elfPlace: Electrostatics-Based Placement for Large-Scale Heterogeneous FPGAs

Yibai Meng, Wuxi Li, *Member, IEEE*, Yibo Lin, *Member, IEEE*, and David Z. Pan, *Fellow, IEEE*

***Abstract*—elfPlace is a flat nonlinear placement algorithm for large-scale heterogeneous field-programmable gate arrays (FPGAs). We adopt the analogy between placement and electrostatic systems initially proposed by ePlace and extend it to tackle heterogeneous blocks in FPGA designs. To achieve satisfiable solution quality with fast and robust numerical convergence, an augmented Lagrangian formulation together with a preconditioning technique and a normalized subgradient-based multiplier updating scheme are proposed. Besides pure-wirelength minimization, we also propose a unified instance area adjustment scheme to simultaneously optimize routability, pin density, and downstream clustering compatibility. We further propose run-to-run deterministic GPU acceleration techniques to speedup the global placement. Our experiments on the ISPD 2016 benchmark suite show that elfPlace outperforms four state-of-the-art FPGA placers UTPlaceF, RippleFPGA, GPlace3.0, and UTPlaceF-DL by 13.5%, 10.2%, 8.8%, and 7.0%, respectively, in routed wirelength with competitive runtime.**

***Index Terms*—Field programmable gate arrays, FPGA placement, GPU acceleration, nonlinear placement.**

I. Introduction

PLACEMENT is becoming ever more crucial and challenging due to the drastic evolution of the field-programmable gate array (FPGA) architecture in the past few decades. Modern FPGAs have thousands of digital signal processing (DSP) and random-access memory (RAM) blocks and millions of lookup table (LUT) and flip-flop (FF) instances. These heterogeneous resources are often exclusively scattered over discrete locations on the FPGA fabric. This complexity and heterogeneity significantly challenge the effectiveness and efficiency of modern FPGA placers, which play an important role in determining the overall FPGA implementation quality.

There are various core FPGA placement algorithms that have been proposed in the literature. Simulated-annealing approaches [1], [2] iteratively perform probabilistic swapping to progressively improve placement solutions. Despite that the global optimum can be reached theoretically, simulated-annealing approaches, in general, suffer from extremely slow convergence. Min-cut approaches [3] distribute instances by recursive netlist partitioning. Despite performing well on small designs, min-cut approaches often produce unacceptably sub-optimal solutions when the design size reaches the scale of millions. Analytical approaches, on the other hand, formulate the entire placement problem as more sophisticated continuous optimization problems. Quadratic approaches [4]–[11] approximate the placement objective using quadratic functions, while nonlinear approaches [12]–[14] use higher order ones. Compared with quadratic approaches, nonlinear approaches often achieve a better solution quality due to their even stronger expressive power.

In contrast to the enormous research endeavor spent on core placement algorithms, there are still very limited works coping with the resource heterogeneity issue in FPGA. Most existing analytical placers only treat highly discrete DSP and RAM blocks specially and eliminate the heterogeneity between LUTs and FFs by either spreading them together with adjusted areas [8], [9], [11] or simply clustering them before placement [13]. These approaches are usually highly sensitive to the heuristics applied, which could hamper the solution quality and placement robustness. A more recent work [15] proposed a multicommodity flow-based algorithm for quadratic placers to spread heterogeneous instances and it demonstrated significant improvement over previous spreading heuristics. However, due to the inherent limitation of quadratic placement, their approach still simplifies spreading as a movement-minimization problem, which cannot explicitly optimize the wirelength nor preserve the relative order among instances of different resource types.

There are also works on optimizing other placement objectives to ease the downstream clustering, legalization, and routing steps. Many works [8], [9], [11] adopted the instance inflation technique to alleviate routing congestions. Alawieh *et al.* [16] used neural networks to predict the routing congestion map at the placement stage. Li and Pan [17] further considered the impact of downstream clustering/legalization and adjusted instance areas accordingly during placement to improve the overall solution quality. However, most of the techniques were originally proposed for quadratic placers. Studies on nonlinear placement for FPGA are still lacking. Meanwhile, nonlinear placement algorithms can enable massive parallelization, opening up possibilities for GPU acceleration [18]. Existing work leverages GPU acceleration in both global placement [19]–[21] and detailed placement [22], [23].

Manuscript received August 9, 2020; revised November 10, 2020; accepted December 28, 2020. Date of publication January 21, 2021; date of current version December 23, 2021. This work was supported in part by the National Science Foundation of China under Grant 62004006, and in part by the Beijing Municipal Science and Technology Program under Grant Z201100004220007. This article was recommended by Associate Editor W. Hung. *(Corresponding author: Yibo Lin.)*

Yibai Meng and Yibo Lin are with the Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, Beijing 100871, China (e-mail: yibolin@pku.edu.cn).

Wuxi Li is with the Vivado Implementation Team, Xilinx Inc., San Jose, CA 95124 USA.

David Z. Pan is with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712 USA.

Digital Object Identifier 10.1109/TCAD.2021.3053191

In this article, we present elfPlace, a general, flat, nonlinear placement algorithm for large-scale heterogeneous FPGAs. elfPlace adopts the idea of casting placement to electrostatic systems initially proposed by ePlace family [24], [25] for application-specific integrated circuits (ASICs), and it is enhanced to tackle the FPGA heterogeneity issue in a unified and elegant way. Besides the conventional wirelength objective, elfPlace also performs routability, pin density, and clustering-aware optimizations to achieve even higher quality and smoother design closure. elfPlace is also massively parallelized with GPU to achieve a competitive runtime.

Our major contributions are summarized as follows.

1) We enhance the original ePlace algorithm [24], [25] for ASICs to deal with heterogeneous resource types in FPGAs.
2) We employ an augmented Lagrangian method (ALM), instead of the multiplier method (MM) used in ePlace, to formulate the nonlinear placement problem.
3) We propose a preconditioning technique to improve the numerical convergence given the wide spectrum of instance sizes and net degrees in FPGA designs.
4) We propose a normalized subgradient method to update density penalty multipliers, which control the spreading of different resource types in a self-adaptive manner.
5) We improve the clustering-aware area adjustment technique proposed in [17] and integrate it, together with routability and pin density optimizations, into elfPlace.
6) We demonstrate more than 7% improvement in routed wirelength, on the ISPD 2016 benchmark suite [26], over four cutting-edge placers with very competitive runtime.
7) We further explore run-to-run deterministic GPU acceleration and achieve 6.18× speedup in global placement and 2.67× speedup in the overall runtime compared with multithreaded CPU.

The remainder of this article is organized as follows. Section II introduces the background knowledge. Section III sketches the overall flow of elfPlace. Section IV describes the core placement algorithms and Section V details the routability, pin density, and clustering-aware optimizations. Section VI explains the GPU acceleration algorithm to global placement. Section VII shows the experimental results, followed by the conclusion and future work in Section VIII.

II. Preliminaries

A. FPGA Architecture

elfPlace is developed based on Xilinx UltraScale [27], which is a representative column-based FPGA architecture that has been also adopted by many other state-of-the-art commercial FPGAs (e.g., Xilinx UltraScale+ series). As shown in Fig. 1(a), each column in this architecture provides one type of logic resource among configurable logic block (CLB), DSP, and RAM. Columns of different resource types are usually unevenly interleaved over the FPGA fabric. Fig. 1(b) details the CLB structure in this architecture, where each CLB consists of eight basic logic elements (BLEs) and each BLE

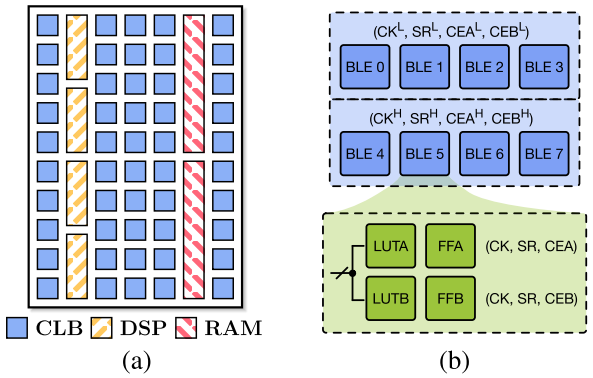

Fig. 1. (a) Simplified column-based FPGA architecture. (b) CLB structure.

further contains two LUTs and two FFs. The two LUTs in the same BLE are subject to a maximum input pin-count constraint. While FFs in the same CLB are subject to control set constraint. More specifically, as shown in Fig. 1(b), a CLB can be divided into two half CLBs, and each of which consists of four BLEs that share the same clock (CK), set/reset (SR), and clock enable (CEA/CEB) signals. Therefore, in each half CLB, FFs must share the same CK/SR and FFs with the same polarity (FFA/FFB) must further share the same CE (CEA/CEB).

B. ePlace Algorithm

ePlace [24], [25] is a leading-edge nonlinear global placement algorithm for ASICs. It approximates half-perimeter wirelength (HPWL)

$$W(\boldsymbol{x},\boldsymbol{y})=\sum_{e\in\mathcal{E}}W_e(\boldsymbol{x},\boldsymbol{y})=\sum_{e\in\mathcal{E}}\left(\max_{i,j\in e}|x_i-x_j|+\max_{i,j\in e}|y_i-y_j|\right) \tag{1}$$

using the weighted-average (WA) model [28], [29]

$$\widetilde{W}_{e_x}(\boldsymbol{x})=\frac{\sum_{i\in e}x_i\exp(x_i/\gamma)}{\sum_{i\in e}\exp(x_i/\gamma)}-\frac{\sum_{i\in e}x_i\exp(-x_i/\gamma)}{\sum_{i\in e}\exp(-x_i/\gamma)}. \tag{2}$$

Here, $\boldsymbol{x}$ and $\boldsymbol{y}$ denote the instance locations, x_i and y_i denote the location of instance i, $\mathcal{E}$ denotes the set of nets in the design, and γ is a parameter to control the modeling smoothness and accuracy. Equation (2) only gives the x-directed WA model of a net and the total wirelength cost is defined as $\widetilde{W}(\boldsymbol{x},\boldsymbol{y})=\sum_{e\in\mathcal{E}}(\widetilde{W}_{e_x}(\boldsymbol{x})+\widetilde{W}_{e_y}(\boldsymbol{y}))$.

The key innovation of ePlace is that it casts the placement density cost to the potential energy of an electrostatic system. With this transformation, each instance i is modeled as a positive charge q_i with the quantity proportional to its area. Given the notations defined in Table I, the electric force $\boldsymbol{F}_i=q_i\boldsymbol{\xi}_i=-q_i\nabla\psi_i$ will guide each black instance/positive charge i toward the direction of minimizing the total potential energy Φ

$$\Phi(\boldsymbol{x},\boldsymbol{y})=\iint_R\rho(x,y)\psi(x,y),\ (x,y)\in R. \tag{3}$$

Here, x and y are the bin indices, representing a bin in the placement region, and $\boldsymbol{x}$ and $\boldsymbol{y}$ still denote the instance locations. The unique solution of the electrostatic system is

TABLE I
NOTATIONS USED IN THE ELECTROSTATIC SYSTEM

R	A finite two-dimensional region
q_i	The electric charge quantity of charge i
$\rho(x,y)$	The electric charge density at $(x,y) \in R$
$\psi_i, \psi(x,y)$	The electric potential at charge i and $(x,y) \in R$
$\boldsymbol{\xi}_i, \boldsymbol{\xi}(x,y)$	The electric field at charge i and $(x,y) \in R$
$\Phi(\boldsymbol{x},\boldsymbol{y})$	The total electric potential energy of placement $(\boldsymbol{x},\boldsymbol{y})$

given by

$$\begin{cases} \nabla \cdot \nabla\psi(x,y) = -\nabla \cdot \boldsymbol{\xi}(x,y) = -\rho(x,y),\ (x,y) \in R & \text{(4a)} \\ \hat{\mathbf{n}} \cdot \nabla\psi(x,y) = -\hat{\mathbf{n}} \cdot \boldsymbol{\xi}(x,y) = \mathbf{0},\ (x,y) \in \partial R & \text{(4b)} \\ \iint_R \rho(x,y) = \iint_R \psi(x,y) = 0,\ (x,y) \in R & \text{(4c)} \end{cases}$$

where (4a) is Poisson's equation to correlate electric potential, electric field, and charge density, (4b) is the Neumann boundary condition (i.e., zero electric field on the boundary of R) to prevent charges from moving out of R, and (4c) neutralizes the overall electric charge and potential to ensure the solution uniqueness of (4). `ePlace` honors the placement density constraints by enforcing the electrostatic equilibrium state, where electric density is evenly distributed and $\Phi(\boldsymbol{x},\boldsymbol{y}) = 0$.

`ePlace` computes the numerical solution of (4) using spectral methods. It divides the placement region into a grid of $m \times m$ bins to construct the charge density map ρ, then the electric potential ψ and electric field $\boldsymbol{\xi} = (\xi_x, \xi_y)$ can be obtained as follows:

$$a_{u,v} = \frac{1}{m^2} \sum_{x=0}^{m-1} \sum_{y=0}^{m-1} \rho(x,y) \cos(\omega_u x) \cos(\omega_v y) \quad \text{(5a)}$$

$$\psi(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \frac{a_{u,v}}{\omega_u^2 + \omega_v^2} \cos(\omega_u x) \cos(\omega_v y) \quad \text{(5b)}$$

$$\xi_x(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \frac{a_{u,v}\omega_u}{\omega_u^2 + \omega_v^2} \sin(\omega_u x) \cos(\omega_v y) \quad \text{(5c)}$$

$$\xi_y(x,y) = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \frac{a_{u,v}\omega_v}{\omega_u^2 + \omega_v^2} \cos(\omega_u x) \sin(\omega_v y). \quad \text{(5d)}$$

Here, x and y are the bin indices, while u and v denote the frequency indices from 0 to $m-1$, and $\omega_u = (2\pi u/m)$ and $\omega_v = (2\pi v/m)$ are the frequencies of sin / cos wave functions. Equation (5) can be efficiently computed using the discrete cosine transform (DCT) and its inverse (IDCT). Note that we eliminate the DC component by setting $a_{0,0} = 0$ to ensure boundary condition (4c).

Finally, with both wirelength cost $\widetilde{W}(\boldsymbol{x},\boldsymbol{y})$ and density penalty $\Phi(\boldsymbol{x},\boldsymbol{y})$ well defined, `ePlace` then iteratively solves the following unconstrained nonlinear optimization problem using MM:

$$\min_{\boldsymbol{x},\boldsymbol{y}} f(\boldsymbol{x},\boldsymbol{y}) = \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) + \lambda\Phi(\boldsymbol{x},\boldsymbol{y}) \quad \text{(6)}$$

where λ is the density penalty multiplier to progressively enforce the density constraint.

TABLE II
NOTATIONS USED IN ELFPLACE

$\mathcal{S}$	Resource type set {LUT, FF, DSP, RAM}
$\mathcal{V}$, $\mathcal{V}_s$	Instance set and its subset with resource type s
$\mathcal{V}^{\mathrm{P}}$, $\mathcal{V}_s^{\mathrm{P}}$	Physical instance set and its subset of resource type s
$\mathcal{V}^{\mathrm{F}}$, $\mathcal{V}_s^{\mathrm{F}}$	Filler instance set and its subset of resource type s
A_i	Area of instance i
$\mathcal{B}_s$	Bin grid for resource type $s \in \mathcal{S}$
A_b^{P}	Physical instance area in bin b
C_b	Resource capacity in bin b
$\boldsymbol{\lambda}$	Density multiplier vector $(\lambda_{\mathrm{LUT}}, \lambda_{\mathrm{FF}}, \lambda_{\mathrm{DSP}}, \lambda_{\mathrm{RAM}})^T$
$\boldsymbol{\Phi}$	Potential energy vector $(\Phi_{\mathrm{LUT}}, \Phi_{\mathrm{FF}}, \Phi_{\mathrm{DSP}}, \Phi_{\mathrm{RAM}})^T$
O_s	Overflow of resource type s

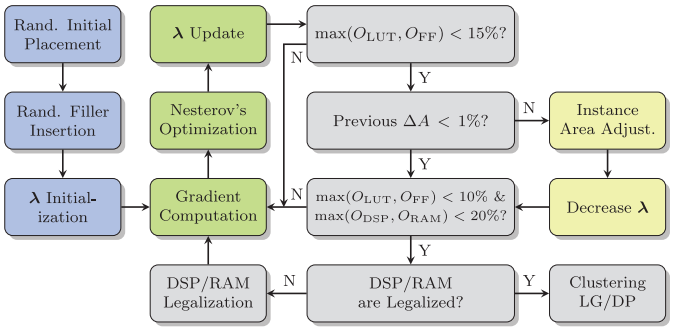

Fig. 2. The Overall flow of `elfPlace`.

III. ELFPLACE OVERVIEW

One major challenge of FPGA placement is heterogeneity handling. `elfPlace` tackles this problem by maintaining separate electrostatic systems for different resource types, including LUT, FF, DSP, and RAM. The notations used in `elfPlace` are given in Table II.

Fig. 2 illustrates the overall flow of `elfPlace`. Different from a typical initial placement that minimizes wirelength by quadratic programming, `elfPlace` starts from a random initial placement, which has been observed to achieve nearly the same quality with a considerable runtime reduction. In the random initial placement, all movable instances are first placed at the centroid of fixed pins and an extra Gaussian noise perturbation is injected with a standard deviation equal to 0.1% of the width and height of the placement region.

After the initial placement, filler instances are created and inserted independently for each resource type. Fillers are needed to pad whitespaces and produce compact placement solutions. For each resource type $s \in \mathcal{S}$ with the bin grid $\mathcal{B}_s$, its total filler area is computed as $\sum_{b \in \mathcal{B}_s} C_b - \sum_{i \in \mathcal{V}_s^P} A_i$. In our experiments, LUT/FF fillers are set to be squares with 1/8 CLB area, and DSP and RAM fillers are set to be rectangles with dimensions 1.0×2.5 and 1.0×5.0 (CLB width), respectively, based on the FPGA architecture. For each resource type s, fillers are randomly inserted based on the resource capacity distribution. More specifically, `elfPlace` first randomly distributes fillers of resource type s into bins based on the probabilities $C_b / \sum_{b \in \mathcal{B}_s} C_b$, and their final locations are then uniformly drawn within bins. By this insertion strategy, fillers can start with relatively low potential energy and it improves the convergence and stability of the later placement optimization.

Based on the initial placement and filler insertion, `elfPlace` initializes the density multiplier vector $\boldsymbol{\lambda} = (\lambda_{\text{LUT}}, \lambda_{\text{FF}}, \lambda_{\text{DSP}}, \lambda_{\text{RAM}})^T$ and then enters the core placement optimization phase. (See Section IV-C for details of how λ is initialized.) In each placement iteration, the gradient of a wirelength-density co-optimization problem is computed and fed to Nesterov's optimizer [24] to take a descent step. After that, $\boldsymbol{\lambda}$ is updated to balance the spreading efforts on different resource types and universally emphasize slightly more density penalties. When both LUT and FF overflows (O_{LUT} and O_{FF}) are reduced down to 15%, `elfPlace` adjusts instance areas (Section V) with the consideration of routability, pin density, and downstream clustering compatibility (i.e., LUT input pin constraint and FF control set constraint described in Section II-A). After the instance area adjustment, the near equilibrium electrostatic states are likely to be damaged; therefore, `elfPlace` reduces the density multipliers $\boldsymbol{\lambda}$ in this case to recover the quality again. This area adjustment step is performed each time that LUT/FF converge to $\max(O_{\text{LUT}}, O_{\text{FF}}) < 15\%$ until the total area change is less than 1%. The overflow of each resource type s is given in

$$O_s = \frac{\sum_{b \in \mathcal{B}_s} \max\left(A_b^{\text{P}} - C_b, 0\right)}{\sum_{b \in \mathcal{B}_s} A_b^{\text{P}}} \quad \forall s \in \mathcal{S}. \tag{7}$$

Once the instance area converges and the overlaps are small enough for all resource types, i.e., $\max(O_{\text{LUT}}, O_{\text{FF}}) < 10\%$ and $\max(O_{\text{DSP}}, O_{\text{RAM}}) < 20\%$, `elfPlace` legalizes and fixes DSP and RAM blocks using the minimum-cost flow approach, such as in [8] and [9]. Here, we set a larger overflow target for DSP/RAM due to their much higher discreteness compared with LUT/FF. After that, LUT/FF placements are further optimized until they both meet the overflow target again ($\max(O_{\text{LUT}}, O_{\text{FF}}) < 10\%$). Finally, `elfPlace` adopts the clustering, legalization, and detailed placement approaches proposed in [17] to produce the final legal solution.

IV. Core Placement Algorithms

A. Augmented Lagrangian Formulation

With a density constraint for each resource type modeled as a separate electrostatic system, `elfPlace` solves the minimization problem defined as follows:

$$\min_{\boldsymbol{x},\boldsymbol{y}} \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) \quad \text{s.t. } \Phi_s(\boldsymbol{x},\boldsymbol{y}) = 0 \quad \forall s \in \mathcal{S}. \tag{8}$$

However, unlike `ePlace`, which solves the density constrained placement problem using the MM given in (6), `elfPlace` uses the ALM, as shown in (9), instead

$$\min_{\boldsymbol{x},\boldsymbol{y}} f(\boldsymbol{x},\boldsymbol{y}) = \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) + \sum_{s \in \mathcal{S}} \lambda_s \left(\Phi_s(\boldsymbol{x},\boldsymbol{y}) + \frac{c_s}{2} \Phi_s(\boldsymbol{x},\boldsymbol{y})^2 \right). \tag{9}$$

Here, λ_s and Φ_s are the density multiplier and electric potential energy for each resource type $s \in \mathcal{S} = \{\text{LUT, FF, DSP, RAM}\}$, and c_s is a parameter to control the relative weight of the quadratic penalty term $\Phi_s(\boldsymbol{x},\boldsymbol{y})^2$. $(1/2)$ is to eliminate the coefficient after differentiation. One can equivalently remove $(1/2)$ and scale down c_s. Slightly different from the typical ALM formulation, where $\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})^2$ has a weight independent to $\boldsymbol{\lambda}$, the magnitude of $\boldsymbol{\Phi}(\boldsymbol{x},\boldsymbol{y})^2$ is also determined by $\boldsymbol{\lambda}$ in (9). This is to better control the overall effort on honoring the density constraints and make `elfPlace` less sensitive to the initial placement.

The ALM formulation in (9) can be viewed as a mixture of the MM and the penalty method. The motivation is that when the resource type s has high potential energy $\Phi_s(\boldsymbol{x},\boldsymbol{y})$, we want the penalty term $(c_s/2)\Phi_s(\boldsymbol{x},\boldsymbol{y})^2$ to dominate [i.e., $(c_s/2)\Phi_s(\boldsymbol{x},\boldsymbol{y})^2 \gg \Phi_s(\boldsymbol{x},\boldsymbol{y})$] and make (9) become the penalty method as shown in (10). Since in this case, the resource type s still has lots of overlaps, using the penalty method can enhance the convexity of the objective function and improve the convergence

$$\min_{\boldsymbol{x},\boldsymbol{y}} f_{\text{PM}}(\boldsymbol{x},\boldsymbol{y}) = \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) + \sum_{s \in \mathcal{S}} \lambda_s \frac{c_s}{2} \Phi_s(\boldsymbol{x},\boldsymbol{y})^2. \tag{10}$$

On the other hand, when the resource type s converges to a relatively small potential energy $\Phi_s(\boldsymbol{x},\boldsymbol{y})$, we want the $\Phi_s(\boldsymbol{x},\boldsymbol{y})$ term to dominate [i.e., $\Phi_s(\boldsymbol{x},\boldsymbol{y}) \gg (c_s/2)\Phi_s(\boldsymbol{x},\boldsymbol{y})^2$] and make (9) become MM as shown in (11). In this case, the overlaps of the resource type s are already relatively small and using the MM can continue the optimization without suffering from the ill-conditioning problem associated with the penalty method

$$\min_{\boldsymbol{x},\boldsymbol{y}} f_{\text{MM}}(\boldsymbol{x},\boldsymbol{y}) = \widetilde{W}(\boldsymbol{x},\boldsymbol{y}) + \sum_{s \in \mathcal{S}} \lambda_s \Phi_s(\boldsymbol{x},\boldsymbol{y}). \tag{11}$$

The key to achieve this penalty method and MM tradeoff is to properly set the value of c_s $\forall s \in \mathcal{S}$. We observe that regardless of the design size, the final potential energy always converges to 10^{-5}–10^{-7} of the initial one. Therefore, we define c_s as follows:

$$c_s = \frac{\beta}{\Phi_s\left(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)}\right)} \quad \forall s \in \mathcal{S} \tag{12}$$

where β is set to 2×10^3 in our experiments and $\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$ is the potential energy of the random initial placement (described in Section III). Under this setting, we will have $(c_s/2)\Phi_s(\boldsymbol{x},\boldsymbol{y})^2 = \Phi_s(\boldsymbol{x},\boldsymbol{y})$ when $\Phi_s(\boldsymbol{x},\boldsymbol{y}) = 10^{-3}\,\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$. Then, the penalty method can smoothly transit to MM at about the halfway of the final convergence. The experimental result in Section VII-B shows that our ALM formulation could improve the final routed wirelength by 1.2% compared with the original MM adopted in `ePlace`.

B. Gradient Computation and Preconditioning

The x-directed gradient of our objective function defined in (9) can be derived as shown in (13). For brevity, only the x-direction will be discussed in the rest of this section and similar conclusions are applicable to the y-direction as well

$$\begin{aligned} \frac{\partial f(\boldsymbol{x},\boldsymbol{y})}{\partial x_i} &= \frac{\partial \widetilde{W}(\boldsymbol{x},\boldsymbol{y})}{\partial x_i} + \lambda_s \left(\frac{\partial \Phi_s(\boldsymbol{x},\boldsymbol{y})}{\partial x_i} + c_s \Phi_s(\boldsymbol{x},\boldsymbol{y}) \frac{\partial \Phi_s(\boldsymbol{x},\boldsymbol{y})}{\partial x_i} \right) \\ &= \frac{\partial \widetilde{W}(\boldsymbol{x},\boldsymbol{y})}{\partial x_i} - \lambda_s q_i \xi_{x_i} (1 + c_s \Phi_s(\boldsymbol{x},\boldsymbol{y})) \quad \forall i \in \mathcal{V}_s. \end{aligned} \tag{13}$$

Although our ALM-based density penalty term is initially motivated by mathematics, there are still physical intuitions

behind its gradient. By the nature of electrostatics, the electric force $q_i\boldsymbol{\xi}_i$ on each charge i will guide the charge toward a nearby low-potential well and this is reflected by the $\lambda_s q_i \boldsymbol{\xi}_{x_i}$ term in (13). Besides, the extra $\lambda_s q_i \boldsymbol{\xi}_{x_i} c_s \Phi_s(\boldsymbol{x}, \boldsymbol{y})$ term further accelerates the charge movement for resource types with high potential energies, which often correspond to relatively large cell overlaps in the placement problem.

The gradient defined in (13) will be preconditioned before being finally fed to the optimizer. Preconditioning can make the local curvature of the objective function become nearly spherical and, hence, alleviates the ill-conditioning problem and improve the numerical convergence and stability. The most commonly used preconditioner is the inverse of the Hessian matrix $\boldsymbol{H}_f$ of the objective function f, and the preconditioned gradient $\boldsymbol{H}_f^{-1}\nabla f$, instead of the original ∇f, will be used as the (opposite of) descent direction. However, due to the scale of the placement problem and the complexity of our objective function, it is impractical to compute the exact Hessian. Instead, `elfPlace` adopts the much cheaper Jacobi preconditioner to approximate the actual Hessian.

The x-directed Jacobi preconditioner is a diagonal matrix with the ith diagonal entry equal to $(\partial^2 f/\partial x_i^2)$. By (13), we have

$$\frac{\partial^2 f(\boldsymbol{x},\boldsymbol{y})}{\partial x_i^2} = \frac{\partial^2 \widetilde{W}(\boldsymbol{x},\boldsymbol{y})}{\partial x_i^2} - \lambda_s q_i \left(\frac{\partial \xi_{x_i}}{\partial x_i}(1 + c_s \Phi(\boldsymbol{x},\boldsymbol{y})) - c_s \xi_{x_i}^2 \right). \tag{14}$$

The closed-form expression of $(\partial^2 \widetilde{W}(\boldsymbol{x},\boldsymbol{y})/\partial x_i^2)$ is too expensive to compute in practice; therefore, we approximate it using

$$\frac{\partial^2 \widetilde{W}(\boldsymbol{x},\boldsymbol{y})}{\partial x_i^2} \sim \sum_{e \in \mathcal{E}_i} \frac{1}{|e| - 1} \tag{15}$$

where $\mathcal{E}_i$ denotes the set of nets incident to instance i and $|e|$ denotes the degree of net e. The second-order derivative of the density term is even more complicated. Although the numerical solution of $([\partial \xi_{x_i}]/[\partial x_i])$ can be computed again through spectral method based on (5), we choose to only keep the $\lambda_s q_i$ term for the sake of efficiency.

Therefore, the overall x-directed second-order derivative of the objective is approximated as follows:

$$\frac{\partial^2 f(\boldsymbol{x},\boldsymbol{y})}{\partial x_i^2} \sim h_{x_i} = \max\left(\sum_{e \in \mathcal{E}_i} \frac{1}{|e| - 1} + \lambda_s q_i,\ 1 \right) \quad \forall i \in \mathcal{V}_s \tag{16}$$

where $\max(\cdot, 1)$ is to avoid extremely small h_{x_i} for filler instances, who do not have incident nets, when λ_s is very small. Finally, the preconditioned gradient

$$H_f^{-1}\nabla f(\boldsymbol{x},\boldsymbol{y}) = \left(\frac{1}{h_{x_1}} \frac{\partial f(\boldsymbol{x},\boldsymbol{y})}{\partial x_1}, \frac{1}{h_{y_1}} \frac{\partial f(\boldsymbol{x},\boldsymbol{y})}{\partial y_1}, \ldots, \right)^T \tag{17}$$

will be fed to Nesterov's optimizer [30] to iteratively update the placement solution. Since in FPGA designs, net degrees (e.g., local signal nets and global clock nets) and instance pin counts and sizes (e.g., small LUT/FF instances and large DSP/RAM blocks) can vary significantly, this wirelength preconditioner is essential to the numerical convergence of our optimization. The experimental result in Section VII-B shows that `elfPlace` can barely converge without our preconditioning technique.

C. Density Multipliers Setting

One important thing we have not yet discussed is the setting of density multipliers $\boldsymbol{\lambda}$, which controls the spreading efforts on different resource types. Since there is often heavy connectivity among different resource types, the spreading process must be capable of achieving target densities for all resource types while not ruining the natural physical clusters consisting of heterogeneous instances.

In `elfPlace`, we set the initial density multipliers $\boldsymbol{\lambda}^{(0)}$ as follows:

$$\boldsymbol{\lambda}^{(0)} = \eta \frac{\left\| \nabla \widetilde{W}(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)}) \right\|_1}{\sum_{i \in \mathcal{V}} q_i \left\| \boldsymbol{\xi}_i^{(0)} \right\|_1} (1, 1, \ldots, 1)^T \tag{18}$$

where $(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$ represent the initial placement, $\boldsymbol{\xi}_i^{(0)}$ denotes the initial electric field at instance i, η is a weighting parameter, and $\| \cdot \|_1$ denotes the $L1$-norm of a vector. In order to emphasize the wirelength optimization in early iterations, η is set to 10^{-4} in our experiments. Note that $\boldsymbol{\lambda}^{(0)}$ is an $|\mathcal{S}|$-dimensional vector, where $|\mathcal{S}|$ is the number of resource types, and by (18), we start from spreading all resource types with the same weight.

Classical optimization approaches use the subgradient method to update $\boldsymbol{\lambda}$ [31]. According to (9), the subgradient of $\boldsymbol{\lambda}$ is defined as

$$\nabla_{\text{sub}} \boldsymbol{\lambda} = \left(\ldots, \Phi_s(\boldsymbol{x},\boldsymbol{y}) + \frac{c_s}{2} \Phi_s(\boldsymbol{x},\boldsymbol{y})^2, \ldots, \right)^T, s \in \mathcal{S}. \tag{19}$$

The reason why $\nabla_{\text{sub}}\boldsymbol{\lambda}$ is called the subgradient instead of gradient is that the dual function, $l(\boldsymbol{\lambda}) = \max f(\boldsymbol{x},\boldsymbol{y})|_{\boldsymbol{\lambda}}$, associated with (9) is not smooth but piecewise linear [31].

However, in our placement problem, the potential energies of different resource types, Φ_s, can differ by order of magnitudes. The very sparse DSP/RAM blocks usually have significantly smaller total potential energies compared with LUT/FF instances. As a result, using the subgradient in (19) to guide the $\boldsymbol{\lambda}$ updating can lead to severely ill-conditioned problems. To mitigate this issue, the normalized subgradient defined in (20) is used instead in `elfPlace`

$$\begin{aligned} \widehat{\nabla}_{\text{sub}} \boldsymbol{\lambda} &= \left(\ldots, \frac{1}{\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})} \left(\Phi_s(\boldsymbol{x},\boldsymbol{y}) + \frac{c_s}{2} \Phi_s(\boldsymbol{x},\boldsymbol{y})^2 \right), \ldots, \right)^T \\ &= \left(\ldots, \widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y}) + \frac{\beta}{2} \widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y})^2, \ldots, \right)^T, s \in \mathcal{S}. \end{aligned} \tag{20}$$

Here, we use $\widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y}) = \Phi_s(\boldsymbol{x},\boldsymbol{y})/\Phi_s(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)})$ to denote the potential energy normalized by the potential energy of the initial placement and c_s is replaced by its definition given in (12). After this normalization, each $\widehat{\Phi}_s(\boldsymbol{x},\boldsymbol{y})$ is approximately upper bounded by 1, which can more accurately reflect the relative level of density violation for each resource type.

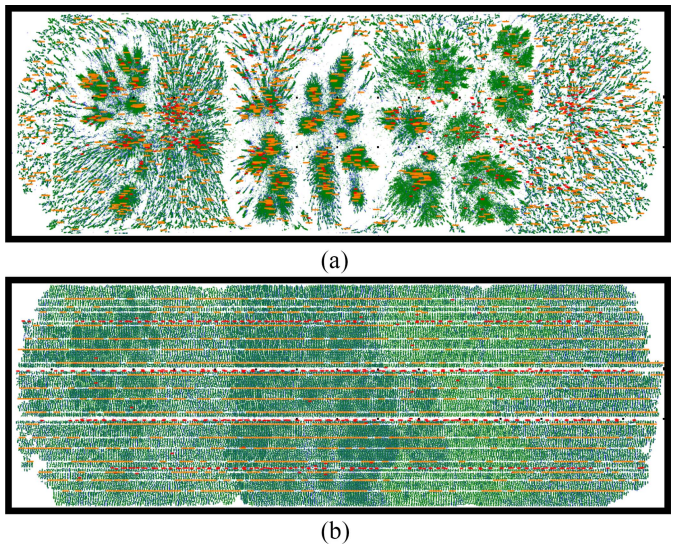

Fig. 3. Distributions of physical LUT (green), FF (blue), DSP (red), and RAM (orange) instances in (a) intermediate placement and (b) placement right before DSP/RAM legalization based on FPGA-10. Both figures are rotated by 90°.

Given $\boldsymbol{\lambda}^{(k)}$ and the step size $t^{(k)}$ at iteration k, we compute $\boldsymbol{\lambda}^{(k+1)}$ by (21) and our step size updating scheme is further presented by (22)

$$\boldsymbol{\lambda}^{(k+1)} = \boldsymbol{\lambda}^{(k)} + t^{(k)} \frac{\widehat{\nabla}_{\text{sub}} \boldsymbol{\lambda}^{(k)}}{\left\| \widehat{\nabla}_{\text{sub}} \boldsymbol{\lambda}^{(k)} \right\|_2} \tag{21}$$

$$t^{(k)} = \begin{cases} \alpha_H - 1, & \text{for } k = 0 \\ t^{(k-1)} \left(\frac{\log(\beta \|\widehat{\boldsymbol{\Phi}}^{(k)}\|_2 + 1)}{1 + \log(\beta \|\widehat{\boldsymbol{\Phi}}^{(k)}\|_2 + 1)} (\alpha_H - \alpha_{\text{L}}) + \alpha_{\text{L}} \right) & \\ \text{for } k > 0. & \end{cases} \tag{22}$$

Here, β is the same weighting parameter used in (12) and the parameter pair $(\alpha_{\text{L}}, \alpha_H)$ defines the range of increasing rate of the step size. The motivation of our step size updating scheme shown in (22) is that the quadratic density penalty term often decays much faster than the linear penalty term in our objective (9). Therefore, we increase the step size faster when the quadratic penalty term dominates (i.e., $\beta \|\widehat{\boldsymbol{\Phi}}^{(k)}\|_2 \gg 1$). As the instance overlaps become smaller, the linear penalty term will start to take over and a slower increasing rate will be used in this case. In our experiments, we set $(\alpha_{\text{L}}, \alpha_H)$ to (1.05, 1.06). It should be noted that although $1.05 \approx 1.06$, their high-order exponents can differ by order of magnitudes [e.g., $(1.06/1.05)^{500} > 100$]. Larger $\alpha_H/\alpha_{\text{L}}$ can result in faster increase of $\boldsymbol{\lambda}$, faster spreading of instances, and fewer iterations, but it may cause wirelength degradation and potential divergence. One the other hand, smaller $\alpha_H/\alpha_{\text{L}}$ can lead to more iterations and runtime overhead.

Fig. 3 illustrates the heterogeneous spreading process in elfPlace. As can be seen from Fig. 3(a), our $\boldsymbol{\lambda}$ updating scheme can greatly preserve those natural physical clusters consisting of heterogeneous instances. The placement right before DSP/RAM legalization shown in Fig. 3(b) further demonstrates the capability of elfPlace to achieve nearly overlap-free solutions even for highly discrete DSP/RAM blocks without explicit legalization.

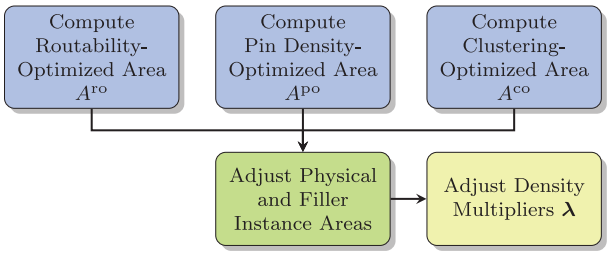

Fig. 4. Area adjustment flow in elfPlace to simultaneously optimize routability, pin density, and downstream clustering compatibility.

V. Instance Area Adjustment

Besides minimizing wirelength, elfPlace is also capable of tackling other practical issues in real-world designs, such as routability, pin density, and clustering compatibility. Routability and pin density optimizations have always been the fundamental requirements of placement to achieve routing-friendly solutions. While clustering compatibility optimization, which was recently discussed in [17], is to further consider the effect of downstream clustering (also referred as packing) early in the placement stage. It turns out that all these issues can be addressed by properly adjusting instance areas on top of our wirelength-driven placement. Therefore, in this section, we propose a unified instance area adjustment approach to simultaneously optimize all of them.

A. Adjustment Scheme

Fig. 4 sketches the algorithm flow of our instance area adjustment. In the beginning, for each physical instance i, we first compute three area terms, each of which is optimized for routability (A_i^{ro}), pin density (A_i^{po}), and clustering compatibility (A_i^{co}), respectively. Let A_i denote the area of instance i before the adjustment, we define the target area increase of each physical instance i as follows:

$$\Delta A_i = \max\left(A_i^{\text{ro}}, A_i^{\text{po}}, A_i^{\text{co}}, A_i\right) - A_i \quad \forall i \in \mathcal{V}^P. \tag{23}$$

In order to prevent the total adjusted area from exceeding the total capacity of each resource type, all ΔA_i need to be further scaled by the following factor according to the resource type s of i:

$$\tau_s = \min\left(\frac{\sum_{i \in \mathcal{V}_s^F} A_i}{\sum_{i \in \mathcal{V}_s^P} \Delta A_i}, 1 \right) \quad \forall s \in \mathcal{S} \tag{24}$$

where $\mathcal{V}_s^F$ denotes the set of filler instances for resource type s and $\sum_{i \in \mathcal{V}_s^F} A_i$ is the total filler area of resource type s before the adjustment. Basically, scaling all ΔA_i by (24) guarantees that the total increased physical instance area is no greater than the total available filler area for each resource type. The final adjusted area A_i' of each physical instance i is then given by

$$A_i' = A_i + \tau_s \Delta A_i \quad \forall i \in \mathcal{V}_s^P \quad \forall s \in \mathcal{S}. \tag{25}$$

Recall that elfPlace relies on the electrostatic neutrality to meet the density constraints $\boldsymbol{\Phi} = \mathbf{0}$; therefore, we also need

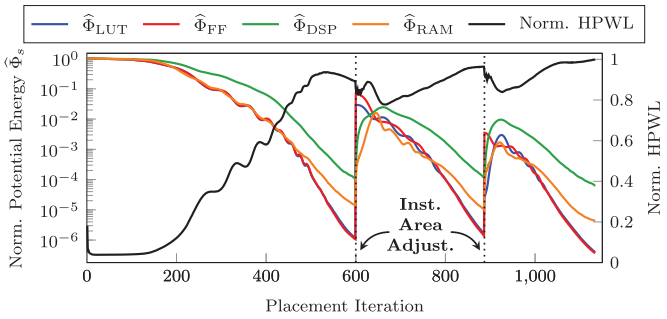

Fig. 5. Normalized potential energy $\widehat{\boldsymbol{\Phi}}$ and HPWL at different placement iterations on FPGA-10.

to downsize filler instances to keep the total positive charge quantity unchanged. The final adjusted area A'_i of each filler instance i is then defined as follows:

$$A'_i = \frac{\sum_{j\in\mathcal{V}_s} A_j - \sum_{j\in\mathcal{V}_s^P} A'_j}{\left|\mathcal{V}_s^F\right|} \quad \forall i \in \mathcal{V}_s^F \quad \forall s \in \mathcal{S}. \tag{26}$$

Our area adjustment step is physically equivalent to redistributing electric charges while preserving overall electrostatic neutrality. After this redistribution, however, the previously achieved state of near equilibrium are likely to be broken. In addition, the adjustment magnitudes and the potential energy increases can be highly uneven across different resource types (e.g., FFs can vary more than LUTs due to the control set rules). Therefore, we reset the density multipliers $\boldsymbol{\lambda}$ by (27) to adapt and recover from this perturbation

$$\boldsymbol{\lambda}' = \eta' \frac{\left\|\nabla\widetilde{W}\right\|_1}{\left\langle\left(\ldots, \sum_{i\in\mathcal{V}_s} q_i\left\|\boldsymbol{\xi}_i\right\|_1, \ldots,\right)^T, \widehat{\nabla}_{\text{sub}}\boldsymbol{\lambda}\right\rangle} \widehat{\nabla}_{\text{sub}}\boldsymbol{\lambda} \tag{27}$$

where $\langle\cdot,\cdot\rangle$ denotes the inner product of two vectors, $(\ldots, \sum_{i\in\mathcal{V}_s} q_i\|\boldsymbol{\xi}_i\|_1, \ldots,)^T$ is an $|\mathcal{S}|$-dimensional vector that contains the L1-norm of the density gradient for each resource type $s \in \mathcal{S}$, $\widehat{\nabla}_{\text{sub}}\boldsymbol{\lambda}$ denotes the normalized subgradient of $\boldsymbol{\lambda}$, as defined in (20), after the area adjustment, and η' is a weighting parameter set to 0.1 in our experiments. Equation (27) essentially redirects $\boldsymbol{\lambda}$ to its current normalized subgradient $\widehat{\nabla}_{\text{sub}}\boldsymbol{\lambda}$ with the scale determined by the gradient norm ratio between wirelength and density. In this way, both the direction and scale of the adjusted $\boldsymbol{\lambda}'$ can adapt the perturbed electrostatic system and help to better heal the placement quality. Besides, in order to smooth the placement convergence, the density multiplier step size is also adjusted by (28), where α_H is the same parameter as used in (22)

$$t' = (\alpha_H - 1)\left\|\boldsymbol{\lambda}'\right\|_2. \tag{28}$$

Fig. 5 illustrates the impact of the instance area adjustment on the placement convergence process. In this example, the adjustment is performed twice at iteration 600 and 887, where the potential energies increase sharply. By using our adaptive density multiplier and step size resetting techniques, the wirelength can be gradually healed and smoothly converges to a nearly overlap-free solution.

B. Optimized Area Computation

As we discussed in Section V-A, for each physical instance i, elfPlace computes three independent areas that are optimized for routability (A_i^{ro}), pin density (A_i^{po}), and clustering compatibility (A_i^{co}), respectively.

1) Routability-Optimized Area: In order to compute the routability-optimized areas, elfPlace first performs a RISA/RUDY-based [32], [33] routing congestion estimation. Let u_i^h and u_i^v denote the resulting horizontal and vertical routing utilizations at instance i, then we compute the routability-optimized area of each physical instance i using (29), where 2 is an empirical constant to avoid overinflation

$$A_i^{\text{ro}} = A_i \min\left(\max\left(u_i^h, u_i^v\right)^2, 2\right) \quad \forall i \in \mathcal{V}^P. \tag{29}$$

2) Pin Density-Optimized Area: Similarly, elfPlace also estimates the pin density by dividing the placement region into bins. Let c^{p} denote the unit-area pin capacity (determined by the FPGA architecture). For each instance i, if we denote its local pin density by u_i^{p} and denote its pin count as $|\mathcal{P}_i|$, then its pin density-optimized area is defined by (30), where 1.5 is an empirical constant to avoid overinflation

$$A_i^{\text{po}} = \frac{|\mathcal{P}_i|}{c^{\text{p}}} \min\left(u_i^{\text{p}}, 1.5\right) \quad \forall i \in \mathcal{V}^P. \tag{30}$$

Different from the routability-optimized area A_i^{ro}, the pin density-optimized area A_i^{po} here is independent to the current instance area A_i. This is because, compared with routing utilization, the local pin density is usually very noisy and sensitive to the placement. If A_i^{po} is self-accumulated as in (29), it can be excessively overinflated.

3) Clustering Compatibility-Optimized Area: One special challenge of flat FPGA placement is that we can barely know the correct LUT and FF areas before the actual downstream clustering solution is formed. Recall the CLB architecture described in Section II-A, if an LUT/FF is incompatible with most of its physical neighbors (e.g., violating the pin count and control set rules), then it tends to occupy a significant portion of a CLB alone. For such an instance, we intuitively should assign it a larger area.

To estimate the instance areas in a feasible clustering solution, we first assume the instance movement $(\Delta\boldsymbol{x}, \Delta\boldsymbol{y})$ during the downstream clustering/legalization approximately follows the Gaussian distribution. That is, we have $\Delta x_i \sim \mathcal{N}(0, \sigma)$ and $\Delta y_i \sim \mathcal{N}(0, \sigma)$, where σ is the assumed standard deviation of the movement. We empirically set σ to $\sqrt{10^{-5} \times |\mathcal{V}^P|}$. Then, we divide the placement region into square bins with bin length equal to σ, and for each LUT/FF instance i, we conduct the area estimation using the bin window $\mathcal{B}_i$, of size 5×5 bins, that is centered at (x_i, y_i). Let $(\mathcal{B}_i^{xl}, \mathcal{B}_i^{yl}, \mathcal{B}_i^{xh}, \mathcal{B}_i^{yh})$ denote the bounding box of the estimation window $\mathcal{B}_i$ of instance i, the expectation of any instance j falling into $\mathcal{B}_i$ then can be defined as

$$\mathrm{E}_{j\in\mathcal{B}_i} = P_\sigma\left(\mathcal{B}_i^{\text{xl}} \le x_j < \mathcal{B}_i^{\text{xh}}\right) P_\sigma\left(\mathcal{B}_i^{\text{yl}} \le y_j < \mathcal{B}_i^{\text{yh}}\right) \tag{31}$$

where $P_\sigma(a \le \mu < b)$ represents the total probability of the Gaussian distribution $\mathcal{N}(\mu, \sigma)$ in the range $[a, b)$.

For each LUT instance i, let $\mathcal{V}_i^P$ and $\overline{\mathcal{V}_i^P}$ denote the sets of LUTs that can and cannot be fitted into the same BLE with i (see Section II-A), respectively, then we define the clustering compatibility-optimized area for LUT i as follows:

$$A_i^{\text{co}} = \frac{1}{16}\frac{\sum_{j\in\mathcal{V}_i^P}\mathrm{E}_{j\in\mathcal{B}_i}}{\sum_{j\in\mathcal{V}_{\text{LUT}}^P}\mathrm{E}_{j\in\mathcal{B}_i}} + \frac{1}{8}\frac{\sum_{j\in\overline{\mathcal{V}_i^P}}\mathrm{E}_{j\in\mathcal{B}_i}}{\sum_{j\in\mathcal{V}_{\text{LUT}}^P}\mathrm{E}_{j\in\mathcal{B}_i}} \quad \forall i \in \mathcal{V}_{\text{LUT}}^P. \tag{32}$$

Equation (32) essentially is a weighted average of the compatible and incompatible expectations for i within the window $\mathcal{B}_i$. Each A_i^{co} $\forall i \in \mathcal{V}_{\text{LUT}}^P$, is in the range $[1/16, 1/8]$ based on our target architecture. The same idea is also adopted in [17] and the proposed equation (32) is its enhancement with Gaussian smoothing.

The estimation for FFs are more subtle due to the complicated control set rules (see Section II-A). For an FF instance i, let θ_i denote its control set (CK, SR, CE) and let Θ_i denote the set of control sets that have the same CK and SR with θ_i. If we use $n_{i,\theta}$ to denote the number of FFs in $\mathcal{B}_i$ with the control set θ, then the area of FF i in the tightest clustering solution formed within $\mathcal{B}_i$ can be estimated by (33), as given in [17]

$$A_i^{\text{co-disc}} = \frac{1}{2n_{i,\theta_i}}\frac{\lceil n_{i,\theta_i}/4\rceil}{\sum_{\theta\in\Theta_i}\lceil n_{i,\theta}/4\rceil}\left\lceil\frac{\sum_{\theta\in\Theta_i}\lceil n_{i,\theta}/4\rceil}{2}\right\rceil \quad \forall i \in \mathcal{V}_{\text{FF}}^P. \tag{33}$$

We omit the derivation of (33) due to the page limit. However, it still can be seen that (33) involves many ceiling operations ($\lceil\cdot\rceil$), which make $A_i^{\text{co-disc}}$ discontinuous (disc) and very sensitive to the estimation window $\mathcal{B}_i$ and the placement solution.

In `elfPlace`, the much smoother (34), instead of (33), is used as the clustering compatibility-optimized areas for FF instances

$$A_i^{\text{co}} = \frac{1}{2\mathrm{E}_{i,\theta_i}}\frac{\texttt{sdc}\left(\mathrm{E}_{i,\theta_i}, 4\right)}{\sum_{\theta\in\Theta_i}\texttt{sdc}\left(\mathrm{E}_{i,\theta}, 4\right)}\texttt{sdc}\left(\sum_{\theta\in\Theta_i}\texttt{sdc}\left(\mathrm{E}_{i,\theta}, 4\right),\ 2\right) \quad \forall i \in \mathcal{V}_{\text{FF}}^P. \tag{34}$$

It has two notable improvements over (33).

1) It replaces each FF count $n_{i,\theta}$ in (33) with the smoother expectation $\mathrm{E}_{i,\theta}$, which denotes the expected number (nonintegral in general) of FFs in window $\mathcal{B}_i$ with the control set θ.
2) It replaces each division-ceiling operation in (33) with the soft division-ceiling function $\texttt{sdc}(x, d)$ defined in

$$\texttt{sdc}(x,d) = \begin{cases} x+(1-d)\lfloor x/d\rfloor, & \text{for } x/d-\lfloor x/d\rfloor < 1/d \\ \lceil x/d\rceil, & \text{otherwise.} \end{cases} \tag{35}$$

The plots of $\texttt{sdc}(x, d)$ function with respect to (w.r.t.) x/d are illustrated in Fig. 6. It smoothes $\lceil x/d\rceil$ by linearizing the beginning $1/d$ of each sharp step. As d approaches to ∞, $\texttt{sdc}(x, d)$ behaves more like $\lceil x/d\rceil$.

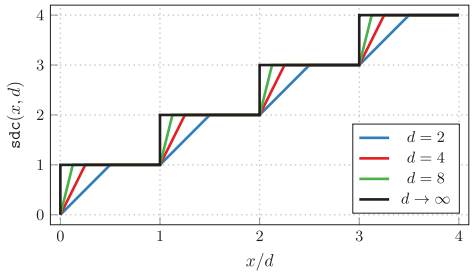

Fig. 6. Plots of the soft division-ceiling function $\texttt{sdc}(x, d)$ w.r.t. x/d.

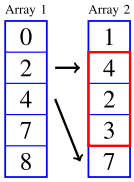

Fig. 7. Two arrays used to store the one-to-many mapping between nets and pins. Here, we show the process to retrieve the pins belonging to net 1. Array 1 stores the starting index of pins in array 2, while array 2 stores the actual pin IDs.

VI. GPU Acceleration

We observe that the majority of runtime is spent on calculating the wirelength gradient and the electrostatic field in global placement. As most of the computation tasks operate on the placement locations of resources, i.e., a high-dimensional vector, there is great potential for parallelization. Therefore, we seek to accelerate the global placement with massive parallelization on GPUs.

A. Choices of Data Structures

Most of the required data structures can be stored as simple arrays. However, processing complicated graph-like data structures on GPU may not be efficient, such as the circuit netlist, which is essentially a hypergraph with information for instances, pins, and nets. The netlist needs to be stored with mappings, between pins and instances and pins and nets. Instances-to-pins and nets-to-pins mappings are one-to-many, which requires structures like a vector of vectors. On GPU, we use two flat arrays to represent nested vectors, similar to compressed sparse row (CSR) format for storing sparse matrices, as shown in Fig. 7. In this way, we can store the netlist on GPU with simple flat arrays.

B. Wirelength Computation With Heterogeneity of Net Degree

When computing the wirelength and its gradient in parallel, we have to deal with workload imbalance caused by heterogeneous net degrees. To compute the wirelength as shown in the WA model in (2), we launch each thread to compute the gradient of pins belonging to each net, as well as each thread for an instance to sum up the gradient of its pins.

The calculation of wirelength gradient involves two rounds of summation: first within a net and then within an instance. Therefore, the most natural idea is to launch a kernel for each individual net, calculate the gradient of each pin for each net.

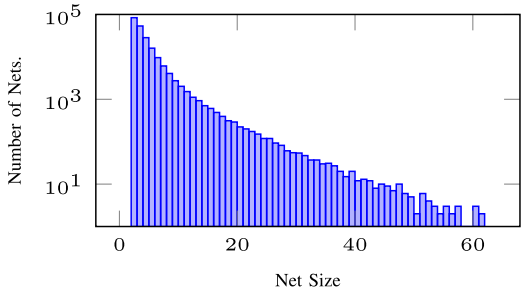

Fig. 8. Distribution of net size in FPGA-03. Note the logarithmic scale. There are also 30 nets with 2000 pins, and one net with 81 507 pins. These extreme outliers are unable to be plotted on the graph.

Then, we launch a kernel for each instance and sum up the gradient of its pins.

However, this simple implementation may end up with extremely long runtime. Take the FPGA-03 design for example, the unaccelerated version has a runtime of 0.076 s for this step, while the "accelerated" version has a runtime of 0.89 s. We observer that the long runtime is mostly due to a few extremely large nets. As we can see in Fig. 8, most of the nets have only a few pins, but one net has 200K pins (a.k.a the global clock signal), 100 times more than the second largest one. To overcome this efficiency challenge, we find it is effective to ignore those nets with large degrees, e.g., degree larger than 3000. The reason is that large nets usually end up being placed everywhere in the layout, so, in general, there is not much room to optimize its HPWL. In our experiments, we also observe almost no effects on the routed wirelength. We also investigate better scheduling by running the large-degree nets first; i.e., we sort the nets from large to small degrees. The resulting runtime is approximately the same as the unsorted case. The possible reason is that the runtime is dominated by computing the several extremely large nets, as the performance of one GPU thread is not as high as that of CPU.

C. Achieving Determinism

Another issue we encounter is the potential indeterminism from limited precisions of floating-point arithmetic when computing the density maps in parallel, which is required when computing the density gradient. The density map is stored as a 2-D array ($M \times N$) with the layout divided into $M \times N$ bins. Due to the fact that the instances may be overlapping during global placement, the density of one bin may be influenced by more than one instance. Algorithm 1 sketches the procedure to compute the density map.

To implement the operation on GPU, we compute the contribution of different instances in parallel, meaning that multiple threads may be concurrently modifying the same bins of the density map. While the possibility of data race can be eliminated by using atomic operations, the order of the operations cannot be guaranteed between different runs. Since IEEE 754 floating-point additions are not completely associative, the computed density map will be slightly different every time, making the final placement result slightly nondeterministic between different runs. Although the quality variation is very small, it can make it difficult to reproduce previously obtained results, and complicates the debugging process.

Algorithm 1: Add Instances to Density Map

```
1  t = blockIdx.x * blockDim.x + threadIdx.x;
2  if t < |V| then
3      v = V_t;    ▷ Get the instance v from thread index;
4      x_l, y_l, x_h, y_h = bottom left and top right corners of v;
5      for x = ⌊x_l/b_w⌋ to ⌈x_h/b_w⌉ do
6          for y = ⌊y_l/b_h⌋ to ⌈y_h/b_h⌉ do
7              instanceDensity = A_i/(b_w b_h) ;
               /* It is possible for other
                  instances to overlap with
                  this bin as well, so we must
                  use atomic operations.     */
8              atomicAdd(densityMap[x][y],
               instanceDensity);
9          end
10     end
11 end
```

To achieve completely deterministic computation, we leverage fixed-point numbers in the `atomicAdd` step of Algorithm 1, i.e., using a density map with fixed-point numbers instead of floating-point numbers. As integer operations are associative, the difference in order would not affect the result. When using fixed-point numbers, we need to tradeoff the precision and data range, as we only have limited budget for the decimal part and the integer part. Due to the limited number of bits, the larger data range we can represent, the less precision we can achieve, vice versa.

To achieve automatic adjustment according to different benchmarks, we developed an *adaptive* approach, estimating the upper bound of the numbers in the density map at the start of the program, shown in the following equation:

$$\rho(x, y) \leq \rho_{\max} = \frac{\sum_{i \in \mathcal{V}_s} A_i}{b_s} \tag{36}$$

where b_s is the area of an individual bin, ρ is the electric charge density defined in Table I, and $\rho_{\max}$ is the upper bound of density. Placing all the instances into one bin results in the maximum density $\rho_{\max}$. We set the decimal point to a specific location such that the integer part is sufficient to represent the upper bound numbers in the density map, and use the rest as the decimal part. We adopt 64 bits fixed-point numbers and guarantee at least 32 bits for the integer part. The bits for integer and decimal parts can be computed as follows:

$$\begin{aligned} \#\text{Bits}_{\text{int}} &= \max\left(\lceil \log_2 \rho_{\max} \rceil, 32\right) \\ \#\text{Bits}_{\text{dec}} &= 64 - \#\text{Bits}_{\text{int}}. \end{aligned} \tag{37}$$

For our problem, 32 bits for the integer part and 32 bits for the decimal part are sufficient. In general, it is unlikely to have numbers larger than 2^{64}, so we only used 64 bits. It would not be difficult to extend the scheme to 128 bits or more, should the need arise.

TABLE III
ISPD 2016 CONTEST BENCHMARKS STATISTICS

Design	#LUT	#FF	#RAM	#DSP	#Ctrl Set
FPGA-01	50K	55K	0	0	12
FPGA-02	100K	66K	100	100	121
FPGA-03	250K	170K	600	500	1281
FPGA-04	250K	172K	600	500	1281
FPGA-05	250K	174K	600	500	1281
FPGA-06	350K	352K	1000	600	2541
FPGA-07	350K	355K	1000	600	2541
FPGA-08	500K	216K	600	500	1281
FPGA-09	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	-

D. Other Acceleration Choices

Most of the computation in global placement can be accelerated using the idea described above. A few other tasks merit special treatment. To solve the Poisson equation, we implemented the spectral method with the NVIDIA's cuFFT [34] library [35]. We also implemented parallel summation reduction kernels with the CUB library [36] to reduce the runtime complexity from linear to $\mathcal{O}(\log N)$.

VII. EXPERIMENTAL RESULTS

We implement `elfPlace` in C++ and perform experiments on a Linux machine running with Intel Xeon Gold 6230 CPU (2.10 GHz and 20 physical cores) and 64-GB RAM. Careful parallelization is applied throughout the whole framework with the support of OpenMP 4.0 [37]. The ISPD 2016 FPGA placement contest benchmark suite [26] released by Xilinx is adopted to demonstrate the effectiveness and efficiency of `elfPlace`.[1] Routed wirelength reported by Xilinx Vivado v2015.4 is used to evaluate the placement quality. The characteristics of the benchmarks are listed in Table III.

A. Comparison With State-of-the-Art Placers

We compare `elfPlace` with four state-of-the-art analytical FPGA placers, namely, `UTPlaceF` [8], `RippleFPGA` [9], [38], `GPlace3.0` [11], and `UTPlaceF-NEP` [17].[2] The executables are obtained from their authors and executed on our machine. Since `UTPlaceF-NEP` and `elfPlace` support multithreading, `RippleFPGA` supports 2-thread, and `UTPlaceF` and `GPlace3.0` only support single-thread, we execute `UTPlaceF` and `GPlace3.0` with a single thread, `RippleFPGA` with two threads, and `UTPlaceF-NEP` and `elfPlace` with a single thread, ten threads, and 20 threads.

Table IV shows the comparison results. Metrics "WL" and "RT" represent the routed wirelength in thousands and runtime in seconds, while "WLR" and "RTR" represent the routed wirelength and runtime ratios normalized to the 10-threaded `elfPlace`. It can be seen that `elfPlace` achieves the best routed wirelength on 11 out of 12 designs and outperforms `UTPlaceF`, `RippleFPGA`, `GPlace3.0`, and `UTPlaceF-NEP` by, on average, 13.5%, 10.2%, 8.8%, and 7.0%, respectively. It is worthwhile to note that these wirelength improvements are fairly consistent from small designs to large ones. With only a single thread, `elfPlace` demonstrates similar runtime compared with `UTPlaceF`, `GPlace3.0`, and `UTPlaceF-NEP`. By exploiting ten threads and 20 threads, `elfPlace` achieves 2.76× and 2.90× speedup, respectively, and shows similar runtime with `RippleFPGA`. Among all the 12 designs, the speedup ratios from the 20-threaded `elfPlace` over single thread are rather consistent, varying between $2.50 \sim 3.29\times$. Meanwhile, the Vivado's routing time for `elfPlace`'s solutions is also comparable with other placers, indicating that the efforts Vivado had to make, or the potential congestion of the placed design, are similar.

We also evaluated these designs on the 372 benchmarks provided by Guelph FPGA CAD Group [39], as show in Table V. These benchmarks are split into 12 groups, each procedurally generated from the corresponding ISPD2016 benchmark, with similar resource usage and complexity. So, the performance on them can be seen as a indicator of the algorithm's robustness and adaptability. As we can see, `elfPlace` produces a comparable number of unroutable results with other placers, while achieving 8%–13% wirelength reduction and keeping its significant runtime superiority. It also processes a comparable routing time with other placers.

B. Individual Technique Validation

Table VI further validates the effectiveness of each proposed technique. The column "w/ MM in (11)" shows the results of using the MM in (11), which is adopted by `ePlace`, instead of our proposed ALM in (9). To make a fair comparison, we set the step size of the MM in a way that the MM and our ALM can converge within about the same amount of time. With this setup, our proposed ALM-based formulation can produce an average of 1.2% better routed wirelength compared with the MM-based formulation. The column "w/o Precond." shows the results without the preconditioning in (16) and it can barely converge due to the wide spectrum of instance sizes and net degrees in FPGA designs. The column "w/ `ePlace` Precond." further gives the results of replacing our preconditioner in (16) with the one proposed in `ePlace` [30]. Although the `ePlace`'s preconditioning technique can achieve similar placement quality and efficiency, it fails to converge on two benchmarks in our experiments. The column "w/ A_i^{co} in [17]" presents the results of using the clustering compatibility-optimized area proposed in [17] instead of our Gaussian and `sdc`-smoothed (32) and (34). With our smoothing techniques, `elfPlace` can converge 15% faster while maintaining essentially the same solution quality. Finally, the columns " w/o A_i^{po} Pin Adj" and "w/o A_i^{ro} Route Adj" presents the results when pin density area adjustment and routability adjustment are not performed, respectively. Routability adjustment has relatively small impacts on the placement quality, as most of the benchmarks are not very

[1] As this work focuses on core placement algorithms, the ISPD 2017 benchmark suite for clock-aware placement is not adopted in our experiments.

[2] NEP stands for no explicit packing. Li *et al.* [8] introduced a direct legalization method that does not require explicit packing.

TABLE IV
ROUTED WIRELENGTH (WL IN 10^3 SITE SIZE) COMPARISON WITH OTHER STATE-OF-THE-ART PLACERS

Design	UTPlaceF [8]					RippleFPGA [9], [38]					GPlace3.0 [11]					UTPlaceF-NEP [17]								
	WL	WLR	VRRR	1-thread RT	1-thread RTR	WL	WLR	VRRR	2-thread RT	2-thread RTR	WL	WLR	VRRR	1-thread RT	1-thread RTR	WL	WLR	VRRR	1-thread RT	1-thread RTR	10-thread RT	10-thread RTR	20-thread RT	20-thread RTR
FPGA-01	357	1.126	1.02	162	5.43	350	1.104	1.05	32	1.08	356	1.122	1.09	83	2.77	340	1.073	1.10	153	5.13	53	1.76	58	1.93
FPGA-02	642	1.105	0.97	273	6.03	682	1.175	1.05	58	1.27	644	1.109	1.03	158	3.49	653	1.124	1.17	270	5.96	94	2.08	89	1.97
FPGA-03	3215	1.122	0.93	778	7.08	3251	1.135	1.11	209	1.90	3101	1.083	1.08	587	5.34	3139	1.096	1.08	757	6.88	343	3.12	336	3.06
FPGA-04	5410	1.115	1.00	768	7.16	5492	1.132	1.08	286	2.67	5403	1.113	1.00	630	5.87	5331	1.099	1.11	758	7.07	375	3.50	349	3.25
FPGA-05	9660	1.049	1.29	973	8.76	9909	1.076	1.01	334	3.01	10507	1.141	0.91	736	6.62	10045	1.091	1.40	864	7.78	401	3.61	381	3.43
FPGA-06	6488	1.138	1.12	1649	8.22	6145	1.078	0.95	518	2.58	5820	1.021	0.96	1189	5.93	5801	1.018	1.03	1341	6.68	635	3.16	596	2.97
FPGA-07	10105	1.156	2.97	1647	8.44	9577	1.096	0.41	558	2.86	9509	1.088	0.42	1277	6.54	9356	1.071	1.28	1282	6.57	622	3.19	597	3.06
FPGA-08	7879	1.026	0.97	1642	10.45	8088	1.054	0.97	412	2.62	8126	1.059	1.03	1400	8.90	8298	1.081	1.65	477	3.03	263	1.67	273	1.73
FPGA-09	12369	1.161	1.64	2483	11.62	11376	1.068	1.20	662	3.10	11711	1.100	0.79	1848	8.65	11633	1.092	1.57	582	2.72	346	1.62	346	1.62
FPGA-10	8795	1.449	2.51	3043	13.36	6972	1.149	0.79	1002	4.40	6836	1.127	1.02	1794	7.87	6317	1.041	1.49	564	2.48	359	1.58	353	1.55
FPGA-11	10196	0.977	3.71	2044	11.48	10918	1.047	2.56	628	3.53	10260	0.984	1.13	1709	9.60	10476	1.004	2.01	552	3.10	326	1.83	309	1.74
FPGA-12	7755	1.196	1.71	2934	13.15	7240	1.116	1.16	847	3.79	7224	1.114	1.01	2263	10.14	6835	1.054	1.42	704	3.16	457	2.05	418	1.87
Norm	-	1.135	1.653	-	9.27	-	1.102	1.11	-	2.73	-	1.088	0.96	-	6.81	-	1.070	1.36	-	5.05	-	2.43	-	2.35

Design	elfPlace [40]									elfPlaceGPU						
	WL	WLR	VRRR	1-thread RT	1-thread RTR	10-thread RT	10-thread RTR	20-thread RT	20-thread RTR	WL	WLR	VRRR	fixed64 RT	fixed64 RTR	float64 RT	float64 RTR
FPGA-01	316	0.997	1.00	273	9.18	95	3.19	96	3.22	317	1.00	1.00	29	0.98	30	1.00
FPGA-02	580	0.999	0.99	498	10.99	185	4.08	175	3.86	581	1.00	1.00	51	1.11	45	1.00
FPGA-03	2862	0.999	1.00	805	7.32	294	2.68	288	2.62	2865	1.00	1.00	110	1.00	110	1.00
FPGA-04	4844	0.998	1.00	798	7.44	279	2.60	281	2.62	4853	1.00	1.00	108	1.00	107	1.00
FPGA-05	9215	1.001	1.00	955	8.60	341	3.07	316	2.84	9206	1.00	1.00	112	1.01	111	1.00
FPGA-06	5727	1.005	1.01	1189	5.93	450	2.24	432	2.15	5699	1.00	1.00	215	1.07	201	1.00
FPGA-07	8749	1.001	1.00	1198	6.14	442	2.26	421	2.16	8740	1.00	1.00	194	0.99	195	1.00
FPGA-08	7661	0.998	1.01	1290	8.21	429	2.73	392	2.49	7676	1.00	1.00	157	1.00	157	1.00
FPGA-09	10657	1.001	0.99	1454	6.81	513	2.40	495	2.32	10650	1.00	1.00	217	1.02	214	1.00
FPGA-10	6058	0.998	1.00	1531	6.72	655	2.88	613	2.69	6068	1.00	1.00	225	0.99	228	1.00
FPGA-11	10421	0.999	1.03	1472	8.27	512	2.88	469	2.63	10432	1.00	1.00	172	0.96	178	1.00
FPGA-12	6480	0.999	0.98	1860	8.34	673	3.02	618	2.77	6484	1.00	1.00	221	0.99	223	1.00
Norm	-	1.000	1.00	-	7.83	-	2.83	-	2.70	-	1.00	1.00	-	1.01	-	1.00

VRRR: Vivado routing runtime, relative to that of elfPlaceGPU's fixed point version. All other relative values (WLR, RTR) are computed relative to elfPlaceGPU's fixed point version as well.

TABLE V
ROUTED WIRELENGTH (WL IN 10^3 SITE SIZE) AND PLACEMENT RUNTIME (RT IN SECONDS) COMPARISON WITH OTHER STATE-OF-THE-ART PLACERS ON 372 GUELPH BENCHMARKS

Group	UTPlaceF [8]				RippleFPGA [9], [38]				GPlace3.0 [11]				UTPlaceF-NEP [17]					elfPlaceGPU			
	WLR	FR	VRRR	1-thread RTR	WLR	FR	VRRR	2-thread RTR	WLR	FR	VRRR	1-thread RTR	WLR	FR	FP	VRRR	20-thread RTR	WLR	FR	VRRR	fixed64 RTR
FPGA-01	1.14	0	1.03	4.90	1.12	0	1.05	0.98	1.15	0	1.09	2.49	1.08	0	0	1.11	1.56	1.00	0	1.00	1.00
FPGA-02	1.12	0	1.03	5.99	1.11	0	1.04	1.21	1.13	0	1.05	3.55	1.08	0	0	1.11	1.89	1.00	0	1.00	1.00
FPGA-03	1.12	0	1.02	6.92	1.18	0	1.10	1.91	1.10	1	1.08	5.28	1.07	0	0	1.09	2.99	1.00	0	1.00	1.00
FPGA-04	1.11	0	1.02	7.11	1.14	0	1.14	2.33	1.10	0	1.04	6.11	1.09	0	0	1.10	2.91	1.00	0	1.00	1.00
FPGA-05	1.04	8	2.80	7.86	1.07	3	3.21	3.84	1.09	1	2.38	6.25	1.09	0	0	2.27	3.10	1.00	0	1.00	1.00
FPGA-06	1.12	0	1.09	7.99	1.09	0	1.06	2.54	1.05	0	1.04	5.85	1.02	0	0	1.07	2.81	1.00	0	1.00	1.00
FPGA-07	1.11	10	1.78	8.90	1.08	1	0.54	4.02	1.06	0	1.23	6.88	1.05	1	0	2.13	3.10	1.00	3	1.00	1.00
FPGA-08	1.05	0	1.09	10.72	1.12	0	1.09	2.88	1.06	0	1.12	9.10	1.10	0	0	1.18	5.28	1.00	0	1.00	1.00
FPGA-09	1.14	16	2.19	11.55	1.11	1	1.55	3.57	1.07	0	1.35	8.27	1.10	1	10	1.83	6.32	1.00	2	1.00	1.00
FPGA-10	1.38	8	2.70	14.11	1.11	0	1.10	5.44	1.08	0	1.10	8.71	1.11	0	0	1.38	3.32	1.00	0	1.00	1.00
FPGA-11	1.05	1	2.81	11.84	1.08	1	2.18	4.25	1.04	0	1.10	9.81	1.07	0	7	1.57	8.96	1.00	0	1.00	1.00
FPGA-12	1.16	1	2.40	13.82	1.06	0	1.22	3.61	1.07	0	1.14	10.33	1.14	0	4	1.44	5.87	1.00	0	1.00	1.00
Norm/Sum	1.13	44	1.75	9.31	1.11	6	1.36	3.05	1.08	2	1.23	6.89	1.08	2	21	1.44	4.01	1.00	5	1.00	1.00

WLR: Average relative routed wirelength of benchmarks in the group. When computing the average numbers, we only consider the benchmarks that are successfully placed and routed for both elfPlace and other placers.
FR: Number of benchmarks Vivado failed to route, or were unable to finish routing within 2 hours.
FP: Number of benchmarks the placer failed to place, or were unable to finish placement within 2 hours.
VRRR: Average relative Vivado routing runtime. Similiar to WLR, only the benchmarks that are successfully placed and routed for all placers are considered.
RTR: Average relative placer runtime.

congested, except for FPGA-05. Pin density area adjustment has larger impacts on the solution quality, without which three benchmarks fail to route due to congestion.

Table VII shows the routed wirelength with and without running the detailed placement. We can see that detailed placement contributes to 1.64% average routed wirelength improvement. Combining the 7%–13% average routed wirelength improvement in Table IV, we can ascribe the benefits mostly to the proposed global placement techniques.

C. Experimental Results on GPU Acceleration

Table IV also compares the quality and total runtime between the CPU version and the GPU accelerated version, shown as elfPlaceGPU. Fig. 9 (also see Table VIII for detailed numbers) compares their global placement runtime. We can see that elfPlaceGPU can achieve an average of 6.18× speedup in global placement and 2.15–3.86× speedup in the overall runtime over elfPlace with 20 threads. Meanwhile, we compare the implementations using fixed-point numbers with run-to-run determinism and floating-point

TABLE VI
NORMALIZED ROUTED WIRELENGTH AND PLACEMENT RUNTIME COMPARISON FOR INDIVIDUAL TECHNIQUE VALIDATION

Design	w/ MM in Eq. (11)		w/o Precond.		w/ ePlace Precond.		w/ A_i^{co} in [17]		w/o A_i^{po} Pin Adj.		w/o A_i^{ro} Route Adj.		elfPlace	
	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR	WLR	RTR
FPGA-01	1.021	1.02	1.009	1.01	1.006	1.02	1.004	1.12	1.129	0.91	1.000	0.96	1.00	1.00
FPGA-02	1.010	0.97	*	*	*	*	1.001	1.16	0.996	0.95	1.000	0.97	1.00	1.00
FPGA-03	1.009	1.03	*	*	0.995	1.03	0.998	1.23	1.014	1.11	1.000	0.99	1.00	1.00
FPGA-04	1.046	0.94	*	*	*	*	1.002	1.16	1.011	1.12	1.002	1.10	1.00	1.00
FPGA-05	1.026	1.05	*	*	1.002	1.03	0.996	1.07	x	1.03	t	0.81	1.00	1.00
FPGA-06	1.008	1.01	*	*	1.030	0.98	1.004	1.25	1.102	1.10	1.000	0.99	1.00	1.00
FPGA-07	1.001	1.04	*	*	0.988	1.04	0.986	1.20	x	1.16	0.998	1.01	1.00	1.00
FPGA-08	0.991	1.01	*	*	0.996	1.01	0.999	1.14	0.974	1.05	1.000	1.04	1.00	1.00
FPGA-09	1.003	1.01	*	*	0.996	1.03	1.002	1.12	1.036	1.06	0.999	1.00	1.00	1.00
FPGA-10	1.006	1.01	*	*	1.000	1.01	0.996	1.03	x	1.03	1.000	0.99	1.00	1.00
FPGA-11	1.011	0.98	*	*	1.002	1.05	1.009	1.15	1.018	1.04	0.999	0.99	1.00	1.00
FPGA-12	1.017	1.02	*	*	0.995	1.04	1.003	1.14	1.121	1.18	1.000	0.99	1.00	1.00
Norm.	1.012	1.01	1.009	1.01	1.001	1.03	1.000	1.15	1.045	1.06	1.000	0.99	1.00	1.00

* Placement fails to converge.
x Vivado fails to route.
t Vivado router times out.

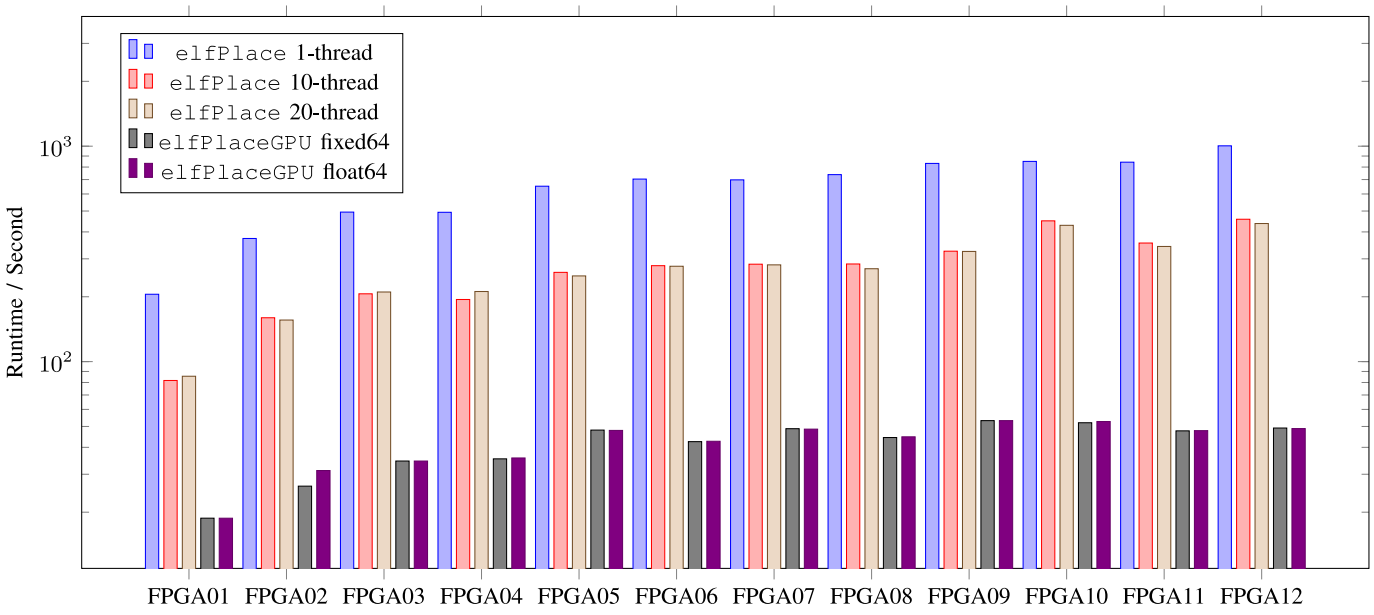

Fig. 9. Comparison of global placement runtime between elfPlace and elfPlaceGPU. We use "fixed64" and "float64" to denote whether using fixed-point numbers or floating-point numbers to compute the density map in Section VI-C.

TABLE VII
COMPARISON OF ROUTED WIRELENGTH W/ AND W/O DETAILED PLACEMENT

Design	WL w. DP	WL w.o. DP	WL Improv.
FPGA-01	326022	316156	3.03%
FPGA-02	601581	580361	3.53%
FPGA-03	2927299	2862003	2.23%
FPGA-04	4906663	4843997	1.28%
FPGA-05	9193812	9215136	-0.23%
FPGA-06	5856876	5727477	2.21%
FPGA-07	8846150	8748886	1.10%
FPGA-08	7714006	7660886	0.69%
FPGA-09	10766509	10657177	1.02%
FPGA-10	6242673	6058331	2.95%
FPGA-11	10475356	10421193	0.52%
FPGA-12	6571599	6479682	1.40%
Norm	-	-	1.64%

numbers without such determinism. It can be seen that our 64-bit fixed-point implementation cannot only guarantee run-to-run determinism but also bring no quality degradation or runtime overhead, compared with the 64-bit floating-point implementation.

D. Runtime Breakdown

Fig. 10 shows the runtime breakdown of the 20-threaded elfPlace [40] based on FPGA-11, as well as that of elfPlaceGPU. Following is the makeup of the three major steps of the algorithm.

1) *Global Placement:* Compute wirelength gradient $\nabla\widetilde{W}$; compute density gradient; update parameters between iterations; and area adjustment.
2) *Legalization:* Clustering and legalization.
3) Detailed Placement.

For the CPU version, the most time-consuming part is to compute the wirelength gradient $\nabla\widetilde{W}$, which takes 33.0% of the total runtime. The density gradient computation is relatively efficient and it consumes 13.3% of the total runtime on constructing the density maps $\boldsymbol{\rho}$, computing the electric

TABLE VIII
COMPARISON OF GLOBAL PLACEMENT RUNTIME IN SECONDS BETWEEN ELFPLACE AND ELFPLACEGPU. WE USE "FIXED64" AND "FLOAT64" TO DENOTE WHETHER USING FIXED-POINT NUMBERS OR FLOATING-POINT NUMBERS TO COMPUTE THE DENSITY MAP IN SECTION VI-C

Design	elfPlace			elfPlaceGPU	
	1-thread	10-thread	20-thread	fixed64	float64
FPGA01	205	82	86	19	19
FPGA02	373	160	156	26	31
FPGA03	494	206	210	35	35
FPGA04	493	194	212	35	36
FPGA05	652	260	250	48	48
FPGA06	704	279	277	43	43
FPGA07	698	283	281	49	49
FPGA08	738	284	270	44	45
FPGA09	832	326	325	53	53
FPGA10	850	450	429	52	53
FPGA11	843	355	342	48	48
FPGA12	1004	458	438	49	49

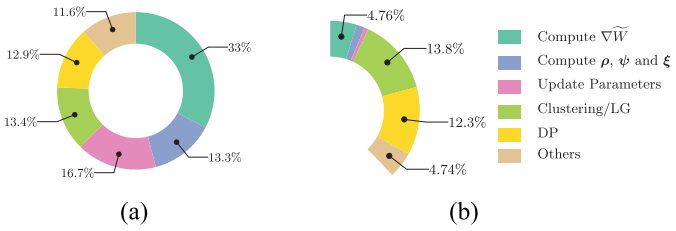

Fig. 10. Runtime Breakdown of FPGA-11. (a) elfPlace with 20 threads. (b) elfPlaceGPU with 64-bit fixed-point numbers.

potential $\boldsymbol{\psi}$ and electric field $\boldsymbol{\xi}$ by (5). The parameter updating, which involves the computation of wirelength, overflow, potential energy $\boldsymbol{\Phi}$, etc., takes 16.7% of the total runtime. The clustering, legalization, and detailed placement algorithms adopted from [17] consume total 26.3% of the runtime. While the remaining 11.6% of the runtime is spent on parsing, placement initialization, and the rest of runtime-insignificant tasks. Actually, this part is also partially accelerated (drop to 4.74% in the GPU version) as placement initialization needs to evaluate wirelength and density overflow according to Section IV.

As for the GPU accelerated version, the wirelength gradient and density gradient computation are no longer the runtime bottlenecks. Legalization and detailed placement become the new bottlenecks, which require further optimizations in the future.

VIII. CONCLUSION

In this article, we have presented elfPlace, a general, flat, nonlinear placement algorithm for large-scale heterogeneous FPGAs. elfPlace resolves the traditional FPGA heterogeneity issue by casting the density constraints of heterogeneous resource types to separate but unified electrostatic systems. An augmented Lagrangian formulation together with a preconditioning technique and a normalized subgradient-based multiplier updating scheme are proposed to achieve satisfiable solution quality with fast and robust numerical convergence. Besides pure-wirelength minimization, elfPlace is also capable of optimizing routability, pin density, and downstream clustering compatibility based on a unified instance area adjustment scheme. Our experiments showed that elfPlace significantly outperforms four state-of-the-art placers in routed wirelength with competitive runtime. In the future, we plan to incorporate timing optimization into elfPlace framework.

ACKNOWLEDGMENT

The authors would like to thank Dr. Gengjie Chen and Prof. Evangeline F. Y. Young for providing the binary of RippleFPGA and Dr. Ziad Abuowaimer, Prof. Shawki Areibi, and Prof. Gary Grewal for providing the binary of GPlace3.0.

REFERENCES

[1] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. FPL*, 1997, pp. 213–222.
[2] G. Chen and J. Cong, "Simultaneous placement with clustering and duplication," *ACM Trans. Design Autom. Electron. Syst.*, vol. 11, no. 3, pp. 740–772, 2006.
[3] P. Maidee, C. Ababei, and K. Bazargan, "Timing-driven partitioning-based placement for island style FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 3, pp. 395–406, Mar. 2005.
[4] Y. Xu and M. A. Khalid, "QPF: Efficient quadratic placement for FPGAs," in *Proc. FPL*, 2005, pp. 555–558.
[5] P. Gopalakrishnan, X. Li, and L. Pileggi, "Architecture-aware FPGA placement using metric embedding," in *Proc. DAC*, 2006, pp. 460–465.
[6] M. Xu, G. Gréwal, and S. Areibi, "StarPlace: A new analytic method for FPGA placement," *Integr. VLSI J.*, vol. 44, no. 3, pp. 192–204, 2011.
[7] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous FPGAs," in *Proc. FPL*, 2012, pp. 143–150.
[8] W. Li, S. Dhar, and D. Z. Pan, "UTPlaceF: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 4, pp. 869–882, Apr. 2018.
[9] G. Chen *et al.*, "RippleFPGA: Routability-driven simultaneous packing and placement for modern FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 10, pp. 2022–2035, Oct. 2018.
[10] W. Li, Y. Lin, M. Li, S. Dhar, and D. Z. Pan, "UTPlaceF 2.0: A high-performance clock-aware FPGA placement engine," *ACM Trans. Design Autom. Electron. Syst.*, vol. 23, no. 4, p. 42, 2018.
[11] Z. Abuowaimer *et al.*, "GPlace3.0: Routability-driven analytic placer for UltraScale FPGA architectures," *ACM Trans. Design Autom. Electron. Syst.*, vol. 23, no. 5, pp. 1–66, 2018.
[12] T.-H. Lin, P. Banerjee, and Y.-W. Chang, "An efficient and effective analytical placer for FPGAs," in *Proc. DAC*, 2013, p. 10.
[13] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang, "Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs," in *Proc. ICCAD*, 2014, pp. 647–654.
[14] Y.-C. Kuo *et al.*, "Clock-aware placement for large-scale heterogeneous FPGAs," in *Proc. ICCAD*, 2017, pp. 519–526.
[15] N. K. Darav, A. Kennings, K. Vorwerk, and A. Kundu, "Multi-commodity flow-based spreading in a commercial analytic placer," in *Proc. FPGA*, 2019, pp. 122–131.
[16] M. B. Alawieh, W. Li, Y. Lin, L. Singhal, M. A. Iyer, and D. Z. Pan, "High-definition routing congestion prediction for large-scale FPGAs," in *Proc. 25th Asia South Pac. Design Autom. Conf. (ASP-DAC)*, 2020, pp. 26–31.
[17] W. Li and D. Z. Pan, "A new paradigm for FPGA placement without explicit packing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 11, pp. 2113–2126, Nov. 2019.
[18] Y. Lin, "GPU acceleration in VLSI back-end design: Overview and case studies," in *Proc. 39th Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 1–4. [Online]. Available: https://doi.org/10.1145/3400302.3415765
[19] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Proc. DAC*, 2019, p. 117.
[20] Y. Lin, D. Z. Pan, H. Ren, and B. Khailany, "DreamPlace 2.0: Open-source GPU-accelerated global and detailed placement for large-scale VLSI designs," in *Proc. China Semicond. Technol. Int. Conf. (CSTIC)*, 2020, pp. 1–4.

[21] J. Gu, Z. Jiang, Y. Lin, and D. Z. Pan, "DreamPlace 3.0: Multi-electrostatics based robust VLSI placement with region constraints," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 1–9.
[22] S. Dhar and D. Z. Pan, "GDP: GPU accelerated detailed placement," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2018, pp. 1–7.
[23] Y. Lin *et al.*, "ABCDPlace: Accelerated batch-based concurrent detailed placement on multithreaded cpus and GPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 5083–5096, Dec. 2020.
[24] J. Lu *et al.*, "ePlace-MS: Electrostatics-based placement for mixed-size circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 5, pp. 685–698, May 2015.
[25] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "RePlAce: Advancing solution quality and routability validation in global placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 9, pp. 1717–1730, Sep. 2019.
[26] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal, "Routability-driven FPGA placement contest," in *Proc. ISPD*, 2016, pp. 139–143.
[27] (2019). *Xilinx Inc.* [Online]. Available: http://www.xilinx.com
[28] M.-K. Hsu, Y.-W. Chang, and V. Balabanov, "TSV-aware analytical placement for 3D IC designs," in *Proc. DAC*, 2011, pp. 664–669.
[29] M.-K. Hsu, V. Balabanov, and Y.-W. Chang, "TSV-aware analytical placement for 3-D IC designs based on a novel weighted-average wire-length model," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 4, pp. 497–509, Apr. 2013.
[30] J. Lu *et al.*, "ePlace: Electrostatics-based placement using fast Fourier transform and Nesterov's method," *ACM Trans. Design Autom. Electron. Syst.*, vol. 20, no. 2, p. 17, 2015.
[31] C. Lemaréchal, "Lagrangian relaxation," in *Computational Combinatorial Optimization*. Berlin, Germany: Springer, 2001, pp. 112–156.
[32] C.-L. E. Cheng, "RISA: Accurate and efficient placement routability modeling," in *Proc. ICCAD*, 1994, pp. 690–695.
[33] P. Spindler and F. M. Johannes, "Fast and accurate routing demand estimation for efficient routability-driven placement," in *Proc. DATE*, 2007, pp. 1226–1231.
[34] Nvidia. *cuFFT*. Accessed: Jan. 6, 2020. [Online]. Available: https://developer.nvidia.com/cufft
[35] Y. Lin *et al.*, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Jun. 22, 2020, doi: 10.1109/TCAD.2020.3003843.
[36] NVIDIA Research. *CUB*. Accessed: Jan. 6, 2020. [Online]. Available: https://nvlabs.github.io/cub/index.html
[37] (2019). *OpenMP 4.0*. Accessed: Apr. 1, 2019. [Online]. Available: http://www.openmp.org/
[38] C.-W. Pui, G. Chen, Y. Ma, E. F. Y. Young, and B. Yu, "Clock-aware ultrascale FPGA placement with machine learning routability prediction," in *Proc. ICCAD*, 2017, pp. 915–922.
[39] D. Maarouff, A. Shamli, T. Martin, G. Grewal, and S. Areibi, "A deep-learning framework for predicting congestion during FPGA placement," in *Proc. 30th Int. Conf. Field Program. Logic Appl. (FPL)*, 2020, pp. 138–144.
[40] W. Li, Y. Lin, and D. Z. Pan, "elfPlace: Electrostatics-based placement for large-scale heterogeneous FPGAs," in *Proc. ICCAD*, 2019, pp. 1–8.

Yibai Meng received the B.S. degree in electrical engineering from Peking University, Beijing, China, in 2020.

He is currently working as a Research Assistant with the Center for Energy-Efficient Computing and Applications, Department of EECS, Peking University. His current research interests include VLSI placement and GPU acceleration for VLSI design.

Wuxi Li (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2013, and the M.S. and Ph.D. degrees in computer engineering from the University of Texas at Austin, Austin, TX, USA, in 2015 and 2019, respectively.

He is currently a Staff Software Engineer with the Vivado Implementation Team, Xilinx, San Jose, CA, USA, where he is primarily working on the physical synthesis field.

Dr. Li has received the Best Paper Award at DAC 2019, the Silver Medal in ACM Student Research Contest at ICCAD 2018, and the first-place awards in the FPGA placement contests of ISPD 2016 and 2017.

Yibo Lin (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiaotong University, Shanghai, China, in 2013, and the Ph.D. degree from the Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX, USA, in 2018.

He is currently an Assistant Professor with the Computer Science Department, Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.

Dr. Lin has received four Best Paper Awards at premier venues (ISPD 2020, DAC 2019, VLSI Integration 2018, and SPIE 2016). He has also served in the Technical Program Committees of many major conferences, including ICCAD, ICCD, ISPD, and DAC.

David Z. Pan (Fellow, IEEE) received the B.S. degree from Peking University, Beijing, China, in 1992, and the M.S. and Ph.D. degrees from the University of California at Los Angeles (UCLA), Los Angeles, CA, USA, in 1998 and 2000, respectively.

From 2000 to 2003, he was a Research Staff Member with IBM T. J. Watson Research Center, Ossining, NY, USA. He is currently an Engineering Foundation Professor with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA. He has published over 360 journal articles and refereed conference papers, and is the holder of eight U.S. patents. His research interests include cross-layer nanometer IC design for manufacturability, reliability, security, machine learning and hardware acceleration, design/CAD for analog/mixed-signal designs, and emerging technologies.

Dr. Pan has received a number of prestigious awards for his research contributions, including the SRC Technical Excellence Award in 2013, the DAC Top 10 Author in Fifth Decade, the DAC Prolific Author Award, the ASP-DAC Frequently Cited Author Award, the 18 Best Paper Awards at premier venues (ASPDAC 2020, DAC 2019, GLSVLSI 2018, VLSI Integration 2018, HOST 2017, SPIE 2016, ISPD 2014, ICCAD 2013, ASPDAC 2012, ISPD 2011, IBM Research 2010 Pat Goldberg Memorial Best Paper Award, ASPDAC 2010, DATE 2009, ICICDT 2009, and SRC Techcon in 1998, 2007, 2012, and 2015), the 15 additional Best Paper Award finalists, Communications of the ACM Research Highlights in 2014, the UT Austin RAISE Faculty Excellence Award in 2014, and many international CAD contest awards. He has served as a Senior Associate Editor for *ACM Transactions on Design Automation of Electronic Systems*, and an Associate Editor for IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART I: REGULAR PAPERS, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: EXPRESS BRIEFS, IEEE DESIGN & TEST, *Science China Information Sciences*, *Journal of Computer Science and Technology*, and IEEE CAS SOCIETY NEWSLETTER. He has served in the Executive and Program Committees of many major conferences, including DAC, ICCAD, ASPDAC, and ISPD. He is the Program Chair of ASPDAC 2017 and ICCAD 2018, the Tutorial Chair of DAC 2014, and the General Chair of ISPD 2008. He is a Fellow of SPIE.