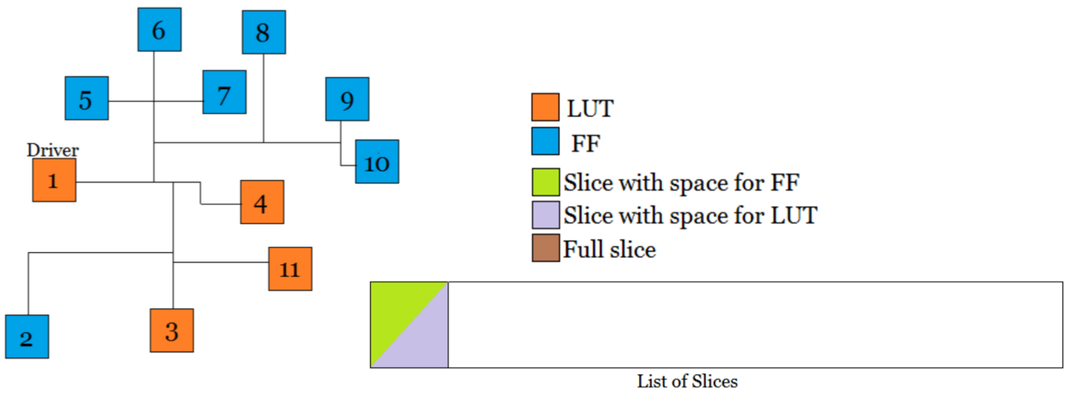


Figure 4.4: Packing CLB cells (FFs and LUTs) into slices part 1.

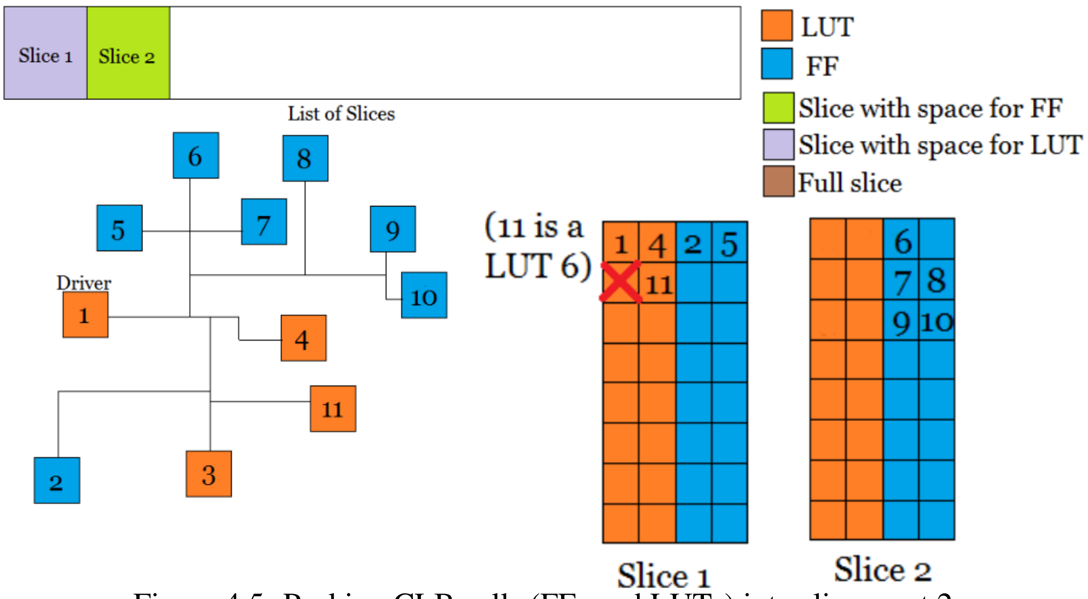


Figure 4.5: Packing CLB cells (FFs and LUTs) into slices part 2.

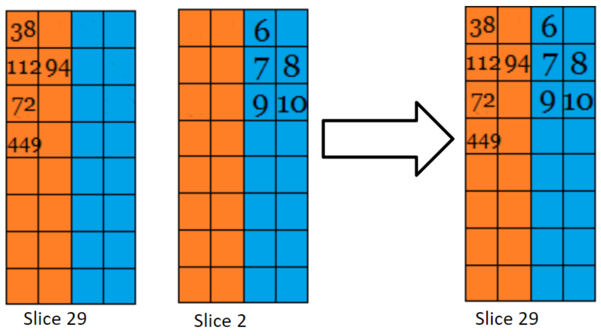


Figure 4.6: Example of merging two slices.

The algorithm for the merges is as follows in Algorithm1:

Algorithm 1 Merge Algorithm

```
forall $slice1$ in slices do
    forall $slice2$ in slices do
        if $slice1$.isLUTOnly and $slice2$.isLUTOnly then
            MERGELUTONLYSLICES($slice1$, $slice2$)
        end
        if $slice1$.isFFOnly and $slice2$.isFFOnly then
            MERGEFFONLYSLICES($slice1$, $slice2$)
        end
    end
end
forall $slice1$ in slices do
    forall $slice2$ in slices do
        if ($slice1$.isLUTOnly() and $slice2$.isFFOnly()) or ($slice1$.isFFOnly() and $slice2$.isLUTOnly()) then
            MERGELUTANDFFSLICES($slice1$, $slice2$)
        end
    end
end
forall $slice1$ in slices do
    forall $slice2$ in slices do
        MERGEMIXEDSLICES($slice1$, $slice2$)
    end
end
```

The reasoning behind this specific order is trying to reduce the number of slices for the last iteration - merge mixed -, because it proved to be the best merge in terms of reducing the number of slices but also very slow. None of the merge functions will do a partial merge - that is, a merge where not all the cells could be reallocated to the other slice. This is because a net that has been absorbed by a slice - that is, the slice contains all the cells of the net - could reappear if we did partial merges. The situation this could occur is: $slice1$ contains 10 LUTs and 3 FFs and $slice2$ contains 10 LUTs and 3 FFs.

Obviously, the slices cannot be merged completely because there are only 16 LUT slots at most. We could do a partial merge such as: $slice1$ contains 16 LUTs and 6 FFs and $slice2$ contains 4 LUTs and 0 FFs.

Since $slice2$ now becomes a great candidate to be merged with another slice, but if the slice 2 contained an entire net, this net would stop being routed for no cost, since connections will appear between the 4 LUTs left on $slice2$ and the cells moved to $slice1$.

The results were better, but not enough; runtime was prohibitive. To improve this situation, we implemented a pseudo-usage attribute for each slice. This attribute is increased as more cells are packed inside the slice. Inserting a LUT will increase it by 1/16; inserting an FF will increase it by 1/16, but inserting one or more FFs that have a LUT as the driver will increase by more. Two FFs and a LUT driver will increase the pseudo-usage by 5/16, and one FF with a LUT driver increase it by 3/16. The pseudo-usage then is used when selecting good candidates to be merged. If the usage of two slices combined is more than a factor, the algorithm doesn't bother trying to merge them, since it is very likely not possible and potentially a waste of precious time.

With this packing flow, we ended up with these results in Table 4.1 in terms of number of slices after each step. It also displays the HPWL acquired for each benchmark. It can be noted that the slice count slowly decreases during each stage. When the slices are merged by the merge mixed algorithm, the count falls very rapidly. Given that the FPGA in question can contain up to 67200 slices, this flow can place all benchmarks but FPGA12 successfully. Figure 4.7 shows the efficiency of merges in terms of reducing the number of slices. It can be noted that the merges that are the most successful are "merge LUT", which merges LUT-only slices, and "merge mixed", which merges mixed slices that have both LUTs and FFs. Slices that contain only FFs are not many, and it might be more difficult to merge the slice's FF area than the LUT area, as there are stringent rules for the signals of the FFs when they are inserted in the slice.

Benchmark	#slices initially	#slices after LUT merge	#slices after FF merge	#Slices after LUT&FF merge	#Slices after Mixed merge	HPWL
FPGA 1	31331	22025	20310	18718	6562	4.10e06
FPGA 2	40297	27239	26639	24397	12268	7.35e06
FPGA 3	130296	79541	70336	60204	31725	4.04e07
FPGA 4	149725	90994	80206	71338	33000	4.61e07
FPGA 5	158684	94565	86762	78129	34821	5.62e07
FPGA 6	234703	157273	148915	133358	53669	9.96e07
FPGA 7	248988	151181	140741	121728	50034	1.09e08
FPGA 8	217897	129558	122590	112692	60576	1.07e08
FPGA 9	307801	167929	157024	134154	62967	1.59e08
FPGA 10	300117	217241	202391	188099	66056	1.34e08
FPGA 11	306115	205091	187716	172879	61419	1.44e08
FPGA 12	359239	266552	242697	225529	69448	1.79e08

Table 4.1: Number of slices before and after the merging steps.

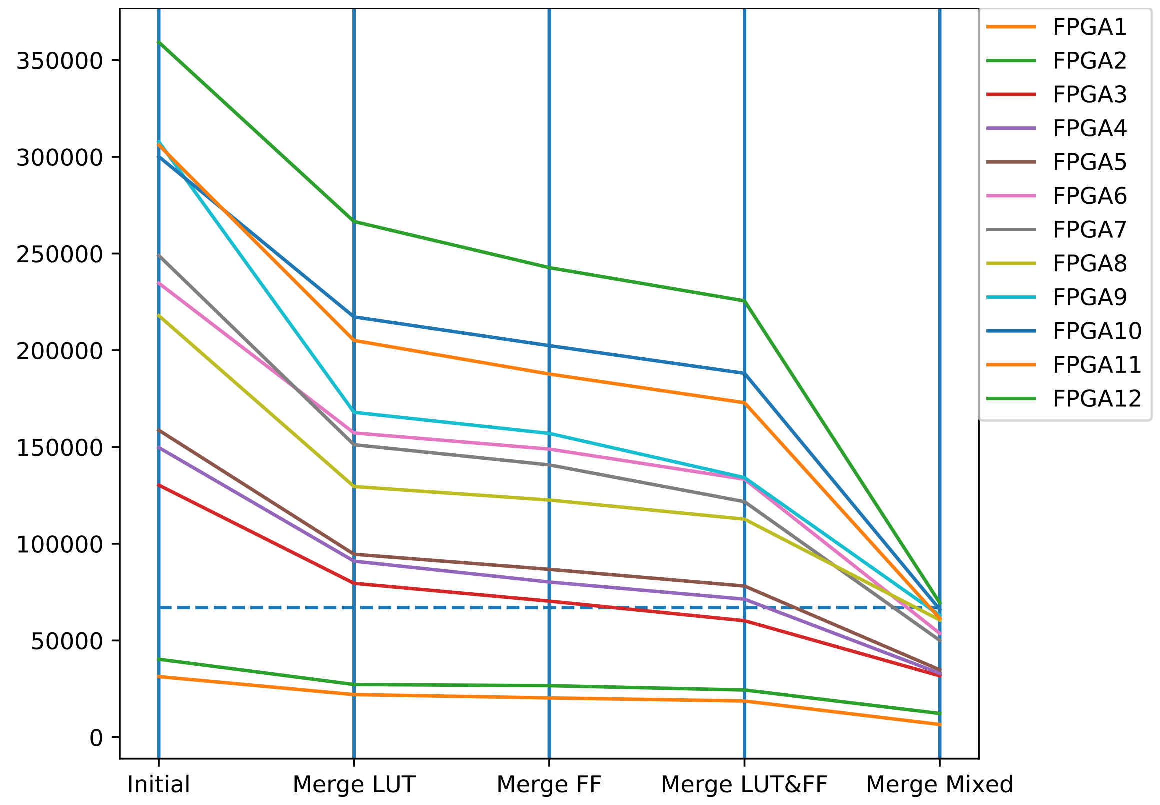


Figure 4.7: Number of slices being reduced through the flow. The dashed horizontal line stands at the number of slices that can fit in the entire FPGA.

After the slices are generated and merged, we perform a simple quadratic placement flow. The complex blocks are considered fixed, and the cells are not considered part of a slice just yet. We perform a spreading force radially from the center of mass of the circuit, and spread the cells with a force proportional to the distance from the center. When the quadratic flow is done, all cells that belong to a slice are imploded into the center of the mass - calculated with the cells in the slice.

Legalization then takes place. It is briefly summarized in Algorithm 2. It goes through all slices and places them in the unused slice slot that is closest to the position of the slice after the quadratic placement flow. Note that if there are more slices to be legalized than slice slots in the FPGA architecture, this algorithm fails. In that case, the program is terminated completely, and an error message is displayed.

Algorithm 2 Legalization

```
forall slice in slices do
    PLACESLICE(slice, FINDCLOSESTLEGALPOSITION(slice.getPosition()))
end
```

Finally, we run a threshold acceptance algorithm as detailed placement. The refinement of the placement solution has the only objective of improving routability, not taking timing or power into consideration. The metric used is HPWL, that is, half the perimeter of the bounding box for a net (Algorithm 3):

Algorithm 3 Threshold Acceptance

```
while $threshold < threshold\_limit$ do
    threshold = threshold*threshold_reduction_constant
    option = RANDOM(0, 100)
    if $option < 90$ then
        hpwl_before = CALCULATEHPWL( )
        SWAPTWORANDOMLYSELECTEDSLICES( )
        hpwl_after = CALCULATEHPWL( )
        if $hpwl\_after - hpwl\_before > threshold$ then
            UNDOSWAP( )
        end
    else
        hpwl_before = CALCULATEHPWL( )
        MOVERANDOMSLICETOEMPTYSPACE( )
        hpwl_after = CALCULATEHPWL( )
        if $hpwl\_after - hpwl\_before > threshold$ then
            UNDOSWAP( )
        end
    end
end
```

Whenever a movement is done, the algorithm checks if it respects the threshold acceptance approach. There are two possible movements, the first being a swap between two cells and the second being a movement of a cell to a free position. The proportion of 90% swaps to 10% moves to empty spaces was determined empirically. After this optimization step is done, the placement solution is outputted to the specified output file so the router can work on it.

4.4 Results of the competition ISPD

Table 4.2 shows the contenders that acquired the first three positions on the competition and the fourth column with our results, as we got the 4th place. It must be noted that this acquisition is, more than anything, due to the fact that it passed every single placement phase for all benchmarks, which means that all the benchmarks were placeable, with the exception of the benchmark FPGA 12, where all the slices were legal, but the number of slice slots in the FPGA is smaller than the number of slices generated. We believe running another merge flow would reduce even further the number of slices generated, making the benchmark FPGA 12 placeable. As for all other benchmarks, our tool was able to place all cells without violating any legalization rule or overusing resources. This leads to future work to be done, covered in the next section.

Designs	TOP 1	TOP 2	TOP 3	OURS
FPGA01	PLACE-ERROR	ROUTED	ROUTED	PLACED
FPGA02	ROUTED	ROUTED	ROUTED	PLACED
FPGA03	ROUTED	ROUTED	ROUTED	PLACED
FPGA04	ROUTED	ROUTED	ROUTED	PLACED
FPGA05	ROUTED	PLACED	PLACED	PLACED
FPGA06	ROUTED	ROUTED	ROUTED	PLACED
FPGA07	ROUTED	ROUTED	ROUTED	PLACED
FPGA08	ROUTED	ROUTED	ROUTED	PLACED
FPGA09	PLACED	ROUTED	PLACED	PLACED
FPGA10	PLACE-ERROR	PLACE-ERROR	PLACED	PLACED
FPGA11	ROUTED	ROUTED	PLACED	PLACED
FPGA12	ROUTED	PLACE-ERROR	PLACED	PLACE-ERROR

Table 4.2: Comparison with top 3 in ISPD 2016 Routability Driven FPGA Placement Contest.

Table 4.3 shows a comparison concerning wirelengths among our tool and three other algorithms, the top 3 of the ISPD 2016 Contest. The values for the three algorithms are the ones reported by Vivado; our values were measured by calculating HPWL. Even though HPWL is not a very precise measurement, this is the measure that we used for comparison because the benchmarks did not pass the Vivado routing stage. It delivered an error message saying that more than 90% of the vertical wiring resources were being used, hence, the solution could not be successfully routed.

Design	UTPlaceF	RippleFPGA	GPlace-flat	Ours (HPWL)
FPGA01	$3.84 * 10^5$	$3.62 * 10^5$	$4.93 * 10^5$	$4.10 * 10^6$
FPGA02	$6.52 * 10^5$	$6.77 * 10^5$	$9.03 * 10^5$	$7.35 * 10^6$
FPGA03	$3.18 * 10^6$	$3.61 * 10^6$	$3.90 * 10^6$	$4.04 * 10^7$
FPGA04	$5.50 * 10^6$	$6.03 * 10^6$	$6.27 * 10^6$	$4.61 * 10^7$
FPGA05	$1.00 * 10^7$	$1.04 * 10^7$	$7.64 * 10^6$	$5.62 * 10^7$
FPGA06	$6.41 * 10^6$	$6.96 * 10^6$	-	$9.96 * 10^7$
FPGA07	$1.00 * 10^7$	$1.02 * 10^7$	$1.12 * 10^7$	$1.09 * 10^8$
FPGA08	$8.11 * 10^6$	$8.87 * 10^6$	$9.32 * 10^6$	$1.07 * 10^8$
FPGA09	$1.36 * 10^7$	$1.29 * 10^7$	$1.40 * 10^7$	$1.59 * 10^8$
FPGA10	$8.86 * 10^6$	$8.56 * 10^6$	-	$1.34 * 10^8$
FPGA11	$1.08 * 10^7$	$1.12 * 10^7$	$1.23 * 10^7$	$1.44 * 10^8$
FPGA12	$8.24 * 10^6$	$8.92 * 10^6$	-	$1.79 * 10^8$

Table 4.3: Comparison with three tools running on ISPD 2016 benchmarks. UTPlaceF, RippleFPGA and GPlace-flat's result are reported in routed wirelength reported by Vivado, and our results were measured in HPWL.

4.5 Placement Flow With HMetis

In this section, the placement flow containing the implementation of the partitioning algorithm hMETIS is presented. It is used mainly before the packing stage described in Chapter 3, as a new bias to clustering the CLB cells into slices. Before, the aim was to try to pack whole net clusters into the same slices, as the routing cost inside a slice is zero. Now, the aim is to pack whole partitions, even though it may only be possible idealistically.

4.5.1 Partitioning

Graph partitioning is a problem in which we consider a graph $G = (V, E, W_v, W_e)$, in which V corresponds to the graph's set of vertices, E corresponds to the set of edges, W_v is the set of vertex weights, and W_e is the set of edge weights. We have to choose a partition solution in clusters $V = V_1UV_2U....UN_p$ such that the sum of the node weights in each set of vertices is distributed evenly (load balance), and the sum of all edge weights of edges connecting all different partitions is minimized, decreasing parallel overhead. (LECTURE..., 2014).

Multilevel partitioning, as stated in (ALPERT; HUANG; KAHNG, 1998), is an approach useful for when the problem size reaches a level of complexity for which it is not enough to perform only one application of a clustering algorithm as described above to produce excellent solutions. A multilevel partitioning algorithm collapses the set of vertices and edges H_1, making them a coarse set of vertices and edges, performing a recursive clustering of each instance in order to make it smaller than a stated threshold and reach H_n. After the threshold is reached, partitioning is performed. After partitioning, the instance has its coarsened vertices and edges uncollapsed to reach stage H_{n-1}, some partitioning refinement is performed, and it follows these steps until reaching H_1, with the final partitioning done, and all the original vertices and edges from the set H_1.

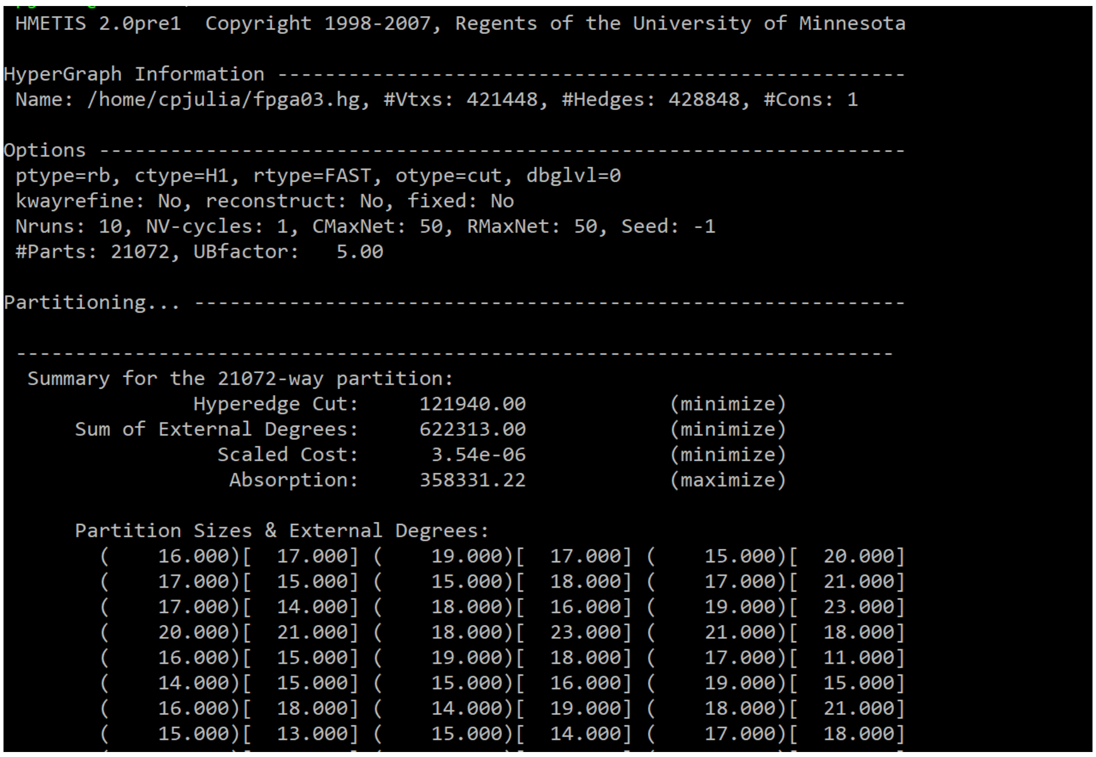
4.6 2018 Placement Flow With hMETIS

This placement flow is identical to the former one, but with one difference. Before, there was a method called packNet(), which tried to pack a whole net altogether within the same slice, as the routing cost within a slice is zero. If it was not accomplished, leftover net cells were packed next within other slices as needed. Using nets as partitions rendered packing difficult, especially, because nets are not disjoint. The ideal solution would be to have slices that had minimal connections in between them, as the cost for routing inside a slice is zero.

In the flow with hMETIS, instead of trying to pack a net altogether within the same slice, the algorithm introduces the concept of partition, and tries to pack as many cells from the same partition within the same slices as possible, as the partitions are as disjointed as possible.

First, using the script developed in Python language in A, we convert the FPGA cells in the UFRGSPlace program's format into hMETIS input rules for partitioning. Then, we partition the FPGA cells of the benchmark according to the average amount of cells wanted per partition. Figure 4.8 shows an example of the hMETIS messages during the execution of the partition algorithm for benchmark FPGA03. The numbers in between parentheses are the amounts of cells per partition (they are approximate to the chosen value, not always the same).

The placement flow was executed for the benchmarks aiming three different distributions: an average of 32 cells per partition, 25 cells, 20 cells and 16 cells. The values were chosen based on the number of cells within each slice, which must be up to 33 (but we set this value to be 32 because there were no CARRY8 cells in the benchmarks), and in the maximum number of cells within the LUT and FF spaces, counting at most 16 FFs and 16 LUTs. The values 20 and 25 were chosen because they are intermediate between 16 and 32.



```
 HMETIS 2.0pre1  Copyright 1998-2007, Regents of the University of Minnesota

HyperGraph Information -----------------------------------------------------
 Name: /home/cpjulia/fpga03.hg, #Vtxs: 421448, #Hedges: 428848, #Cons: 1

Options --------------------------------------------------------------------
 ptype=rb, ctype=H1, rtype=FAST, otype=cut, dbglvl=0
 kwayrefine: No, reconstruct: No, fixed: No
 Nruns: 10, NV-cycles: 1, CMaxNet: 50, RMaxNet: 50, Seed: -1
 #Parts: 21072, UBfactor:   5.00

Partitioning... ------------------------------------------------------------

 -------------------------------------------------------------------------
  Summary for the 21072-way partition:
                Hyperedge Cut:     121940.00           (minimize)
      Sum of External Degrees:     622313.00           (minimize)
                  Scaled Cost:       3.54e-06          (minimize)
                   Absorption:     358331.22           (maximize)

      Partition Sizes & External Degrees:
        (    16.000)[  17.000] (    19.000)[  17.000] (    15.000)[  20.000]
        (    17.000)[  15.000] (    15.000)[  18.000] (    17.000)[  21.000]
        (    17.000)[  14.000] (    18.000)[  16.000] (    19.000)[  23.000]
        (    20.000)[  21.000] (    18.000)[  23.000] (    21.000)[  18.000]
        (    16.000)[  15.000] (    19.000)[  18.000] (    17.000)[  11.000]
        (    14.000)[  15.000] (    15.000)[  16.000] (    19.000)[  15.000]
        (    16.000)[  18.000] (    14.000)[  19.000] (    18.000)[  21.000]
        (    15.000)[  13.000] (    15.000)[  14.000] (    17.000)[  18.000]
```

Figure 4.8: Display of hMETIS message during the execution of the partitioning of benchmark FPGA03.

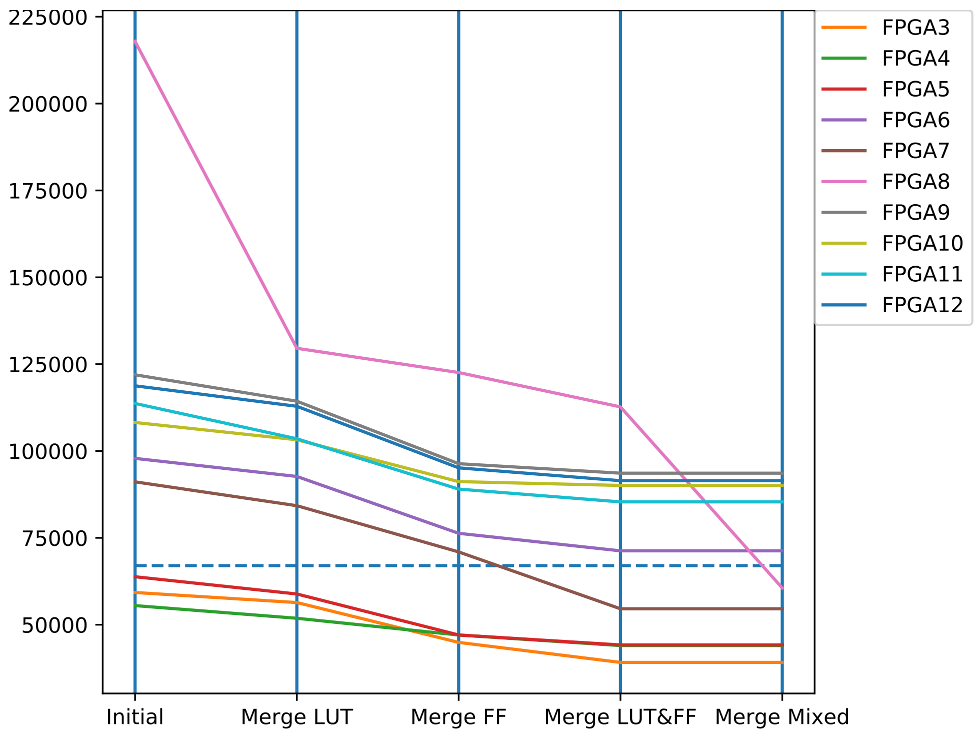


Figure 4.9: Number of slices being reduced through the flow when using hMetis to generate the initial partitions. The dashed horizontal line stands at the number of slices that can fit in the entire FPGA.

Figure 4.9 displays the statistics of the merges for the benchmarks when we choose 25 cells per partition approximately. The blue dashed line is the maximum amount of slice resources in the FPGA: 67200 slices. As can be noted from the graph, half of the benchmarks were not aided by the "merge mixed algorithm and cannot be placed due to lack of resources. The benchmarks that are placeable are not routable because of the same error message by Vivado that happened with the prior flow: the use of vertical wiring resources reached more than 90%.

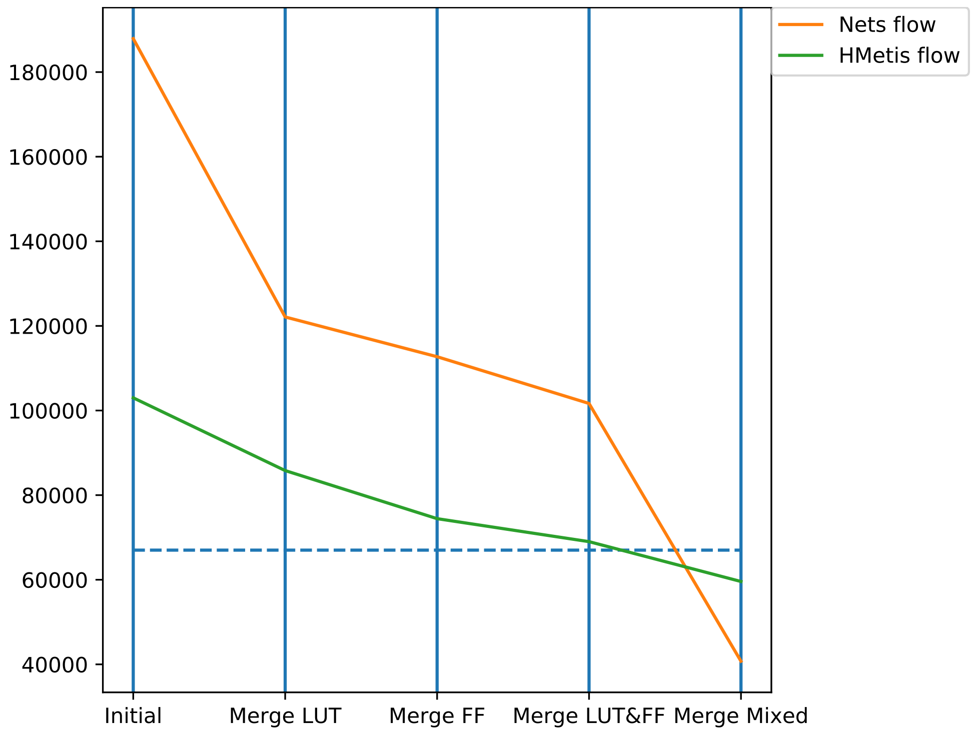


Figure 4.10: Average progress of both flows regarding number of slices generated.

Figure 4.10 shows a comparison in the progress of the merges for the flow that packs nets and the flow that packs partitions with hMetis. The blue dashed line is the maximum number of slice resources, which is 67200. As seen, the hMetis approach did not take much advantage of the "merge mixed" algorithm, which is the one that decreases the number of slices the most.

Table 4.4 shows the HPWL for benchmarks from FPGA03 to FPGA06. The ones above FPGA06 exceeded the resources and were not placed. From observing the table, it seems that it is inconclusive which partition distribution is the best, as there is no choice of the number of cells per partition that outperformed the others. Also these values, 16, 20, 25, 32, are very close, so, aside from the fact that the packing method can be improved to fit the partition approach better, another modification that could be done would be to choose the number of cells per partition from a bigger range, therefore, a greater amount of cells per partition. On the other hand, as in packing nets, the main objective would be to have a partition fully packed within the same slice, as the inner connection cost of the slice is zero, and, with a bigger number of cells per partition, it would be much more difficult to have one whole partition contained inside the same slice.

	16 partitions	20 partitions	25 partitions	32 partitions
FPGA03	3.07E+07	2.90E+07	2.97E+07	3.14E+07
FPGA04	3.76E+07	3.95E+07	3.68E+07	3.89E+07
FPGA05	4.52E+07	4.75E+07	4.62E+07	4.41E+07
FPGA06	8.43E+07	8.13E+07	9.14E+07	9.10E+07

Table 4.4: HPWL of benchmarks after placement.

5 CONCLUSIONS AND FUTURE WORK

In this work, an EDA tool called UFRGSPlace for homogeneous and heterogeneous FPGA placement was presented. It implements a complete FPGA placement flow with packing, global placement, legalization and detailed placement.

The tool supports the ISPD contest input and output formats (which are called extended bookshelf). Our placement flow was very robust and respected all the FPGA technology constraint rules. Moreover, all benchmarks were successfully placed, with exception of FPGA12. This result achieved fourth place in the ISPD contest of 2016.

We also implemented an alternative flow using hMETIS to partition the netlist. The secondary flow uses the partitions generated to perform the initial packing as opposed to the nets. Therefore, in the packing stage, we try to maintain entire partitions within the same slices as opposed to entire nets. However, this approach has proven faulty because of two main reasons. Firstly, it is difficult to partition the design in such a way that most of the partitions fit in a single slice. Secondly, the slices generated have proven to be harder to merge. This can be caused by FF signal problems when attempting to merge.

The main contribution of this work was the development of a complete FPGA placement tool. Even though the results of the packing flow have been satisfactory, the placement still requires refinement. Our estimated routed wire-length is one order of magnitude larger than the other tools.

It can be concluded that the hierarchical approach to the placement, that is, packing the cells into slices based on logic alone, is a less efficient strategy when compared to flat placement. This is further strengthened by the studies of the state-of-the-art algorithms, all of which use flat placement and perform significantly better.

Future works that are interested in studying the hierarchical approach could attempt to incorporate physical information into the partitioning algorithm. One way to achieve this is by performing a flat placement and considering the relative positions of the cells when packing. Also, other methods can be employed in order to increase the chance of each partition being packed into a single slice. One of the methods could be forcing restraints on the number of signal nets that are part of each partition. If the packing stage with partitioning does not improve after employing new techniques, partitioning still can be applied in another placement stage.

Concerning placement, we understand that a way of estimating congestion needs to be implemented. HPWL, while being very fast and arguably accurate for both large and very small nets, lacks this property.

REFERENCES

ALPERT, C. J.; HUANG, J.-H.; KAHNG, A. B. Multilevel circuit partitioning. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 17, n. 8, p. 655–667, 1998.

ALTERA Cyclone 90nm FPGA Family. Available from Internet: <<https://www.altera.com/products/fpga/cyclone-series/cyclone-ii/overview.tablet.highResolutionDisplay.html. Visited2016Jul18.>, year=2016.

BETZ, V.; ROSE, J. Vpr: A new packing, placement and routing tool for fpga research. In: LUK, W.; CHEUNG, P. Y. K.; GLESNER, M. (Ed.). **FPL**. Springer, 1997. (Lecture Notes in Computer Science, v. 1304), p. 213–222. ISBN 3-540-63465-7. Available from Internet: <http://dblp.uni-trier.de/db/conf/fpl/fpl1997.html#BetzR97>.

BRENNER, U.; VYGEN, J. Analytical methods in vlsi placement. **Handbook of Algorithms for VLSI Physical Design Automation**, Taylor and Francis, p. 327–346, 2008.

CHEN, G.; CONG, J. Simultaneous timing driven clustering and placement for fpgas. In: BECKER, J.; PLATZNER, M.; VERNALDE, S. (Ed.). **FPL**. Springer, 2004. (Lecture Notes in Computer Science, v. 3203), p. 158–167. ISBN 3-540-22989-2. Available from Internet: <http://dblp.uni-trier.de/db/conf/fpl/fpl2004.html#ChenC04>.

FIDUCCIA, C. M.; MATTHEYSES, R. M. A linear-time heuristic for improving network partitions. In: ACM. **Papers on Twenty-five years of electronic design automation**. [S.l.], 1988. p. 241–247.

FPGA Generic Physical Design Flow. 2016. Available from Internet: <<http://www.embedded.com/print/4014814>>. Visited 2016 Jul 18 .

FPGA Technology. 2016. Available from Internet: <<http://www.tutorialspoint.com/vlsi_design/vlsi_design_fpga_technology.html/>>. Visited 2016 Jul 18 .

GORT, M.; ANDERSON, J. H. Analytical placement for heterogeneous fpgas. In: KOCH, D.; SINGH, S.; TøRRESEN, J. (Ed.). **FPL**. IEEE, 2012. p. 143–150. ISBN 978-1-4673-2257-7. Available from Internet: <http://dblp.uni-trier.de/db/conf/fpl/fpl2012.html#GortA12>.

HADJIDIMOS, A. Successive overrelaxation (sor) and related methods. **Journal of Computational and Applied Mathematics**, v. 123, n. 1, p. 177 – 199, 2000. ISSN 0377-0427. Numerical Analysis 2000. Vol. III: Linear Algebra. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0377042700004039>.

HESTENES, M. R.; STIEFEL, E. **Methods of conjugate gradients for solving linear systems**. [S.l.]: NBS, 1952.

HMETIS Manual available at. Available from Internet: <http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download>.

ISPD 2016 Contest. 2016. Available from Internet: <<http://www.ispd.cc/contests/16/ispd2016_contest.html>>. Visited 2016 Jul 18 .

KAHNG, A. B. et al. **VLSI Physical Design - From Graph Partitioning to Timing Closure.** [S.l.]: Springer. I-XI, 1-310 p. ISBN 978-90-481-9590-9.

KARYPIS, G. et al. Multilevel hypergraph partitioning: applications in vlsi domain. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 7, n. 1, p. 69–79, 1999.

KARYPIS, G.; KUMAR, V. Multilevel k-way hypergraph partitioning. **VLSI design**, Hindawi, v. 11, n. 3, p. 285–300, 2000.

KERNIGHAN, B. W.; LIN, S. An efficient heuristic procedure for partitioning graphs. **The Bell system technical journal**, Nokia Bell Labs, v. 49, n. 2, p. 291–307, 1970.

KIM, M.-C.; LEE, D.; MARKOV, I. L. Simpl: An effective placement algorithm. In: SCHEFFER, L.; PHILLIPS, J. R.; HU, A. J. (Ed.). **ICCAD**. IEEE, 2010. p. 649–656. ISBN 978-1-4244-8192-7. Available from Internet: <http://dblp.uni-trier.de/db/conf/iccad/iccad2010.html#KimLM10>.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. **science**, American Association for the Advancement of Science, v. 220, n. 4598, p. 671–680, 1983.

LAM, J.; DELOSME, J.-M. Performance of a new annealing schedule. In: **DAC**. [s.n.], 1988. p. 306–311. Available from Internet: <http://dblp.uni-trier.de/db/conf/dac/dac88.html#LamD88>.

LECTURE Notes. 2014. Available from Internet: <http://adl.stanford.edu/cme342/Lecture_Notes_files/lecture7-14.pdf>.

LEE, Y.-S.; WU, A. C.-H. A performance and routability-driven router for fpgas considering path delays. **IEEE Trans. on CAD of Integrated Circuits and Systems**, v. 16, n. 2, p. 179–185, 1997. Available from Internet: <http://dblp.uni-trier.de/db/journals/tcad/tcad16.html#LeeW97>.

LEMIEUX, G. G.; BROWN, S. D.; VRANESIC, D. On two-step routing for fpgas. In: **ISPD**. [s.n.], 1997. p. 60–66. Available from Internet: <http://dblp.uni-trier.de/db/conf/ispd/ispd1997.html#LemieuxBV97>.

LI, W.; DHAR, S.; PAN, D. Z. Utplacef: A routability-driven fpga placer with physical and congestion aware packing. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 37, n. 4, p. 869–882, April 2018. ISSN 0278-0070.

MEAD, C.; CONWAY, L. **Introduction to VLSI systems**. [S.l.]: Addison-Wesley Reading, MA, 1980.

MONTEIRO, J. et al. An analytical timing-driven algorithm for detailed placement. In: IEEE. **Circuits & Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on**. [S.l.], 2015. p. 1–4.

PATTISON, R. et al. Gplace: A congestion-aware placement tool for ultrascale fpgas. In: IEEE. **Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on**. [S.l.], 2016. p. 1–7.

PAUL, S.; DAS, R.; BANERJEE, P. A study on detailed placement for fpgas. In: IEEE. **VLSI Systems, Architecture, Technology and Applications (VLSI-SATA), 2015 International Conference on**. [S.l.], 2015. p. 1–6.

PUI, C.-W. et al. Ripplefpga: a routability-driven placement for large-scale heterogeneous fpgas. In: ACM. **Proceedings of the 35th International Conference on Computer-Aided Design**. [S.l.], 2016. p. 67.

QUARTUS II. Available from Internet: <<http://dl.altera.com/13.0sp1/?edition=web> .Visited2016Jul18.>, year=2016.

SHAHOOKAR, K.; MAZUMDER, P. Vlsi cell placement techniques. **ACM Computing Surveys (CSUR)**, ACM, v. 23, n. 2, p. 143–220, 1991.

SHERWANI, N. **Algorithms for VLSI Physical Design Automation**. 1. ed. [S.l.]: Kluwer Academic Publishers, 1993.

TRIMBERGER, S. M. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. **Proceedings of the IEEE**, IEEE, v. 103, n. 3, p. 318–331, 2015.

VIVADO Suite. 2016. Available from Internet: <<http://www.xilinx.com/products/design-tools/vivado.html>>. Visited 2016 Jul 18 .

XILINX Ultrascale. 2016. Available from Internet: <<http://www.xilinx.com/products/technology/ultrascale.html/>>. Visited 2016 Jul 18 .

XILINX Ultrascale Packing. 2014. Available from Internet" <<http://eecatalog.com/fpga/2014/01/08/xilinx-ultrascale-moves-fpga-from-%E2%80%98critical%E2%80%99-accelerator-to-%E2%80%98central%E2%80%99-soc/>>. Visited 2016 Jul 18 .

XIU, L. **VLSI Circuit Design Methodology Demystified**. 1. ed. [S.l.]: John Wiley Sons, Inc., 2008. 10-11 p.

XU, M.; GRÉWAL, G.; AREIBI, S. Starplace: A new analytic method for fpga placement. **Integration**, v. 44, n. 3, p. 192–204, 2011. Available from Internet: <http://dblp.uni-trier.de/db/journals/integration/integration44.html#XuGA11>.

ZHUO, Y.; LI, H.; MOHANTY, S. P. A congestion driven placement algorithm for fpga synthesis. In: **FPL**. IEEE, 2006. p. 1–4. ISBN 1-4244-0312-X. Available from Internet: <http://dblp.uni-trier.de/db/conf/fpl/fpl2006.html#ZhuoLM06>.

APPENDIX A — SCRIPT FOR CONVERTING UFRGSPLACE FORMAT CELLS TO HMETIS INPUT RULES

```
import sys

if len(sys.argv) != 4:
        print("Usage:␣python3␣converter␣<design.nodes>␣<design.nets>
␣␣␣␣␣␣␣␣<output.hg>")
        sys.exit(-1)

nodes_file_path = sys.argv[1]
nets_file_path = sys.argv[2]
output_file_path = sys.argv[3]

nodes_file = open(nodes_file_path, 'r')
nets_file = open(nets_file_path, 'r')


nodes = dict()
for line in nodes_file:
        words = line.strip().split("␣")
        nodes[words[0]] = len(nodes) + 1

hyperedges = []
for line in nets_file:
        words = line.strip().split("␣")
        net_name = words[1]
        number_of_cells = int(words[2])
        hyperedge = (net_name, []*number_of_cells)
        for i in range(number_of_cells):
                net_line = nets_file.readline()
                net_line = net_line.strip().split("␣")
                hyperedge[1].append(net_line[0])
        hyperedges.append(hyperedge)
        dummy_final_line = nets_file.readline().split()

output_file = open(output_file_path, 'w')

first_line = "{}␣{}\n".format(len(hyperedges), len(nodes))
output_file.write(first_line)
for hyperedge in hyperedges:
        nodes_list = hyperedge[1]
        line = ""
        for node in nodes_list:
                line = line + str(nodes[node.strip()]) + "␣"
        line = line + "\n"
        output_file.write(line)
sys.exit(1)
```