# Placement Algorithms for Large-Scale Heterogeneous FPGAs

Wuxi Li

Ph.D. Dissertation Proposal, University of Texas at Austin
September, 2018

**Abstract**

In recent years, the drastically enhanced architecture and capacity of Field-Programmable Gate Array (FPGA) devices have led to the rapid growth of customized hardware acceleration for modern applications, such as machine learning, cryptocurrency mining, and high-frequency trading. However, this growing capability raises ever more challenges to FPGA placement engines. A modern FPGA device often consists of heterogeneous logic resources that are unevenly distributed across the layout. This heterogeneity and nonuniformity bring difficulties to achieve smooth and high-quality placement convergences. In addition, FPGA devices also contain complex clocking architectures to deliver flexible clock networks. The physical structure of these clock networks, however, are pre-manufactured, unadjustable, and of only limited routing resources. Conventional placement approaches without clock feasibility consideration, hence, can easily lead to clock routing failures and fail the entire FPGA implementation flow. Finally, given the special standing of FPGAs in fast prototyping and frequent reprogramming, the implementation time is a crucial determining factor to get customers' favor. Therefore, as a runtime bottleneck of the whole FPGA implementation flow, ultra-fast and efficient placement engines are highly desirable.

In my Ph.D. study, I propose various algorithms and methodologies to address the aforementioned challenges, including (1) two high-performance placement core engines that explore different methodologies to deliver high-quality solutions on modern heterogeneous large-scale FPGAs; (2) a generic clock-aware placement algorithm to ensure clock network feasibility; (3) an efficient global placement parallelism scheme exploiting today's multi-core systems to significantly reduce placement runtime.

# Contents

# 1 Introduction

The Field Programmable Gate Array (FPGA) is a type of pre-manufactured integrated circuit designed to be configured by customers or designers. Historically, due to the limited logic resources, FPGAs were only used for fast realization of small digital circuits. In recent years, the drastically enhanced architecture and capacity of FPGAs have led to the rapid growth of customized hardware acceleration for modern applications, such as machine learning, cryptocurrency mining, and high-frequency trading. However, this growing capability of FPGA devices brings ever more challenges to FPGA placement engines.

Figure 1 illustrates a representative column-based FPGA architecture that has been widely adopted by many state-of-the-art commercial FPGA devices (e.g., Xilinx *Virtex UltraScale* and *UltraScale+* series [1]). In this specific architecture, each column provides one type of logic resource among *digital signal processor* (DSP), *random access memory* (RAM), I/O, and *configurable logic block* (CLB), where each CLB site further consists of multiple *lookup table* (LUT) and *flip-flop* (FF) slots. Placement is an essential stage in FPGA implementation flows that assigns exact physical locations for circuit components (e.g., LUTs, FFs, DSPs, and RAMs) within the FPGA layout (e.g., Fig. 1). Given the significance of placement in determining the overall implementation quality and efficiency, a superior placer must optimize various objectives to ensure the implementation performance and feasibility. Common placement optimization objectives include wirelength, routability, timing, and power. Among them, minimizing wirelength is typically treated as the most important objectives, since it is a good first-order approximation of the other three metrics (routability, timing, and power). However, unfortunately, even the pure-wirelength minimization placement problem is extremely challenging and has been proved to be $\mathcal{NP}$-hard [2]. Therefore, core placement algorithms and methodologies that can effectively and efficiently explore the very large placement solution space are urgently desired.

The placement challenges also come from the layout constraints imposed by modern FPGA clocking architectures. Modern FPGA devices often contain complex clocking architectures to achieve high-performance and flexible clock networks. The physical structure of these clock networks, however, are pre-manufactured, unadjustable, and of only limited routing resources. As shown in Fig. 1, the clocking architecture of an FPGA device typically consists of a grid of *clock regions*. Due to the predetermined number of pre-manufactured clock routing tracks, each clock region can only accommodate a limited number of clock networks routing through. With the recent increase in design complexity, there can be tens to even hundreds of global clocks in a single FPGA design. As a result, placement without careful clock network planning can easily lead to clock routing failures and fails the whole implementation flow. Thus, novel placement approaches with clock

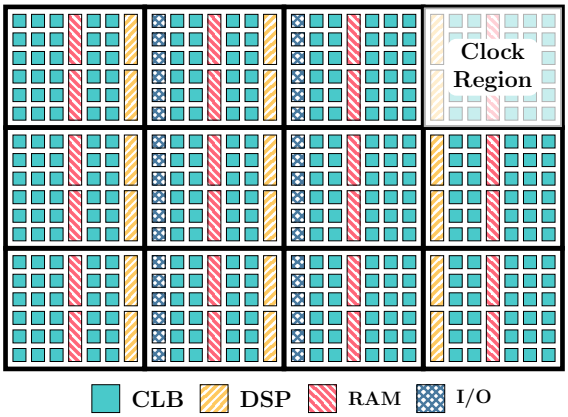



Figure 1: A representative column-based FPGA architecture adopted in Xilinx UltraScale and UltraScale+ series. This device is composed of 3 × 4 clock regions.

awareness are becoming imperative to ensure design closure of modern FPGAs.

As a type of hardware accelerators, FPGAs need to be frequently reconfigured to adapt rapid and continuous changes from users in many scenarios. Therefore, as a runtime bottleneck of the whole FPGA implementation flow, ultra-fast and efficient placement engines are highly desirable to get customers' favor. With the recent aggressive evolution of FPGA devices, the gate count of state-of-the-art commercial FPGAs has been growing steadily and reached the scale of millions [1,3]. However, due to the plateau of CPU single-core performance in the past decade, placement algorithms no longer automatically run faster as prescribed by Moore's Law [4]. In order to alleviate this runtime crisis, one promising solution is leveraging today's powerful multi-core CPUs to achieve efficient placement parallelism. Therefore, innovative approaches to achieve satisfiable placement parallelism under quality requirement is also a key challenge we are facing.

In my Ph.D. study, I propose various algorithms and methodologies to address the aforementioned challenges. I present two distinct core placement methodologies that explore the very large solution space of placement problem in different ways. Both methodologies can achieve state-of-the-art solution quality with competitive runtime. In addition, a generic clock-aware placement framework is proposed to ensure clock network feasibility during placement. I also propose an efficient global placement parallelism scheme that can significantly reduce placement runtime with controllable quality degradation. The key contributions include the following.

- A packing-based core placement engine that exploits physical clustering techniques to deliver high-quality placement solutions.
- A packing-free core placement engine that explores the very large solution spaces in a massively parallel manner inspired by the *College Admission Problem* [5].

- A generic clock-aware placement framework based on an iterative minimum-cost flow approach to achieve clock-feasible solutions with minimized quality degradation.
- An efficient global placement parallelism scheme leveraged by the block-Jacobi preconditioning and additive multigrid method [6] to achieve significant runtime speedup with satisfiable solution quality.

# 2 Core FPGA Placement Algorithm and Methodology Exploration

## 2.1 UTPlaceF: A Packing-Based FPGA Placement Engine

In a packing-based FPGA implementation flow, logic synthesis and technology mapping first translate a design into a netlist consisting of *lookup tables* (LUTs) and *flip-flops* (FFs). In the packing stage, LUTs and FFs are first clustered into *basic logic elements* (BLEs) and then BLEs are further grouped into *configurable logic blocks* (CLBs). After packing, placement is responsible for determining the physical locations of all CLBs and complex blocks (DSPs/RAMs) while optimizing some cost metrics (e.g., wirelength, routability, timing, power, etc.). Finally, routing is conducted to physically connect all the placed blocks.

Packing algorithms typically can be divided into three different categories: (1) seed-based approaches, (2) partitioning-based approaches, and (3) placement-guided and cluster-merging-based approaches. Seed-based packing approaches iteratively choose a BLE to form an initial CLB, then keep adding other unpacked BLEs into the CLB based on an attraction function until no more BLEs can be added. VPack [7], T-VPack [8], RPack [9], iRAC [10], T-NDPack [11], and MO-Pack [12] are representative examples of seed-based algorithms with different objectives and attraction functions. Partitioning-based approaches, like [13] and PPack [14], first apply a k-way partitioning to get a set of potential CLBs, and then perform a sequence of inter-partition moves to legalize the packing solution. HDPack [15] is an example of placement-guided and cluster-merging-based methods. It incorporates physical information using a min-cut-partitioning based global placement and applies the idea of *hybrid first choice clustering* (HFCC) from [16] to recursively group clusters with the highest attraction until no more merging could be performed.

FPGA placement algorithms are very similar to ASIC's placement and typically fall into one of the following three categories: (1) simulated-annealing-based approaches, (2) min-cut-partitioning-based approaches, and (3) analytical approaches. Simulated annealing based placers, like VPR [17], SCPlace [18], and [19], apply a probabilistic searching to approximate the global optimal solution. Min-cut-partitioning based placers, e.g. [20, 21], recursively apply min-cut partitioning until cells are fully spread out. Analytical placers typically approximate cost metrics, like wirelength and bin density, with a smooth objective function, then use numeric solvers to find the optimal solution. Different analytical placement approaches have been extensively

studied in [22–30].

### 2.1.1 Preliminaries and Overall Flow

#### 2.1.1.1 FPGA Architecture

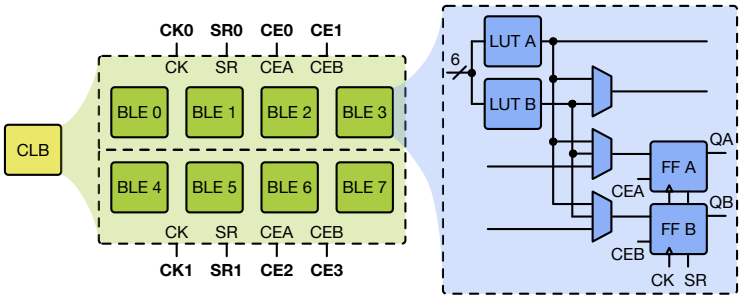



Figure 2: CLB and BLE in Xilinx UltraScale architecture.

We adopt the Xilinx UltraScale VU095 [1] used in the ISPD 2016 FPGA placement contest [31] as the target FPGA device. Its BLE and CLB architectures are shown in Fig. 2. In this particular architecture, a CLB slice consists of 8 BLEs and each BLE further contains 2 LUTs and 2 FFs. The 2 LUTs in a BLE can be either implemented as a single 6-input LUT or 2 smaller LUTs with a total number of distinct inputs no greater than 5. The 2 FFs in a BLE must share the same clock (CK) and set/reset (SR) signals, however, their clock enable (CEA and CEB) signals can be different. A CLB can be divided into two half CLBs, each of which consists of 4 BLEs that share the same set of CK, SR, CEA, and CEB.

#### 2.1.1.2 Quadratic Placement

A FPGA netlist can be represented as a hypergraph $H = (V, E)$, where $V = \{v_1, v_2, ..., v_{|V|}\}$ is the set of cells, and $E = \{e_1, e_2, ..., e_{|E|}\}$ is the set of nets. Let $\mathbf{x} = \{x_1, x_2, ..., x_{|V|}\}$ and $\mathbf{y} = \{y_1, y_2, ..., y_{|V|}\}$ be the $x$ and $y$ coordinates of all cells. The wirelength-driven global placement problem is to determine position vectors $\mathbf{x}$ and $\mathbf{y}$ that minimize the total wirelength and obey bin density constraint. Wirelength is measured by the half-perimeter wirelength (HPWL),

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \sum_{e \in E} \{\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j|\}. \tag{1}$$

As HPWL is not differentiable everywhere, quadratic placers approximate it by squared Euclidean distance between cells. Therefore, the wirelength cost function in quadratic placer is defined as,

$$W(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_x^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_y^T \mathbf{y} + const. \tag{2}$$

#### 2.1.1.3 UTPlaceF Overall Flow

Fig. 3 shows the flowchart of UTPlaceF. The overall flow is composed of four parts: (1) *flat initial placement* (FIP), (2) *physical and congestion aware packing* (PCAP), (3) global placement, and (4) legalization and detailed placement.

FIP is responsible for generating cells' physical locations, and detecting cells that are likely to be placed into routing congested regions to better guide packing. In the packing stage (PCAP), LUTs and FFs are first grouped into BLEs, then BLEs are clustered into CLBs. We assume that FIP yields the optimal cell relative position, and packing should not perturb it too much. Therefore PCAP, with cell physical locations information, disallows long-distance packing and prefers close packing. Similar to iRAC, absorbing small nets is treated as the main objective during packing stage of PCAP to reduce channel width and routing demand, which in turn improves wirelength and routability. Besides considering grouping connected cells, PCAP also packs unconnected cells based on their physical locations to further reduce the number of CLBs. Leveraged by routing congestion information from FIP, PCAP can perform loose packing only for cells that are likely to be placed into routing hotspots, and avoid blindly depopulating throughout whole netlists for routability enhancement. By using this congestion-aware depopulation technique, PCAP is able to achieve both good wirelength and routability.

Our global placement basically shares the same framework with FIP but handles CLBs instead of LUTs and FFs. In detailed placement stage, a bipartite-matching-based minimum-pin-movement legalization is applied first. Then a hierarchical independent set matching is performed to further reduce wirelength. To preserve the routability optimized global placement solution, white spaces and cells in congested regions are handled specially throughout the detailed placement stage.

### 2.1.2 Flat Initial Placement

Our FIP adopts the main framework of an ASIC placer *POLAR 2.0* [32]. Its overall flow is shown in Fig. 4. In each iteration of wirelength-driven placement, a quadratic program is solved followed by rough legalization [33] for reducing cells overlaps. Then the density preserving global move [34] is applied to improve the wirelength of the rough-legalized placement while preserving bin densities. A sequence of pure wirelength-driven placement iterations is performed until the gap between the upper bound wirelength and the lower bound wirelength is less than a certain number, which is 50% in PCAP. In the routability-driven placement stage, after a certain number of placement iterations, a fast global router NCTUgr [35] is called for routing congestion estimation, then similar to POLAR 2.0, cells in congested regions are inflated by a small ratio, and the inflation accumulates to the end of FIP. Different from the first stage, DSPs, RAMs, and I/O

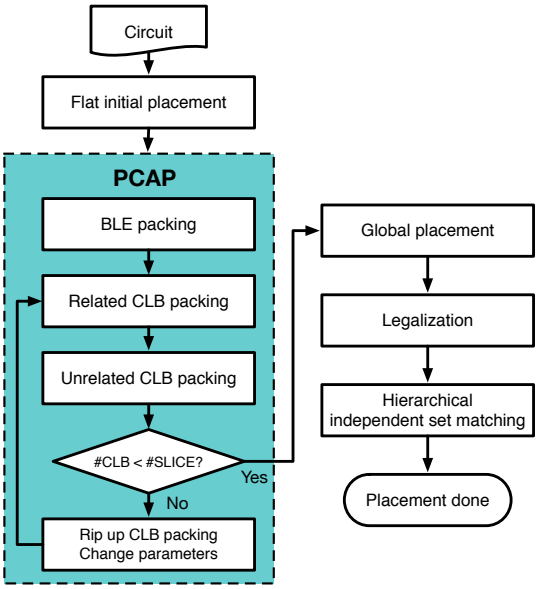



Figure 3: Overview of UTPlaceF.

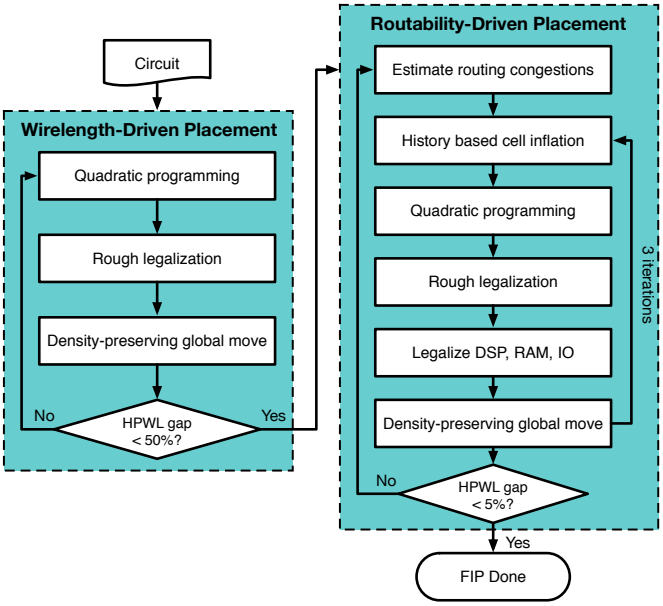



Figure 4: Overall flow of FIP.

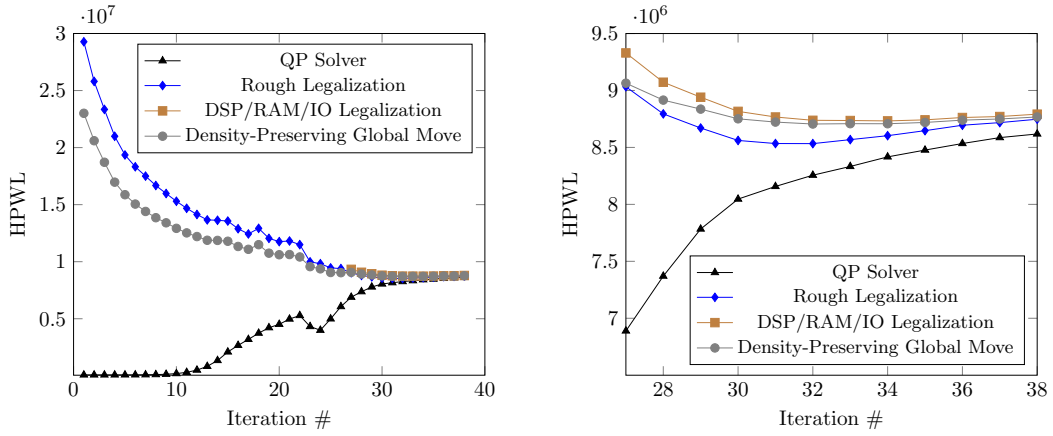



Figure 5: HPWL at different steps in FIP of benchmark FPGA-5. (a) All placement iterations. (b) Placement iterations in routability-driven stage.

blocks are immediately legalized after rough legalization in this stage. Since sites for these cells are typically discrete and scattered on FPGAs, if we handle them like LUTs and FFs in rough legalization, they might be far away from their legal positions in the final FIP solution. This discrepancy would introduce inaccuracy of cell relative positions into FIP. To eliminate this discrepancy, UTPlaceF performs an extra legalization step for DSPs, RAMs, and I/Os right after the conventional rough legalization in the second stage of FIP. By simply doing this, DSPs, RAMs, and I/Os will use their legal positions as their anchor points in placement iterations, and the discrepancy will be eliminated in the final FIP solution. The legalization approach here

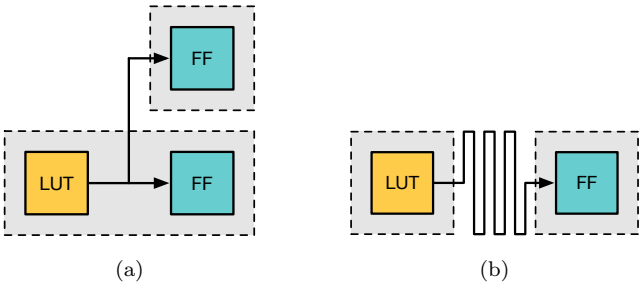



Figure 6: Different LUT and FF pairing scenarios. (a) LUT fanouts to multiple FFs, and groups with the closest one. (b) LUT fanouts to one FF that is far away and the grouping is rejected.

will be further discussed in Section 2.1.3.2. Fig. 5(a) illustrates the progression of the placement solution with respect to HPWL in different steps. A zoomed-in view of the last few iterations, with the legalization of DSPs, RAMs, and I/Os enabled, is shown in Fig. 5(b).

The two objectives of FIP are: (1) generating physical locations for each LUT and FF, and (2) detecting LUTs and FFs that are in routing congested regions. Cell physical locations are explicitly generated by the wirelength and routability co-optimized placement. Congestion information associated with cells is implicitly obtained from history based cell inflation. The insight of cell inflation is quantifying the possibility of a cell lying in routing congested regions using its area – a larger cell area indicates a larger possibility of being placed into congested regions. After FIP, each LUT and FF would have a physical location and a cell area, which indicates the congestion level associated with it.

#### 2.1.2.1 Physical and Congestion Aware Packing

#### 2.1.2.2 Max-Weighted-Matching-Based BLE Packing

As a BLE in our target FPGA architecture, Xilinx Ultrascale, contains 2 LUTs and 2 FFs, existing VPR-style BLE packing algorithms cannot be directly applied. To address this new BLE architecture, we propose a two-step BLE packing algorithm that comprises: (1) LUT and FF pairing, and (2) LUT-FF pairs matching.

In our *physical and congestion aware packing* (PCAP), we apply the LUT and FF pairing in a similar manner to the BLE packing in VPack. As shown in Fig. 6, we group each LUT to the closest FF in its fanout. Besides that, long-distance packing is rejected, and only packing within *maximum packing distance of BLE* ($\overline{\lambda_b}$) is allowed. This step is mainly to make full use of fast connections between LUTs and FFs that are in the same BLEs.

In the second step, *max-weighted matching* is used for finalizing the BLE packing. We construct an undirected weighted graph $UWG = (V, E)$, where each $v_i$ in $V = \{v_1, v_2, ..., v_{|V|}\}$ is an LUT-FF pair, a single LUT, or a single FF. $E = \{e_1, e_2, ..., e_{|E|}\}$ represents the set of legal mergings. To apply high-attraction and

close packing, we say a merging $(v_i, v_j)$ is legal if and only if,

1. $v_i$ and $v_j$ are connected in the netlist.
2. Merging $v_i$ and $v_j$ into the same BLE does not violate any packing rules.
3. The merging attraction is greater than the *minimum packing attraction for BLEs* ($\underline{\phi_b}$).

In the UWG, edge weights are set as merging attractions. The attraction value for a merging is defined as,

$$\phi_b(v_i, v_j) = (1 - e^{\gamma_b(dist(v_i,v_j)-\overline{\lambda_b})}) \sum_{p \in Net(v_i \cap v_j)} \frac{k_b}{deg(p) - 1}. \tag{3}$$

where $\gamma_b$ is a constant value being experimentally set as 0.2, $dist(v_i, v_j)$ is the Manhattan distance between $v_i$ and $v_j$, $\overline{\lambda_b}$ is the maximum packing distance of BLE which is 4 in PCAP, $Net(c_i \cap c_j)$ is the set of nets shared between $v_i$ and $v_j$, $deg(p)$ is the total number of pins of net $p$ that is exposed in cluster level, and $k_b$ is 2 for 2-pin nets and 1 for other nets.

The first term, $1 - e^{\gamma_b(dist(v_i,v_j)-\overline{\lambda_b})}$, is a packing distance penalty factor in range $(-\infty, 1 - e^{-\gamma_b\overline{\lambda_b}})$. This factor is very close to 1 for mergings of distance much less than $\overline{\lambda_b}$. It drops quickly as $dist(v_i, v_j)$ gets close to $\overline{\lambda_b}$, and becomes negative once $dist(v_i, v_j)$ is greater than $\overline{\lambda_b}$. By using the first term, short-distance mergings in PCAP are always preferred. The second term, $\sum_{p \in Net(v_i \cap v_j)} \frac{k_b}{deg(p)-1}$, is introduced for reducing the number of nets exposed in cluster level, which in turn improves wirelength and routability. With the second term, merging two clusters that share more small nets is of high priority, and 2-pin nets are given even higher weight by the factor $k_b = 2$. In PCAP, the minimum packing attraction for BLEs ($\underline{\lambda_b}$) is set to 0 by default.

Fig. 7 shows a simple cluster matching example. Due to our rules for legal mergings, the constructed UWG typically is not connected and comprises many small connected subgraphs. Mergings in different connected subgraphs are independent, so PCAP performs a max-weighted matching algorithm on each of these subgraphs and all matched cluster pairs would be merged. The location of a merged cluster is set as the average location of all cells (LUTs and FFs) it contains, and the cluster area is simply the sum of cell areas. After each pass of matching and merging, PCAP rebuilds the UWG for clusters generated from the previous stage and resolves the max-weighted matching for each new connected subgraph until no more legal merging exists. The pseudo-code of the max-weighted-matching-based BLE packing is summarized in Alg. 1.

**Algorithm 1** Max-Weighted-Matching-Based BLE Packing

**Input:** Post-FIP netlist.
**Output:** BLE level netlist with external nets reduced.

```
Pair LUTs and FFs;
while legal cluster merging exists do
    Construct an UWG using Eq. (3);
    for each connected subgraph do
        Run max-weighted matching;
        for each matched edge do
            Merge corresponding clusters;
            Set location and area of the merged cluster;
        end for
    end for
end while
```

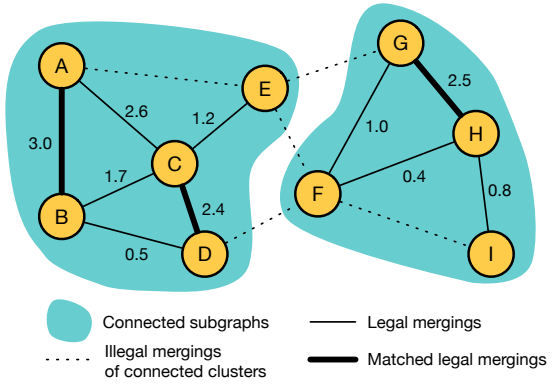



Figure 7: A simple max-weighted cluster matching example.

#### 2.1.2.3 Related CLB Packing with Congestion-Aware Depopulation

After BLE packing, we create a CLB for every single BLE. CLB packing is done by successively merging smaller CLBs into larger ones. CLBs that share common nets are said to be related, and in this stage, only related CLBs mergings are considered.

The *BestChoice Clustering* (BC) [36] is used as our main engine for related CLB packing. In BC, the attractions of all legal CLB mergings are calculated first, then the algorithm iteratively merges CLB pairs with the highest attraction using a priority queue (PQ). The location and area of a merged CLB are set as the average location and total area of cells it contains, respectively. After each merging, the legality and attraction of mergings related to the new CLB are updated accordingly. The pseudo-code of our BC-based related CLB packing is summarized in Alg. 2.

In PCAP, a related CLB merging $(c_i, c_j)$ is said to be legal if and only if

1. $c_i$ and $c_j$ are connected in the netlist.
2. Merging $c_i$ and $c_j$ into the same CLB does not violate any packing rules.
3. The merging attraction is greater than the *minimum packing attraction for related CLBs* ($\underline{\phi_{rc}}$).
4. Total area of $c_i$ and $c_j$ is no greater than the *maximum CLB area* ($\overline{a_c}$).

The first three rules are inherited from our BLE packing. The fourth rule is introduced to perform congestion-aware depopulation and avoid overpacking in routing congested regions. Note that all the cell areas used in the fourth rule are from the accumulated cell inflation in FIP. As discussed in Section 2.1.2, cells with larger areas indicate a higher possibility to be placed into routing congested regions. By constraining area of each CLB, PCAP would apply loose packing in routing congested regions, and tight packing in other

**Algorithm 2** BC-based Related CLB Packing

**Input:** Post-BLE-packing netlist.
**Output:** CLB level netlist with external nets reduced.

Create an empty priority queue PQ;
Find all legal mergings $(c_i, c_j)$ and their attraction $\phi_{rc}(c_i, c_j)$;
Insert all legal mergings with $\phi_{rc}(c_k, c_j) \geq \underline{\phi_{rc}}$ into PQ;
**while** PQ is not empty **do**
    Pick merging $(c_i, c_j)$ with the highest attraction from PQ;
    **if** $c_i$ or $c_j$ has been merged already **then**
        continue;
    **end if**
    Merge $c_i$ and $c_j$ to form $c_{ij}$;
    Set location and area of $c_{ij}$;
    **for** each CLB $c_k$ that connects to $c_{ij}$ **do**
        **if** merging $(c_k, c_{ij})$ violates packing rules **then**
            continue;
        **end if**
        Calculate attraction $\phi_{rc}(c_k, c_{ij})$;
        **if** $\phi_{rc}(c_k, c_{ij}) \geq \underline{\phi_{rc}}$ **then**
            Insert $(c_k, c_{ij})$ into PQ;
        **end if**
    **end for**
**end while**

**Algorithm 3** CLB Packing and Rip-up and Repacking

**Input:** BLE packing is done.
**Input:** Maximum FPGA CLB utilization $U_{max}$, maximum unrelated CLB packing distance incrasing rate $\beta_{\overline{\lambda_{uc}}}$, and minimum related CLB packing attraction increasing rate $\Delta\underline{\phi_{rc}}$.
**Output:** Maximum FPGA CLB utilization constraint is satisfied.

**while** true **do**
    Perform related CLB packing
    Perfrom unrelated CLB packing
    **if** CLB utilization $\leq U_{max}$ **then**
        **return**
    **end if**
    $\overline{\lambda_{uc}} \leftarrow \overline{\lambda_{uc}} \cdot \beta_{\overline{\lambda_{uc}}}$
    $\underline{\phi_{rc}} \leftarrow \underline{\phi_{rc}} + \Delta\underline{\phi_{rc}}$
**end while**

regions. This congestion awareness makes PCAP able to achieve a good trade-off between wirelength and routability.

The attraction function of related CLB mergings $(c_i, c_j)$ is defined as,

$$\phi_{rc}(c_i, c_j) = (1 - e^{\gamma_{rc}(dist(c_i,c_j) - \overline{\lambda_{rc}})}) \sum_{p \in Net(c_i \cap c_j)} \frac{k_{rc}}{deg(p) - 1}. \quad (4)$$

Eq. (4) is basically a replica of our BLE packing attraction function defined in Eq. (3), but differs only by some constant parameters. We experimentally set $\gamma_{rc}$ to 0.2, and $\overline{\lambda_{rc}}$ to 6. $k_{rc}$ is 2 for 2-pin nets and 1 for other nets. The minimum packing attraction for related CLBs ($\underline{\phi_{rc}}$) is set to 0.1, and the maximum CLB area ($\overline{a_c}$) is set as 1.8 times average CLB area by default.

#### 2.1.2.4 Size-Prioritized $K$-Nearest-Neighbor Unrelated CLB Packing

CLBs without common nets are said to be unrelated. After related CLB packing stage, unrelated CLB mergings are considered. Different from related CLB packing, in which reducing external nets is the main objective, unrelated CLB packing aims to reduce the number of CLBs.

BC-based approaches typically could yield very good packing solutions for given attraction functions. However, they have an inherent drawback – inability of making tight packing. Generally, BC would generate a large number of medium-sized clusters that are difficult to merge further due to the cluster capacity

constraint. To mitigate this issue, we proposed a size-prioritized BC-based unrelated CLB packing. By assigning higher priority to mergings producing larger CLBs, medium-sized CLBs would be promoted quickly. As a result, much tighter packing solutions can be achieved.

Unlike the related CLB packing technique in Alg. 2, where all merging candidates are in one single PQ, we have a separate PQ for each merging size in the unrelated CLB packing. In other words, merging candidates are separated by the number of BLEs in their resulting CLBs, and only mergings result in same BLE count could be placed into the same PQ. The PQ corresponding to larger merging size is grant higher priority and always be processed first.

Within each PQ, BC-based unrelated CLB packing is performed in a manner similar to our related CLB packing. For a CLB, however, instead of considering all its connected CLBs, its $K$-nearest neighbors (in terms of physical distance) within distance $\overline{\lambda_{uc}}$ would be considered in our unrelated CLB packing. Besides, a different attraction function defined in Eq. (5) is used.

$$\phi_{uc}(c_i, c_j) = 1 - e^{\gamma_{uc}(dist(c_i,c_j)-\overline{\lambda_{uc}})}. \tag{5}$$

The attraction function $\phi_{uc}$ is a packing distance penalty factor similar to the first terms of Eq. (3) and Eq. (4). In UTPlaceF, we set $\gamma_{uc}$ to 0.2, $\overline{\lambda_{uc}}$ to 8, and $K$ to 30 by default. Note that, although the objective of our unrelated CLB packing is to reduce the number of CLBs and deliver tight packing, the congestion-aware depopulation technique described in Section 2.1.2.3 is still applied in this stage to maintain good routability.

For high-utilization designs, our default unrelated CLB packing might still not be able to generate tight enough packing solutions that satisfy FPGA capacity constraint. In this case, the existing packing solution will be ripped up, and new related and unrelated CLB packing parameters will be adopted to generate a tighter packing solution in the next packing pass. PCAP would iteratively perform this rip-up and repacking loop until the FPGA capacity constraint is satisfied. The details of this rip-up and repacking phase will be further discussed in Section 2.1.2.5.

#### 2.1.2.5 Net Reduction and Packing Tightness Trade-off

Our related CLB packing works effectively to reduce the number of external nets, however, it often yields relatively loose packing due to the inherent shortcoming of BC mentioned in section 2.1.2.4. In contrast, our unrelated CLB packing is capable of aggressively reducing the number of CLBs and achieving tight packing. Therefore, if more packing is performed in the related CLB packing stage, a loose packing solution with less external nets would be delivered. However, if we only do a small portion of packing in the related CLB packing stage and leave most of the work to unrelated CLB packing, the final packing would be more inclined

to the "tight" side with more external nets.

In PCAP, the minimum related CLB packing attraction ($\underline{\phi_{rc}}$) and the maximum unrelated CLB packing distance ($\overline{\lambda_{uc}}$) are used to control the amount of packing work for each (related/unrelated) CLB packing stage. Initially, $\underline{\phi_{rc}}$ is set as 0.1 to aggressively reduce the number of external nets, and $\overline{\lambda_{uc}}$ is set as 8 to only allow close packing in unrelated CLB packing stage. This initial setting typically results in a loose packing with a large amount of net reduction. For high-utilization designs, however, the packing solution generated by the initial setting could be sparse to the extent that the number of CLBs exceeds the FPGA capacity. To address this problem, PCAP would discard the existing CLB packing solution (but respect BLE packing solution) and perform a repacking step, which applies related and unrelated CLB packing again. In the repacking phase, however, $\underline{\phi_{rc}}$ is increased to reduce related CLB packing, and $\overline{\lambda_{uc}}$ is also increased to allow unrelated CLB packing of longer distance. As results, the repacking step would achieve tighter packing but sacrifice net reduction. The repacking step is repeated until the CLBs utilization target is satisfied.

The pseudo-code of our rip-up and repacking is summarized in Alg. 3. In UTPlaceF, we experimentally set $U_{max}$ to 0.999, $\Delta\underline{\phi_{rc}}$ to 0.3, and $\beta_{\overline{\lambda_{uc}}}$ to 1.414.

### 2.1.3 Post-Packing Placement

#### 2.1.3.1 Global Placement

After PCAP, the global placement is performed immediately to further optimize wirelength and routability. Our global placement shares the same framework and parameter settings with FIP, but instead of optimizing flat LUT/FF netlist, it considers each CLB as a whole. It is an incremental placement using the FIP solution as the starting point to speed up wirelength convergence. Since the initial CLB-level placement induced from FIP is more or less close to the optimal solution, we skip the wirelength-driven phase in Fig. 4 and directly apply the routability-driven phase to further reduce the runtime. To avoid global placement being stuck in the local optimal around FIP, the weight of pseudo-nets for cell spreading is reduced at the beginning of global placement.

#### 2.1.3.2 Min-Cost Bipartite Matching Based Legalization

A notable difference between ASIC and FPGA legalization is that ASIC standard cells have different dimensions whereas FPGA CLBs have the same size. Because of this special property, FPGA legalization problem can be formulated as a min-cost-max-cardinality bipartite matching problem with pin movement as cost. By solving the corresponding bipartite matching problem, global placement can be legalized with minimum total pin movement. However, solving a complete bipartite matching for large designs is impractical in

**Algorithm 4** Min-Cost Bipartite Matching Based Legalization

**Input:** Rough-legalized CLB level netlist.
**Output:** Legalized placement with minimum movement.

```
1: Split the placement region into non-overlapping windows;
2: Sort windows by their routing congestion in descending order;
3: for each unlegalized window do
4:     while Num. CLBs > Num. slices do
5:         Add neighboring unoccupied slices into the window;
6:     end while
7:     Run min-cost bipartite matching between CLBs and slices with
   Manhattan distance as cost.
8:     Move each CLB to its matched slice, and set the slice as occu-
   pied.
9: end for
```

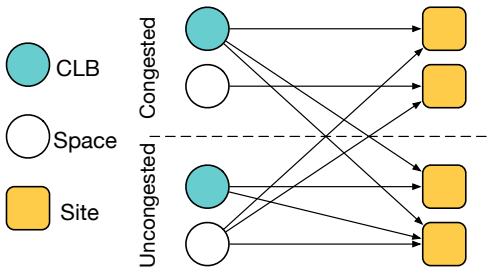



Figure 8: Illustration of our congestion-aware ISM.

terms of runtime. To address this problem, we partition the placement region into a set of uniform rectangle partitions, then apply a min-cost bipartite matching for each partition. To further speed up runtime, edges in bipartite graphs are pruned based on Manhattan distance and are incrementally added when necessary. Our legalization approach is summarized in Alg. 4.

Similar to CLBs, heterogeneous blocks like DSPs, RAMs, and I/Os also have the regularity of sizes, so they are legalized separately using Alg. 4 with minor variations in UTPlaceF as well.

#### 2.1.3.3 Congestion-Aware Hierarchical Independent Set Matching

The idea of bipartite matching can also be applied to optimize wirelength. For a given set of legalized cells, a wirelength optimization problem can be formulated as a min-cost bipartite matching with edge weights as HPWL increase of moving cells to different sites. However, solving this matching problem cannot guarantee the optimal HPWL improvement, since the edge weight of a cell depends on the positions of other connected cells in the same matching set. To overcome this drawback, we adopt the *independent set matching* (ISM) idea from NTUPlace3 [37] and only apply matching within a set of cells that do not share any nets. Besides, white spaces are also considered in our matching to further increase the solution space.

In UTPlaceF, ISM is hierarchically applied to CLBs, BLEs and LUT pairs. One main objective of our packing stage (PCAP) is to absorb small nets into clusters (BLEs, CLBs). Therefore, most CLBs essentially are clusters of LUTs and FFs that have strong connectivity. One of our key observation is that moving cells with strong connectivity together helps to jump out of local optima in terms of wirelength, so ISM is applied to CLBs first in UPlaceF. However, even though most CLBs contain strongly connected cells, they are clustered only based on physical distance and connectivity but are not aware of wirelength. Thus ISM for BLEs and LUT pairs are introduced to fix our CLB packing and BLE packing respectively after CLB level ISM.

The ISM works effectively for optimizing HPWL. However, it could ruin the local cell density optimized for routability, especially when spaces are considered in our ISM. To mitigate this problem, we propose a congestion-aware ISM with three extra constraints introduced: 1) cells can be moved out of but not into routing congested regions, 2) spaces can be moved into but not out of congested regions, and 3) moves within congested regions are disallowed. Fig. 8 shows a simple matching example with the extra constraints applied. To get accurate congestion information, the routing congestion map is updated after a certain number of ISM iterations. By applying our congestion-aware ISM, HPWL can be optimized without routability degradation.

#### 2.1.4 Experimental Results

UTPlaceF was implemented in C++ and tested on a Linux machine with 3.40 GHz CPU and 32GB RAM. The benchmark suite released by Xilinx for ISPD'16 FPGA placement contest was used to validate the effectiveness of UTPlaceF. Related executables, placement solutions, and benchmarks are released at link (`http://wuxili.net/project/utplacef/`).

##### 2.1.4.1 Benchmark Characteristics

Table 1: ISPD'16 Placement Contest Benchmarks Statistics

| Benchmark | #LUT | #FF | #RAM | #DSP | #Ctrl Set |
|---|---|---|---|---|---|
| FPGA-1 | 50K | 55K | 0 | 0 | 12 |
| FPGA-2 | 100K | 66K | 100 | 100 | 121 |
| FPGA-3 | 250K | 170K | 600 | 500 | 1281 |
| FPGA-4 | 250K | 172K | 600 | 500 | 1281 |
| FPGA-5 | 250K | 174K | 600 | 500 | 1281 |
| FPGA-6 | 350K | 352K | 1000 | 600 | 2541 |
| FPGA-7 | 350K | 355K | 1000 | 600 | 2541 |
| FPGA-8 | 500K | 216K | 600 | 500 | 1281 |
| FPGA-9 | 500K | 366K | 1000 | 600 | 2541 |
| FPGA-10 | 350K | 600K | 1000 | 600 | 2541 |
| FPGA-11 | 480K | 363K | 1000 | 400 | 2091 |
| FPGA-12 | 500K | 602K | 600 | 500 | 1281 |
| Resources | 538K | 1075K | 1728 | 768 | - |

The characteristics of ISPD'16 benchmark suite are listed in Table 1. This benchmark suite has cell count ranging from 0.1 to 1.1 million, which is much larger than existing academic FPGA benchmarks. Note that several benchmarks have extremely high cell utilization, which raises two requirements to FPGA placement packing and placement engines: 1) the capability to yield tight packing solutions to satisfy the CLB capacity constraint, and 2) the capability to reduce routing resource demand, since little white space is available for cell and routing demand spreading.

#### 2.1.4.2 Comparison with Previous Works

We compare our results with the top 3 winners of ISPD'16 placement contest and other state-of-the-art FPGA placers. The results are shown in Table 2 and Table 3. All routed wirelength are reported by Xilinx Vivado v2015.4, and runtime of the contest winners are evaluated on a Linux Machine with 3.20 GHz CPU and 32GB RAM. Normalized results in the last row of Table 2 and Table 3 are based on comparisons with our results, and only benchmarks that other placers completed are considered in each comparison. It can be seen that UTPlaceF achieves the best overall routed wirelength. On average UTPlaceF outperforms by 6.2%, 11.6%, 29.1%, 3.7%, 7.3%, and 16.8% in routed wirelength compared with the top 3 contest winners, [28], RippleFPGA [29], and GPlace [30] respectively. It should be noted that only UTPlaceF and RippleFPGA are able to route all 12 benchmarks. In terms of runtime, as all placers are evaluated on different machines, it is not fair to compare them directly. However, we still can see that the runtime of UTPlaceF is only worse than GPlace, and is about 1.5× to 3.1× faster than other placers.

#### 2.1.4.3 Runtime Analysis

The runtime breakdown of UTPlaceF is shown in Table 4. On average, 55.0% of the total runtime is taken by FIP, while PCAP, global placement, and hierarchical ISM respectively take 16.2%, 5.1% and 21.4% of the total runtime, and legalization only takes 1.6% of the total runtime. PCAP is further divided into three components: BLE packing takes 0.5% of the total runtime, related CLB packing takes 2.3% of the total runtime, and the remaining 13.4% is taken by the unrelated CLB packing. In hierarchical ISM, CLB, BLE, and LUT-pair level ISM respectively take 2.6%, 6.7%, and 12.1% of the total runtime.

Table 2: Comparison with ISPD'16 Contest Winners on ISPD 2016 Benchmark Suite

| Benchmark | 1st Place | | 2nd Place | | 3rd Place | | UTPlaceF | |
|---|---|---|---|---|---|---|---|---|
| | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) |
| FPGA-1 | * | - | 379932 | 118 | 581975 | 97 | 356769 | 185 |
| FPGA-2 | 677877 | 435 | 679878 | 208 | 1046859 | 191 | 642108 | 305 |
| FPGA-3 | 3223042 | 1527 | 3660659 | 1159 | 5029157 | 862 | 3215087 | 831 |
| FPGA-4 | 5628519 | 1257 | 6497023 | 1149 | 7247233 | 889 | 5409765 | 824 |
| FPGA-5 | 10264769 | 1266 | † | - | † | - | 9659958 | 1237 |
| FPGA-6 | 6630179 | 2920 | 7008525 | 4166 | 6822707 | 8613 | 6487628 | 1041 |
| FPGA-7 | 10236827 | 2703 | 10415871 | 4572 | 10973376 | 9169 | 10104837 | 1721 |
| FPGA-8 | 8384338 | 2645 | 8986361 | 2942 | 12299898 | 2741 | 7879022 | 1686 |
| FPGA-9 | † | - | 13908997 | 5833 | † | - | 12369055 | 2537 |
| FPGA-10 | * | - | * | - | † | - | 8794515 | 3182 |
| FPGA-11 | 11091383 | 3227 | 11713479 | 7331 | † | - | 10196038 | 2151 |
| FPGA-12 | 9021769 | 4539 | * | - | † | - | 7755443 | 2944 |
| Norm. | +6.2% | 1.55× | +11.6% | 2.30× | +29.1% | 3.10× | +0.0% | 1.00× |

*: Placement error. †: Unroutable placement.

Table 3: Comparison with State-of-the-Art Academic FPGA Placers on ISPD 2016 Benchmark Suite

| Benchmark | [28] * | | RippleFPGA [29] | | GPlace [30] | | UTPlaceF | |
|---|---|---|---|---|---|---|---|---|
| | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) | Routed WL | Runtime(s) |
| FPGA-1 | 384709 | 215 | 362563 | 74 | 493788 | 30 | 356769 | 185 |
| FPGA-2 | 652690 | 399 | 677563 | 167 | 903099 | 61 | 642108 | 305 |
| FPGA-3 | 3181331 | 1555 | 3617466 | 1037 | 3908244 | 289 | 3215087 | 831 |
| FPGA-4 | 5504083 | 1289 | 6037293 | 621 | 6277878 | 280 | 5409765 | 824 |
| FPGA-5 | 10068879 | 1237 | 10455204 | 1012 | † | - | 9659958 | 1237 |
| FPGA-6 | 6411247 | 2827 | 6960037 | 2772 | 7643382 | 600 | 6487628 | 1041 |
| FPGA-7 | 10040562 | 2588 | 10248020 | 2170 | 11255351 | 691 | 10104837 | 1721 |
| FPGA-8 | 8113483 | 2705 | 8874454 | 1426 | 9323360 | 734 | 7879022 | 1686 |
| FPGA-9 | 13616625 | 3407 | 12954350 | 2683 | 14002965 | 974 | 12369055 | 2537 |
| FPGA-10 | 8866049 | 4091 | 8564363 | 5555 | † | - | 8794515 | 3182 |
| FPGA-11 | 10834629 | 3267 | 11226088 | 3636 | 12367773 | 923 | 10196038 | 2151 |
| FPGA-12 | 8246410 | 4625 | 8928528 | 9748 | † | - | 7755443 | 2944 |
| Norm. | +3.7% | 1.47× | +7.3% | 1.61× | +16.8% | 0.38× | +0.0% | 1.00× |

* This is the preliminary version of UTPlcaeF that was published on ICCAD'16.

Table 4: Runtime Breakdown of UTPlaceF

| Benchmark | FIP | PCAP | | | GP | Legalization | Hierarchical ISM | | | Others | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | BLE | Rel. CLB | Unrel. CLB | | | CLB | BLE | LUT Pair | | |
| FPGA-1 | 115 | 1 | 2 | 3 | 9 | 1 | 5 | 26 | 22 | 1 | 185 |
| FPGA-2 | 187 | 2 | 4 | 4 | 17 | 1 | 8 | 38 | 42 | 2 | 305 |
| FPGA-3 | 528 | 5 | 12 | 13 | 58 | 1 | 28 | 57 | 120 | 9 | 831 |
| FPGA-4 | 500 | 6 | 12 | 17 | 56 | 2 | 33 | 61 | 127 | 10 | 824 |
| FPGA-5 | 663 | 7 | 13 | 23 | 77 | 3 | 41 | 67 | 136 | 11 | 1041 |
| FPGA-6 | 1029 | 8 | 44 | 171 | 101 | 30 | 49 | 77 | 197 | 15 | 1721 |
| FPGA-7 | 974 | 9 | 52 | 195 | 126 | 19 | 54 | 81 | 205 | 16 | 1731 |
| FPGA-8 | 1004 | 9 | 26 | 39 | 91 | 16 | 49 | 174 | 274 | 4 | 1686 |
| FPGA-9 | 1217 | 13 | 100 | 538 | 119 | 41 | 60 | 116 | 314 | 19 | 2537 |
| FPGA-10 | 1343 | 11 | 65 | 1154 | 95 | 59 | 60 | 148 | 229 | 18 | 3182 |
| FPGA-11 | 1248 | 12 | 50 | 133 | 115 | 23 | 55 | 203 | 308 | 4 | 2151 |
| FPGA-12 | 1720 | 12 | 66 | 279 | 116 | 107 | 54 | 226 | 346 | 18 | 2944 |
| Norm. | 55.0% | 0.5% | 2.3% | 13.4% | 5.1% | 1.6% | 2.6% | 6.7% | 12.1% | 0.7% | 100.0% |

## 2.2 A New Paradigm for FPGA Placement without Explicit Packing

Given the significance of FPGA placement and packing in determining the overall implementation quality and efficiency, lots of research efforts have been devoted to them over the past two decades. Figure 9 summarizes several representative placement and packing flows in previous works. In the early age of FPGAs, *Pack-Place-Legalize* flows (the red path), such as [7–14, 17], dominate industry and academic research. In this type of flow, the packing solution is first determined based on logical interconnects, then the placement and legalization are performed successively to produce a legal solution. Despite the efficiency, *Pack-Place-Legalize* flows do not incorporate placement/spatial information during their packing decision making, thus are likely to cause poor design quality. *Place-Pack-Place-Legalize* flows (the blue path) then emerged as a remedy to this issue [15, 38–40]. In such a flow, a flat initial placement (FIP) is first performed. Then, both logical and spatial information is considered during the packing stage before the final placement and legalization. Another category of flows, namely *Place-SemiPack-Legalize* flows (the orange path), blur the boundary between placement and packing [41, 42]. In particular, after a FIP similar to that in *Place-Pack-*

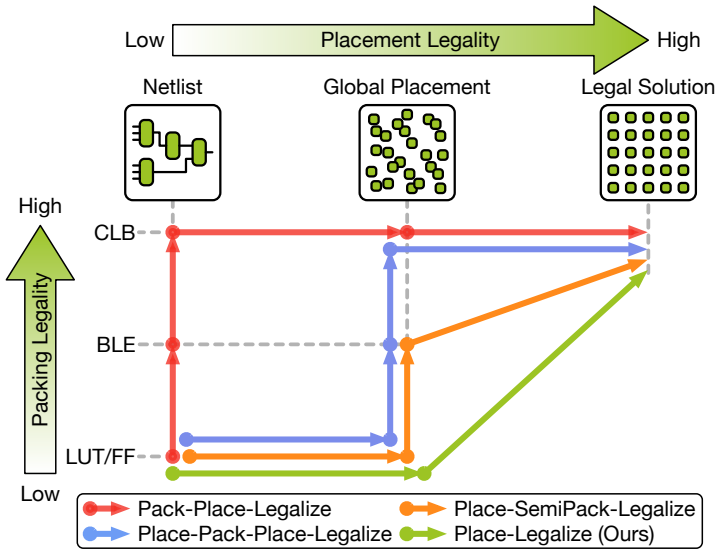



Figure 9: Representative FPGA placement and packing flows.

*Place-Legalize* flows, they only group LUTs and FFs into intermediate clusters (e.g., BLEs). Then, the rest of packing work and the final placement are combined into a single legalization process.

Among all these methodologies, *Place-Pack-Place-Legalize* flows (the blue path) currently dominate the state-of-the-art industrial FPGA CAD tools [38]. However, large discrepancies between FIP solutions and final legal solutions can still be observed, which implies that metrics, like wirelength, timing, and routability, that are carefully optimized in FIPs can be ruined in final legal solutions. The reason is usually twofold. Firstly, most existing FIP techniques only seek to optimize the placement without considering the effect of packing. As a result, large perturbations can be introduced in the later packing stage, especially for those hard-to-pack designs. Secondly, when forming a BLE/CLB in the packing stage, its location is typically estimated by the average location of its containing cells, which, however, can be far away from its final legal position.

To remedy the aforementioned deficiencies in previous works, we propose a new paradigm for FPGA placement without explicit packing. We call it *Place-Legalize* flow. As shown in Fig. 9, in the proposed flow (the green path), a final legal solution can be achieved directly from a FIP by incorporating the previously separated packing and final placement/legalization steps. Our experiments show that the overall implementation quality, as well as the correlation between FIPs and final legal solutions, can be significantly improved with the proposed flow.

#### 2.2.0.1 The FPGA Direct Legalization Problem

In our *Place-Legalize* flow, the *direct legalization* (DL) is a step that produces a legal solution directly from a FIP. Given the notations defined in Table 5, the FPGA DL problem can be defined as follows:

$$\max_{\mathbf{x},\mathbf{y}} \quad \sum_{s\in\mathcal{S}} \phi(\{v \in \mathcal{V} \mid z_{v,s} = 1\}) - \lambda \cdot \text{HPWL}(\mathbf{x},\mathbf{y}), \tag{6a}$$

$$\text{s.t.} \quad \sum_{s\in\mathcal{S}} z_{v,s} = 1, \forall v \in \mathcal{V}, \tag{6b}$$

$$\{v \mid v \in \mathcal{V}, z_{v,s} = 1\} \text{ is arch. legal}, \forall s \in \mathcal{S}, \tag{6c}$$

$$|x_v - x'_v| + |y_v - y'_v| \leq D, \forall v \in \mathcal{V}. \tag{6d}$$

The objective (6a) is to maximize the total clustering score $\sum_{s\in\mathcal{S}} \phi(\{v \in \mathcal{V} \mid z_{v,s} = 1\})$ and minimize a wirelength term $\lambda \cdot \text{HPWL}(\mathbf{x},\mathbf{y})$ normalized by a positive parameter $\lambda$. The clustering score function $\phi(c)$ typically captures the pin/net sharing, timing impact, and so on, within each CLB slice, like affinity functions used in conventional packing algorithms [8,10,12]. In general, $\phi(c)$ can be customized for different optimization targets. The details of the objective setting used in our framework will be elaborated in Section 2.2.2.3. The constraint (6b) guarantees that each LUT and FF is assigned to one and only one CLB slice. The constraint (6c) assures that all the architecture rules stated in Section 2.1.1.1 are satisfied. The maximum displacement constraint (6d) is introduced to better preserve the FIP. We treat (6d) as a soft constraint, since a legal solution may not always exist for a given $D$.

Unlike previous approaches, our DL formulation guarantees both placement and packing legality while optimizing the objective (6a). Besides, the maximum displacement is explicitly considered. It is, however, much harder to be dealt with in traditional methods (e.g., *Place-Pack-Place-Legalize* flow).

Table 5: Notations used in the direct legalization problem

| | |
|---|---|
| $\mathcal{V}$ | The set of LUTs and FFs |
| $\mathcal{S}$ | The set of CLB slices available on the target FPGA device |
| $\mathbf{x}', \mathbf{y}'$ | The x and y coordinates of cells in $\mathcal{V}$ in FIP |
| $\mathbf{x}, \mathbf{y}$ | The x and y coordinates of cells in $\mathcal{V}$ in the final legal solution |
| $z_{v,s}$ | Binary variables that represent if cell $v \in \mathcal{V}$ is assigned to CLB slice $s \in \mathcal{S}$ |
| $\phi(c)$ | The clustering score of a set of LUTs and FFs $c$ |
| $D$ | The cell maximum displacement constraint |

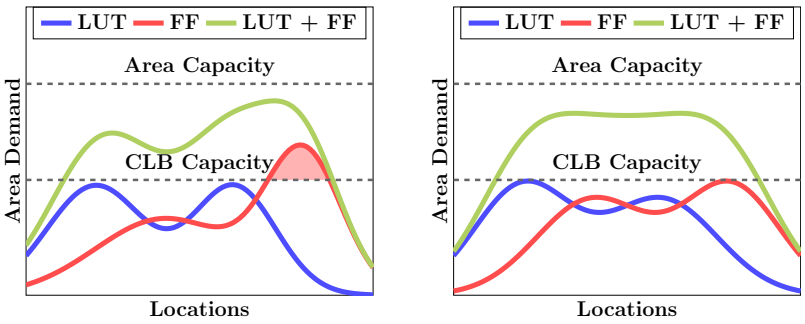



Figure 10: (a) An area overflow-free placement but with FF demand overflow (the shaded red region). (b) A legal placement.

### 2.2.1 Challenges of Place-Legalize Flow

Although *Place-Legalize* flows have lots of potential merits, there are several new challenges come into the field with it.

To achieve a smooth DL process, the FIP needs to be as near as possible to a legal solution. Otherwise, the final legal solution cannot be obtained with a small placement perturbation. In a FIP, each cell is associated with an area, and the final FIP solution heavily depends on the cell area assignment. Therefore, a proper cell area assignment is essential for achieving a reasonably legal FIP. In most of the previous works, cell areas were set statically based on some empirical estimations [30, 41]. However, the area of a cell should be largely determined by its resource demand, which is, in fact, determined by its packing solution. For example, if a FF occupies (the FF portion of) a half CLB slice alone in the final solution due to the control set conflict, it should be assigned a larger area in the FIP. Considering packing solutions are not actually available in FIPs, how to model the effect of packing in cell area assignment is indeed a challenge.

Another important problem in FIP is how to properly distribute cells of different types. In quadratic placers, to achieve a rough-legal placement, overlapping removal techniques (e.g., rough legalization) distribute cells by evening out area demand throughout the layout. However, an area overflow-free placement can be far away from a truly legal solution, since the area metric alone cannot capture utilizations of different cell types. This issue can be better illustrated in Fig. 10. Here the LUT and FF area demands (the blue curve and the red curve) represent the numbers of CLBs required by LUTs and FFs, respectively, in different locations. The CLB capacity (the lower dashed line) represents the numbers of CLB slices available in each location, and the area capacity (the upper dashed line) is the maximum LUT + FF area constraint used by the placers. In Fig. 10(a), despite the satisfaction of the LUT + FF area constraint, large displacement will still be introduced in the later legalization step due to the FF demand overflow (the shaded red region). Figure 10(b) gives a legal case where both LUT and FF demands are lower than the CLB capacity. This

issue, of course, can be avoided by over-constraining the area capacity, but at the cost of resource wasting.

To legalize a placement, many previous works adopted *Tetris*-like approaches [43]. In such a greedy approach, only one cell/cluster is considered and legalized at a time, which leads to a narrow solution space exploration. To explore a broader solution space, however, much more expensive computational effort is typically required. The scalability issue is even more severe in the *Place-Legalize* flow, since the flat netlist without any pre-clustering is directly considered. Therefore, how to explore a sufficiently large solution space while maintaining good runtime scalability is also a challenge for the *Place-Legalize* flow.

To sum up, there are several issues discussed in this section that need to be resolved before we can confidently adopt the *Place-Legalize* flow. No existing work has considered these issues, therefore, how to overcome them is a major challenge of this work.

### 2.2.2 Proposed Algorithms

#### 2.2.2.1 Overall Flow

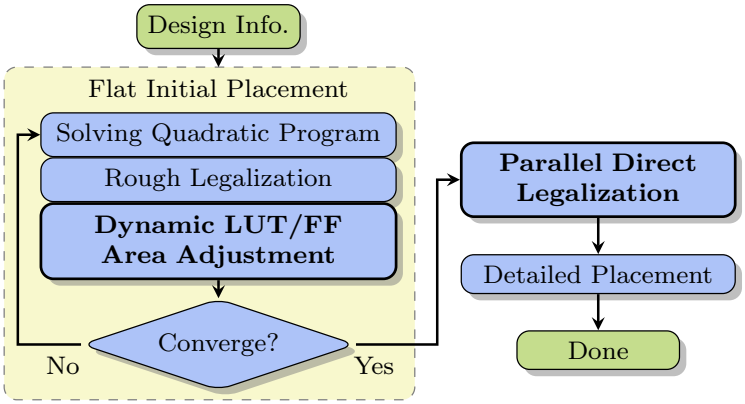



Figure 11: The proposed overall flow.

The proposed overall flow is illustrated in Fig. 11. The whole flow starts with a flat initial placement (FIP). Besides the conventional quadratic program solving and rough legalization, a new dynamic LUT/FF area adjustment step is performed after each placement iteration. This new step is aiming at resolving the two FIP-related issues pointed out in Section 2.2.1. More specifically, the area of each LUT and FF is dynamically adjusted to account for the impact of both packing and utilizations of different cell types. Once the FIP converges to a roughly legal solution, a high-quality legal placement can be straightly produced by the parallel direct legalization, in which the Formulation (6a) – (6d) is solved. The detailed placement then conducts further optimization on the legal placement before the final solution is delivered.

As the centerpieces of this work, the dynamic LUT/FF area adjustment and the parallel direct legalization

techniques will be detailed in Section 2.2.2.2 and Section 2.2.2.3, respectively.

#### 2.2.2.2 Dynamic LUT/FF Area Adjustment

Since packing and cell utilization calculation are both very local-scoped in nature, it is reasonable to analyze them in a small spatial neighborhood context for each cell. Additionally, considering that cells of different types do not affect the packing legality or compete for logic resources against each other, it is also reasonable to consider each cell type individually from the perspective of solution legality. Therefore, the area of each cell can be largely determined based on its spatial neighbors of the same type.

In this section, a cell $u$ is said to be a neighbor of a cell $v$ if $u$ and $v$ have the same cell type and $u$ falls into the box region $(x_v - L, y_v - L, x_v + L, y_v + L)$ centering at $v$, where $(x_v, y_v)$ is the location of $v$ and $L$ is a constant being empirically set to 5 CLB slices in our framework. In the rest of this section, we will use $\mathcal{N}_v$ to denote the set of neighbors of $v$, and use $\mathcal{N}_v^+$ to denote $\{v\} \cup \mathcal{N}_v$.

**Area Updating** To determine a cell area, the following three aspects are considered in our framework: (1) the cell resource demand, (2) the local resource utilization, and (3) the routability impact. As discussed in Section 2.2.1, the resource demand of a cell is mainly determined by its packing solution, so the effect of packing is our major consideration here. Besides, the local resource utilization also needs to be considered and, as pointed out in Section 2.2.1, this should be done for different cell types (e.g., LUT and FF) individually. We also take the routability impact into consideration to avoid over-congested solutions.

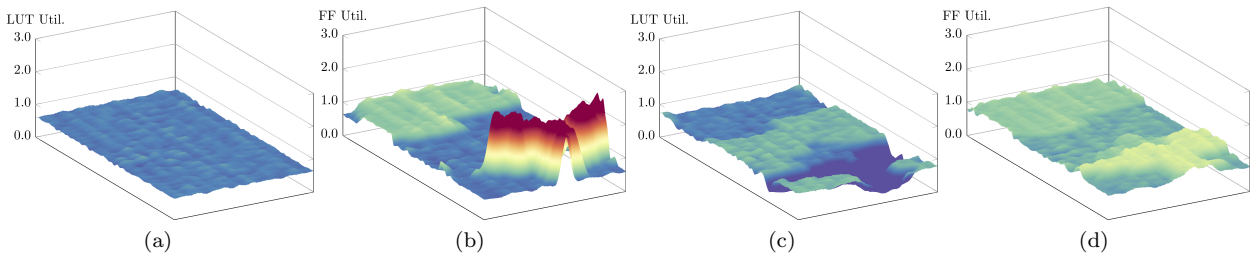
Given a LUT (FF) $v$, we first define the local LUT (FF) utilization at $v$, denoted by $U_v$, as follows:

$$U_v = \frac{\sum_{i \in \mathcal{N}_v^+} A_i}{C_v}, \tag{7}$$

where $C_v$ denotes the number of CLB slices within the box region $(x_v - L, y_v - L, x_v + L, y_v + L)$ we used to define $\mathcal{N}_v$, and $A_i$ denotes the LUT (FF) resource demand of the cell $i$. The magnitude of $A_i$ here can also be interpreted as the degree of difficulty to pack $i$. The methods to compute $A_i$ for LUTs and FFs will be detailed in Section 2.2.2.2 (Eq. (9)) and Section 2.2.2.2 (Eq. (10)), respectively.

We then define the new area of $v$, denoted by $a_v$, as follows:

$$a_v = \begin{cases} \min(a'_v \beta_+,\ A_v U_v \gamma_v), & \text{if } A_v U_v \gamma_v > a'_v, \\ \max(a'_v \beta_-,\ A_v U_v \gamma_v), & \text{if } A_v U_v \gamma_v < a'_v \text{ and } R_v < R_{\max}, \\ a'_v, & \text{otherwise,} \end{cases} \tag{8}$$




(a) (b) (c) (d)

Figure 12: (a) and (b) are the 3-D LUT and FF utilization (Eq. (7)) maps without our dynamic area adjustment. (c) and (d) are the ones with it applied. The area constraint is satisfied in both placement solutions. This is an example of the challenging case illustrated in Fig. 10. The experiments are based on design *FPGA-10* in ISPD 2016 benchmark suite [31].

where $a'_v$ denotes the current area of $v$, $\gamma_v \geq 1$ denotes the cumulative inflation ratio of $v$ for routability optimization, $R_v$ denotes the local routing utilization at $v$, and $R_{\max}$ is an empirically determined constant representing the maximum routing utilization for area shrinking. Besides, $\beta_+ > 1$ and $\beta_- < 1$ are two parameters to control the rates of area increasing/decreasing.

Intuitively, if a LUT (FF) $v$ is hard to pack (large $A_v$) and is located in a region with high LUT (FF) utilization (large $U_v$), as well as high routing congestion (large $\gamma_v$), its area will be inflated. Otherwise, we will shrink its area to allow other cells to get in, but only when the region is not routing-congested ($R_v < R_{\max}$). To achieve a smooth area adjustment, $\beta_+$ and $\beta_-$ are set to 1.1 and 0.95, respectively. The $\gamma_v$ is cumulatively updated using a history-based cell inflation technique similar to [39, 41]. The routing congestion is estimated by a fast global router NCTUgr [35], and $R_{\max}$ is empirically set to 0.65 in our framework.

In the proposed flow, we perform rough legalization for LUTs and FFs together using the same fence regions to maintain the relative order between them. Considering LUTs and FFs do not occupy the same logic resources, adjusting their areas directly targeting to their resource demand $A_v$ (Eq. (9) and Eq. (10)) can result in low resource usage. This issue is prevented by scaling $A_v$ using the local resource utilization $U_v$ in Eq. (8). By doing so, cells in low-utilization regions will be assigned areas that are much smaller than their actual resource demand to leave spaces for other cells.

Our dynamic area adjustment technique can effectively mitigate the "unbalanced resource issue" illustrated in Fig. 10, and resolve "packing hotspots" that are harmful to the smoothness of the subsequent direct legalization process. Figure 12 shows the LUT/FF utilization (Eq. (7)) maps of a design with/without this technique when area constraint is satisfied (see Fig. 10). It can be seen that, with this technique, a huge FF hotspot (Fig. 12(b)) is resolved (Fig. 12(d)). More interestingly, the LUTs that are original in the FF hotspot (Fig. 12(a)) tend to follow the movement of their connected FFs, which results in a "valley" (Fig. 12(c)). This is actually an expected behavior, since it implicitly preserves the relative cell ordering,

which is important for the placement quality.

**LUT Resource Demand Estimation** Now we present the method to estimate the resource demand (the $A_i$ in Eq. (7)) of a LUT, or more precisely, the amount of LUT portion in a CLB needed by a given LUT. Recall that, in the target device, each CLB contains 8 BLEs and each BLE can accommodate up to 2 LUTs if they are architecturally compatible. Therefore, each LUT can take 1 or 1/2 BLE, which are equivalent to 1/8 and 1/16 CLB in a compact packing solution. Considering a packing solution is not yet available in a FIP, a probabilistic estimation defined in Eq. (9) is used instead for a LUT $v$ with its neighbors $\mathcal{N}_v$.

$$A_v^{(\mathrm{LUT})} = \frac{|\widehat{\mathcal{N}}_v|}{|\mathcal{N}_v|} \cdot \frac{1}{16} + \frac{|\mathcal{N}_v| - |\widehat{\mathcal{N}}_v|}{|\mathcal{N}_v|} \cdot \frac{1}{8}, \tag{9}$$

where $\widehat{\mathcal{N}}_v$ denotes the set of LUTs in $\mathcal{N}_v$ that can be fitted into the same BLE with $v$. Intuitively, if a LUT is architecturally compatible with most of its neighbors, its resource demand will tend to be small, otherwise, it will end up with a larger resource demand.

**FF Resource Demand Estimation** The resource demand (the $A_i$ in Eq. (7)) estimation for FFs is more subtle due to their complicated control set rules. Given a FF $v$, one can, of course, enumerate all possible packing solutions of $\mathcal{N}_v^+$ and come up with a weighted sum similar to Eq. (9) to get an average-case estimation. However, this method is too computationally expensive to be practical. Instead, we turn to a best-case estimation based on the tightest packing solution, which can provide us a firm lower bound of the FF resource demand.

For the notation simplicity, here we define every 4 FFs that share the same (CK, SR, CE) in a CLB slice as a *quarter CLB*[1], and define every 8 FFs that share the same (CK, SR) in a CLB slice as a *half CLB*[1]. For instance, in Fig. 2, the 4 "FF A"s in BLE 0 – 3 is a quarter CLB and the 8 FFs in BLE 0 – 3 is a half CLB.

An instant observation is that, to produce the tightest packing solution, FFs with the same control set need to be packed together as much as possible. Given this observation, our approach to estimate the lower-bound FF demands can be illustrated by Fig. 13. Using the (CK, SR) group of (B, P) in Fig. 13 as an example, there are 5 and 2 FFs with CE of X and Z. In the tightest packing solution, at least $\lceil 5/4 \rceil = 2$ and $\lceil 2/4 \rceil = 1$ quarter CLBs are required for the control sets (B, P, X) and (B, P, Z). Since any two of these $2 + 1 = 3$ quarter CLBs can be fitted into the same half CLB, there will be a minimum of $\lceil 3/2 \rceil = 2$ half CLBs for the (B, P) group. Considering the control sets of any two half CLBs are independent, we can safely set the resource demand of each half CLB as 1/2 (of a CLB). Therefore, the minimum demand of the (B, P) group will be $1/2 \cdot 2 = 1$. After that, we can divide this demand of 1 into $2/(2 + 1) = 2/3$ and

[1]The quarter/half CLBs here only contain FFs, since LUTs are irrelevant in this subsection.

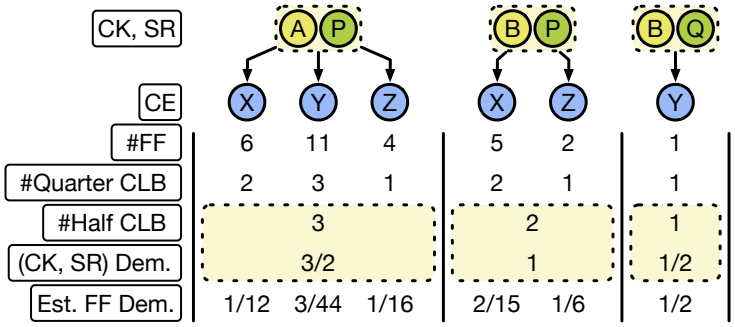



Figure 13: Illustration of FF resource demand estimation.

$1/(2+1) = 1/3$ based on the quarter CLB counts in (B, P, X) and (B, P, Z). Finally, the resource demands of each FF in (B, P, X) and (B, P, Z) can be estimated as $2/3/5 = 2/15$ and $1/3/2 = 1/6$, respectively, by evenly distributing the demand of 2/3 and 1/3 to 5 and 2 FFs.

To generalize the above discussion, let $v$ be a FF with control set ($\mathrm{CK}_0$, $\mathrm{SR}_0$, $\mathrm{CE}_0$), and let {$\mathrm{CE}_0$, $\mathrm{CE}_1$, ..., $\mathrm{CE}_m$} be the set of CE nets in $\mathcal{N}_v^+$. If we denote the number of FFs in $\mathcal{N}_v^+$ with the control set ($\mathrm{CK}_0$, $\mathrm{SR}_0$, $\mathrm{CE}_i$) as $n_i$, for $0 \leq i \leq m$, the estimated FF demand of $v$ can be expressed as follows:

$$A_v^{(\mathrm{FF})} = \frac{1}{2} \cdot \frac{\lceil \frac{n_0}{4} \rceil}{\sum_{i=0}^{m} \lceil \frac{n_i}{4} \rceil} \cdot \left\lceil \frac{\sum_{i=0}^{m} \lceil \frac{n_i}{4} \rceil}{2} \right\rceil \cdot \frac{1}{n_0} \cdot \alpha^{(\mathrm{FF})}. \tag{10}$$

Note that our analysis in Fig. 13 corresponds to the tightest packing, which realizes the lower bound FF demands. In practice, however, a packing solution is likely to be looser than that when considering net sharing and other optimization metrics. Therefore, we introduce $\alpha^{(\mathrm{FF})} \geq 1$ in Eq. (10) to give more flexibility to the packing. In our framework, we empirically set $\alpha^{(\mathrm{FF})}$ to 1.1.

If we ignore the $\alpha^{(\mathrm{FF})}$ term, the FF demand estimated by Eq. (10) is in the range $[1/16, 1/2]$, which implies that FF demands can vary up to 8× (e.g., the FF demands of (A, P, Z) and (B, Q, Y) in Fig. 13). In a traditional flow, the discrepancy between FIPs and legal solutions is mainly from their incapability of capturing this large demand variance, as shown in Fig. 12. Therefore, our FF demand estimation method detailed in this section is essential for the proposed *Place-Legalize* flow.

It is worthwhile to mention that, although the LUT/FF resource demand estimations elaborated in Section 2.2.2.2 and Section 2.2.2.2 are specific to the architecture detailed in Section 2.1.1.1, the same idea is also applicable to other FPGA devices with different architectures.

#### 2.2.2.3 Fully Parallelizable Direct Legalization

Our direct legalization (DL) takes a rough-legal FIP and produces a legal solution by solving the Formulation (6a) – (6d). The idea is partially similar to the clustering algorithms in [38, 44], but we are aiming at a legal placement directly. Besides, the cell displacement is explicitly considered in our formulation. Compared to traditional greedy methods, our method can explore a significantly larger solution space by exploiting the strength of parallelism.

For the simplicity, we will use "slice" to denote "CLB slice" in this section.

**A College Admission Analogy** The proposed DL algorithm is inspired by the *College Admission Problem* [5]. One can think that each slice is a "college", each cell is a "student", and cells sharing common nets are "friends". Then, DL can be regarded as a process of colleges admitting students. By using this analogy, the DL process is fully parallelizable in nature in a sense that all colleges can make decisions simultaneously. Our DL formulation, however, is more complicated than the original college admission problem for the following two reasons: 1) a student (cell) rates a college (slice) not only based on the college (slice) itself, but also depending on the decision making of its friends (connected cells); 2) some students (cells) cannot be in the same college (slice) for the FPGA architectural legality.

**The Node-Centric Algorithm** The key idea of our DL algorithm is that each slice finds the cluster of cells that fits it best, then offers this cluster to all the cells involved. "Admission" happens when all the cells in this cluster accept this "offer". The process of sending and accepting/rejecting "offers" between slices and cells is iteratively performed until no potential "admission" could happen anymore. Different slices here can create cluster candidates and make their own decisions independently. Moreover, a cluster can be simultaneously created and considered by multiple slices, which implicitly explores the solution space of placement together with packing. This elegant property overcomes the drawback of all existing methods, where placement and packing cannot be considered at the same time.

In our DL algorithm, the computation is mainly centered at each slice. Thus, we call each slice as a *computation node* and each computation node can run in parallel against each other.

The node-centric DL algorithm flow at each computation node is shown in Fig. 14. Each computation node maintains a set of cells that have been determined in the slice ($det$), a priority queue of cluster candidates ($pq$), a set of neighbor cells ($nbr$), and a list of seed clusters ($scl$) for new candidate generation. The $pq$ stores the K best clusters found so far at each computation node and we use K = 10. Clusters are created by adding cells in $nbr$ into seed clusters in $scl$ in our algorithm. Besides, a variable $i$ is also maintained by each computation node to record the number of iterations since the last change of the best candidate in $pq$.

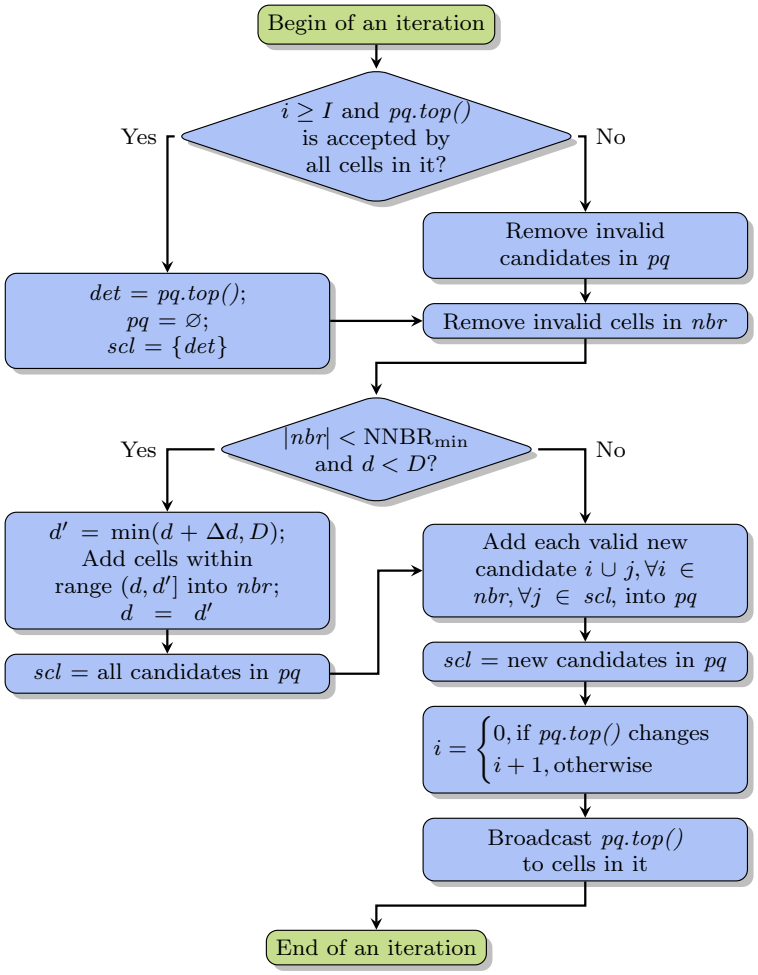



Figure 14: The node-centric DL algorithm flow at each computation node (slice).

We will use *pq.top()* to denote the best candidate in a *pq*. Before the DL starts, for each computation node, we initialize $i = 0$, $det = \varnothing$, $pq = \varnothing$, $nbr$ as the set of cells within a predefined distance $d$ (default is 1) to the slice, and $scl$ to contain an empty cluster without any cells.

In every DL iteration, as shown in Fig. 14, each computation node first checks if its *pq.top()* (the best candidate in *pq*) can be committed into the slice. The *pq.top()* will be committed only if it has been stable for a long enough time ($i \geq I$) and it has been accepted by all the cells in it. $I$ is set to 3 in our algorithm to prevent committing premature candidates. Once the *pq.top()* is valid to commit, we will update *det* as *pq.top()* and reset *pq*. To guarantee that all subsequent candidates contain the set of determined cells *det*, *scl* is set to only have *det* after each *pq.top()* committing.

If the *pq.top()* is not yet ready to commit, invalid candidates and cells will be removed from *pq* and *nbr*, respectively. A cell can become invalid if it has been added to another slice or it is no longer architecturally compatible with *det*. A cluster becomes invalid if one or more cells in it are invalid. After that, if there are

too few available cells in $nbr$ ($|nbr| < \text{NNBR}_{\min}$), the current maximum cell displacement constraint $d$ will be relaxed by $\Delta d$, and more neighbor cells will be added to guarantee the scope of solution space we can explore. Besides, we will also copy all candidates in $pq$ to $scl$ to guarantee that they will be considered by these newly added neighbor cells. To honor the ultimate maximum displacement constraint $D$, we will stop increasing $d$ when it reaches $D$. $\text{NNBR}_{\min}$, $\Delta d$, and $D$ are set to 10, 1, and 12, respectively, in our algorithm.

To create new cluster candidates, we add each cell in $nbr$ into each cluster in $scl$. Those valid new candidates will be added into $pq$. Meanwhile, we also store them in $scl$ for the candidate generation in the next DL iteration. After that, $i$ will be increased by 1 if *pq.top()* does not change, otherwise, we reset it to 0.

Finally, the computation node broadcasts its *pq.top()* to the involved cells and let each of them make a decision. The decisions of these cells will determine if this *pq.top()* can be committed in the next DL iteration.

After all computation nodes finish their broadcasting, a cell may receive multiple "offers" from a set of different slices $\mathcal{S}$. In such a case, the cell will select the slice $s \in \mathcal{S}$ with the highest score improvement $\Delta\text{SCORE}(s)$. The score improvement of a slice $s$ is defined as

$$\Delta\text{SCORE}(s) = \text{SCORE}(pq.top() \text{ of } s, s) - \text{SCORE}(det \text{ of } s, s), \tag{11}$$

where $\text{SCORE}(c,\ s)$ represents the score of cluster $c$ in slice $s$, and it will be given in Section 2.2.2.3. Intuitively, $\Delta\text{SCORE}(s)$ is the amount of score improvement by committing the *pq.top()* in slice $s$.

Algorithm 5 summarizes the whole fully parallelizable DL process. All computation nodes are initialized in parallel in line 1 – 7. In each DL iteration, the node-centric algorithm presented in Fig. 14 is executed by each slice individually in line 9 – 11. After that, each cell receives "offers" from slices with *pq.top()* containing it, and it picks the one with the highest score improvement $\Delta$SCORE (Eq. (11)) and inform its acceptance to the selected slice in line 12 – 16. The loop from line 8 to line 17 is repeated until no more valid candidate exists.

**Convergence Requirement** In general, a valid *pq.top()* may never be accepted by all the cells in it. In such a case, our DL algorithm will fall into an infinite loop. To guarantee the convergence of the algorithm, an extra requirement is imposed as stated in Lemma 1.

**Lemma 1** *Suppose $s_1$ and $s_2$ are two different computation nodes (slices). If $\Delta SCORE(s_1) \neq \Delta SCORE(s_2)$ holds for any choice of $s_1$ and $s_2$ in the same DL iteration, the convergence of the proposed DL algorithm is guaranteed.*

**Algorithm 5** Fully Parallelizable Direct Legalization

**Input:** A rough-legal placement, the set of cells $\mathcal{C}$, the set of slice (computation node) $\mathcal{S}$, and the initial maximum displacement constraint $d$.
**Output:** A legal placement.

**parallel for** each $s \in \mathcal{S}$ **do**
  $s.i \leftarrow 0$;
  $s.det \leftarrow \varnothing$;
  $s.pq \leftarrow \varnothing$;
  $s.nbr \leftarrow \{c \in \mathcal{C} \mid \text{dist}(c, s) \leq d\}$;
  $s.scl \leftarrow \{\text{an empty cluster}\}$;
**end parallel for**
**while** exists valid *pq.top()* for $s \in \mathcal{S}$ **do**
  **parallel for** each $s \in \mathcal{S}$ **do**
    Run the node-centric algorithm shown in Fig. 14;
  **end parallel for**
  **parallel for** each $c \in \mathcal{C}$ **do**
    Among $\{s \in \mathcal{S} \mid c \in s.pq.top()\}$, pick $s^\star$ that
    has the highest $\Delta\text{SCORE}(s^\star)$ defined in Eq. (11)
    and inform $s^\star$ that $c$ accepts its *pq.top()*;
  **end parallel for**
**end while**

Lemma 1 basically requires a tie-breaking mechanism for the case that multiple computation nodes offer the same score improvement (Eq. (11)) to a set of cells. In our implementation, a tie is broken based on the unique identifier of each computation node, since a cell can receive at most one "offer" from a computation node in each iteration.

**Score Function** Since our DL explores the solution spaces of placement and packing simultaneously, the score function needs to capture both placement- and packing-related metrics. Given a slice $s$ and a cluster $c$, the score of $c$ in $s$ is defined as follows:

$$\text{SCORE}(c, s) = \sum_{e \in \text{Net}(c)} \frac{\text{InternalPins}(e, c) - 1}{\text{TotalPins}(e) - 1} - \lambda \cdot \Delta\text{HPWL}(c, s), \tag{12}$$

where $\text{Net}(c)$ denotes the set of nets that have at least one cell in $c$, $\text{TotalPins}(e)$ denotes the total pin count of net $e$, $\text{InternalPins}(e, c)$ represents the number of pins of net $e$ in $c$, and $\Delta\text{HPWL}(c, s)$ represents the HPWL increase of moving cells in $c$ from their FIP locations to $s$. $\lambda$ is a positive weighting parameter, which is empirically set to 0.02 in our algorithm. The first term defines the clustering score $\phi(c)$ in Eq. (6a), and it here grants a higher score to clusters that absorb more external nets as internal ones, which can effectively reduce routing demands and improve routability. The second term gives a higher preference to candidates that lead to a large wirelength reduction.

**Parallelization Scheme** As illustrated in Algorithm 5, our DL algorithm is massively parallelizable. Like colleges independently making decisions in the analogy of college admission (Section 2.2.2.3), different computation nodes (slices) can execute the flow illustrated in Fig. 14 perfectly in parallel in each DL iteration (Algorithm 5 line 9 – 11). Moreover, cells can also process the results generated by slices and send back their decisions individually (Algorithm 5 line 12 – 16).

Since the number of slices and cells in a modern FPGA is typically at the scale of $10^4$ or more (e.g., our target FPGA contains 67K slices), a fine-grained parallelization with even tens of threads is potentially viable in our DL algorithm. Our experiments in Section 2.2.3.2 demonstrates the near-linear runtime scalability of our DL algorithm with respect to the number of threads. This extreme parallelizability can further facilitate efficient implementations of our DL algorithm on hardware accelerators, like FPGAs and GPUs.

Another strong property of our DL algorithm is serial equivalency. Serial equivalency is a property to guarantee that a parallel algorithm always produces exactly the same solution as its serial version does. That is, our DL algorithm guarantees to produce the same solution, regardless of the number of threads used. Therefore, we can enable as much as available parallel computational resources without sacrificing the quality of results.

**Post-DL Exception Handling** Although the convergence of our algorithm can be guaranteed by Lemma 1, after the regular DL process described in Algorithm 5, a small portion (typically $< 1\%$) of cells may still not be able to find legal positions within a given maximum displacement constraint $D$. To legalize these remaining cells, one can, of course, keep relaxing $D$ until a legal solution is reached, like a *Tetris*-based legalizer. However, this approach can result in a huge displacement. Instead of *Tetris*-based approaches, we choose to rip up some already determined clusters and reallocate these ripped cells together with those originally illegal ones to produce a legal solution. Since our FIP is nearly legal, this method is likely able to find a legal solution with very little placement perturbation.

One of the key questions here is which cluster to break. For an illegal cell $v$ and a slice $s$ with its determined cluster $c$, we first define the score of breaking $c$ in $s$ for $v$ as

$$\text{SCORE}_{\text{ripup}}(v, s, c) = -\lambda_1 \cdot \Delta\text{HPWL}(v, s) - \lambda_2 \cdot \text{SCORE}(c, s) - \lambda_3 \cdot \text{Area}(c), \tag{13}$$

where $\Delta$HPWL($v$, $s$) denotes the HPWL increase of moving $v$ to $s$, SCORE($c$, $s$) denotes the score of $c$ in $s$ as defined in Eq. (12), and Area($c$) represents the total cell area (from FIP) in $c$. $\lambda_1$, $\lambda_2$, and $\lambda_3$ are three positive weighting parameters. In our experiments, we empirically set them to 0.02, 1.0, and 4.0, respectively. Intuitively, we prefer to move $v$ to a low-score slice with little wirelength increase. The Area($c$)

term is introduced to evaluate if the ripped cells are easy to legalize. If $c$ has a large area, it either contains many cells or the cells it contains are hard to pack (recall Eq. (9) and Eq. (10)). For both cases, we tend to not break it.

### 2.2.3 Experimental Results

We implemented the proposed framework in C++ based on UTPlaceF [39] and performed the experiments on a Linux machine running with Intel Core i9-7900X CPUs (3.30 GHz, 10 cores, and 13.75 MB L3 cache) and 128 GB RAM. OpenMP 4.0 [45] is used to support multi-threading. The benchmark suite released by Xilinx for ISPD 2016 FPGA placement contest [31] is used to validate the effectiveness of the proposed approaches. All the routings are conducted by Xilinx Vivado v2015.4 [46]. The characteristics of ISPD 2016 benchmark suite are listed in Table 1.

#### 2.2.3.1 Effectiveness Validation of Proposed Techniques

Table 6 demonstrates the effectiveness of the proposed dynamic area adjustment (DAA) and direct legalization (DL) techniques. Here we compare four different placement methodologies, as listed in the four columns. Column "UTPlaceF" represents the original UTPlaceF [39] flow. Column "UTPlaceF + DAA" applies DAA on top of the original UTPlaceF. Column "Proposed" employs both DAA and the proposed DL by replacing the packing, CLB-level placement, and legalization subroutines in "UTPlaceF + DAA" flow with the proposed DL. To further demonstrate the effectiveness of the proposed DL, we also implemented a greedy Tetris-based direct legalization (greedy DL) for comparison. This greedy DL adopts the same score function Eq. (12) used by the proposed DL, and it legalizes one cell at a time to maximize the score improvement defined in Eq. (11). The results of substituting the proposed DL in "Proposed" with this greedy DL are shown in column "DAA + Greedy DL". Metrics "WL" and "RT" represent the routed wirelength and runtime, while "WLR" and "RTR" represent the wirelength and runtime ratios normalized to the "Proposed" column. Note that these four flows share the same underlying global and detailed placement engines, so that noises from parts that are irrelevant to this work can be completely decoupled in this comparison.

It is worthwhile to mention that, the proposed DL should not be applied without DAA, since this can result in very suboptimal or even illegal solutions. In the proposed DL, all cells simultaneously seek to legalize themselves. Without DAA, however, the FIP solution can be considerably far away from a truly legal placement, and in this case, a significant portion of cells can fail to find nearby legal positions. Therefore, we always employ the proposed DL together with the DAA technique in our experiments.

In this experiment, we enable 16 threads for our DL algorithm due to its parallel nature, and all other parts (global/detailed placement) are single-threaded. While the UTPlaceF is universally executed with a

Table 6: Routed Wirelength ($\times 10^3$) and Runtime (Seconds) Comparison with UTPlaceF$^\dagger$ [39]

| Designs | UTPlaceF$^\dagger$ [39] | | | | UTPlaceF$^\dagger$ + DAA | | | | DAA + Greedy DL | | | | Proposed (DAA + DL) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR |
| FPGA-01 | 346 | 70 | 1.016 | 1.15 | 342 | 70 | 1.005 | 1.15 | 371 | 55 | 1.092 | 0.90 | 340 | 61 | 1.000 | 1.00 |
| FPGA-02 | 652 | 111 | 0.999 | 0.97 | 643 | 109 | 0.985 | 0.96 | 691 | 102 | 1.059 | 0.89 | 653 | 114 | 1.000 | 1.00 |
| FPGA-03 | 3173 | 318 | 1.011 | 0.95 | 3107 | 331 | 0.990 | 0.99 | 3283 | 309 | 1.046 | 0.93 | 3139 | 333 | 1.000 | 1.00 |
| FPGA-04 | 5440 | 399 | 1.020 | 1.01 | 5250 | 411 | 0.985 | 1.04 | 5479 | 362 | 1.028 | 0.92 | 5331 | 394 | 1.000 | 1.00 |
| FPGA-05 | 9775 | 432 | 0.973 | 1.00 | 9687 | 449 | 0.964 | 1.03 | 10162 | 396 | 1.012 | 0.91 | 10045 | 434 | 1.000 | 1.00 |
| FPGA-06 | 6505 | 718 | 1.121 | 1.24 | 6173 | 803 | 1.064 | 1.39 | 6185 | 541 | 1.066 | 0.94 | 5801 | 578 | 1.000 | 1.00 |
| FPGA-07 | 9876 | 792 | 1.056 | 1.20 | 9718 | 849 | 1.039 | 1.28 | 9754 | 626 | 1.043 | 0.95 | 9356 | 662 | 1.000 | 1.00 |
| FPGA-08 | 7735 | 641 | 0.932 | 0.92 | 7942 | 690 | 0.957 | 0.99 | 8451 | 650 | 1.018 | 0.94 | 8298 | 695 | 1.000 | 1.00 |
| FPGA-09 | 12062 | 1409 | 1.037 | 1.59 | 11904 | 1385 | 1.023 | 1.56 | 12229 | 830 | 1.051 | 0.94 | 11633 | 887 | 1.000 | 1.00 |
| FPGA-10 | 8178 | 1783 | 1.295 | 2.33 | 7935 | 1782 | 1.256 | 2.33 | 7399 | 730 | 1.171 | 0.95 | 6317 | 766 | 1.000 | 1.00 |
| FPGA-11 | 9966 | 1001 | 0.951 | 1.12 | 10386 | 1023 | 0.991 | 1.15 | 10833 | 848 | 1.034 | 0.95 | 10476 | 890 | 1.000 | 1.00 |
| FPGA-12 | 7635 | 1343 | 1.117 | 1.36 | 7557 | 1551 | 1.106 | 1.57 | 7534 | 914 | 1.102 | 0.93 | 6835 | 988 | 1.000 | 1.00 |
| Norm. | - | - | 1.044 | 1.24 | - | - | 1.030 | 1.29 | - | - | 1.060 | 0.93 | - | - | 1.000 | 1.00 |

$^\dagger$ : This UTPlaceF version is fine tuned for even better routed wirelength and runtime compared with the original publication [39].

single thread for the following two reasons: (1) the packing and legalization algorithms in UTPlaceF are inherently sequential and hard to be parallelized without quality degradation or extra threading communication overhead; (2) the packing and legalization in UTPlaceF only take about 15% of the total runtime on average, therefore, the overall performance gain would be still limited even if they have been carefully parallelized.

We first compare columns "UTPlaceF" and "UTPlaceF + DAA", where the only difference is whether or not DAA is applied. On average, "UTPlaceF + DAA" achieves 1.4% better routed wirelength with only 5% runtime overhead compared with "UTPlaceF". The reason is that, with DAA applied, the FIP solution can be considerably closer to a truly legal solution due to its packing effect consideration. This can be further proved by the experimental results shown in Table 7, which we will discuss in details later in this section.

We then compare columns "UTPlaceF + DAA" and "DAA + Greedy DL" with "Proposed" to demonstrate the effectiveness of the proposed DL. These three methodologies share the same FIP solutions and only differ by their packing and legalization steps. As can be seen, on average, "Proposed" outperforms "UTPlaceF + DAA" and "DAA + Greedy" by 3.0% and 6.0%, respectively, in routed wirelength. It should be noted that "Proposed" outperforms "DAA + Greedy DL" on all twelve benchmarks in routed wirelength with only 7% longer runtime. Therefore, compared with the *Pack-Place-Legalize* methodology in "UTPlaceF + DAA" and the greedy DL, the proposed DL algorithm can effectively explore a larger solution space and achieves better solution quality.

By comparing columns "Proposed" and "UTPlaceF", we can see that, with both DAA and DL employed, the proposed flow outperforms the original UTPlaceF by 4.4% in routed wirelength, while runs 1.24× faster. It is worthwhile to mention that, among all the designs, *FPGA-10* contains the largest number of control sets and flip-flops, which make it severely difficult to pack and legalize. On this particular design, our

approach outperforms UTPlaceF by 29.5% in routed wirelength. Besides, our approach also consistently excel on other control set intensive designs, like *FPGA-06*, *FPGA-07*, and *FPGA-09*. Thus, our approach is especially effective for hard-to-pack designs.

Another notable merit of our approach is that the correlation between FIPs and post-legalization solutions can be significantly improved. This property is appealing in a sense that metrics, like wirelength, timing, and routability, optimized in FIPs can be greatly preserved in legal solutions. Table 7 shows the average and maximum cell displacements between the FIPs and the post-legalization/DL solutions by using the four different methodologies listed in Table 6. As can be seen, the original UTPlaceF introduces huge cell displacements with average and maximum values of 21.4 and 162.5, respectively. DAA technique alone can effectively reduce them down to 11.7 and 135.0 in "UTPlaceF + DAA". In "DAA + Greedy DL", by applying the greedy DL instead of the packing-based methodology, the average and maximum displacements can be further reduced to 1.5 and 57.4, respectively. For all twelve benchmarks, the proposed methodology with DAA and the proposed DL together can achieve maximum displacements that are less than 12.0, which is the predefined constraint in our DL algorithm. Meanwhile, the average displacements are all in the range from 1.0 to 1.5. As expected, our DAA and DL techniques can greatly help to preserve the FIP solution even after a legal solution is obtained.

Table 7: Displacement Comparison with UTPlaceF

| Designs | UTPlaceF [39] | | UTPlaceF + DAA | | DAA + Greedy DL | | Proposed | |
|---|---|---|---|---|---|---|---|---|
| | Avg. | Max. | Avg. | Max. | Avg. | Max. | Avg. | Max. |
| FPGA-01 | 5.7 | 42.6 | 5.6 | 26.1 | 1.2 | 9.2 | 1.0 | 11.4 |
| FPGA-02 | 5.2 | 305.0 | 5.9 | 191.1 | 1.1 | 11.3 | 1.0 | 11.8 |
| FPGA-03 | 12.2 | 146.9 | 7.7 | 90.9 | 1.2 | 23.8 | 1.0 | 11.5 |
| FPGA-04 | 20.3 | 124.3 | 7.8 | 128.0 | 1.3 | 64.3 | 1.1 | 11.5 |
| FPGA-05 | 12.1 | 185.1 | 10.5 | 126.7 | 1.2 | 47.8 | 1.1 | 11.8 |
| FPGA-06 | 60.4 | 187.8 | 17.0 | 209.0 | 1.7 | 52.7 | 1.3 | 11.8 |
| FPGA-07 | 21.1 | 232.6 | 13.9 | 166.8 | 1.7 | 92.5 | 1.3 | 11.5 |
| FPGA-08 | 11.8 | 95.4 | 6.3 | 107.9 | 1.1 | 13.3 | 1.0 | 10.9 |
| FPGA-09 | 13.0 | 150.6 | 10.3 | 144.7 | 1.6 | 86.8 | 1.2 | 11.7 |
| FPGA-10 | 25.5 | 164.7 | 26.2 | 127.4 | 2.6 | 166.7 | 1.4 | 11.7 |
| FPGA-11 | 40.8 | 138.1 | 15.6 | 124.8 | 1.4 | 46.4 | 1.0 | 11.5 |
| FPGA-12 | 28.7 | 177.6 | 13.2 | 176.6 | 1.5 | 74.1 | 1.1 | 11.5 |
| Norm. | 21.4 | 162.5 | 11.7 | 135.0 | 1.5 | 57.4 | 1.2 | 11.6 |

Figure 15 visualizes the distributions of cell displacement between the FIPs and the post-legalization/DL solutions based on design *FPGA-03*. Compared with UTPlaceF, most cells in the proposed methodology are much closer to their original locations in the FIP. However, tremendously large displacements can be observed in UTPlaceF.

#### 2.2.3.2 Runtime Scaling of the Direct Legalization

Figure 16 shows the runtime scaling of our DL algorithm for different design sizes under 1, 2, 4, 8, and 16 threads. It can be seen that, with a fixed number of available threads, the algorithm scales linearly with

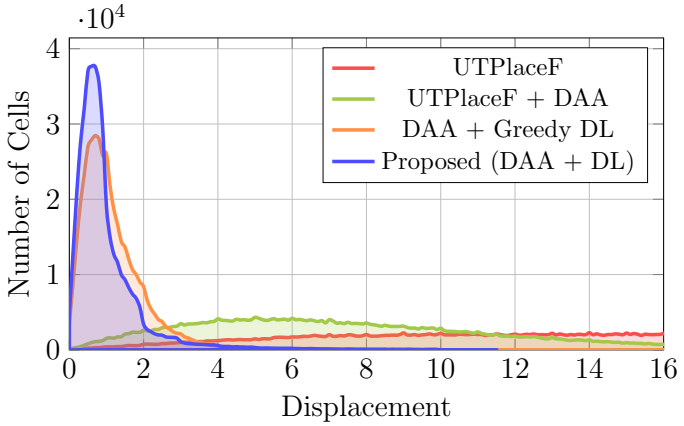



Figure 15: The cell displacement distributions of UTPlaceF, UTPlaceF + DAA, DAA + Greedy DL, and the proposed flow (DAA + DL) based on design *FPGA-03*.

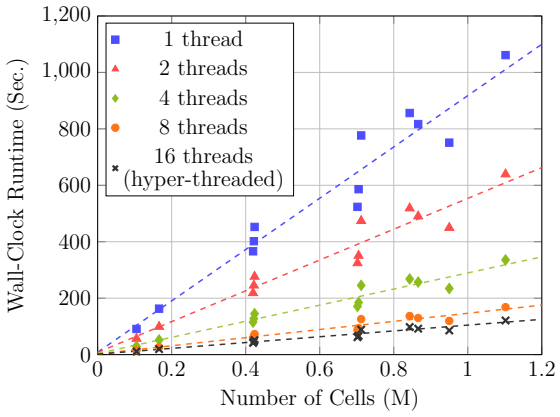



Figure 16: Runtime scaling of the direct legalization.

Table 8: Routed Wirelength ($\times 10^3$) and Runtime (Seconds) Comparison with Other State-of-the-Art Academic Placers on ISPD 2016 Benchmark Suite

| Designs | 1st Place | | | | 2nd Place | | | | 3rd Place | | | | GPlace [30] | | | | RippleFPGA [41] | | | | Proposed | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR |
| FPGA-01 | * | * | - | - | 380 | 118 | 1.117 | 1.93 | 582 | 97 | 1.711 | 1.59 | 494 | 30 | 1.451 | 0.49 | 353 | 31 | 1.037 | 0.51 | 340 | 61 | 1.000 | 1.00 |
| FPGA-02 | 678 | 435 | 1.038 | 3.82 | 680 | 208 | 1.041 | 1.82 | 1047 | 191 | 1.603 | 1.68 | 903 | 61 | 1.383 | 0.54 | 645 | 57 | 0.989 | 0.50 | 653 | 114 | 1.000 | 1.00 |
| FPGA-03 | 3223 | 1527 | 1.027 | 4.59 | 3661 | 1159 | 1.166 | 3.48 | 5029 | 862 | 1.602 | 2.59 | 3908 | 289 | 1.245 | 0.87 | 3262 | 201 | 1.039 | 0.60 | 3139 | 333 | 1.000 | 1.00 |
| FPGA-04 | 5629 | 1257 | 1.056 | 3.19 | 6497 | 1449 | 1.219 | 3.68 | 7247 | 889 | 1.359 | 2.26 | 6278 | 280 | 1.178 | 0.71 | 5510 | 224 | 1.033 | 0.57 | 5331 | 394 | 1.000 | 1.00 |
| FPGA-05 | 10265 | 1266 | 1.022 | 2.92 | † | - | - | - | † | - | - | - | † | - | - | - | 9969 | 270 | 0.992 | 0.62 | 10045 | 434 | 1.000 | 1.00 |
| FPGA-06 | 6330 | 2920 | 1.091 | 5.05 | 7009 | 4166 | 1.208 | 7.21 | 6823 | 8613 | 1.176 | 14.90 | 7643 | 600 | 1.318 | 1.04 | 6180 | 424 | 1.065 | 0.73 | 5801 | 578 | 1.000 | 1.00 |
| FPGA-07 | 10237 | 2703 | 1.094 | 4.08 | 10416 | 4572 | 1.113 | 6.91 | 10973 | 9196 | 1.173 | 13.89 | 11255 | 691 | 1.203 | 1.04 | 9640 | 493 | 1.030 | 0.74 | 9356 | 662 | 1.000 | 1.00 |
| FPGA-08 | 8384 | 2645 | 1.010 | 3.81 | 8986 | 2942 | 1.083 | 4.23 | 12300 | 2741 | 1.482 | 3.94 | 9323 | 734 | 1.124 | 1.06 | 8157 | 425 | 0.983 | 0.61 | 8298 | 695 | 1.000 | 1.00 |
| FPGA-09 | † | - | - | - | 13909 | 5833 | 1.196 | 6.58 | † | - | - | - | 14003 | 974 | 1.204 | 1.10 | 12305 | 589 | 1.058 | 0.66 | 11633 | 887 | 1.000 | 1.00 |
| FPGA-10 | * | * | - | - | * | * | - | - | † | - | - | - | † | - | - | - | 7140 | 649 | 1.130 | 0.85 | 6317 | 766 | 1.000 | 1.00 |
| FPGA-11 | 11091 | 3227 | 1.059 | 3.63 | 11713 | 7331 | 1.118 | 8.24 | † | - | - | - | 12368 | 923 | 1.181 | 1.04 | 11023 | 542 | 1.052 | 0.61 | 10476 | 890 | 1.000 | 1.00 |
| FPGA-12 | 9022 | 4539 | 1.320 | 4.59 | * | * | - | - | † | - | - | - | † | - | - | - | 7363 | 650 | 1.077 | 0.66 | 6835 | 988 | 1.000 | 1.00 |
| Norm. | - | - | 1.080 | 3.96 | - | - | 1.140 | 4.90 | - | - | 1.444 | 5.84 | - | - | 1.254 | 0.88 | - | - | 1.041 | 0.64 | - | - | 1.000 | 1.00 |

*: Placement error. †: Unroutable placement.

respect to the design size. On the other hand, given a design, its runtime also decreases nearly linearly as the number of threads increases (except the 16-thread case with hyper-threading). On average, 1.65×, 3.15×, 6.19×, and 8.68× speedups can be achieved with 2, 4, 8, and 16 threads, respectively, compared with the single-thread execution. Note that the sacling starts to saturate from 8 threads to 16 threads. This is because a CPU core can lanuch 2 hyper-threads that share the same execution resources and cannot be truly parallelized. Considering there are only 10 cores in our machine, at least 6 threads will not be running at their maximum speed in the case of 16 threads.

#### 2.2.3.3 Comparison with Other State-of-the-Art Placers

To further demonstrate the effectiveness of our approach, we also compare our result with other state-of-the-art academic placers, including RippleFPGA [41], GPlace [30] as well as the top-3 winners of ISPD 2016 contest, on ISPD 2016 benchmark suite. The comparisons of routed wirelength and runtime are presented

in Table 8.

As the experiment setup in Section 2.2.3.1, we enable 16 threads only for our DL algorithm and keep all other parts single-threaded. All other placers are executed using a single thread. Although other placers can also gain some performance by parallelizing their packing and legalization algorithms, the improvement might be limited. This is because most of their packing and legalization algorithms are not the runtime bottleneck, just like in UTPlaceF. For example, RippleFPGA only takes about 5% of the total runtime on packing and legalization [41].

Despite the different global/detailed placement engines and the execution machines, our approach still shows the best overall routed wirelength. On average, our approach outperforms the three contest winners, GPlace, and RippleFPGA by 8.0%, 14.0%, 44.4%, 25.4%, and 4.1%, respectively. Again, our approach especially excels in control set intensive designs, like *FPGA-06*, *FPGA-07*, *FPGA-09*, and *FPGA-10*. which further evidences the effectiveness of our approach on hard-to-pack designs. As for the runtime, our approach is 3.96×, 4.90×, and 5.84× faster than the three contest winners. While comparing with GPlace and RippleFPGA, our approach runs 1.14× and 1.56× slower.

#### 2.2.3.4 Runtime Breakdown

The runtime breakdown of our approach based on all twelve designs in ISPD 2016 benchmark suite is shown in Figure 17. Again, we enable 16 threads only for DL and keep all other parts single-threaded. On average, 64.5% and 18.5% of the total runtime are taken by the quadratic placement (quadratic programming and rough legalization) and the detailed placement, respectively. While the proposed dynamic area adjustment, direct legalization, and post-DL exception handling techniques consume 2.2%, 12.5%, and 0.7% of the total runtime.

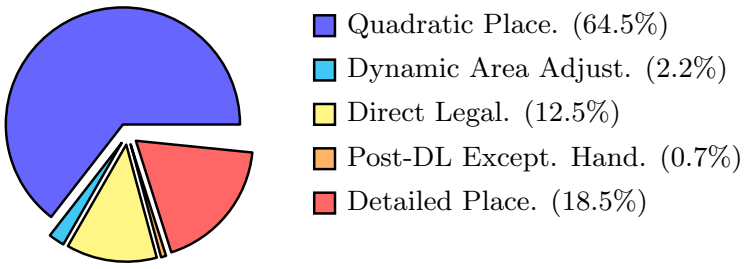



Figure 17: The runtime breakdown of our approach based on designs in ISPD 2016 benchmark suite.

# 3 UTPlaceF 2.0: A Clock-Aware FPGA Placement Engine

Clock rules, among various FPGA architecture constraints, are of great importance, not only because of their significant impact on timing closure and power dissipation but their imposed layout constraints that affect how efficiently other logics can be mapped. With the consideration of clock rules, placement turns out to be much more difficult for FPGAs even to find a feasible solution. For modern FPGAs, clock network planning must be involved during or even before placement stage due to their pre-manufactured clock networks, which cannot be adjusted to different designs, as well as their limited clock routing resources. The interdependency between clock sink placement and clock network planning makes the clock-aware placement for FPGAs a challenging chicken and egg problem.

While many existing works have proposed fairly mature placement techniques for FPGAs to optimize conventional design metrics such as wirelength, routability, power, and timing [8, 10, 12, 17, 24–27, 29, 30, 38, 39, 47], there has been limited research effort on placement with the awareness of clock rules. The closest related previous work is [48], which proposes a cost function for cell swapping that penalizes high-clock-usage placement and integrates it into a conventional simulated-annealing-based placement engine to produce clock-legal solutions. However, their approach suffers from slow annealing process and its quality heavily depends on the cost function tuning. Moreover, their approach can easily get stuck in some local swappings and hence unable to resolve global clock congestions.

To address the clock legalization challenges in FPGA placement for large-scale state-of-the-art commercial FPGAs, we proposed a high-performance clock-aware placement engine, UTPlaceF 2.0, which simultaneously optimizes the conventional wirelength objective and honors complicated global and detailed clock constraints.

## 3.1 Preliminaries and Overall Flow

In this section, we will briefly introduce the targeted FPGA clocking architecture and give the constraints and problem definition for clock-aware placement. At the end of this section, an overview of the proposed UTPlaceF 2.0 framework will be exposed.

## 3.2 Clocking Architecture

UTPlaceF 2.0 is targeted to Xilinx UltraScale VU095 [1], which was adopted in both ISPD'16 and ISPD'17 FPGA placement contests [31, 49]. In this particular architecture, each FPGA device is divided into 5 by 8 of 40 clock regions and each clock region contains synchronous elements such as configurable logic blocks

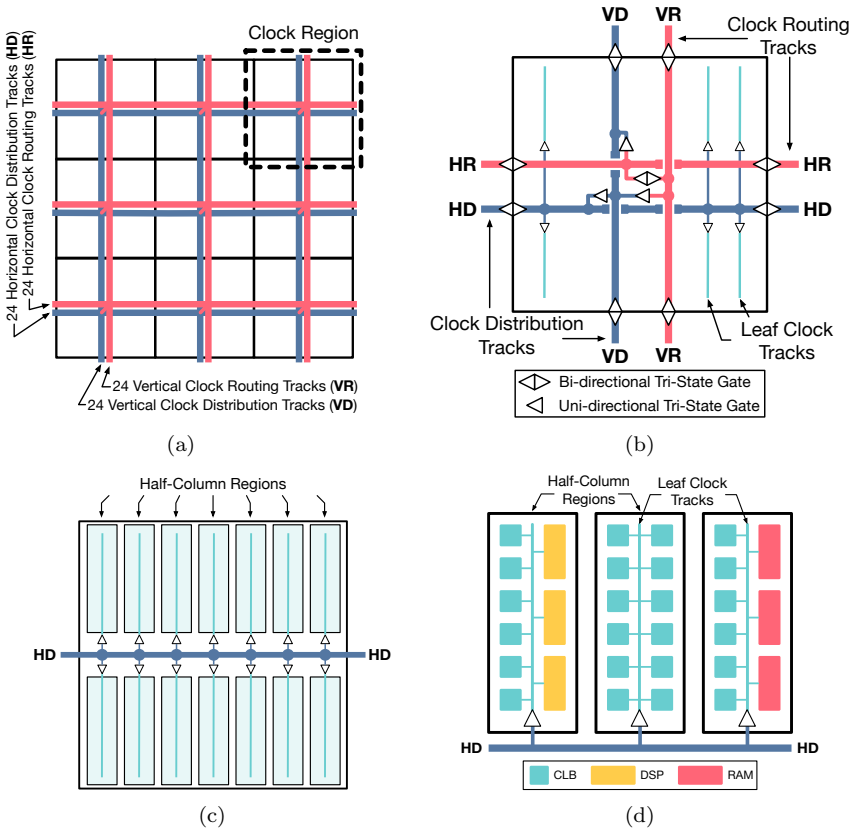



Figure 18: Clocking architecture of Xilinx Ultrascale. (a) A global view of 3 by 3 of 9 clock regions, the clock routing network (red), and the clock distribution network (blue). (b) A detailed view of clock routing tracks, clock distribution tracks, and leaf clock tracks within a clock region. (c) A global view of half-column regions within a clock region. (d) A detailed view of three adjacent half-column regions with different logic resources.

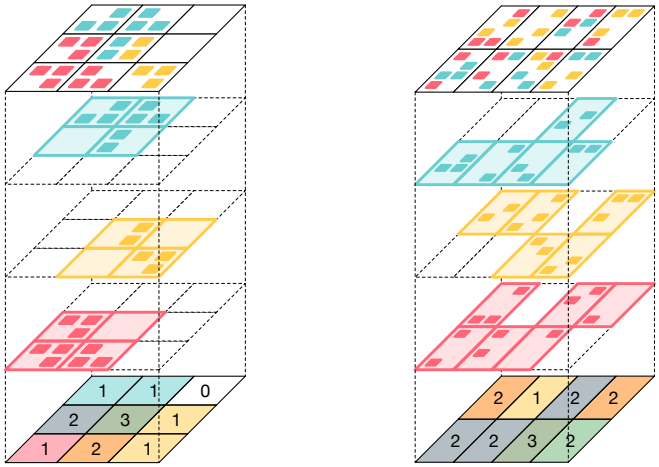
(CLBs), digital signal processors (DSPs), random-access memories (RAMs), and so on. Each CLB further consists of lookup tables (LUTs) and flip-flops (FFs).

The global clocking architecture is a two-level structure containing a clock routing network on top of a clock distribution network. As shown in Figure 18(a), both routing and distribution networks have 24 horizontal and 24 vertical tracks running through each clock region. A detailed view of clocking architecture within a clock region is exposed in Figure 18(b). The connection between any horizontal and vertical clock routing tracks (HR and VR) is bidirectional but the same connection (HD and VD) is unidirectional (from VD to HD) in clock distribution network. Besides, a clock can hop onto VD through their corresponding HR and VR, but there is no path back vice versa. Therefore, once a clock is hooked on the clock distribution network (HD and VD), it can only stay on it or go to the leaf-level clock tracks.

For the leaf clock networks, a clock region is further divided into multiple half-column regions of two-site width and half-clock-region height as shown in Figure 18(c). Figure 18(d) illustrates a detailed view of three

adjacent half-column regions. Different half-column regions might contain distinct logic resources, such as CLBs, DSPs, and RAMs, but they have similar leaf clock networks to directly drive clock sinks within them. More detailed clocking architecture can be found in [49].

### 3.2.1 Clock Constraints for Placement




(a) (b)

Figure 19: Illustration of clock demand calculation for clock regions and half-column regions. Different colors represent different clocks. (a) Global clock demand calculation for clock regions. (b) Clock demand calculation for half-column regions within a clock region.

Restricted by the clocking architecture, two major constraints, namely clock region constraint and half-column region constraint, are imposed in the placement stage.

Clock region constraint is introduced by the limited clock routing/distribution tracks and it restricts the global clock demand in each clock region must be equal to or less than 24 in our targeted FPGA architecture. For a clock region, its global clock demand is defined as the total number of clock nets that have their bounding boxes intersected with it. Figure 19(a) illustrates how global clock demands are calculated for a simple placement with three clock nets. The top layer shows the sink distribution for each clock, the three middle layers represent the spanning clock regions of each clock and the layer in the bottom gives the final global clock demand in each clock region.

Similarly, imposed by the limited leaf clock tracks, half-column region constraint requires that each half-column region can only contain at most 12 clocks. Figure 19(b) illustrates the clock demand calculation for half column regions within a clock region. Different from clock regions, the clock demand in a half-column region is only the number of clocks inside it, regardless of clock net bounding boxes.

### 3.2.2 Problem Definition

In modern FPGA placement, the optimization usually includes multiple objectives, such as wirelength, which is measured by Half-Perimeter Wirelength (HPWL), routability, timing, and power. Wirelength is still regarded as the major objective, since it is a good first-order approximation of many other metrics, e.g., power and timing. Therefore, in UTPlaceF 2.0, our objective is to minimize wirelength.

To produce legal placement solutions, apart from clock constraints, packing rules also need to be satisfied when multiple LUTs and FFs are placed into the same CLB site. The detailed packing rules for our targeted FPGA are elaborated in [31].

With all constraints and objectives defined, we now define our clock-aware placement problem as follows.

**Problem 1 (Clock-Aware Placement)** *Given a netlist of LUTs, FFs, DSPs, RAMs, and I/Os, produce a legal placement solution with minimized routed wirelength, meanwhile packing rules, clock region constraint, and half-column region constraint are satisfied.*

### 3.2.3 UTPlaceF 2.0 Overview

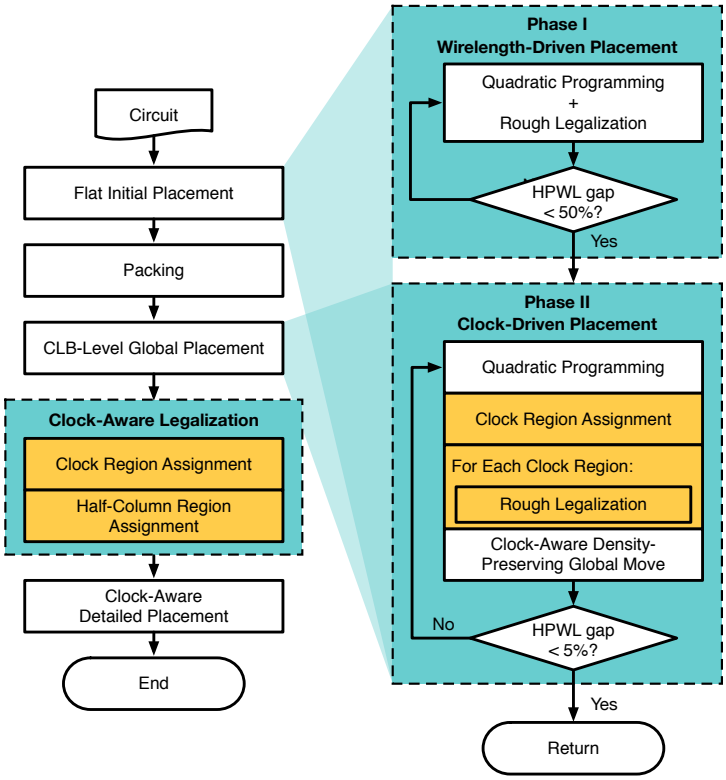



Figure 20: The overall flow of UTPlaceF 2.0. Yellow-shaded blocks indicate major new/modified steps that differ from UTPlaceF.

The overall flow of UTPlaceF 2.0 is shown in Figure 20. UTPlaceF 2.0 is a natural extension of UTPlaceF [39] and consists of five major steps: 1) flat initial placement (FIP), 2) packing, 3) CLB-level global placement, 4) legalization, and 5) detailed placement. On top of conventional wirelength optimizations performed in UTPlaceF, clock constraints are explicitly considered throughout the UTPlaceF 2.0 framework.

FIP is responsible for producing physical location of each cell to better guide decision making in the packing stage. It consists of two phases: 1) wirelength-driven phase and 2) clock-driven phase. In each iteration of the first phase, a quadratic analytical placement is solved to minimize wirelength, followed by a rough legalization [34] for cell overlapping removal. In the second phase, besides the conventional wirelength optimization, an additional clock region assignment (see Section 3.3.1) step is called after the quadratic placement. It produces a cell-to-clock-region assignment solution that satisfies the clock region constraint. Then, the rough legalization is only performed within each clock region to honor the assignment result. UTPlaceF 2.0 stops FIP once the wirelength converges.

CLB-level global placement is performed immediately after packing to further optimize the placement for the post-packing netlist. It shares the same framework with the second phase of FIP, and use the final solution of FIP as the starting point to speed-up placement convergence.

In legalization stage, while minimizing the conventional objective of pin movement, clock region constraint and half-column region constraint are rigorously respected by clock region assignment technique and half-column region assignment technique (see Section 3.3.6), respectively.

Finally, detailed placement techniques in UTPlaceF is extended to be clock-aware and maintain the clock legality throughout the wirelength optimization process before the final solution is produced.

## 3.3 UTPlaceF 2.0 Algorithms

In this section, we will explain UTPlaceF 2.0 algorithms, including clock region assignment and half-column region assignment, in details.

### 3.3.1 Clock Region Assignment

Clock region assignment is a key step in UTPlaceF 2.0 to ensure the satisfaction of clock region constraint. It is intensively called throughout major steps, such as FIP, CLB-level global placement, and legalization, in UTPlaceF 2.0. Therefore, it has a significant impact on both solution quality and runtime. Here we formally define the clock region assignment problem as follows.

**Problem 2 (Clock Region Assignment)** *Given a rough-legalized placement and logic resource capacity of each clock region, assign cells to clock regions to minimize total pin movement without logic resource and*

*global clock overflow.*

Given the notations defined in Table 9, Problem 2 can be written as a binary minimization problem with linear and boolean logical constraints as shown in Formulation (14).

Table 9: Notations Used in Clock Region Assignment

| | |
|---|---|
| $\mathcal{V}$ | The set of cells. |
| $\mathcal{V}^{(s)}$ | The set of cells of resource type $s \in \{\text{CLB, DSP, RAM}\}$. |
| $P_i$ | The number of pins in cell $i$. |
| $A_i^{(s)}$ | The cell $i$'s demand for resource type $s \in \{\text{CLB, DSP, RAM}\}$. |
| $\mathcal{R}$ | The set of clock regions. |
| $C_j^{(s)}$ | The clock region $j$'s capacity for resource type $s \in \{\text{CLB, DSP, RAM}\}$. |
| $D_{i,j}$ | The physical distance between cell $i$ and clock region $j$. |
| $\mathcal{E}$ | The set of clock nets. |
| $Z_{i,e}$ | A binary value represents whether cell $i$ is in clock net $e$. |
| $L_j^+$ | The set of clock regions that are left to or in the same column of clock region $j$. |
| $R_j^+$ | The set of clock regions that are right to or in the same column of clock region $j$. |
| $B_j^+$ | The set of clock regions that are below or in the same row of clock region $j$. |
| $T_j^+$ | The set of clock regions that are above or in the same row of clock region $j$. |

$$\underset{\mathbf{x}}{\text{minimize}} \quad \sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{R}} P_i \cdot D_{i,j} \cdot \mathbf{x}_{i,j}, \tag{14a}$$

$$\text{subject to} \quad \mathbf{x}_{i,j} \in \{0,1\}, \forall i \in \mathcal{V}, \forall j \in \mathcal{R}, \tag{14b}$$

$$\sum_{j\in\mathcal{R}} \mathbf{x}_{i,j} = 1, \forall i \in \mathcal{V}, \tag{14c}$$

$$\sum_{i\in\mathcal{V}} A_i^{(s)} \cdot \mathbf{x}_{i,j} \le C_j^{(s)}, \forall j \in \mathcal{R}, \forall s \in \{\text{CLB, DSP, RAM}\} \tag{14d}$$

$$\sum_{e\in\mathcal{E}} \Big[\Big(\bigvee_{q\in\mathcal{L}_j^+} Z_{i,e}\cdot\mathbf{x}_{i,q}\Big) \wedge \Big(\bigvee_{q\in\mathcal{R}_j^+} Z_{i,e}\cdot\mathbf{x}_{i,q}\Big) \wedge \Big(\bigvee_{q\in\mathcal{B}_j^+} Z_{i,e}\cdot\mathbf{x}_{i,q}\Big) \wedge \Big(\bigvee_{q\in\mathcal{T}_j^+} Z_{i,e}\cdot\mathbf{x}_{i,q}\Big)\Big] \le 24, \forall j \in \mathcal{R}. \tag{14e}$$

Formulation (14) is optimized over binary variables $\mathbf{x}_{i,j}$ to minimize the objective (14a) of total pin movement. If cell $i \in \mathcal{V}$ is assigned to clock region $j \in \mathcal{R}$, then $\mathbf{x}_{i,j} = 1$, otherwise $\mathbf{x}_{i,j} = 0$. The constraint (14c) ensures that each cell is assigned to one and only one clock region. The constraint (14d) guarantees the demands are no greater than the capacities for all logic resource types (e.g., CLB, DSP, and RAM) in each clock region. The boolean logical constraint (14e) ensures the satisfaction of clock region constraint. Intuitively, for a given clock region $j$, if and only if a clock net has cells located in all $L_j^+$, $R_j^+$, $B_j^+$, and $T_j^+$, its bounding box would intersect with $j$ and occupy global clock resource in it. The constraint (14e) sums

up all such clock nets for each clock region and restricts the clock usage no more than the clock capacity, which is 24 in our target FPGA architecture.

### 3.3.2 The Relaxation Algorithm

With proper modeling, the boolean logical constraint (14e) can be transformed to a set of linear constraints and then Formulation (14) can be optimally solved utilizing integer linear programming techniques. However, integer linear programming is computationally expensive and suffers from unaffordable runtime for our application. Therefore, here we relax Formulation (14) to an easier problem that can be efficiently solved, as shown in Formulation (15).

$$\underset{\mathbf{x},\lambda}{\text{minimize}} \quad \sum_{\mathrm{i}\in\mathcal{V}}\sum_{\mathrm{j}\in\mathcal{R}}(\mathrm{P_i}\cdot\mathrm{D_{i,j}}+\lambda_{\mathrm{i,j}})\cdot\mathbf{x}_{\mathrm{i,j}}, \tag{15a}$$

$$\text{subject to} \quad \mathbf{x}_{i,j}\in\{0,1\},\forall i\in\mathcal{V},\forall j\in\mathcal{R}, \tag{15b}$$

$$\sum_{j\in\mathcal{R}}\mathbf{x}_{i,j}=1,\forall i\in\mathcal{V}, \tag{15c}$$

$$\sum_{i\in\mathcal{V}}A_i\cdot\mathbf{x}_{i,j}\leq C_j,\forall j\in\mathcal{R}. \tag{15d}$$

Compared to the original Formulation (14), Formulation (15) has two major differences: 1) instead of considering capacity constraint (14d) simultaneously for all logic resource types, here we only consider the capacity constraint (15d) for one resource type at a time (i.e., we consider three resource types separately); 2) we remove the boolean logical constraint (14e) and add a penalty multiplier $\lambda_{i,j}$ for each potential cell $i\in\mathcal{V}$ to clock region $j\in\mathcal{R}$ assignment $\mathbf{x}_{i,j}$ to the objective (15a).

The main idea of our relaxation can be explained as follows. We first ignore the clock region constraint (14e) and only respect each logic resource constraint separately. Each time after solving Formulation (15), we properly penalize assignments causing clock overflow by updating the corresponding penalty multipliers $\lambda_{i,j}$ in the objective (15a). By repeating the process of solving and updating Formulation (15), the assignment will progressively converge to a clock-feasible solution.

The proposed relaxation-based clock region assignment algorithm is summarized in Algorithm 6. Cells are first divided into three groups based on their resource types, including CLB, DSP, and RAM, in line 1. Then all penalty multipliers $\lambda_{i,j},\forall i\in\mathcal{V},\forall j\in\mathcal{R}$ are initialized to zero in line 2, which degenerates the Formulation (15) into a pure logic-resource-constrained pin-movement minimization problem without clock

**Algorithm 6** Clock Region Assignment with Relaxation

**Input:** A rough-legalized placement.
**Output:** A movement-minimized assignment without logic resource and clock overflow.
1: Divide cells $\mathcal{V}$ into three groups, $\mathcal{V}^{(\text{CLB})}$, $\mathcal{V}^{(\text{DSP})}$, and $\mathcal{V}^{(\text{RAM})}$, based on their logic resource types.
2: $\lambda_{i,j} \leftarrow 0, \forall i \in \mathcal{V}, \forall j \in calR$;
3: **while** clock overflow exists **do**
4: **for** each $\mathcal{U} \in \{\mathcal{V}^{(\text{CLB})}, \mathcal{V}^{(\text{DSP})}, \mathcal{V}^{(\text{RAM})}\}$ **do**
5: Solve Formulation (15) for $\mathcal{U}$; ▷ See Section 3.3.3
6: **end for**
7: Update penalty multiplier $\lambda$; ▷ See Section 3.3.4
8: **end while**

legality consideration. Within the loop from line 3 to line 8, we first solve Formulation (15) for each cell group under current penalty multiplier settings, and then in line 7, penalty multipliers $\lambda_{i,j}$ are updated according to the assignment solutions produced from line 4 to line 6. The penalty multipliers are incrementally updated to penalize assignments causing clock overflow and new assignments produced by the Formulation (15) with these updated penalty multipliers will be progressively closer to clock-legal solutions after each iteration. The process of solving and updating Formulation (15) from line 3 to line 8 is repeated until a clock-overflow-free assignment is reached.

### 3.3.3 Minimum-Cost Flow Transformation

One notable merit of our relaxation in Formulation (15) is that it can be transformed into a minimum-cost-flow problem, which is a fairly mature field with many efficient algorithms [50].

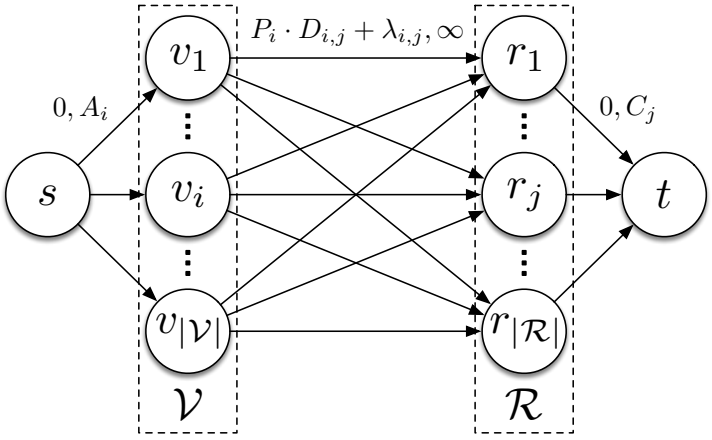



Figure 21: The minimum-cost flow representation of Formulation (15). Pair of numbers (e.g., $P_i \cdot D_{i,j} + \lambda_{i,j}, \infty$) on each edge represents cost and capacity, respectively, and $\infty$ means unlimited capacity.

Figure 21 illustrates the minimum-cost flow representation of Formulation (15). If all cells in $\mathcal{V}$ have unit resource demand ($A_i = 1, \forall i, j \in \mathcal{V}$), optimally solving Formulation (15) is equivalent to computing the

minimum-cost flow of amount $\sum_{i \in \mathcal{V}} A_i$ on the graph. For cases with non-unit resource demands, however, the assignment solutions corresponding to their minimum-cost flow results may contain cells that are split into multiple clock regions, which require extra post-processing steps and slight relaxation on constraint (15d) to guarantee solution feasibility. In UTPlaceF 2.0, we always apply unit cell resource demand ($A_i = 1$) to ensure the solutions produced by our minimum-cost flows transformation are feasible.

### 3.3.4 Penalty Multiplier Updating

Algorithm 6 is a generalized relaxation framework for solving the clock region assignment problem. One key step within this framework is how to update the penalty multiplier $\lambda$ after each assignment iteration. Various updating strategies can converge to different feasible solutions. In this section, we will only discuss one specific updating method, which is applied in UTPlaceF 2.0, and its effectiveness is demonstrated by our experiments.

The guiding idea of our $\lambda$ updating method is that we hope clock overflow can be mitigated by forbidding some clocks from occupying the overflowed clock regions. We observe that to block a clock net from taking clock resources in a given clock region, all the cells in this clock net must be strictly restricted in the region below, above, left, or right to the clock region, as shown in Figure 22. Otherwise, the clock bounding box must intersect with the clock region. Thus, intuitively, clock congestion can be resolved by iteratively pushing clock nets to these four escaping regions corresponding to overflowed clock regions.

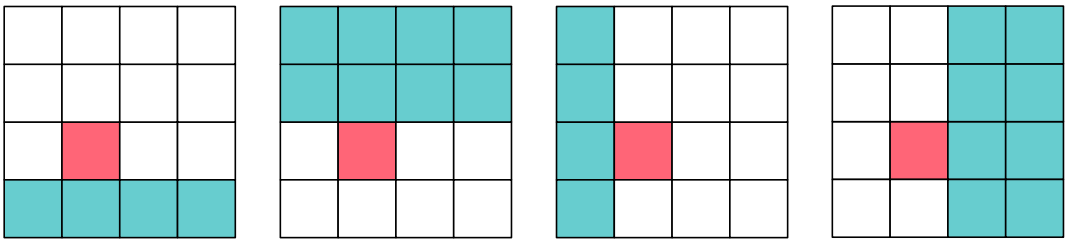



Figure 22: The four escaping regions (green) that are defined as the regions (a) below, (b) above, (c) left, and (d) right to a given clock region (red), respectively. To avoid employing clock resources of the given clock region, all cells of a clock net must be simultaneously placed into one of these four escaping regions.

Inspired by this observation, we propose our penalty multiplier updating method summarized in Algorithm 7. We first choose the most overflowed clock region as our target clock region to resolve in line 1 and locate the four escaping regions illustrated in Figure 22 for the target clock region from line 2 to line 5. Then from line 6 to line 11, for clock nets occupying clock resources in the target clock region, we compute their moving costs to each of these four escaping regions and store the calculation results as candidates for later overflow resolving. Within the loop from line 16 to line 30, these candidates are accessed in ascending order

**Algorithm 7** Penalty Multiplier Updating For Clock Region Constraint

**Input:** Penalty multiplier $\lambda$. The maximum clock overflow descent step $I_{max}$ for each clock region in each iteration.

**Output:** An updated penalty multiplier $\lambda$ that will result in less clock overflow.

1: Find the clock region $r$ with the largest clock overflow $O_{max}$;
2: $B_r \leftarrow$ the region below to $r$;
3: $T_r \leftarrow$ the region above to $r$;
4: $L_r \leftarrow$ the region left to $r$;
5: $R_r \leftarrow$ the region right to $r$;
6: $l \leftarrow \varnothing$;
7: **for** each clock $e \in \mathcal{E}$ occupying clock resource of $r$ **do**
8: **for** each region $b \in \{B_r, T_r, L_r, R_r\}$ **do**
9: $l \leftarrow l \cup \{\texttt{Cost}(e, b)\}$; ▷ See Equation (16)
10: **end for**
11: **end for**
12: Sort candidates in $l$ by ascending order of their cost;
13: $I \leftarrow \texttt{min}(O_{max}, I_{max})$;
14: $i \leftarrow 0$;
15: $chosen[e] \leftarrow false, \forall e \in \mathcal{E}$;
16: **for** each $\texttt{Cost}(e, b) \in$ sorted $l$ **do**
17: **if** $chosen[e]$ **or** $\neg\texttt{IsFeasible}(e, b, \lambda)$ **then**
18: Continue;
19: **end if**
20: **for** each cell $i$ in clock $e$ **do**
21: **for** each clock region $j$ that is not in region $b$ **do**
22: $\lambda_{i,j} \leftarrow \infty$;
23: **end for**
24: **end for**
25: $chosen[e] \leftarrow true$;
26: $i \leftarrow i + 1$;
27: **if** $i \geq I$ **then**
28: Return;
29: **end if**
30: **end for**

of their costs. For each candidate, in line 17, we first ensure that no other candidate associated with the same clock net has been chosen and then call function `IsFeasible` to reject candidates leading to infeasible solutions. The feasibility checking and function `IsFeasible` will be further discussed later in this section. If a candidate is feasible, we block all assignments between cells in the candidate clock net and clock regions outside the candidate escaping region by setting the corresponding penalty multipliers to infinity from line 20 to line 24. This operation prevents the target clock region from being further employed by the candidate clock in the later assignment iterations. The loop from line 16 to line 30 is repeated until all overflow in the target clock is fully resolved or the number of resolved overflow reaches the limit $I_{max}$, which is 2 in UTPlaceF 2.0 by default. Intuitively, $I_{max}$ is the maximum descent step for clock overflow resolving. Under smaller $I_{max}$, clock congestion can be resolved more smoothly and evenly among different clock regions with the cost of more assignment iterations.

Now we explain the function $\texttt{Cost}(e, b)$ that computes the cost of pushing clock $e$ to escaping region $b$. The objective of our assignment is to minimize total pin movement. In addition, each cell movement may lead to logic resource overflow in the target escaping region. Thus the cost consists of two components: pin movement cost and logic resource cost, shown as follows,

$$Cost(e, b) = \sum_{i \in \mathcal{V}(e)} (P_i \cdot d_{i,b} + \alpha \cdot A_i^{(\text{CLB})} + \beta \cdot A_i^{(\text{DSP})} + \gamma \cdot A_i^{(\text{RAM})}). \tag{16}$$

where $\mathcal{V}(e)$ denotes the set of cells in clock net $e$, and $d_{i,b}$ denotes the physical distance between cell $i$ and box region $b$. The trade-off weights $\alpha$, $\beta$, and $\gamma$ are set to 5, 50, and 50, respectively, in our experiments.

The only thing left now is the feasibility checking function $\texttt{IsFeasible}(e, b, \lambda)$. For an intermediate solution with the logic resource constraint satisfied, arbitrarily blocking a set of assignments cannot further guarantee the existence of feasible solutions. For example, if clock nets are restricted in small regions without enough logic resources to accommodate all the cells, the corresponding assignment problem will be infeasible. One straightforward method to avoid this issue is to solve the updated minimum-cost flow for the graph in Figure 21 to check if a feasible solution exists before actually applying the new assignment blockages. However, this method is far too cumbersome to be applied in our iterative framework, which motivates us to propose a light-weight yet effective feasibility checking method.

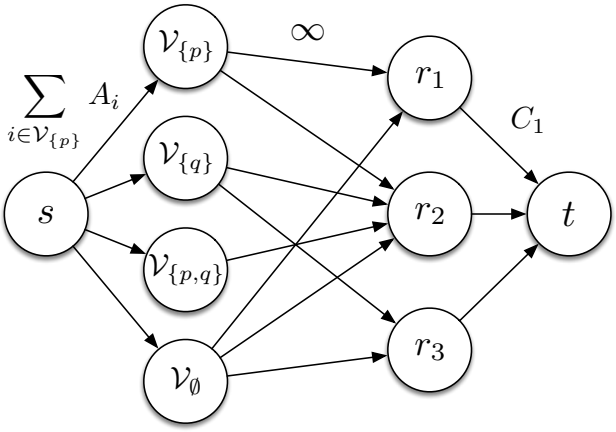



Figure 23: Illustration of our maximum-flow-based feasibility checking. Numbers on edges represent their capacities.

Instead of solving the minimum-cost flow for the graph in Figure 21, we construct a maximum-flow graph, as illustrated in Figure 23, to perform fast feasibility checking. Here we are trying to assign a netlist with two clock nets $p$ and $q$ to three clock regions $r_1$, $r_2$, and $r_3$. We divide all cells into four mutually exclusive groups $\mathcal{V}_{\{p\}}$, $\mathcal{V}_{\{q\}}$, $\mathcal{V}_{\{p,q\}}$, and $\mathcal{V}_{\varnothing}$, based on the clock nets they belong to, where $\mathcal{V}_p$ and $\mathcal{V}_q$ denote the

groups of cells belonging only to $p$ and $q$, respectively, $\mathcal{V}_{\{p,q\}}$ denotes the group of cells belonging to both $p$ and $q$, and $\mathcal{V}_{\varnothing}$ is the set of cells without clock nets. In this graph construction, we only introduce edges for unblocked assignments ($\lambda_{i,j} \neq \infty$), and specific to this example, clock $p$ cannot be assigned to $r_3$ and clock $q$ cannot be assigned to $r_1$. Then, with the proper edge capacity settings as shown in Figure 23, checking the assignment feasibility of Formulation (15) is equivalent to computing the maximum flow of amount $\sum_{i\in\mathcal{V}} A_i$ in this graph. If the resulting maximum flow value is equal to $\sum_{i\in\mathcal{V}} A_i$, a feasible solution with the set of new assignment blockages applied must exist, otherwise, the assignment problem becomes infeasible.

It should be noted that our maximum-flow-based checking is performed for three resource types (CLB, DSP, and RAM) separately each time and we only conclude the updated problem is feasible when all three checks are passed.

Compared with the minimum-cost flow graph in Figure 21, the problem size is dramatically reduced by the clock grouping in the graph construction in Figure 23. In addition, edge costs are removed in the maximum-flow graph since we solve it only for feasibility checking purpose. Therefore, the proposed maximum-flow-based technique enables a much faster yet reliable feasibility checking process.

### 3.3.5 Speed-up Techniques

The minimum-cost flow formulation shown in Figure 21 is optimal for given $\lambda$ and unit logic resource demand but may suffer from long runtime for large designs. Instead of solving flat minimum-cost flow problems, we here propose a geometric clustering technique to reduce the problem size by grouping cells that are physically close and share the same set of clock nets together. Our proposed geometric clustering technique can significantly speed up the minimum-cost flow solving process while preserving good solution quality.

$$\underset{\mathbf{x},\lambda}{\text{minimize}} \quad \sum_{\mathrm{k}\in\mathcal{K}}\sum_{\mathrm{j}\in\mathcal{R}}\left(\frac{\mathrm{P_k}}{\mathrm{A_k}}\cdot \mathrm{D_{k,j}} + \lambda_{\mathrm{k,j}}\right)\cdot \mathbf{x}_{\mathrm{k,j}}, \tag{17a}$$

$$\text{subject to} \quad \mathbf{x}_{k,j} \in \{0,1,2,\dots\}, \forall k \in \mathcal{K}, \forall j \in \mathcal{R}, \tag{17b}$$

$$\sum_{j\in\mathcal{R}} \mathbf{x}_{k,j} = A_k, \forall k \in \mathcal{K}, \tag{17c}$$

$$\sum_{k\in\mathcal{K}} \mathbf{x}_{k,j} \leq C_j, \forall j \in \mathcal{R}. \tag{17d}$$

With our clustering technique applied, instead of solving Formulation (15), we solve a very similar

Formulation (17), where $\mathcal{K}$ denotes the set of clusters, $P_k$ and $A_k$ denote the total pin count and total logic resource demand of cells in cluster $k$, and $D_{k,j}$ denotes the physical distance between the average location of cells in cluster $k$ and clock region $j$. Besides, unit logic resource demand is assumed for all cells (not clusters).

Formulation (17) can still be solved utilizing the minimum-cost flow transformation illustrated in Figure 21. However, comparing to Formulation (15), several key differences need to be emphasized.

Firstly, since inaccuracy is injected to pin movement calculation by using averaged pin count $\frac{P_k}{A_k}$ and averaged cell locations for $D_{k,j}$ in the objective (17a), less optimal solutions will be obtained as the cluster size increases. On the other hand, a larger cluster size can dramatically reduce the problem size, which results in much faster runtime. Therefore, with different partition sizes, we can achieve different tradeoffs between quality and runtime. In UTPlaceF 2.0, the partition width and height are empirically set to 5 CLB sites.

Secondly, in the minimum-cost flow graph with our clustering technique applied, each assignment object represents a cluster of cells and does not have unit logic resource demand anymore (i.e., $A_k \geq 1$). Thus, the final minimum-cost flow solution of Formulation (17) might assign non-zero flows to more than one edge of an assignment object. It is physically equivalent to splitting a cluster into subgroups and assigning them to multiple clock regions. For each of these split cases, to decide which cell goes to which clock region, one more level of minimum-cost flow corresponding to Formulation (15) is needed. Note that, this second-level minimum-cost flow is only for cells within the split cluster and the set of clock regions that this cluster has been assigned to. Since we assume unit resource demand for all cells, split assignment would not happen again in these second-level minimum-cost flows and the solution feasibility can be guaranteed.

#### 3.3.6 Half-Column Region Assignment

As shown in Figure 20, half-column region constraint is explicitly respected in legalization stage by our half-column region assignment technique. Here we formally define the half-column region assignment problem as follows.

**Problem 3 (Half-Column Region Assignment)** *Given a rough-legalized placement, logic resource capacity of each half-column region, and a feasible clock region assignment solution, assign cells within each clock region to half-column regions to minimize total pin movement without logic resource and clock overflow.*

Problem 3 is very similar to the clock region assignment problem defined in Problem 2. It can also be formulated into Formulation (14) but with a simpler clock constraint (14e). So the same relaxation and

minimum-cost flow framework described in Section 3.3.2 and Section 3.3.3 can be seamlessly applied to solve half-column region assignment problem as well.

**Algorithm 8** Penalty Multiplier Updating For Half-Column Region Constraint

**Input:** Penalty multiplier $\lambda$.
**Output:** An updated penalty multiplier $\lambda$ that will result in less clock overflow.

```
1: Find the half-column region r with the largest clock overflow O_max;
2: for each clock e in r do
3:     l ← l ∪ {Cost(e, r)};                          ▷ See Equation (18)
4: end for
5: Sort candidates in l by ascending order of their cost;
6: i ← 0;
7: for each Cost(e, r) ∈ sorted l do
8:     if ¬IsFeasible(e, r, λ) then
9:         Continue;
10:    end if
11:    for each cell i in clock e do
12:        λ_{i,r} ← ∞;
13:    end for
14:    i ← i + 1;
15:    if i ≥ O_max then
16:        Return;
17:    end if
18: end for
```

The only notable difference from the clock region assignment is the method to update penalty multipliers for clock overflow resolving. Our proposed penalty multiplier updating algorithm for half-column region constraint is summarized in Algorithm 8. It should be noted that, given a feasible clock region assignment solution, the half-column region assignment can be performed within each clock region independently without impacting clock legality. So the scope of Algorithm 8 is only limited in a single clock region.

Within a clock region, we first find the target half-column region with the largest clock overflow in line 1. Then, the cost of moving each clock out of the target half-column region is calculated and stored in a list from line 2 to line 4. The cost of moving clock $e$ out of half-column $r$ is defined as follows,

$$Cost(e, r) = \sum_{i \in \mathcal{V}(e) \cap \mathcal{V}(r)} P_i, \tag{18}$$

where $\mathcal{V}(e) \cap \mathcal{V}(r)$ denotes the set of cells that are simultaneously in clock $e$ and half-column region $r$, and $P_i$ represents the total number of pins associated with cell $i$.

In the loop from line 7 to line 18, we iteratively pick the clock with the smallest moving cost and block it out of the target half-column region in the subsequent assignment iterations. A feasibility check similar to the one illustrated in Figure 23 is performed in line 8 to ensure the existence of feasible assignments after the edge blocking. This process is repeated until the number of iterations hits the overflow limit in line 15.

## 3.4 Experimental Results

UTPlaceF 2.0 was implemented in C++ and compiled by g++ 4.7.2. The official contest evaluation results conducted by Xilinx is used here to demonstrate the effectiveness of UTPlaceF 2.0.

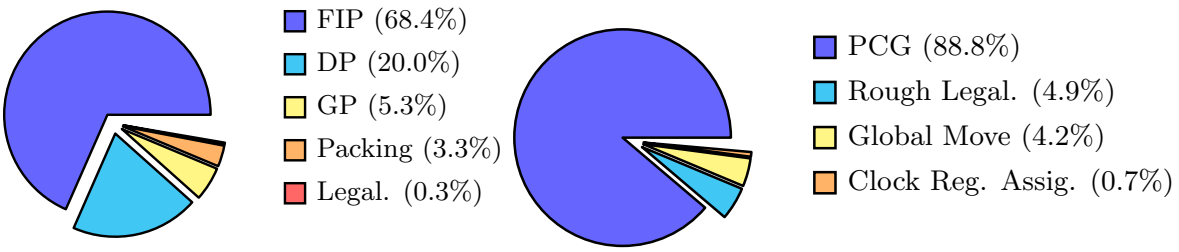
The characteristics of ISPD'17 benchmark suite are listed in Table 10. This benchmark suite consists of industry-strength designs with gate counts ranging from 0.45 million to about 1 million. Several designs in this suite have extremely high resource utilization and clock usage.

Table 10: ISPD'17 Placement Contest Benchmarks Statistics

| Benchmark | #LUT | #FF | #RAM | #DSP | #Clocks |
|---|---|---|---|---|---|
| CLK-FPGA01 | 215K | 236K | 170 | 75 | 30 |
| CLK-FPGA02 | 215K | 236K | 170 | 75 | 30 |
| CLK-FPGA03 | 242K | 270K | 255 | 112 | 33 |
| CLK-FPGA04 | 268K | 300K | 340 | 150 | 36 |
| CLK-FPGA05 | 295K | 325K | 425 | 187 | 39 |
| CLK-FPGA06 | 322K | 354K | 510 | 225 | 42 |
| CLK-FPGA07 | 350K | 384K | 595 | 262 | 45 |
| CLK-FPGA08 | 376K | 414K | 680 | 300 | 48 |
| CLK-FPGA09 | 392K | 431K | 765 | 337 | 51 |
| CLK-FPGA10 | 408K | 449K | 850 | 375 | 54 |
| CLK-FPGA11 | 424K | 450K | 900 | 397 | 55 |
| CLK-FPGA12 | 440K | 484K | 950 | 420 | 56 |
| CLK-FPGA13 | 456K | 503K | 1000 | 442 | 57 |
| Resources | 538K | 1075K | 1728 | 768 | N/A |

As UTPlaceF 2.0 won the first place in ISPD'17 placement contest, we here compare our results with the second- and third-place contest winners. Table 11 shows the routed wirelength comparison results that reported by Xilinx Vivado v2016.4. It can be seen that UTPlaceF 2.0 achieves the best overall routed wirelength and outperforms by 4.0% and 11.0% in routed wirelength compared with the other two contest winners, respectively.

The runtime comparison is shown in Table 12. Running in a single thread, UTPlaceF 2.0 completes the largest benchmark CLK-FPGA13 (0.96M cells) in 20 minutes. We also report the runtime breakdown of UTPlaceF 2.0 in Figure 24(a). On average, the majority (68.4%) of the total runtime is taken by FIP, while detailed placement, CLB-level global placement, packing, and legalization respectively consume 20.0%, 5.3%, 3.3%, and 0.3% of the total runtime. We further divide the runtime of FIP into four components, as shown in Figure 24(b), where the preconditioned conjugate gradient for quadratic programming takes 88.8% of the FIP runtime, followed by 4.9% and 4.2% for rough legalization and density-preserving global move, respectively, and only 0.7% is taken by the clock region assignment.




(a) (b)

Figure 24: Runtime breakdown of (a) UTPlaceF 2.0 and (b) flat initial placement (FIP).

Table 11: Routed Wirelength Comparison with ISPD'17 Contest Winners

| Benchmark | 2nd Place | | 3rd Place | | UTPlaceF 2.0 (1st Place) | |
|---|---|---|---|---|---|---|
| | Routed WL | Ratio | Routed WL | Ratio | Routed WL | Ratio |
| CLK-FPGA01 | 2209328 | 1.001 | 2268532 | 1.027 | 2208170 | 1.000 |
| CLK-FPGA02 | 2273729 | 0.998 | 2504444 | 1.099 | 2279171 | 1.000 |
| CLK-FPGA03 | 6229292 | 1.164 | 5803110 | 1.084 | 5353071 | 1.000 |
| CLK-FPGA04 | 3817377 | 1.032 | 4085670 | 1.105 | 3697950 | 1.000 |
| CLK-FPGA05 | 4995177 | 1.065 | 5180916 | 1.104 | 4692356 | 1.000 |
| CLK-FPGA06 | 5605573 | 1.003 | 6216898 | 1.112 | 5588507 | 1.000 |
| CLK-FPGA07 | 2504544 | 1.024 | 2676088 | 1.095 | 2444837 | 1.000 |
| CLK-FPGA08 | 1989632 | 1.055 | 2057117 | 1.091 | 1885632 | 1.000 |
| CLK-FPGA09 | 2583442 | 0.995 | 2813538 | 1.084 | 2596654 | 1.000 |
| CLK-FPGA10 | 4770168 | 1.069 | 4839765 | 1.084 | 4464341 | 1.000 |
| CLK-FPGA11 | 4207699 | 1.006 | 4777177 | 1.142 | 4184233 | 1.000 |
| CLK-FPGA12 | 3376930 | 1.002 | 3739517 | 1.110 | 3368698 | 1.000 |
| CLK-FPGA13 | 3920965 | 1.019 | 4320345 | 1.123 | 3847832 | 1.000 |
| Norm. | 1.040 | - | 1.100 | - | 1.000 | - |

# 4 UTPlaceF 3.0: A Parallelization Framework for FPGA Global Placement

Placement is one of the most time-consuming optimization steps in modern FPGA implementation flow. In the past several decades, most of the research attention and endeavor was focused on improving placement solution quality. However, ultra-fast and efficient placement core engines are now in great demand. Firstly, with the increasing capacity and complexity of modern FPGA devices, the gate count of state-of-the-art commercial FPGAs has reached the scale of millions [1, 3]. Moreover, as a type of hardware accelerators, FPGAs need to be frequently reconfigured to adapt rapid and continuous changes from users in many scenarios, like datacenter-based cloud computing. Therefore, it is particularly desirable to develop a high-performance placement engine to reduce the FPGA implementation time. Besides, placement also has significant impacts on the quality of design mapping, hence, good performance-boosting techniques should still maintain competitive placement solution quality.

The scalability issue has constantly pushed the evolution of placement algorithms in the past few decades. In the early age of placement, simulated annealing-based placers, such as [17–19], dominated industry and

Table 12: Runtime (Seconds) Comparison with ISPD'17 Contest Winners

| Benchmark | 2nd Place | | 3rd Place | | UTPlaceF 2.0 (1st Place) | |
|---|---|---|---|---|---|---|
| | Runtime | Ratio | Runtime | Ratio | Runtime | Ratio |
| CLK-FPGA01 | 3023 | 5.68 | 354 | 0.67 | 532 | 1.00 |
| CLK-FPGA02 | 3153 | 6.15 | 333 | 0.65 | 513 | 1.00 |
| CLK-FPGA03 | 4066 | 3.91 | 666 | 0.64 | 1039 | 1.00 |
| CLK-FPGA04 | 3077 | 4.33 | 464 | 0.65 | 711 | 1.00 |
| CLK-FPGA05 | 3631 | 3.87 | 680 | 0.72 | 939 | 1.00 |
| CLK-FPGA06 | 3836 | 3.60 | 695 | 0.65 | 1066 | 1.00 |
| CLK-FPGA07 | 3953 | 4.68 | 410 | 0.49 | 845 | 1.00 |
| CLK-FPGA08 | 4395 | 8.31 | 277 | 0.52 | 529 | 1.00 |
| CLK-FPGA09 | 5428 | 6.45 | 414 | 0.49 | 842 | 1.00 |
| CLK-FPGA10 | 3305 | 3.39 | 516 | 0.53 | 974 | 1.00 |
| CLK-FPGA11 | 4341 | 4.06 | 548 | 0.51 | 1068 | 1.00 |
| CLK-FPGA12 | 4949 | 6.39 | 413 | 0.53 | 774 | 1.00 |
| CLK-FPGA13 | 3748 | 3.20 | 548 | 0.47 | 1172 | 1.00 |
| Norm. | 4.63 | - | 0.57 | - | 1.00 | - |

academic research. Although the annealing technique worked well for small designs, as the capacity of FPGAs kept growing, it was no longer scalable for larger FPGA devices. Industry and academia then turned to min-cut-partitioning-based placers, e.g. [20, 21], which performed well for designs with tens of thousands gates. When the gate count of FPGAs arrived at the scale of millions, analytical placers, e.g. [22–30] started to steadily outperform min-cut-partitioning-based approaches in both runtime and quality.

Despite the effectiveness and efficiency of analytical placers, their runtime still increases rapidly as the complexity and scale of FPGA devices has not stopped growing. One promising solution is leveraging today's powerful multi-core CPUs to achieve efficient placement parallelism. For quadratic placers, wirelength is approximated as a quadratic objective, which can be minimized by solving symmetric and positive-definite (SPD) linear systems. The solution can be quickly approached by various Krylov-subspace methods, among which Preconditioned Conjugate Gradient (PCG) method is the most efficient known algorithm for placement problem. As solving SPD linear systems dominates the total runtime of placement, some previous effort has been made on parallelizing PCG from various angles. SimPL [33], a quadratic placer for application-specific integrated circuits (ASICs), achieved 1.89× speedup for PCG by solving x- and y-directed wirelength simultaneously with 2 threads. Further speedup is possible by parallelizing matrix-vector operations in PCG, however, [51] showed that this technique cannot even achieve 2× speedup by using 16 threads in their experiments.

For nonlinear placers, wirelength and other constraints (e.g. cell density) are modeled into one single nonlinear objective, optimizing which becomes the runtime bottleneck. Unlike quadratic placers, the x and y directions in nonlinear placers cannot be decomposed into two independent components, therefore, parallelizing nonlinear placers is even harder.

A recent work, POLAR 3.0 [51], presented a quadratic placement parallelization framework for ASICs

based on geometric partitioning. In particular, they divided cells into partitions based on their physical locations, then performed PCG and rough legalization [33] for each partition separately in parallel. In order to reduce solution quality loss, full-netlist placement was performed periodically to allow inter-partition cell moving. Although their approach achieved promising results (4× speedup with 1.2% wirelength degradation by using 16 threads) on ASIC benchmarks, similar performance gain might not be reproducible to FPGA designs for the following two reasons: 1) placeable cells and nets in FPGA designs typically have more pins compared with ASIC designs, which leads to a stronger inter-partition connection and a larger quality degradation and 2) various FPGA architecture constraints (e.g. clock legalization rules [49]) impose difficulties to the parallelization of rough legalization.

In this section, we propose a highly parallelized quadratic placement framework for FPGAs, UTPlaceF 3.0, which takes full advantages of modern multi-core CPUs and delivers an appealing performance boost. Besides, the placement quality loss also is taken care of with only very little runtime overhead.

## 4.1 Preliminaries

### 4.1.1 Quadratic Placement

An FPGA netlist can be represented as a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$ is the set of cells, and $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$ is the set of nets. Let $\mathbf{x} = \{x_1, x_2, ..., x_{|\mathcal{V}|}\}$ and $\mathbf{y} = \{y_1, y_2, ..., y_{|\mathcal{V}|}\}$ be the x and y coordinates of all cells. The wirelength-driven global placement problem is to determine cell position vectors $\mathbf{x}$ and $\mathbf{y}$ such that the total wirelength is minimized. Wirelength is measured by half-perimeter wirelength (HPWL) defined as follows.

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \sum_{e \in \mathcal{E}} \{\max_{i,j \in e} |x_i - x_j| + \max_{i,j \in e} |y_i - y_j|\}. \tag{19}$$

As HPWL is not differentiable everywhere, quadratic placers approximate it by squared Euclidean distance between cells using various net models, such as hybrid net model [52] and bound-to-bound (B2B) net model [53]. Therefore, the wirelength cost function in quadratic placers is defined as

$$W(\mathbf{x}, \mathbf{y}) = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_x^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_y^T \mathbf{y} + const. \tag{20}$$

Since both Hessian matrices $Q_x$ and $Q_y$ in Eq. (20) are SPD, minimizing $W(\mathbf{x}, \mathbf{y})$ is equivalent to solving

the following two linear systems.

$$Q_x \mathbf{x} = -\mathbf{c}_x, \tag{21a}$$

$$Q_y \mathbf{y} = -\mathbf{c}_y. \tag{21b}$$

To eliminate cell overlapping, most state-of-the-art quadratic placers [28–30] adopted a cell-spreading technique, called rough legalization [33], which adds extra spreading force pointing to cells' anchor locations (i.e., locations that satisfy cell density constraint). After each rough legalization execution, linear systems (21a) and (21b) are updated accordingly and solved again. The loop of solving linear systems and performing rough legalization is repeated until cells are fully spread out.

## 4.2 Empirical Runtime Study

In this section, we will empirically analyze the runtime bottleneck of a state-of-the-art quadratic placer and evidence that achieving a highly scalable parallel placement is indeed a challenging problem.

A representative rough legalization-based quadratic placement flow is presented in Fig. 25. To show its runtime bottlenecks, we implemented a pure wirelength-driven UTPlaceF [28] and performed experiments on ISPD'16 benchmark suites with a single thread. As illustrated in Fig. 26, the three major runtime contributors, on average, are solving linear systems (85.1%), rough legalization (7.5%), and linear system construction (7.3%), respectively.

Due to the decomposability of x- and y-directed wirelength in quadratic placement, the linear systems (21a) and (21b) can be constructed and solved perfectly in parallel using two threads. However, further parallelization becomes harder and inefficient.

The most efficient linear system construction approach is to scan and fill Hessian matrices $Q_x$ and $Q_y$ in a net-by-net manner. Considering multiple nets might contribute to the same entry in $Q_x$ and $Q_y$ (e.g. multiple nets share the same cell), there are two potential parallelization strategies: (1) processing nets in parallel and adding mutex lock[2] to each non-zero entry and (2) partitioning nets into groups and constructing partial Hessian matrices for each group in parallel, then, joining all partial Hessian matrices together. As can be seen, both strategies introduce considerable runtime overhead, therefore, linear system construction is difficult to be parallelized.

Solving linear systems, as the most time-consuming step, is inherently hard to be parallelized as well due to the iterative nature of PCG. Experiments in [51] showed that only 1.8× speedup can be achieved by using

[2]Mutex (mutually exclusive) lock is a mechanism that guarantees a resource can only be accessed by one thread at a time in a multi-threading environment.

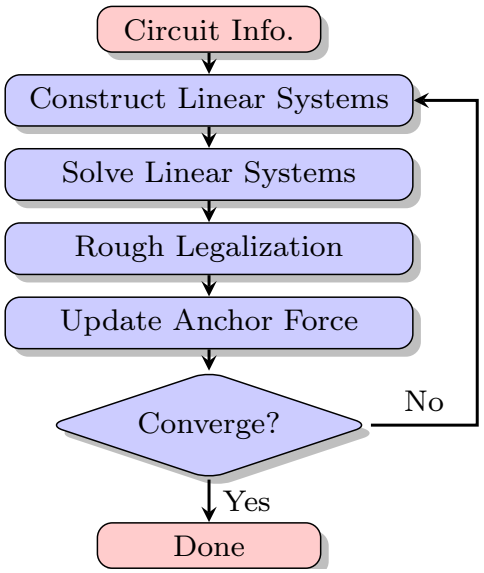



Figure 25: A representative rough legalization-based quadratic placement flow.

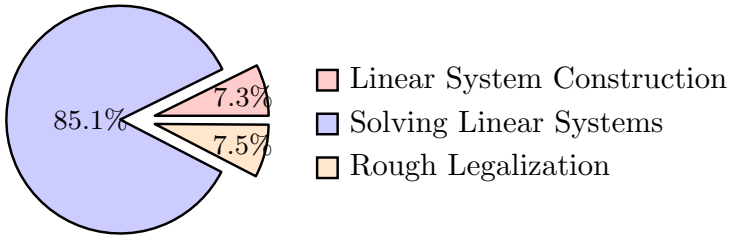



Figure 26: Normalized runtime of the three major runtime contributors in our pure wirelength-driven UTPlaceF on ISPD'16 benchmark suite.

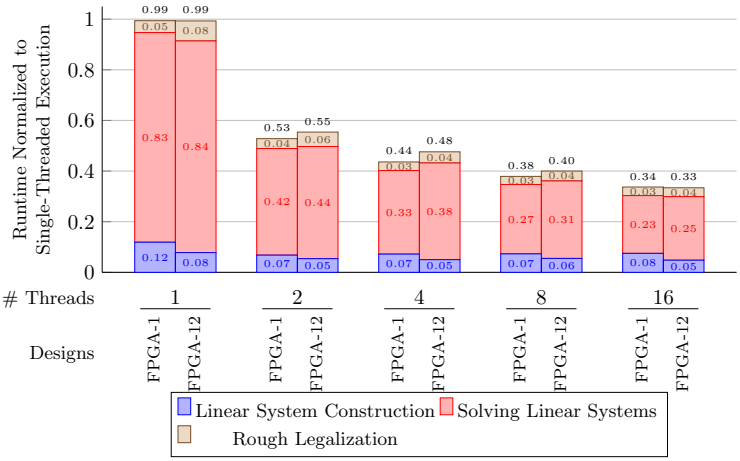



Figure 27: Runtime scaling of the three major runtime contributors in our pure wirelength-driven UTPlaceF by multi-threading. FPGA-1 and FPGA-12 contain 0.1M and 1.1M cells, respectively.

the cutting-edge PCG solver in Intel MKL library [54] on an 8-core machine.

Unless special care is taken, rough legalization can only handle one cell density hotspots at a time, since spreading windows of multiple hotspots might intersect to each other. For each hotspot, the frequent cell sortings for horizontal and vertical cell spreading dominates the runtime. Although parallelism can be applied here, experiments in [33] only achieved 1.62× speedup by using 8 threads.

To manifest the placement parallelization crisis, we further parallelized our pure wirelength-driven UTPlaceF in the following way utilizing OpenMP [45]: 1) constructing and solving linear systems for x and y in parallel, 2) enabling parallelization for matrix-vector operations in PCG if more than two threads are available, and 3) substituting sequential sortings in rough legalization with their equivalent parallel version in GNU libstdc++ parallel mode [55]. We then performed experiments using a 2.60 GHz Intel Xeon E5-2690 v3 CPU with 24 cores on two representative (the smallest and the largest) designs in ISPD'16 benchmark suite. 1, 2, 4, 8, and 16 threads were enabled respectively. Fig. 27 presents the runtime scaling of the

three major runtime contributors. As shown, most PCG speedup is from simultaneously solving x and y (1 thread vs 2 threads) and it plateaus out quickly on matrix-vector level parallelization (from 2 threads to 16 threads). Both linear system construction and rough legalization scale poorly. The overall speedup saturates at around 3X.

In summary, decomposing x and y and low-level parallelization (i.e., sortings and matrix-vector operations) alone can only achieve about 3× speedup. In this work, we will explore innovative high-level approaches to break this parallelism wall.

## 4.3 UTPlaceF 3.0 Algorithms

### 4.3.1 Overall Flow

The overall flow of UTPlaceF 3.0 is illustrated in Fig. 28. The whole flow starts with the initial placement consisting of one pass of linear system construction and solving followed by rough legalization and anchor force updating. After that, the netlist is divided into several sub-netlists utilizing hypergraph min-cut partitioning techniques. It should be noted that the resulting sub-netlists are not necessarily physically separated. In practice, cells from different sub-netlists are most likely mixed together. Once the partitioning is done, the linear systems of each sub-netlist are constructed and solved independently in parallel. However, because of the interdependency among sub-netlists (i.e., inter-partition nets), sub-netlist placement might not converge to the optimal solution, which leads to placement quality loss. Therefore, an incremental placement correction step, which takes the inter-partition dependency into consideration, is performed to reduce the quality degradation after all sub-netlist placements are done. Finally, rough legalization and anchor force updating are executed. The loop of parallelized placement, rough legalization, and anchor force updating is repeated until the wirelength converges.

### 4.3.2 Placement-Driven Block-Jacobi Preconditioning

As demonstrated by the empirical study in Section 4.2, PCG dominates the total runtime of global placement. Thus, parallelizing PCG is of top priority. An effective PCG parallelization scheme is to leverage the strength of block-Jacobi method. The block-Jacobi method essentially exploits the divide-and-conquer approach to transform a large linear system into a set of independent smaller sub-systems and solve each of them individually.

Fig. 29 shows an example of applying an 8-way block-Jacobi method on a sparse SPD matrix $Q$. The first step (from Fig. 29(a) to Fig. 29(b)) is to find a good row/column permutation such that the sum of off-block-diagonal non-zero entries is minimized. As an SPD matrix can be represented as an undirected

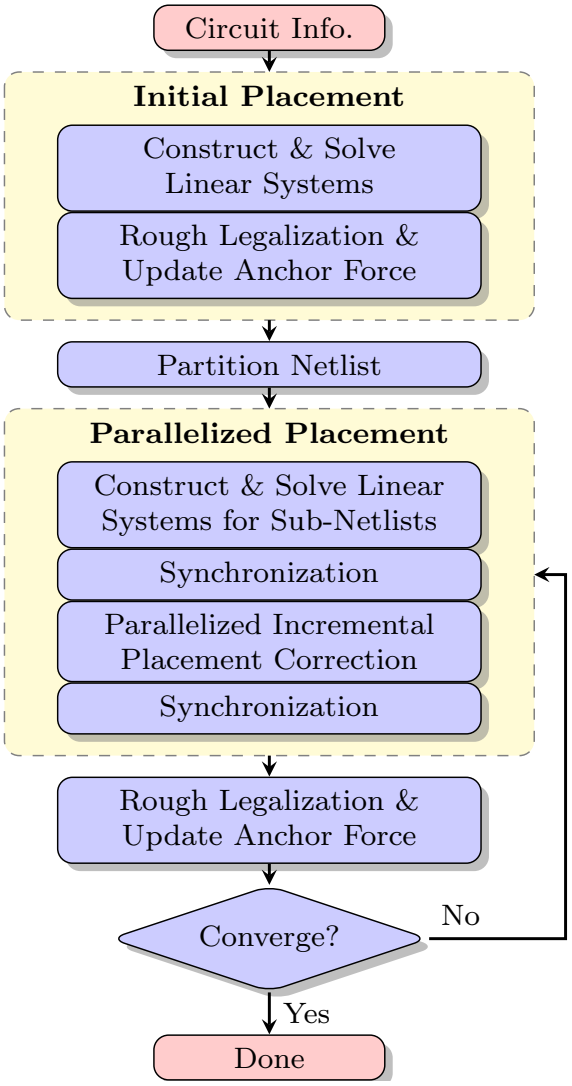



Figure 28: The overall flow of UTPlaceF 3.0.

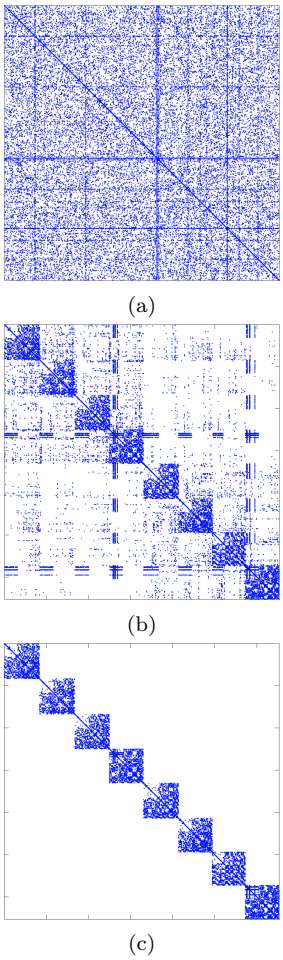



Figure 29: An example of building a block-Jacobi preconditioner from a sparse SPD matrix $Q$ based on an 8-way partitioning. (a) The matrix $Q$. (b) After an 8-way partitioning by row/column permutation. (c) The resulting block-Jacobi preconditioner of $Q$.

weighted graph (UWG), finding the best permutation for $k$ diagonal blocks is equivalent to finding the $k$-way min-cut partitioning of the induced UWG. It should be noted that the permutated $Q$ (Fig. 29b) is mathematically equivalent to the original $Q$ (Fig. 29a), thus, with proper permutation on decision vectors ($\mathbf{x}$ and $\mathbf{y}$) and right-hand sides ($\mathbf{c}_x$ and $\mathbf{c}_y$), a permutated linear system is also equivalent to the original one. After the permutation, the block-Jacobi preconditioner in Fig. 29(c) can be obtained by simply ignoring all off-block-diagonal non-zero entries in the permutated matrix in Fig. 29(b).

From placement point of view, each diagonal block in the block-Jacobi preconditioner corresponds to a partition (sub-netlist) and solving linear systems (21a) and (21b) with block-Jacobi preconditioner is physically equivalent to performing placement for each partition individually by assuming all off-partition

cells are fixed at location $(x = 0, y = 0)$[3]. Although this approach scales almost linearly with the number of partitions created, the placement quality would degrade dramatically as wrong physical locations are assumed for cells in inter-partition nets.

To mitigate this problem, rather than using (0, 0) as the fixed point, off-partition cells are considered fixed at their locations induced from the linear system solutions of the previous placement iteration. An example of a three-pin net spanning three partitions is shown in Fig. 30. We call this process "placement-driven block-Jacobi preconditioning".

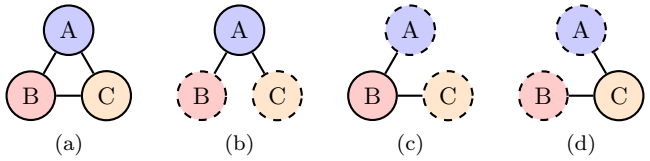



Figure 30: An inter-partition net spanning three partitions. (a) illustrates the net in the original netlist. (b), (c), and (d) show the net in the partition containing A, B, and C, respectively. Dashed circles represent off-partition cells that are fixed at their last locations.

Given a netlist consisting of the set of cell $\mathcal{V}$, a partitioning $\mathcal{P}$ on $\mathcal{V}$, and the $\mathcal{P}$-permutated linear systems (21a) and (21b), if we denote the belonging partition of cell $i \in \mathcal{V}$ by $\pi(i)$ and denote the placement-driven block-Jacobi preconditioned linear systems of (21a) and (21b) by

$$\widehat{Q}_x \mathbf{x} = -\widehat{\mathbf{c}}_x, \tag{22a}$$

$$\widehat{Q}_y \mathbf{y} = -\widehat{\mathbf{c}}_y, \tag{22b}$$

then, $\widehat{Q}$ ($\widehat{Q}_x$ and $\widehat{Q}_y$) and $\widehat{\mathbf{c}}$ ($\widehat{\mathbf{c}}_x$ and $\widehat{\mathbf{c}}_y$) can be defined as

$$\widehat{Q}_{i,j} = \begin{cases} 0, & \text{if } \pi(i) \neq \pi(j), \forall i, j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise,} \end{cases} \tag{23a}$$

$$\widehat{\mathbf{c}}_i = \mathbf{c}_i + \sum_{k \in \mathcal{V} \setminus \pi(i)} Q_{i,k} \cdot l_k, \forall i \in \mathcal{V}, \tag{23b}$$

where $l_k$ denotes the (x or y) coordinate of cell $k \in \mathcal{V}$ in the previous placement iteration. Compared with normal block-Jacobi preconditioning, our placement-driven block-Jacobi preconditioning only differs by the right-hand sides in linear systems. Therefore, the SPD property is retained in linear systems (22a) and (22b) and diagonal blocks can still be solved independently by PCG.

[3]The wirelength cost in x direction for two cells can be written as $w_{i,j}(x_i - x_j)^2$. If cell $i$ and $j$ are not in the same partition, the term $-2w_{i,j}x_i x_j$ would be ignored in the block-Jacobi preconditioner. Therefore, this cost term becomes $w_{i,j}(x_i - 0)^2 + w_{i,j}(x_j - 0)^2$. For cell $i$ ($j$), this is equivalent to assuming cell $j$ ($i$) is fixed at $x = 0$. A similar conclusion is hold for y direction.

Since the matrices $Q_x$ and $Q_y$ are placement dependent, from quality wise, it is ideal to perform two (one each for x and y) UWG min-cut partitionings for each placement iteration. However, doing so will result in a dramatic amount of runtime overhead. Therefore, only one partitioning is done right before the first parallelized placement step as shown in Fig. 28. Considering $Q_x$ and $Q_y$ change after every placement iteration, hypergraph min-cut partitioning is adopted to approximate the varied optimal matrix partitionings.

#### 4.3.3 Parallelized Incremental Placement Correction (PIPC)

Although applying placement-driven block-Jacobi preconditioning can reduce quality loss to some extent, it is no longer effective enough once more partitions (e.g. more than 4) are created. To further mitigate placement quality degradation introduced by partitioning, the idea of additive correction multigrid method [6] is adopted to incrementally correct the solutions of linear systems (22a) and (22b).

Without loss of generality, we only discuss x direction in the rest of this session, but conclusions that hold for x are also applicable to y.

Let $\mathbf{x}^{(0)}$ be the solution of linear system (22a), that is

$$\widehat{Q}\mathbf{x}^{(0)} = -\widehat{\mathbf{c}}, \tag{24}$$

where $\widehat{Q}$ and $\widehat{\mathbf{c}}$ are defined in Eq. (23a) and Eq. (23b), respectively. If we can find $\Delta\mathbf{x}$ satisfying

$$Q\Delta\mathbf{x} = -\mathbf{c} - Q\mathbf{x}^{(0)}, \tag{25}$$

the solution of the original linear system (21a) will be $\mathbf{x}^{(0)} + \Delta\mathbf{x}$. Intuitively, $\Delta\mathbf{x}$ represents the discrepancy between current placement $\mathbf{x}^{(0)}$ and the optimal placement $\mathbf{x}^{\star}$ and it is, physically, a very local and smooth perturbation around the placement $\mathbf{x}^{(0)}$.

Now we present the approach to find $\Delta\mathbf{x}$. Let $\mathbf{r}^{(0)} = -\mathbf{c} - Q\mathbf{x}^{(0)}$ and $N = \widehat{Q} - Q$. Then, solving linear system (25) is equivalent to solving

$$\widehat{Q}\Delta\mathbf{x} = N\Delta\mathbf{x} + \mathbf{r}^{(0)}. \tag{26}$$

The iterative method now becomes solving $\Delta\mathbf{x}^{(k)}$ for $k \geq 1$ with given $\Delta\mathbf{x}^{(0)}$ by

$$\widehat{Q}\Delta\mathbf{x}^{(k+1)} = N\Delta\mathbf{x}^{(k)} + \mathbf{r}^{(0)}, \tag{27}$$

which can be further written as

$$\Delta \mathbf{x}^{(k+1)} = \Delta \mathbf{x}^{(k)} + \widehat{Q}^{-1}(\mathbf{r}^{(0)} - Q\Delta \mathbf{x}^{(k)}). \tag{28}$$

Intuitively, if $\Delta \mathbf{x}^{(k)}$ and $\Delta \mathbf{x}^{(k+1)}$ are becoming closer and closer (i.e., $\Delta \mathbf{x}^{(k)}$ converges), we have $\mathbf{r}^{(0)} - Q\Delta \mathbf{x}^{(k)}$ approaches to 0. This indicates that once $\Delta \mathbf{x}^{(k)}$ converges, it will always converge to the same solution. To formally prove the convergence, consider the following lemma [56].

**Lemma 2** *Let $Q = \widehat{Q} - N$ , with $Q$ and $\widehat{Q}$ symmetric and positive definite. If the matrix $2\widehat{Q} - Q$ is positive definite, then the iterative method defined in Eq.* (28) *is convergent for any choice of the initial $\Delta \mathbf{x}^{(0)}$. Moreover, the convergence of the iteration is monotone with respect to the norm $||\cdot||_Q$ (i.e., $||\mathbf{r}^{(0)} - Q\Delta \mathbf{x}^{(k+1)}|| < ||\mathbf{r}^{(0)} - Q\Delta \mathbf{x}^{(k)}||$, $k = 0, 1, ...$).*

Now we can conclude the following proposition.

**Proposition 1** *By picking any placement-driven block-Jacobi preconditioner of $Q$ as $\widehat{Q}$, the iterative method defined in Eq.* (28) *is monotonically convergent for any choice of $\Delta \mathbf{x}^{(0)}$.*

**Proof 1** *Since $|Q_{i,i}| > \sum_{j \neq i} |Q_{i,j}|, \forall i \in \mathcal{V}$, both $Q$ and $\widehat{Q}$ are symmetric strictly diagonally dominant matrices. By the definition of $\widehat{Q}$ in Eq.* (23a)*, the matrix $\Omega = 2\widehat{Q} - Q$ can be defined as*

$$\Omega_{i,j} = \begin{cases} -Q_{i,j}, & \text{if } \pi(i) \neq \pi(j), \forall i, j \in \mathcal{V}, \\ Q_{i,j}, & \text{otherwise.} \end{cases}$$

*So $\Omega = 2\widehat{Q} - Q$ is also a symmetric strictly diagonally dominant matrix. A symmetric strictly diagonally dominant real matrix with nonnegative diagonal entries is positive definite. Therefore, $2\widehat{Q} - Q$ is positive definite and, thus, the iterative scheme defined by Eq.* (28) *will converge monotonically.*

Proposition 1 reveals two important properties of the iterative method in Eq. (28): 1) with sufficient iterations, the exact solution, $\Delta \mathbf{x}$, of linear system (25) can be reached, hence, we can obtain the exact solution, $\mathbf{x}^{(0)} + \Delta \mathbf{x}$, of linear system (21a) and 2) since the convergence is monotone, more iterations guarantees better solutions (i.e., solutions that are closer to the exact solution). These two properties imply that, by using this iterative method, placement quality and runtime can be perfectly traded to each other. Thus, different trade-offs can be made accordingly under different scenarios.

Another attractive property of the iterative method in Eq. (28) is its strong parallelizability. Almost all

the runtime of solving Eq. (28) is taken by computing $\widehat{Q}^{-1}(\mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)})$, which is equivalent to solving

$$\widehat{Q}\mathbf{x} = \mathbf{r}^{(0)} - Q\Delta\mathbf{x}^{(k)}. \tag{29}$$

Recall that $\widehat{Q}$ is the placement-driven block-Jacobi preconditioner of Q defined in Eq. (23a). Fig. 31 illustrates the parallelization scheme of solving this linear system by an example with four partitions. By blocking matrix $Q$ along rows, each strap in $Q$ can be constructed independently and solving linear system (29) becomes solving the following sub-system for each partition separately and then assembling their solutions ($\mathbf{x}_i$) together.

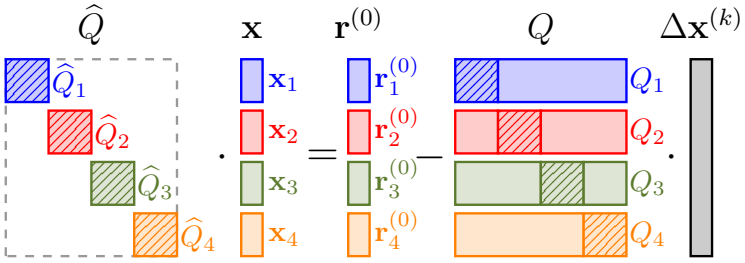



Figure 31: Parallelization scheme of solving linear system (29) with four partitions. Shaded regions represent diagonal blocks in $\widehat{Q}$ and $Q$. Different colors denote different partitions that can be solved in parallel.

$$\widehat{Q}_i\mathbf{x}_i = \mathbf{r}_i^{(0)} - Q_i\Delta\mathbf{x}^{(k)}, \forall i \in \mathcal{P}, \tag{30}$$

where $\widehat{Q}_i$, $x_i$, $\mathbf{r}_i^{(0)}$, and $Q_i$ are partial matrices and vectors corresponding to partition $i$ as illustrated in Fig. 31, and $\mathcal{P}$ denotes the set of partitions.

Algorithm 9 summarizes the overall flow of our parallelized incremental placement correction (PIPC) scheme. The partial matrix $Q_i$ of $Q$ for each partition is constructed in parallel from line 1 to line 3. Then the linear system (30) for each partition is solved in parallel from line 7 to line 9. After that, the solution of linear system (29) is obtained by assembling partial solutions together in line 10. The solution after each correction iteration is incrementally updated in line 11. The loop from line 6 to line 13 is repeated until the correction iteration limit $I$ is reached.

#### 4.3.4 Varied PIPC Configuration

In general, placement quality degrades as the number of partitions increases. So it is not wise to perform the same amount of PIPC for different number of partitions, which might be overkill for small partition counts but insufficient for a large number of partitions. To resolve this issue, PIPC is configured based on the number of partitions created accordingly.

In UTPlaceF 3.0, we configure three related variables: 1) the frequency of applying PIPC, 2) the number

**Algorithm 9** Parallelized Incremental Placement Correction

**Input:** The set of partitions $\mathcal{P}$, the solution of linear system (22a) $\mathbf{x}^{(0)}$, and the number of correction iterations $I$.
**Output:** A better solution than $\mathbf{x}^{(0)}$.

**parallel for** each $i \in \mathcal{P}$ **do**
  Construct $Q_i$; ▷ See Fig. (31)
**end parallel for**
$k \leftarrow 0$;
$\Delta\mathbf{x}^{(0)} \leftarrow 0$;
**while** $k < I$ **do**
  **parallel for** each $i \in \mathcal{P}$ **do**
    Solve $\widehat{Q}_i \Delta\mathbf{x}_i^{(k+1)} = \mathbf{r}_i^{(0)} - Q_i \Delta\mathbf{x}^{(k)}$;
  **end parallel for**
  $\Delta\mathbf{x}^{(k+1)} \leftarrow (\Delta\mathbf{x}_1^{(k+1)}, \Delta\mathbf{x}_2^{(k+1)}, ..., \Delta\mathbf{x}_{|\mathcal{P}|}^{(k+1)})$;
  $\Delta\mathbf{x}^{(k+1)} \leftarrow \Delta\mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k+1)}$;
  $k \leftarrow k + 1$;
**end while**
**return** $\mathbf{x}^{(0)} + \Delta\mathbf{x}^{(I)}$;

of correction iterations in each PIPC, and 3) the number of PCG iterations in PIPC. For small partition counts, good placement quality can still be maintained by infrequent PIPC (e.g. once every 3 placement iterations) with only a few correction iterations (e.g. 1 or 2) each time. As the partition count increases, PIPC frequency and correction iterations need to be increased properly to control quality degradation. The third variable, PCG iteration count in PIPC, is tuned to further save runtime. Since the solution of PIPC ($\Delta\mathbf{x}$) is essentially a local perturbation of the existing placement ($\mathbf{x}^{(0)}$), its magnitude is typically much smaller than $\mathbf{x}^{(0)}$. Therefore, we can save some PCG iterations in PIPC without losing too much placement quality. The detailed configuration will be further discussed in Section 4.4.1.

## 4.4 Experimental Results

UTPlaceF 3.0 is implemented in C++ and compiled by g++ 4.8.4. OpenMP 4.0 [45] is used to support multi-threading, GNU libstdc++ parallel mode [55] is used to parallelize critical sortings in rough legalization, hypergraph partitioning tool PaToh [57] is used to partition netlists, and the PCG in Eigen3 [58] is used to solve sparse SPD linear systems. All the experiments are performed on a Linux machine running with Intel Xeon E5-2690 v3 CPUs (2.60 GHz, 24 cores, and 30M L3 cache) and 64 GB RAM.

The benchmark suite released by Xilinx for ISPD'16 FPGA placement contest [59] is used to evaluate the efficiency of UTPlaceF 3.0. The statistics of the benchmarks are listed in Table 1. Since this work is focused on placement parallelization, all routability optimizations are discarded and only wirelength is considered. In the experiments, the CLB resource demands of all LUTs and FFs are set to 0.08 and the target density is set to 1.0 for all benchmarks. In the target FPGA architecture, considering a vertical route needs to go

through about twice as many switch boxes as a horizontal route needs for the same length, the wirelength is measured by the scaled HPWL (sHPWL) defined as

$$sHPWL = 0.5 \cdot HPWL_x + HPWL_y, \tag{31}$$

where $HPWL_x$ and $HPWL_y$ denote the x- and y-directed components of HPWL in Eq. (19), respectively.

### 4.4.1 Parallelization Configuration

Table 13: Parallelization Configuration of UTPlaceF 3.0 Under Different Number of Threads

| # Threads | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| # Partitions | 1 | 1 | 2 | 4 | 8 |
| # PCG Iter. | 250 | 250 | 250 | 250 | 250 |
| PIPC Period | - | - | - | 3 | 1 |
| # Iter. of each PIPC | - | - | - | 1 | 1 |
| # PCG Iter. in PIPC | - | - | - | 150 | 50 |

Table 14: Comparison of UTPlaceF 3.0 without PIPC Under Different Number of Threads

| Designs | 1 Thread | | | | 2 Threads | | | | 4 Threads | | | | 8 Threads | | | | 16 Threads | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | WL | RT | WLR* | RTR† | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR | WL | RT | WLR | RTR |
| FPGA-1 | 227 | 44 | 1.000 | 1.000 | 227 | 24 | 1.000 | 0.546 | 225 | 15 | 0.987 | 0.341 | 233 | 10 | 1.025 | 0.220 | 233 | 7 | 1.027 | 0.165 |
| FPGA-2 | 437 | 72 | 1.000 | 1.000 | 437 | 40 | 1.000 | 0.557 | 427 | 29 | 0.977 | 0.394 | 450 | 17 | 1.029 | 0.238 | 474 | 12 | 1.085 | 0.172 |
| FPGA-3 | 1663 | 232 | 1.000 | 1.000 | 1663 | 131 | 1.000 | 0.567 | 1676 | 85 | 1.008 | 0.368 | 1750 | 51 | 1.052 | 0.222 | 1771 | 39 | 1.065 | 0.169 |
| FPGA-4 | 3375 | 250 | 1.000 | 1.000 | 3375 | 148 | 1.000 | 0.589 | 3361 | 90 | 0.996 | 0.359 | 3408 | 61 | 1.010 | 0.242 | 3449 | 43 | 1.022 | 0.172 |
| FPGA-5 | 6504 | 293 | 1.000 | 1.000 | 6504 | 173 | 1.000 | 0.589 | 6322 | 112 | 0.972 | 0.380 | 6489 | 71 | 0.998 | 0.243 | 6579 | 54 | 1.011 | 0.185 |
| FPGA-6 | 3144 | 450 | 1.000 | 1.000 | 3144 | 249 | 1.000 | 0.555 | 3188 | 153 | 1.014 | 0.341 | 3106 | 103 | 0.988 | 0.229 | 3301 | 75 | 1.050 | 0.167 |
| FPGA-7 | 5851 | 446 | 1.000 | 1.000 | 5851 | 251 | 1.000 | 0.562 | 5910 | 164 | 1.010 | 0.368 | 6004 | 99 | 1.026 | 0.223 | 6500 | 81 | 1.111 | 0.181 |
| FPGA-8 | 5555 | 526 | 1.000 | 1.000 | 5555 | 298 | 1.000 | 0.567 | 5667 | 179 | 1.020 | 0.340 | 6059 | 119 | 1.091 | 0.225 | 6077 | 86 | 1.094 | 0.163 |
| FPGA-9 | 7318 | 580 | 1.000 | 1.000 | 7318 | 323 | 1.000 | 0.557 | 7551 | 206 | 1.032 | 0.355 | 7564 | 136 | 1.034 | 0.235 | 7523 | 101 | 1.028 | 0.174 |
| FPGA-10 | 3062 | 579 | 1.000 | 1.000 | 3062 | 316 | 1.000 | 0.546 | 3059 | 200 | 0.999 | 0.346 | 3058 | 128 | 0.999 | 0.221 | 3130 | 96 | 1.022 | 0.165 |
| FPGA-11 | 6975 | 601 | 1.000 | 1.000 | 6975 | 335 | 1.000 | 0.558 | 7004 | 202 | 1.004 | 0.336 | 7053 | 133 | 1.011 | 0.221 | 8807 | 96 | 1.263 | 0.160 |
| FPGA-12 | 3837 | 824 | 1.000 | 1.000 | 3837 | 407 | 1.000 | 0.494 | 3723 | 260 | 0.970 | 0.316 | 3943 | 168 | 1.028 | 0.205 | 3926 | 125 | 1.023 | 0.151 |
| Geo. Mean | - | - | 1.000 | 1.000 | - | - | 1.000 | 0.557 | - | - | 0.999 | 0.353 | - | - | 1.024 | 0.227 | - | - | 1.065 | 0.169 |

* WLR: Wirelength ratio compared to 1-thread execution.
† RTR: Runtime ratio compared to 1-thread execution.

The parallelization configuration used for our experiments is presented in Table 13. If $N$ ($N > 1$) threads are available, we always generate $\lfloor \frac{N}{2} \rfloor$ partitions and deal x and y separately using 2 threads in each partition. The number of PCG iterations for solving linear systems (21a), (21b), (22a), and (22b) are universally set to 250. PIPC is disabled for 1, 2, and 4 threads, since good placement quality can be achieved with placement-driven block-Jacobi preconditioning alone. For the case of 8 threads, PIPC is applied every 3 placement iterations and each time only one correction iteration is performed. For 16 threads, PIPC needs to be applied at every placement iteration to maintain good solution quality. The numbers of PCG iterations in PIPC are set to 150 and 50, respectively, for 8 and 16 threads. Although fewer PCG iterations are performed for 16 threads, the quality can be guaranteed by its more frequent PIPC calls.

Table 15: Comparison of UTPlaceF 3.0 with PIPC Under Different Number of Threads

| Designs | 8 Threads | | | | 16 Threads | | | |
|---|---|---|---|---|---|---|---|---|
| | WL | RT | WLR | RTR | WL | RT | WLR | RTR |
| FPGA-1 | 230 | 11 | 1.013 | 0.256 | 232 | 9 | 1.020 | 0.200 |
| FPGA-2 | 448 | 19 | 1.024 | 0.261 | 453 | 15 | 1.037 | 0.207 |
| FPGA-3 | 1719 | 57 | 1.034 | 0.247 | 1674 | 46 | 1.007 | 0.201 |
| FPGA-4 | 3406 | 68 | 1.009 | 0.270 | 3379 | 51 | 1.001 | 0.204 |
| FPGA-5 | 6413 | 85 | 0.986 | 0.290 | 6549 | 64 | 1.007 | 0.217 |
| FPGA-6 | 3125 | 114 | 0.994 | 0.254 | 3203 | 91 | 1.019 | 0.203 |
| FPGA-7 | 6160 | 115 | 1.053 | 0.258 | 6507 | 90 | 1.112 | 0.201 |
| FPGA-8 | 5848 | 130 | 1.053 | 0.247 | 6151 | 103 | 1.107 | 0.197 |
| FPGA-9 | 7576 | 156 | 1.035 | 0.268 | 7341 | 114 | 1.003 | 0.197 |
| FPGA-10 | 3043 | 150 | 0.994 | 0.258 | 3036 | 116 | 0.992 | 0.200 |
| FPGA-11 | 7077 | 154 | 1.015 | 0.256 | 7287 | 113 | 1.045 | 0.189 |
| FPGA-12 | 3845 | 187 | 1.002 | 0.227 | 3897 | 144 | 1.016 | 0.175 |
| Geo. Mean | - | - | 1.017 | 0.257 | - | - | 1.030 | 0.199 |

Table 16: Runtime Breakdown of UTPlaceF 3.0 Under Different Number of Threads

| Components | 1 Threads | | 2 Threads | | 4 Threads | | 8 Threads | | 16 Threads | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Norm. RT* | RT %† | Norm. RT | RT % | Norm. RT | RT % | Norm. RT | RT % | Norm. RT | RT % |
| Solve Linear System | 0.851 | 85.1% | 0.449 | 80.6% | 0.231 | 65.6% | 0.125 | 48.5% | 0.076 | 38.0% |
| Constr. Linear System | 0.073 | 7.3% | 0.047 | 8.4% | 0.047 | 13.3% | 0.034 | 13.4% | 0.025 | 12.6% |
| Rough Legalization | 0.075 | 7.5% | 0.053 | 9.5% | 0.042 | 11.9% | 0.036 | 14.0% | 0.033 | 16.7% |
| Partition Netlists | - | - | - | - | 0.005 | 1.5% | 0.009 | 3.6% | 0.016 | 7.9% |
| PIPC | - | - | - | - | - | - | 0.033 | 12.8% | 0.034 | 16.9% |
| Others | 0.001 | 0.1% | 0.008 | 1.5% | 0.027 | 7.7% | 0.020 | 7.7% | 0.016 | 7.9% |
| Total | 1.000 | 100.0% | 0.557 | 100.0% | 0.353 | 100.0% | 0.257 | 100.0% | 0.199 | 100.0% |

* Norm. RT: Normalized runtime of each component compared to the total runtime of 1-thread execution.
† RT %: The percentage of total runtime taken by each component under different threads.

#### 4.4.2 PIPC Effectiveness Validation

The experimental results without PIPC (but with placement-driven block-Jacobi preconditioning) are presented in Table 14. We report the sHPWL (WL) in the unit of $10^3$, runtime (RT) in the unit of second, and their ratios (WLR and RTR) compared to corresponding 1-thread execution.

On average, without PIPC, UTPlaceF 3.0 achieves 1.7X, 2.8X, 4.4X, and 5.9X speedup by using 2, 4, 8, and 16 threads. With 8 and 16 threads, the wirelength degrades by 2.4% and 6.5%, respectively.

Since PIPC is not applied for the cases of 1, 2, and 4 threads as shown in Table 13, we only report the results with PIPC enabled for 8 and 16 threads in Table 15. As can be seen, PIPC effectively reduces the wirelength degradation from 2.4% and 6.5% to 1.7% and 3.0% with only a little runtime overhead. With PIPC, UTPlaceF 3.0 can still achieve 5X speedup by using 16 threads.

It is worthwhile to mention that PIPC is a general technique that can be configured for different runtime and quality trade-offs. The results shown in Table 15 is only a set of experiments to demonstrate the effectiveness of PIPC by using the configuration in Table 13.

#### 4.4.3 Runtime Analysis

Table 16 presents the runtime breakdown of UTPlaceF 3.0 under different number of threads. We divide the total runtime into components of solving linear systems, linear system construction, rough legalization, netlists partitioning, PIPC, and others. The normalized runtime (Norm. RT) and runtime percentage (RT %) are reported for each component under each thread count. As the number of threads increases, the runtime of solving linear systems reduces nearly linearly. However, the speedup of linear system construction and rough legalization saturate quickly. For the case of 16 threads, the runtime taken by each component becomes pretty much comparable.

According to the runtime breakdown, we can see that further speedup with more threads becomes difficult for the following four main reasons: 1) solving linear systems does not dominate the total runtime anymore, thus, the overall gain from it becomes limited, 2) linear system construction and rough legalization scales poorly, 3) the runtime of netlist partitioning is almost linear to the number of partitions, and 4) more PIPC would be needed to maintain good placement quality. However, it is still possible to push the runtime down to the limit by the following several improvements: 1) combine low-level (matrix-vector operations) parallelization with our partition-based framework, 2) combine the high-level parallelization scheme proposed by [33] with our parallelization for critical sortings, and 3) use parallelized partitioning tools (the one in UTPlaceF 3.0 now is single-threaded).

## 5 Future Works

In section 2, 3 and 4, I have described the major works done until now at UT. So far, my research includes general FPGA placement algorithms and methodologies for better implementation quality and efficiency under FPGA architecture constraints (e.g., clock network feasibility). In the remaining part of my Ph.D., I will focus on FPGA placement techniques that target to more specific applications.

One of the emerging applications of FPGAs is deploying trained deep neural networks (DNN) to achieve high-performance and low-power inference [60]. A DNN design typically contains many repetitive blocks for convolution computation. These repetitive blocks, in nature, are logically regular, which implies that they should be also topologically regular in the physical layout for best performance. However, no existing work that targets to this specific design flavor with the consideration of the repetitive property. I plan to explore and develop placement techniques that directly target to DNN applications for the improvement in both inference performance and deployment time.

# References


[1] Xilinx Inc., "http://www.xilinx.com," 2018.

[2] S. Sahni and T. Gonzalez, "P-complete approximation problems," *J. ACM*, vol. 23, no. 3, pp. 555–565, 1976.

[3] Intel (Altera) Corp. http://www.altera.com. Accessed: 2017-8-1.

[4] P.-H. Ho, "Interesting problems in physical synthesis," p. 131, 2017.

[5] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962.

[6] B. R. Hutchinson and G. D. Raithby, "A multigrid method based on the additive correction strategy," *Numerical Heat Transfer, Part A: Applications*, vol. 9, no. 5, pp. 511–537, 1986.

[7] V. Betz and J. Rose, "Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size," pp. 551–554, 1997.

[8] A. S. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," pp. 37–46, 1999.

[9] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh, "RPack: routability-driven packing for cluster-based FPGAs," pp. 629–634, 2001.

[10] A. Singh, G. Parthasarathy, and M. Marek-Sadowska, "Efficient circuit clustering for area and power reduction in FPGAs," vol. 7, no. 4, pp. 643–663, 2002.

[11] H. Liu and A. Akoglu, "Timing-driven Nonuniform Depopulation-based Clustering," Article 3, 2010.

[12] S. T. Rajavel and A. Akoglu, "MO-Pack: Many-objective clustering for FPGA CAD," pp. 818–823, 2011.

[13] Z. Marrakchi, H. Mrabet, and H. Mehrez, "Hierarchical FPGA clustering based on a multilevel partitioning approach to improve routability and reduce power dissapation," pp. 25–28, 2005.

[14] W. Feng, "K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint," pp. 8–15, 2012.

[15] D. T. Chen, K. Vorwerk, and A. Kennings, "Improving timing-driven FPGA packing with physical information," pp. 117–123, 2007.

[16] K. Vorwerk and A. Kennings, "An improved multi-level framework for force-directed placement," pp. 902–907, 2005.

[17] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," pp. 213–222, 1997.

[18] G. Chen and J. Cong, "Simultaneous timing driven clustering and placement for fpgas," pp. 158–167, 2004.

[19] G. Chen and J. Cong, "Simultaneous placement with clustering and duplication," vol. 11, no. 3, pp. 740–772, 2006.

[20] P. Maidee, C. Ababei, and K. Bazargan, "Fast timing-driven partitioning-based placement for island style FPGAs," pp. 598–603, 2003.

[21] P. Maidee, C. Ababei, and K. Bazargan, "Timing-driven partitioning-based placement for island style FPGAs," vol. 24, no. 3, pp. 395–406, 2005.

[22] Y. Xu and M. A. Khalid, "QPF: efficient quadratic placement for FPGAs," pp. 555–558, 2005.

[23] P. Gopalakrishnan, X. Li, and L. Pileggi, "Architecture-aware FPGA placement using metric embedding," pp. 460–465, 2006.

[24] M. Xu, G. Gréwal, and S. Areibi, "StarPlace: A new analytic method for FPGA placement," vol. 44, no. 3, pp. 192–204, 2011.

[25] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous FPGAs," pp. 143–150, 2012.

[26] T.-H. Lin, P. Banerjee, and Y.-W. Chang, "An efficient and effective analytical placer for FPGAs," pp. 10:1–10:6, 2013.

[27] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang, "Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs," pp. 647–654, 2014.

[28] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," pp. 66:1–66:7, 2016.

[29] C.-W. Pui, G. Chen, W.-K. Chow, K.-C. Lam, J. Kuang, P. Tu, H. Zhang, E. F. Young, and B. Yu, "RippleFPGA: A routability-driven placement for large-scale heterogeneous FPGAs," pp. 67:1–67:8, 2016.

[30] R. Pattison, Z. Abuowaimer, S. Areibi, G. Gréwal, and A. Vannelli, "GPlace: A congestion-aware placement tool for ultrascale FPGAs," pp. 68:1–68:7, 2016.

[31] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal, "Routability-driven fpga placement contest," pp. 139–143, 2016.

[32] T. Lin and C. C. Chu, "POLAR 2.0: An effective routability-driven placer," pp. 1–6, 2014.

[33] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: An Effective Placement Algorithm," vol. 31, no. 1, pp. 50–60, 2012.

[34] T. Lin, C. C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: Placement based on novel rough legalization and refinement," pp. 357–362, 2013.

[35] W.-H. Liu, Y.-L. Li, and C.-K. Koh, "A fast maze-free routing congestion estimator with hybrid unilateral monotonic routing," pp. 713–719, 2012.

[36] G.-J. Nam, S. Reda, C. J. Alpert, P. G. Villarrubia, and A. B. Kahng, "A fast hierarchical quadratic placement algorithm," vol. 25, no. 4, pp. 678–691, 2006.

[37] T. C. Chen, Z. W. Jiang, T. C. Hsu, H. C. Chen, and Y. W. Chang, "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," vol. 27, no. 7, pp. 1228–1240, 2008.

[38] L. Singhal, M. A. Iyer, and S. Adya, "Lsc: A large-scale consensus-based clustering algorithm for high-performance fpgas," pp. 30:1–30:6, 2017.

[39] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," 2017.

[40] Y.-C. Kuo, C.-C. Huang, S.-C. Chen, C.-H. Chiang, Y.-W. Chang, and S.-Y. Kuo, "Clock-aware placement for large-scale heterogeneous fpgas," pp. 519–526, 2017.

[41] G. Chen, C. W. Pui, W. K. Chow, K. C. Lam, J. Kuang, E. F. Y. Young, and B. Yu, "RippleFPGA: Routability-driven simultaneous packing and placement for modern FPGAs," 2017.

[42] C.-W. Pui, G. Chen, Y. Ma, E. F. Y. Young, and B. Yu, "Clock-aware ultrascale fpga placement with machine learning routability prediction," pp. 915–922, 2017.

[43] D. Hill, "Method and system for high speed detailed placement of cells within an integrated circuit design," 2002. US Patent 6,370,673.

[44] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," pp. 135–146, 2010.

[45] OpenMP 4.0. `http://www.openmp.org/`. Accessed: 2017-8-1.

[46] Xilinx Vivado Design Suite, "`https://www.xilinx.com/products/design-tools/vivado.html`," 2018.

[47] M. Tom, D. Leong, and G. Lemieux, "Un/DoPack: re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs," pp. 680–687, 2006.

[48] J. Lamoureux and S. J. E. Wilton, "On the Trade-off Between Power and Flexibility of FPGA Clock Networks," vol. 1, no. 3, pp. 13:1–13:33, 2008.

[49] S. Yang, C. Mulpuri, S. Reddy, M. Kalase, S. Dasasathyan, M. E. Dehkordi, M. Tom, and R. Aggarwal, "Clock-Aware FPGA Placement Contest," pp. 159–164, 2017.

[50] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[51] T. Lin, C. Chu, and G. Wu, "Polar 3.0: An ultrafast global placement engine," pp. 520–527, 2015.

[52] N. Viswanathan and C. C. Chu, "FastPlace: efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model," vol. 24, no. 5, pp. 722–733, 2005.

[53] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2-A fast force-directed quadratic placement approach using an accurate net model," vol. 27, no. 8, pp. 1398–1411, 2008.

[54] Intel Math Kernel Library. `https://software.intel.com/en-us/mkl`. Accessed: 2017-8-1.

[55] GNU libstdc++ Parallel Mode. `https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`. Accessed: 2017-8-1.

[56] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical mathematics*. Springer Science & Business Media, 2010.

[57] PaToH. `http://bmi.osu.edu/umit/software.html`. Accessed: 2017-8-1.

[58] G. Guennebaud, B. Jacob, *et al.*, "Eigen3." http://eigen.tuxfamily.org.

[59] ISPD 2016 Routability-Driven FPGA Placement Contest. `http://www.ispd.cc/contests/16/ispd2016_contest.html`. Accessed: 2017-8-1.

[60] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," pp. 65–74, 2017.

# 6 Publication List at UT-Austin

**Journal Papers**

[J4] **Wuxi Li**, David Z. Pan, "A New Paradigm for FPGA Placement without Explicit Packing", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2018.

[J3] Meng Li, Bei Yu, Yibo Lin, Xiaoqing Xu, **Wuxi Li**, David Z. Pan, "A Practical Split Manufacturing Framework for Trojan Prevention via Simultaneous Wire Lifting and Cell Insertion", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2018.

[J2] **Wuxi Li**, Yibo Lin, Meng Li, Shounak Dhar, David Z. Pan, "UTPlaceF 2.0: A High-Performance Clock-Aware FPGA Placement Engine", ACM Transactions on Design Automation of Electronic Systems (TODAES), 2018. (**1st-Place Award of ISPD 2017 Contest**)

[J1] **Wuxi Li**, Shounak Dhar, David Z. Pan, "UTPlaceF: A Routability-Driven FPGA Placer with Physical and Congestion Aware Packing", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2017.

**Conference Papers**

[C4] Meng Li, Bei Yu, Yibo Lin, Xiaoqing Xu, **Wuxi Li**, David Z. Pan, "A Practical Split Manufacturing Framework for Trojan Prevention via Simultaneous Wire Lifting and Cell Insertion", IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC), 2018.

[C3] **Wuxi Li**, Meng Li, Jiajun Wang, David Z. Pan, "UTPlaceF 3.0: A Parallelization Framework for Modern FPGA Global Placement", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017. (**Invited Paper**)

[C2] Wei Ye, Yibo Lin, Xiaoqing Xu, **Wuxi Li**, Yiwei Fu, Yongsheng Sun, Canhui Zhan, David Z. Pan, "Placement Mitigation Techniques for Power Grid Electromigration", IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2017.

[C1] **Wuxi Li**, Shounak Dhar, David Z. Pan, "UTPlaceF: A Routability-Driven FPGA Placer with Physical and Congestion Aware Packing", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2016. (**Invited Paper, 1st-Place Award of ISPD 2016 Contest**)

# 7 Course Work Summary

| Major Work | | | | |
|---|---|---|---|---|
| EE382M | VLSI I | Fall 2013 | Dr. Michael Orshansky | A |
| EE382N | Computer Architecture | Fall 2013 | Dr. Aater Suleman | A |
| EE382V | Optimization Issues in VLSI CAD | Fall 2013 | Dr. David Pan | A |
| EE382M | VLSI II | Spring 2014 | Dr. Jacob Abraham | A |
| EE380L | Engineer Programming Languages | Spring 2014 | Dr. Craig Chase | A |
| EE382M | Verification of Digitial Systems | Spring 2014 | Dr. Jayanta Bhadra | A |
| EE382V | VLSI Physical Design Automation | Spring 2015 | Dr. David Pan | A |
| EE382N | High-Speed Computer Arithmetic I | Fall 2015 | Dr. Earl Swartzlander | A |
| Supporting Work | | | | |
| INF385M | Database Management | Spring 2015 | Dr. Stan Gunn | A |
| INF385T | Metadata Generation/Interface for Massive Dataset | Spring 2015 | Dr. Unmil Karadkar | A |
| EE380N | Optimization in Engineering Systems | Spring 2016 | Dr. Ross Baldick | CR |
| CS383C | Numerical Analysis: Linear Algebra | Fall 2017 | Dr. George Biros | CR |