GPlace: A Congestion-Aware Placement Tool for UltraScale FPGAs

(Invited Paper)

Ryan Pattison, Ziad Abuowaimer, Shawki Areibi, *Gary Grewal, Anthony Vannelli
School of Engineering, *School of Computer Science
University of Guelph
Guelph, ON, Canada
{rpattiso,abuowaiz,sareibi,ggrewal,vannelli}@uoguelph.ca

ABSTRACT

Traditional FPGA flows that wait until the routing stage to tackle congestion are quickly becoming less effective. This is due to the increasing size and complexity of FPGA architectures and the designs targeted for them. In this paper, we present two new congestion-aware placement tools for Xilinx UltraScale architectures, called GPlace-pack and GPlace-flat, respectively. The former placer participated in the ISPD 2016 Routability-driven Placement Contest for FPGAs, and finished in third place overall. The latter placer was subseqently developed based on our experience in the contest with GPlace-pack. Results obtained indicate that GPlace-flat is on average $5.3\times$ faster than GPlace-pack. The post routing results show that GPlace-flat is able to obtain a further 22.5% improvement in wirelength and a 40.0% improvement in runtime compared to GPlace-pack.

Keywords

Placement, Field Programmable Gate Array, Congestion, Routing-aware, Heterogeneous, UltraScale Architecture

1. INTRODUCTION

In this paper, we share our experience developing two analytic placement flows for performing FPGA placement: GPlace-pack and GPlace-flat. The former flow participated in the ISPD 2016 Routability-Driven FPGA Placement Contest [16], where it finished third out of nineteen participating institutions. The latter flow was developed post-contest, and outperforms GPlace-pack in terms of quality-of-result, runtime, and total number of successfully routable contest benchmarks.

As the contest title suggests, the goal of this year's challenge was a departure from earlier years, where the focus was on solving problems relevant to ASIC design. We chose to enter this year's contest because we had earlier success developing both serial [14] and parallel [8] analytic placement algorithms for homogeneous FPGAs, and wanted to better understand the challenging issues that arise when targeting modern, heterogeneous FPGA architectures, like the Xilinx UltraScale devices.

ICCAD '16, November 07-10, 2016, Austin, TX, USA

 ISBN 978-1-4503-4466-1/16/11...$15.00

DOI: http://dx.doi.org/10.1145/2966986.2980085

Our development team consisted of the five authors on this paper. Development of GPlace-pack began shortly after November 30, 2015, when the first sample benchmark was released by the contest organizers, and was completed on April 6, 2016, when the contest officially completed. The four contest benchmarks provided to contestants all had very different features leading to a very wide design space containing different placement strategies and configurations. Especially challenging was finding ways to satisfy the many hard constraints that arise from the complex, heterogeneous architecture present in the UltraScale FPGA device. Satisfying these constraints during placement, however, does not guarantee that the subsequent routing phase will complete successfully. This is because congested regions in the placement may exhaust the routing resources available in those regions causing the router to fail.

Our first approach to addressing these issues was to develop GPlace-pack, a placer which seeks to satisfy hard constraints through judicious packing of logic into slices. We found, however, that this approach leads to poor wirelength or solutions that do not fit on the FPGA. Therefore, following the contest we developed GPlace-flat, which solves congestion and legalization constraints during global placement, leading to better wirelength and lower congestion.

The main contributions of this paper can be summarized as follows:

1. Two novel congestion-aware placement tools for Xilinx's UltraScale architectures are presented.
2. A fast method for estimating congestion that is independent of the placement quality is implemented, making its use early in the placement process feasible.
3. A novel bi-partitioning procedure for legalizing flat placements is presented. The procedure satisfies all hard constraints imposed by the UltraScale architecture, while minimzing cell displacement. This procedure eliminates the need for a packing step in the FPGA flow, and allows the placer to optimize globally while enforcing legalization constraints.

The remainder of this paper is organized into the following sections. Section 2 provides an overview of the FPGA placement problem. Section 3 presents related work. In Section 4, the GPlace algorithm is presented with two different implementations. Section 5 reviews the results of placing the contest benchmarks. Finally, the conclusions are presented in Section 6.

2. BACKGROUND

2.1 Problem Formulation

FPGA placement is the process of mapping the cells in a circuit net-list to sites on an FPGA. A net-list is represented by a hyper-graph $G_h(V, E_h)$, where vertices V is a set of cells (i.e., LUTs, FFs, RAMs, DSPs, and IOs) to be placed and the hyper-edges E_h are a set of nets. Each net is a subset of V that specifies the connections to be made between cells. The architecture defines a rectangular region with width W and height H divided into columns each dedicated to one cell type (i.e., slice, RAMs, DSPs, and IOs). The placement must satisfy all architectural constraints associated with FF control sets, LUT-sharings, block and column type matching. In routability-driven placement, the goal is to successfully route the design by placing cells on the chip such that the nets have minimal overlap with one another.

2.2 UltraScale FPGA Architecture

Xilinx UltraScale FPGAs [13] consist of heterogeneous programmable logic blocks, such as general logic (i.e., LUTs), random access memory blocks (i.e., BRAMs), and digital-signal processing blocks (i.e., DSPs). As well, there are prefabricated routing segments of different lengths in both horizontal and vertical directions. Slices contain both LUTs and FFs, which grouped together share a single switch for routing. Switch boxes provide both intra- and inter-slice connectivity. Slices are placed side by side such that each pair of slices share the same switch box in a back-to-back formation. Each slice contains 8, 6-input LUTs and sixteen FFs. The 6-input LUTs can be configured to implement either one logic function of up to 6-inputs with one output, or two distinct logic functions of up to 5-inputs and two separate outputs. To implement two distinct functions in the same LUT, the sum of the distinct inputs of the two functions must not exceed five. There are only two clock (CLK) signals and two set/reset signals per slice. The bottom 8 FFs share the same CLK and RESET signals, while the upper 8 FFs use the second CLK and RESET signals. Each group of 8 FFs is further divided into two subgroups, each of which must share the same clock-enable (CE) signal. These control-set constraints, coupled with the other placement constraints, make the placement problem extremely challenging [16]. Even if all of the architectural constraints are satisfied by the current placement, there is no guarantee that the placement will be routable. Moreover, differ placement solutions may require the router to perform more or less work to find a feasible routing solution.

3. RELATED WORK

FPGA placement algorithms can be classified into three main categories based on the optimization approach employed: simulated-annealing, partitioning, and analytic. Simulated-annealing based placers, like the state-of-the-art academic placer VPR [9], are able to achieve high-quality placement solutions, but at the expense of long runtimes. Placement algorithms based on partitioning, like PPFF [7], employ recursive top-down partitioning to divide the circuit and FPGA into sub-circuits and sub-regions by minimizing the number of cuts between sub-regions. However, placement quality often suffers because wirelength is not directly minimized. Analytic placement algorithms, like QPF [15], CAPRI [5], StarPlace [14], HeAP [6], and LLP [12] use smooth functions to approximate a non-smooth wirelength cost function, and solve a system of equations using efficient numerical methods. However, analytic placers often have difficulty dealing with hard constraints imposed by the FPGA architecture. Moreover, most analytic FPGA placers [15], [14], [5] consider homogeneous, island-style FPGA architectures. More recently, academic analytic placers for heterogeneous FPGA architectures have started to appear in [6], [12], [4], and [3].

Many of the previous FPGA placers (e.g., [6], [12], [4]) employ wirelength models that do not consider the type of segmented-routing architectures used in today's modern FPGA architectures. Two exceptions are [5] and [3], both of which seek to minimize segmented-routing wirelength. None of the previous FPGA analytic placers directly model congestion.

Packing has been previously used in some FPGA flows to address congestion. The approach in [10] employs loose packing to avoid over utilization of Configurable Logic Blocks (CLBs), thus reducing congestion. However, this strategy can rapidly lead to the under utilization of CLBs resulting in the circuit failing to fit on the FPGA. Moreover, the uniform under utilization of CLBs can also lead to an increase in total wirelength and hence congestion. To compensate, Un/DoPack [11] first identifies congested regions, unpacks CLBs in those regions, and then re-packs those CLBs with a reduced cluster size. However, the enitre process is extremely expensive, as it requires using the actual router to identify the congested regions of CLBs. Also, Un/DoPack does not handle the congestion that is introduced by "flyover" nets (i.e., nets which do not have any pins in that congested region).

Unlike academic FPGA architectures, commercial FPGAs have more hard constraints that limit and complicate traditional packing algorithms that are primarily based on circuit connectivity. The authors in [1] introduce specialized packing techniques directly into a placer targeted for the Xilinx Virtex-5 architecture. First, a flat placement which ignores all architecture constraints is performed. Then, based on the physical locations of the cells and circuit connectivity, packing is performed to obtain slices that satisfy the architecture constraints. Finally, the packed net-list is placed onto the FPGA as slices. Although this work applies packing techniques to optimize for timing and power, it does not consider congestion and routability. In this paper, we borrow some ideas from the flow in [1] and employ them in our GPlace-pack flow. However, in the spirit of the ISPD 2016 Routablity-Driven FPGA Placement contest, our focus is on performing congestion-driven placement rather than on placement objectives like power or timing.

4. GPLACE-PACK/FLAT PLACERS

In this Section, we describe the development of GPlace-pack, which was used in the ISPD 2016 FPGA Design contest, and GPlace-flat, which was developed post-contest. We begin with a brief description of our earlier analytic placement algorithm, StarPlace [14]. Both GPlace-pack and GPlace-flat employ the same wirelength model used in [14], as well as a similar legalization procedure. Although StarPlace does not directly support heterogeneous architectures and does not optimize for congestion, we chose to start with StarPlace because of the excellent results it was able to achieve both in terms of runtime and critical-path delay compared with the state-of-the-art academic placer, VPR [9].

4.1 Original StarPlace (Homogeneous)

The StarPlace flow is illustrated in Figure 1. StarPlace uses VPR's TV-Pack to first pack LUTs and FFs into CLBs to make the placement homogeneous. Then, in order to avoid trivial solutions, I/O blocks are initially fixed; logic blocks are assigned random locations. StarPlace then performs a maximum number of iterations where wirelength is optimized based on the Star+ [14] model. Both the Star+ model and the legalization procedure are described in the subsections that follow.

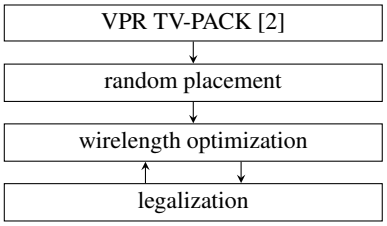


Figure 1: StarPlace (Homogeneous) Flow

4.1.1 Wirelength Optimization

In the Star+ model, a net is converted to a star by adding a centre-of-gravity node at the mean position (c_{yl}, c_{xl}) of the cells in the net by Equation 1. The Star+ cost is computed for each net and dimension in Equation 2. Equation 3 updates the x-position of cell i at iteration k using the Jacobi method.

$$c_{xl} = \frac{1}{|\mathrm{Net}_l|} \sum_{i \in \mathrm{Net}_l} x_i \tag{1}$$

$$S_{xl} = \sqrt{1 + \sum_{i \in \mathrm{Net}_l} (x_i - c_{xl})^2} \tag{2}$$

$$x_i^{(k+1)} = \sum_{l: i \in \mathrm{Net}_l} \frac{c_{xl}^{(k)}}{S_{xl}^{(k)}} \Bigg/ \sum_{l: i \in \mathrm{Net}_l} \frac{1}{S_{xl}^{(k)}} \tag{3}$$

4.1.2 Legalization

After solving for wirelength, the placement has real-valued coordinates that (likely) violate the constraints imposed by the architecture, thus requiring legalization. StarPlace uses recursive bi-partitioning to legalize the placement. In bi-partitioning, the FPGA is recursively bisected into smaller regions and arranged in a 2-dimensional tree. Each node in the tree stores the location and size of the region, as well as a capacity for each site type. Initially, all cells belong to the root of the tree in a region that spans the entire FPGA. Cells are stored in two arrays and each array is sorted by the cell's position with one array for each dimension. Next, the array corresponding to the dimension being cut is partitioned into two. The split point for the partitions in the array must not be higher than the left-child's capacity and not lower than the total number of cells minus the right child's capacity to avoid overflow. The point in this legal range that is closest to the dividing line of the left and right child regions is selected as the split point and the cells below the split point go to the left child and the rest go to the right child. This process repeats until the leaf nodes in the tree are reached and each cell is assigned its own site.

4.2 GPlace-pack

Targeted at homogeneous, island-style FPGAs, StarPlace is not directly applicable to the UltraScale architecture. Therefore, it had to be adapted for the heterogeneous UltraScale architecture. The resulting flow, called GPlace-pack, is summarized in Figure 2. GPlace-pack uses the same wirelength solver as StarPlace, but the legalization procedure was modified to use different capacities for each cell type to roughly legalize the placement. The placement is only roughly legalized because control-set constraints and LUT sharing constraints are not considered during bi-partitioning in GPlace-pack. Instead, these constraints are solved in the packing stage. After packing, however, bi-partitioning is sufficient to legalize the packed slices. GPlace-pack's packing algorithm was inspired by the flow of [1]. To improve congestion, we added a congestion model to estimate the routability, then adjusted packing to repack the whole circuit with a reduced limit on the external pin counts for each slice to reduce local congestion around the slices.

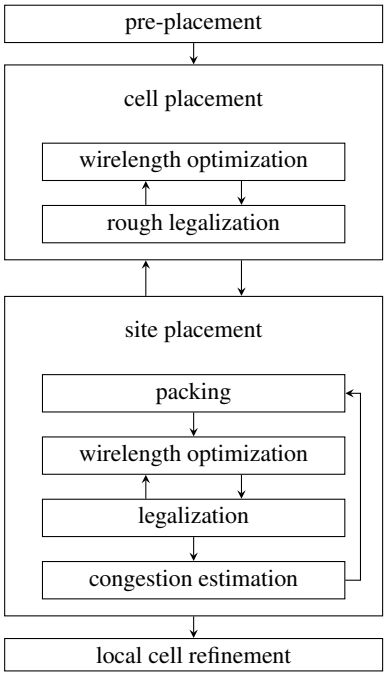


Figure 2: GPlace-pack Flow

4.2.1 Pin Propagation Pre-Placement

Initially, a placement is generated by propagating the positions of fixed I/O cells through the net-list. First, the net-list is converted to a graph by creating a directed edge from the driver of the net to each sink in the net. The cells are iterated in topological order and moved to the average position of their inputs, placing cells near the input pads they are most connected to. The entire process is repeated, but with the direction of the edges reversed, thereby placing cells near their ultimate outputs. Finally, the results of propagating from inputs and outputs are averaged.

4.2.2 Placement-aware Packing

Packing begins by dividing the FPGA into a regular grid. The size of each bin in the grid is initially (1×1) but can grow up to (12×12) if the placement remains congested. Packing proceeds in each bin by selecting an arbitrary cell as a seed to start the cluster. A cell becomes a candidate if adding it to the cluster does not violate any capacity, control-set, or LUT-sharing constraints, and does not result in too many external pins. The maximum number of allowable external pins is initially high (i.e., 104) and is decreased later if the congestion is too high, before repacking. Next, the candidates are ranked by their affinity to the current cluster and the candidate with the highest affinity is added to the cluster. The affinity is computed as the number of edges between the cluster and the candidate, treating a net as a clique. Candidates are added in this way until there are no more suitable candidates. From the remaining unclustered cells, a new seed is selected arbitrarily and the clustering repeats until there are no more seeds. Next, the wirelength optimization and legalization are applied to the packed net-list.

4.2.3 Congestion Estimation

GPlace-pack estimates congestion using the Wire-Length-Per Area (WLPA) model with the Weighted Half-Perimeter Wire-Length (WHPWL) estimate from VPR [2]. WLPA is used to estimate congestion because it can be efficiently computed and can accurately identify areas of high congestion (i.e., hot spots).

To further improve the accuracy, the bounding box for each net is converted from slice coordinates to switch coordinates. The vertical and horizontal WLPA ($\text{WLPA}_x, \text{WLPA}_y$) are computed by Equation 4–5.

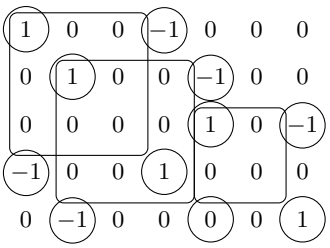
$$\text{WLPA}_{xl} = q(|\text{Net}|)/h_l \tag{4}$$

$$\text{WLPA}_{yl} = q(|\text{Net}|)/w_l \tag{5}$$

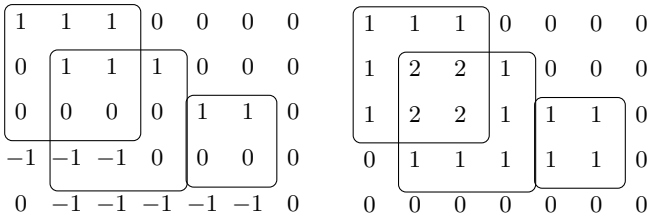
where q is the compensation function from VPR and w_l, h_l are the width and height of the bounding box, respectively.

4.2.4 Computing Switch Utilization

GPlace computes the switch utilization using prefix sums. The benefit to this approach is that the computation time is independent of the quality of placement and can, therefore, be applied early in the placement process. Normally the computation of WLPA involves iterating over the bounding boxes of all nets in the netlist. Using this naïve method for a netlist with a total of P pins and a $W \times H$ sized FPGA results in a complexity of $\mathcal{O}(PWH)$, while the prefix-sum method outlined here has the complexity $\Theta(P + WH)$.



(a) mark corners



(b) sum rows (c) sum columns

Figure 3: Computing WLPA with Prefix Sums (WLPA=1)

The steps to computing the switch utilization are illustrated in Figure 3. First, the bounding box for each net is computed, converted to switch coordinates, and then the WLPA is estimated. For simplicity, Figure 3 assumes the WLPA is one for all nets. Next, a matrix of zeros is allocated with the same size as the FPGA. For each net, the WLPA value is added to the matrix near the corners of the bounding box. More specifically, for a bounding box, (x_1, y_1, x_2, y_2), the WLPA value is added at (x_1, y_1) and $(x_2 + 1, y_2 + 1)$, while the WLPA is subtracted at $(x_2 + 1, y_1)$ and $(x_1, y_2 + 1)$. Next, an inclusive prefix-sum is performed on each row of the matrix, and then for each column. The resulting matrix contains the total wirelength estimate of each switch, and is converted to a utilization estimate by dividing by the channel width. Both the horizontal and vertical switch utilization are computed using prefix sums.

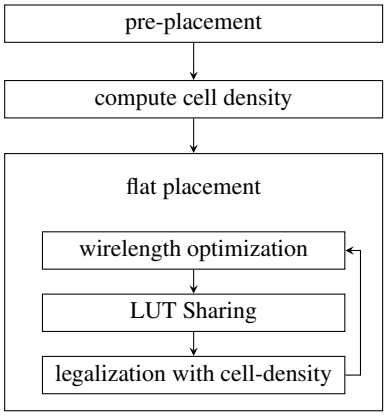


Figure 4: GPlace-flat Flow

4.2.5 Local Refinement

Local refinement performs a single iteration of wirelength optimization to find a more ideal, but likely illegal, position for each cell. Next, local refinement attempts to move each cell from its current legal position to the slice closest to its ideal position, provided the move will not violate any slice-constraints. This process is repeated for several iterations.

4.2.6 Preliminary Results of GPlace-pack

Based on the results presented later in Section 5, it was observed that GPlace-pack may unpack and repack the entire net-list if the placement is congested. This global repacking leads to a rapid increase in wirelength for highly-congested designs, and an increase in the number of slices used in the FPGA. Consequently, a circuit may not fit on the FPGA if packed too loosely. Furthermore, any rapid increase in wirelength may increase congestion further, thus reducing routability.

4.3 GPlace-flat Placement Algorithm

To address the drawbacks of GPlace-pack described above, a new flow, called GPlace-flat, was developed post contest. The GPlace-flat framework adds cell density to address congestion locally as opposed to modifying global packing parameters. Furthermore, the legalization phase is improved to handle FF control-set constraints, as well as LUT sharing. The latter removes the need to perform packing, and thus avoids the risk of the circuit not fitting on the FPGA. The flow of GPlace-flat is summarized in Figure 4, where the packing and local refinement steps are eliminated and where cell density is added to solve congestion. The new steps in the flow are explained in the following subsections.

4.3.1 Cell Density in Bi-partitioning

With cell-density, there are two types of overflow that can occur: hard and soft. Hard overflow occurs when a region does not have enough sites to legalize the cells, and is always avoided by bi-partitioning. Soft overflow occurs when the total cell density exceeds the capacity of the region. When soft overflow occurs, it is treated the same as hard overflow unless both child regions overflow. If both of the child regions overflow, the cell density is kept roughly balanced between the two partitions. BRAM and DSP cells are legalized using recursive bi-partitioning as previously de-

scribed. However, the handling of LUTs and FFs requires some modifications in order to allow for LUT-sharing and control-set constraints for FFs, as described below.

4.3.2 LUT Sharing

The goal of LUT-sharing is to reduce the utilization of the chip by combining LUTs. A slice in the UltraScale architecture contains 8, 6-input LUTs. However, a single 6-input LUT can implement 2 LUTs if together the number of unique inputs is less than 6. GPlace-flat pairs LUTs while aiming to maximize the number of shared inputs and minimize the distance between the paired LUTs in the placement. First, GPlace-flat searches for groups of LUTs that have n common inputs. LUTs with common inputs are found by hashing all possible subsets of their inputs of size n; that is, for sharing n inputs, a k-input LUT hashes all (k choose n) subsets of its inputs and is added to the corresponding buckets. After all LUTs have been hashed, each bucket is partitioned into groups that share the same n inputs. Next, each LUT is paired to the nearest unpaired LUT in the same partition, provided the distance between them is less than 1 in the placement. The pairing process is repeated for all values of n, starting at the highest number of shared inputs (5) and ending with the fewest (0).

4.3.3 LUT Sharing in Bi-partitioning

The dependence between the sharing of LUTs and the LUT cell capacity of a slice makes identifying overflowing regions difficult. A slice may contain 8, 6-input LUTs, however, if the LUTs are smaller and share, a slice can contain up to 16 LUTs. To resolve this dependence, GPlace-flat pairs LUTs prior to legalization and removes one LUT of each pair from the placement. Removing one LUT from each pair allows bi-partitioning to use a capacity of 8 LUTs per slice. After legalization, the removed LUTs are added back to the placement in the same position as their paired LUT.

4.3.4 Flip Flop Control Sets in Bi-partitioning

A slice may contain 16 FFs provided that they satisfy several constraints on their clock, clock-enable, and reset signals. The 16 FFs form a hierarchy based on their control signals. At the top level, the 16 FFs can be subdivided into two groups of 8, with each group sharing the same clock and reset signal. Next, each group of 8 must be subdivided into 2 groups of 4, each group sharing the same clock-enable signal. The bi-partitioning algorithm requires that the number of slices needed for all FFs in a region be computed and updated as FFs move between regions. To compute the slice count, GPlace-flat maintains a tree structure organizing the FFs in the circuit during partitioning, as shown in Figure 5. The root of the tree has a child for each region in the tree used for bi-partitioning. Each region node has a child for each clock and reset combination in the region. In the next level, the clock and reset nodes have a child for each clock-enable. The next level contains a count of the number of FFs that match the clock-enable, reset, clock, and region as its node's branch in the tree. Computing the number of slices required in each region is carried out in a bottom-up pass through the tree. At the bottom, the clock-enable node returns the number of groups of 4 it will require ($\lceil \mathrm{FF}/4 \rceil$). The clock and reset node sums the value from each child and returns $\lceil \mathrm{sum}/2 \rceil$, thereby computing the number of half slices. Finally, the region node sums the half slice values from each child and returns $\lceil \mathrm{sum}/2 \rceil$, thereby computing the number of slices required in the region. This tree is used to identify overflow in a region and its branches are updated when moving an FF between regions.

4.3.5 Cell Density

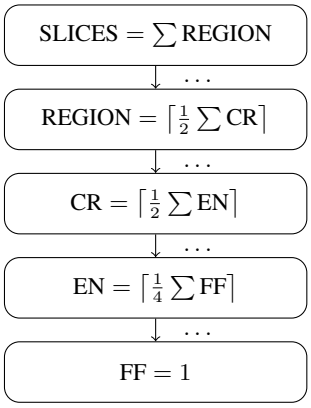


Figure 5: FF Slice Counting Tree

GPlace-flat uses cell density to avoid congestion in the circuit. The cell density is computed for LUTs by Equation 6 which uses the average number of pins of LUTs in the net-list μ_{LUT}. The density of FFs, RAM, and DSP are not varied.

$$\mathrm{density(LUT)} = \frac{\mathrm{inputs(LUT)} + \mathrm{outputs(LUT)}}{0.55\mu_{LUT}} \quad (6)$$

Bi-partitioning uses the density of cells to decide if a region is overflowing when moving cells between regions. If both regions are overflowing, the legalization will attempt to evenly distribute the density between regions.

5. RESULTS

In this section, we first introduce our experimental setup. This is followed by a description of the contest benchmarks used to evaluate each placer. Finally, we present the results for both GPlace-pack and GPlace-flat.

5.1 Experimental Setup

The experiments were carried out using a Linux machine (CentOS release 6.8) running on an Intel (Xeon CPU E3-1270 V2 @ 3.50GHz) processor. The algorithms were developed using C and compiled using gcc 4.4.7 20120313 (Red Hat 4.4.7-17) compiler. The routing is performed by Vivado (version 15) with a patch applied to make it compatible with the modified bookshelf benchmark format of the contest. The target device is xcvu095, part of the Virtex UltraScale family.

5.2 Benchmarks

Table 1 shows the Xilinx benchmarks that were used to evaluate the placers for the ISPD contest [16]. The benchmarks were generated using a netlist-generation tool based on Generate Netlist (Gnl). Several properties of the netlists were varied among the ISPD benchmarks including, the number of LUTs, DPSs, BRAMs and FFs. Different Rent exponents were used to vary the interconnection complexity. The final parameter used in generating the benchmarks was the number of resets which poses challenges and restrictions on the placer. It is clear from Table 1 that these circuits vary in their placement and routing complexities since they have different components and interconnections.

5.3 Routing Results

To test GPlace, the 12 benchmarks from the 2016 ISPD placement contest [16] are placed and routed. The routed wirelength for

Table 1: Xilinx Benchmarks: Statistics

BM	#LUTs	#FF	#BRAM	#DSP	#CSet	#IO	R.E
FPGA-1	49K	55K	0	0	12	150	0.4
FPGA-2	98K	74K	100	100	121	150	0.4
FPGA-3	245K	170K	600	500	1281	400	0.6
FPGA-4	245K	172K	600	500	1281	400	0.7
FPGA-5	246K	174K	600	500	1281	400	0.8
FPGA-6	345K	352K	1000	600	2541	600	0.6
FPGA-7	344K	357K	1000	600	2541	600	0.7
FPGA-8	485K	216K	600	500	1281	400	0.7
FPGA-9	486K	366K	1000	600	2541	600	0.7
FPGA-10	346K	600K	1000	600	2541	600	0.6
FPGA-11	467K	363K	1000	400	2091	600	0.7
FPGA-12	488K	602K	600	500	1281	400	0.6

Table 2: Routed Wirelength: GPlace-pack vs. GPlace-flat

Benchmark	GPlace-pack	GPlace-flat	% Improvement
FPGA-1	581,975	493,788	15.1%
FPGA-2	1,046,859	903,099	13.7%
FPGA-3	5,029,157	3,908,244	22.2%
FPGA-4	7,247,233	6,277,878	13.3%
FPGA-5	–	–	–
FPGA-6	10,648,413	7,643,382	28.2%
FPGA-7	–	11,255,351	–
FPGA-8	12,299,898	9,323,360	24.1%
FPGA-9	–	14,002,965	–
FPGA-10	–	–	–
FPGA-11	–	12,367,773	–
FPGA-12	–	–	–

each benchmark is provided in Table 2. It is clear that GPlace-flat improves the wirelength on all the routable benchmarks over the original GPlace-pack. On average GPlace-flat achieves 22.5% improvement in routed wirelength. The improvement in wirelength achieved by GPlace-flat over GPlace-pack is attributed to GPlace-flat solving the architectural constraints and congestion (by cell-density) during global placement. It is also evident from Table 2 that GPlace-flat is capable of routing several benchmarks that GPlace-pack failed to route including FPGA-7, FPAG-9 and FPGA-11.

The CPU time of both GPlace-pack and GPlace-flat are shown in Table 3 Where on average, GPlace-flat is $5.3\times$ faster than GPlace-pack.

Table 3: Placer CPU Time: GPlace-pack vs. GPlace-flat

Benchmark	GPlace-pack	GPlace-flat	% Improvement
FPGA-1	1m34s	0m30s	77.1%
FPGA-2	3m7s	1m1s	67.3%
FPGA-3	15m18s	4m49s	68.5%
FPGA-4	15m41s	4m40s	70.2%
FPGA-5	16m32s	4m41s	71.6%
FPGA-6	15m2s	10m0s	33.4%
FPGA-7	156m14s	11m31s	92.6%
FPGA-8	15m40s	12m14s	21.9%
FPGA-9	93m59s	16m14s	82.7%
FPGA-10	86m41s	15m53s	81.6%
FPGA-11	91m0s	15m23s	83.1%
FPGA-12	117m17s	20m36s	82.4%

The CPU time used by the Vivado 2015.4 router is shown in Table 4. The Xilinx Vivado router is able to route placements produced by GPlace-flat with less effort by cutting down the CPU time by 39.8% on average compared to those produced by GPlace-pack. This indicates that the quality of placements produced by GPlace-flat are less congested and of better quality than those produced by GPlace-pack.

Table 4: Vivado Router CPU Time: GPlace-pack vs. GPlace-flat

Benchmark	GPlace-pack	GPlace-flat	% Improvement
FPGA-1	3m41s	3m43s	0%
FPGA-2	4m41s	4m34s	2.4%
FPGA-3	9m55s	9m48s	1.1%
FPGA-4	10m56s	10m27s	4.4%
FPGA-5	–	–	–
FPGA-6	50m52s	16m29s	67.5%
FPGA-7	–	21m13s	–
FPGA-8	20m47s	15m42s	24.4%
FPGA-9	–	135m26s	–
FPGA-10	–	–	–
FPGA-11	–	98m25s	–
FPGA-12	–	–	–

The progress of GPlace-flat is illustrated in Figures 6 and 7. Although these figures are for a single benchmark, FPGA-3, the behaviour that they illustrate is typical of that for other benchmarks.

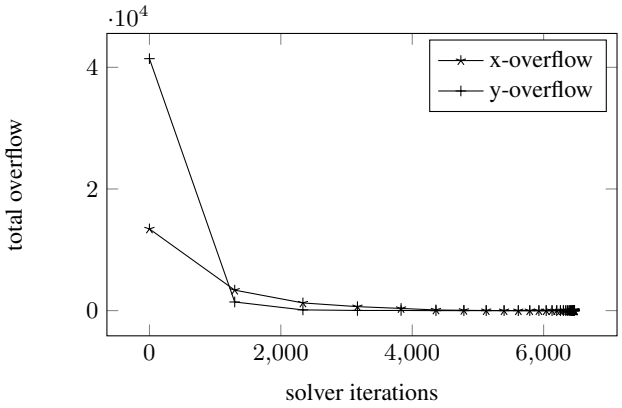


Figure 6: Total Overflow FPGA-3

6. CONCLUSIONS

In this paper we presented a new congestion-aware placement tool for Xilinx's UltraScale architectures called GPlace. GPlace participated in the ISPD 2016 Routability-driven Placement Contest for FPGAs, where it placed third. GPlace has since been improved with a finer grain congestion solution based on cell density and improved legalization. The legalization scheme now fully legalizes the placement, removing the need for a packing stage and allowing for fine grain adjustments to the placement throughout the flow. GPlace-flat is on average $5.3\times$ faster than GPlace-pack and is able to obtain a further 22.5% improvement in wirelength and a 40.0% improvement in routing time compared to GPlace-pack. The solver of GPlace is based on proven parallelization techniques that are well suited to many-core architectures for runtime improvements. Future work will focus on improving the congestion resolution using more detailed routing estimates as well as parallelization of the entire placement flow.

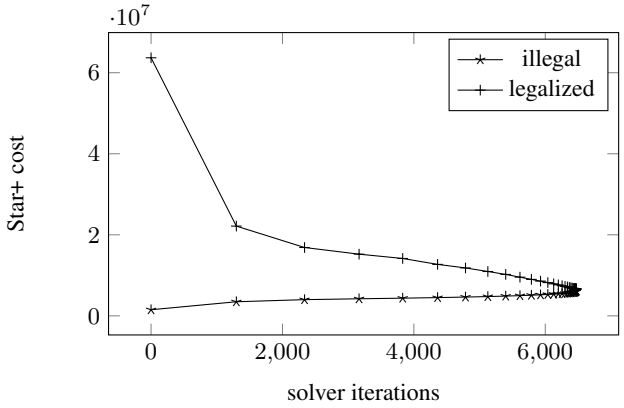


Figure 7: Upper and Lower Star+ Cost FPGA-3

7. REFERENCES

[1] T. Ahmed, P. D. Kundarewich, and J. H. Anderson. Packing techniques for Virtex-5 FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(3):18, 2009.

[2] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[3] S.-Y. Chen and Y.-W. Chang. Routing-architecture-aware analytical placement for heterogeneous FPGAs. In *Proceedings of the 52nd Annual Design Automation Conference*, page 27. ACM, 2015.

[4] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 647–654. IEEE Press, 2014.

[5] P. Gopalakrishnan, X. Li, and L. Pileggi. Architecture-aware FPGA placement using metric embedding. In *Proceedings of the 43rd annual Design Automation Conference*, pages 460–465. ACM, 2006.

[6] M. Gort and J. H. Anderson. Analytical placement for heterogeneous FPGAs. In *Field Programmable Logic and Applications (FPL), 22nd International Conference on*, pages 143–150. IEEE, 2012.

[7] P. Maidee, C. Ababei, and K. Bazargan. Timing-driven partitioning-based placement for island style FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(3):395–406, 2005.

[8] R. Pattison, S. Areibi, and G. Grewal. Scalable Analytic Placement for FPGAs on GPGPU. In *International Conference on Reconfigurable Computing and FPGAs*, pages 1–6, Cancun, Mexico, December 2015.

[9] J. Rose, J. Luu, C. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR project: Architecture and CAD for FPGAs from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86. ACM, 2012.

[10] R. Tessier and H. Giza. Balancing logic utilization and area efficiency in FPGAs. In *International workshop on Field Programmable Logic and Applications*, pages 535–544. Springer, 2000.

[11] M. Tom, D. Leong, and G. Lemieux. Un/DoPack: re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs. In *Proceedings of the IEEE/ACM international conference on Computer-aided design*, pages 680–687. ACM, 2006.

[12] D. Xie, J. Xu, and J. Lai. A new FPGA placement algorithm for heterogeneous resources. In *ASIC, ASICON'09. IEEE 8th International Conference on*, pages 742–746. IEEE, 2009.

[13] Xilinx. "UltraScale Architecture Configurable Logic Block User Guide". `http://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf`.

[14] M. Xu, G. Grewal, S. Areibi, C. Obimbo, and D. Banerji. StarPlace: An Efficient and Effective Analytic Method for FPGA Placement. *Integration, The VLSI Journal*, 44(3):192–204, June 2011.

[15] Y. Xu and M. A. Khalid. QPF: Efficient quadratic placement for FPGAs. In *Field Programmable Logic and Applications. International Conference on*, pages 555–558. IEEE, 2005.

[16] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal. Routability-driven FPGA placement contest. In *Proceedings of the International Symposium on Physical Design*, pages 139–143. ACM, 2016.