Chapter 9
Deep Learning Framework for Placement

Yibo Lin, Zizheng Guo, and Jing Mai

9.1 Introduction

Placement is an important stage in modern VLSI design automation [1]. It determines the locations of millions of instances, optimizing for multiple objectives such as wirelength, routability, timing, etc. Its performance and efficiency can significantly impact the design closure and turnaround time of the backend design flow. Thus, both industry and academia have been exploring efficient and high-performance placement engines.

The increasing design scale and complexity challenge placement algorithms in both efficiency and performance. We need to deal with tens of millions of instances and optimize for multiple objectives, considering complicated design constraints from both high-level architecture and low-level manufacturing technology such as region constraints and design rules. As modern placement algorithms follow iterative procedures in optimization, complicated objectives and constraints with large problem sizes can result in low computing efficiency and slow convergence.

State-of-the-art placement engines usually include wirelength-driven analytical placement kernels, which can be categorized into quadratic placement and nonlinear placement. The kernel placement can be extended to consider other objectives and constraints. Quadratic placement formulates the wirelength optimization problem into a quadratic programming [2–6]. As simply minimizing the wirelength can cause overlaps between instances, quadratic placement adopts iterative quadratic programming and rough legalization to spread instances in the layout. Nonlinear placement formulates a nonlinear wirelength objective, subjecting to a density constraint for overlap minimization [7–14]. By relaxing the density constraint into

Y. Lin (✉) · Z. Guo · J. Mai
Peking University, Beijing, China
e-mail: yibolin@pku.edu.cn; gzz@pku.edu.cn; magic3007@pku.edu.cn

H. Ren, J. Hu (eds.), *Machine Learning Applications in Electronic Design Automation*, https://doi.org/10.1007/978-3-031-13074-8_9

the objective and gradually increasing the overlap penalty, we can spread the instances with minimum wirelength. Many commercial tools like Cadence Innovus and Synopsys IC Compiler adopt nonlinear placement algorithms for ASIC [15, 16], so we focus on nonlinear placement in this book chapter.

Nonlinear placement algorithms may need thousands of gradient descent iterations for convergence, which is computationally expensive. The literature has investigated multi-threading on CPUs using partitioning [17–19]. The speedup ratio typically saturates at 5× with 2–6% quality degradation. GPU acceleration has been explored to accelerate a placement algorithm consisting of clustering, declustering, and nonlinear optimization [10]. Around 15× speedup has been reported with less than 1% quality degradation. The speedup ratio is mostly limited by the sequential clustering and declustering steps in the algorithm. With the recent advances in nonlinear placement [13, 14] and deep learning [20], Lin et al. establish an analogy between the nonlinear placement problem and the neural network training problem [21]. With such an analogy, deep learning frameworks/toolkits like `PyTorch` [22] can be adopted to develop placement algorithms with native support to GPU acceleration and highly optimized kernel operators. Eventually, 30–40× speedup can be achieved for the nonlinear placement kernel. In this book chapter, we target this line of studies [21, 23–27] and survey how deep learning frameworks help placement development with high efficiency and performance. We also introduce how to customize kernel operators for further speedup [28] as well as how to handle cutting-edge objectives and constraints with both conventional and machine learning techniques in such a framework [21, 25, 26].

9.2 DL Analogy for the Kernel Placement Problem

Modern placement consists of three steps: global placement (GP), legalization (LG), and detailed placement (DP). Global placement determines the rough locations of instances by relaxing the placement problem into continuous optimization. It can achieve roughly legal solutions with small amount of overlaps. Legalization removes the remaining overlaps and satisfies all design rules. Detailed placement refines the results by local perturbation to further improve the solution quality. The performance of global placement is critical to the eventual design closure.

The kernel problem of global placement is to solve a wirelength minimization problem subjecting to density constraints [11]:

$$\min_{\mathbf{x},\mathbf{y}} \quad \sum_{e \in E} \mathrm{WL}(e; \mathbf{x}, \mathbf{y}), \tag{9.1a}$$

$$\text{s.t.} \quad d(\mathbf{x}, \mathbf{y}) \leq d_t, \tag{9.1b}$$

where $\mathrm{WL}(\cdot;\cdot)$ denotes the wirelength cost function that takes any net instance e and returns the wirelength, $d(\cdot)$ denotes the density of a location in the layout, and

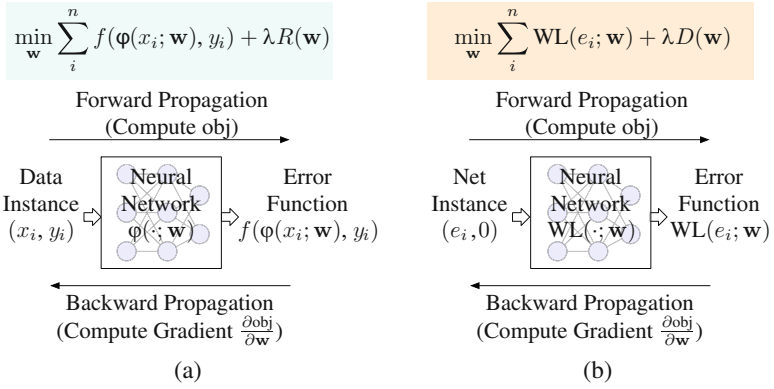


Fig. 9.1 Analogy between neural network training and analytical placement [21]. (**a**) Train a neural network for weights $\mathbf{w}$. (**b**) Solve a placement, for instance, locations $\mathbf{w} = (\mathbf{x}, \mathbf{y})$

d_t is a given target density. The target of solving this problem is to spread instances in the layout with minimum wirelength.

As the density constraints are non-convex and nonlinear, it is difficult to solve the constrained optimization directly. The widely adopted nonlinear placement algorithm relaxes the constraints into the objective and formulates an unconstrained nonlinear optimization:

$$\min_{\mathbf{x},\mathbf{y}} \quad \left(\sum_{e \in E} \text{WL}(e;\mathbf{x},\mathbf{y})\right) + \lambda D(\mathbf{x},\mathbf{y}), \tag{9.2}$$

where $D(\cdot)$ is the density penalty to spread cells out in the layout. We satisfy the density constraints by gradually increasing the density weight λ.

Since both solving the global placement and training a neural network are essentially solving a nonlinear optimization problem, we examine the underlying similarity between these two problems [21]. We first review the neural network training process, as shown in Fig. 9.1a. Given a neural network with trainable weights, we feed data samples to the network and compute error functions (or loss functions) at the output. The target of neural network training is to optimize a nonlinear objective consisting of two terms, the total error term and the regularization term. The total error term is related to both the data samples and the weights of the network, while the regularization term, adopted to avoid overfitting, is usually only related to the weights of the network. By iterative forward propagation to compute the objective and backward propagation to compute the gradient, we can find the best weights that minimize the objective.

If we write the placement objective in the same format as neural network training, as shown in Fig. 9.1b, we can observe that the wirelength term can correspond to the total error term in neural network training and the density term can correspond to the regularization term. More specifically, we can view each net instance as a data

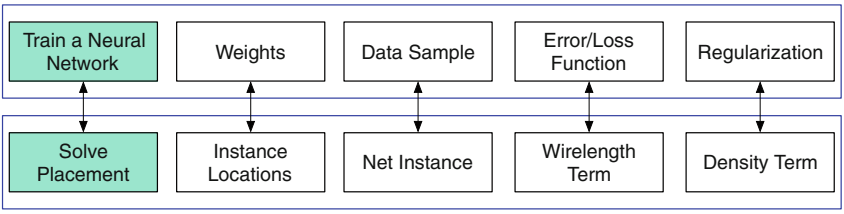


Fig. 9.2 Summary of analogy between neural network training and placement [21]

sample and the locations of instances as trainable weights. Then, we can construct a conceptual network to compute the wirelength of a net at the output. In this way, the total error term in the objective of neural network training becomes equivalent to the total wirelength term in the placement objective. The density term is only related to the locations of instances, which is similar to the regularization term. Figure 9.2 summarizes the analogy between the two problems.

By casting the placement problem into neural network training, we can solve the placement problem following the procedure of neural network training, with iterative forward and backward propagation. We can then leverage highly optimized deep learning frameworks to develop placement engines. Note that with this analogy, each time when we solve a placement problem, we essentially perform neural network training once. In other words, there is no testing phase like that in machine learning applications.

Figure 9.3 illustrates the software architecture of the placement developed with deep learning frameworks [21, 24]. Deep learning frameworks like `TensorFlow` and `PyTorch` consist of three stacks, i.e., low-level operators (OPs), automatic gradient derivation, and optimization engines. The low-level OPs are usually implemented and optimized on both CPU and GPU. The frameworks also provide APIs to extend the existing set of OPs so that we can customize the OPs for wirelength and density computation. The other parts of the frameworks are usually implemented in Python, which enables convenient development and customization of optimization engines. Thus, we can easily ensemble placement flows with low-level OPs, automatic gradient derivation, and optimization engines in deep learning frameworks.

9.3 Speedup Kernel Operators

`DREAMPlace` adopts the `ePlace` algorithm where each instance is analogous to an electric particle in an electrostatics system, which is summarized in Fig. 9.4. In this analogy, instance density corresponds to charge density, density penalty corresponds to electric potential energy, and density gradient corresponds to electric field. We can rewrite the objective in Eq. (9.2) as the following:

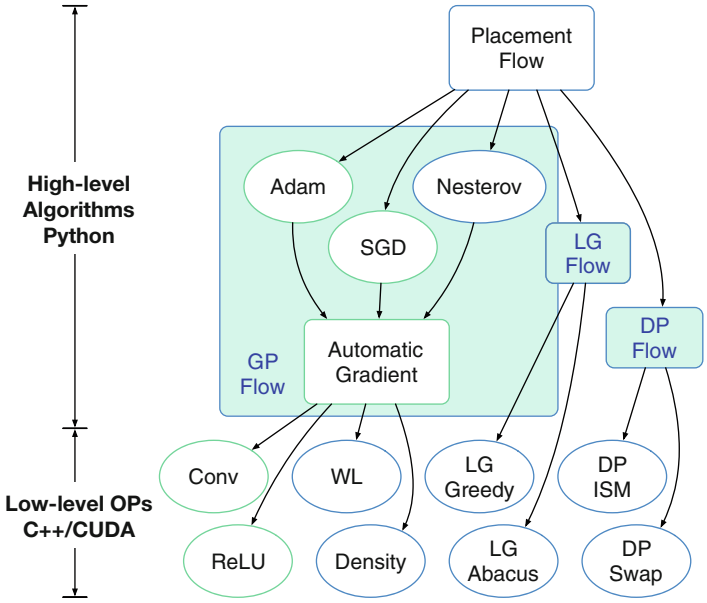


Fig. 9.3 Software architecture consisting of low-level operators written in C++/CUDA and high-level algorithms written in Python [24]. In the low-level operators, "Conv" and "ReLU" denote the existing convolution and activation OPs in the deep learning toolkit, respectively; "WL" denotes the custom OP to compute wirelength term in the placement objective; "Density" denotes the custom OP to compute the density term; "LG Greedy" and "LG Abacus" are two custom OPs for legalization algorithms leveraging a greedy approach [11] and a row-based dynamic programming method [29], respectively; "DP ISM" and "DP Swap" are two custom OPs for detailed placement algorithms leveraging independent set matching [11] and global swap [30], respectively. In the high-level algorithms, placement flows, i.e., GP flow for global placement, LG flow for legalization, and DP flow for detailed placement, can be ensembled using low-level operators with the automatic gradient derivation component and optimizers such as Adam [31], stochastic gradient descent (SGD), and Nesterov's accelerated gradient descent method

$$\min_{\mathbf{x},\mathbf{y}} \left(\sum_{e \in E} \mathrm{WL}(e; \mathbf{x}, \mathbf{y}) \right) + \lambda \Phi(\mathbf{x}, \mathbf{y}), \tag{9.3}$$

where $\Phi(\cdot)$ is the electric potential energy. The electric potential energy can be computed by solving a Poisson's equation from the instance density $\rho(x, y)$ by spectral methods. Given an $M \times M$ grid of bins and $w_u = \frac{2\pi u}{M}$ and $w_v = \frac{2\pi v}{M}$ with $u = 0, 1, \ldots, M-1, v = 0, 1, \ldots, M-1$, the solution can be computed as the following [13]:

$$a_{u,v} = \frac{1}{M^2} \sum_{x=0}^{M-1} \sum_{y=0}^{M-1} \rho(x, y) \cos(w_u x) \cos(w_v y), \tag{9.4a}$$

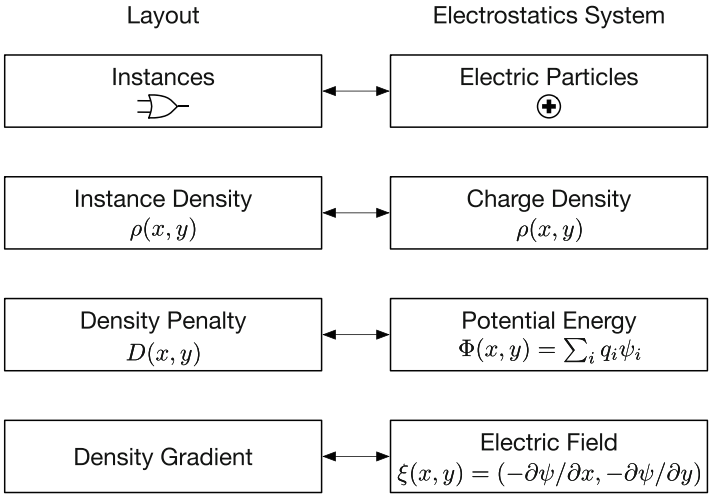


Fig. 9.4 An analogy between a layout and an electrostatics system [13, 32]

$$\psi_{\mathrm{DCT}}(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v}}{w_u^2 + w_v^2} \cos(w_u x) \cos(w_v y), \tag{9.4b}$$

$$\xi_{\mathrm{DSCT}}^{X}(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v} w_u}{w_u^2 + w_v^2} \sin(w_u x) \cos(w_v y), \tag{9.4c}$$

$$\xi_{\mathrm{DCST}}^{Y}(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v} w_v}{w_u^2 + w_v^2} \cos(w_u x) \sin(w_v y), \tag{9.4d}$$

where ψ_{DCT} denotes the numerical solution of the potential function and ξ_{DSCT}^{X} and ξ_{DCST}^{Y} denote the solution of the electric field in horizontal and vertical directions, respectively. Eq. (9.4) requires discrete cosine transformation (DCT) and its variations to solve efficiently.

`DREAMPlace` also adopts the weighted-average wirelength (WAWL) as a smooth approximation to the half-perimeter wirelength (HPWL) [33, 34]:

$$\mathrm{WA}_e = \frac{\sum_{i \in e} x_i e^{\frac{x_i}{\gamma}}}{\sum_{i \in e} e^{\frac{x_i}{\gamma}}} - \frac{\sum_{i \in e} x_i e^{-\frac{x_i}{\gamma}}}{\sum_{i \in e} e^{-\frac{x_i}{\gamma}}}, \tag{9.5}$$

where γ is a parameter to control the smoothness and accuracy. The smaller γ is, the more accurate it is to approximate HPWL, but the less smooth.

The efficiency of the wirelength and density operators is critical to the overall efficiency of placement, as the placement algorithm needs to iteratively perform

Table 9.1 Notations

Notation	Description	Notation	Description
V	Set of cells	E	Set of nets
P	Set of pins	B	Set of bins
x_e^+	$\max_{i \in e} x_i, \forall e \in E$	x_e^-	$\min_{i \in e} x_i, \forall e \in E$
a_i^+	$e^{\frac{x_i - x_e^+}{\gamma}}, \forall i \in e, e \in E$	a_i^-	$e^{-\frac{x_i - x_e^-}{\gamma}}, \forall i \in e, e \in E$
b_e^+	$\sum_{i \in e} a_i^+, \forall e \in E$	b_e^-	$\sum_{i \in e} a_i^-, \forall e \in E$
c_e^+	$\sum_{i \in e} x_i a_i^+, \forall e \in E$	c_e^-	$\sum_{i \in e} x_i a_i^-, \forall e \in E$
$\mathbf{x}^+$	$\{x_e^+\}, \forall e \in E$	$\mathbf{x}^-$	$\{x_e^-\}, \forall e \in E$
$\mathbf{a}^+$	$\{a_i^+\}, \forall i \in P$	$\mathbf{a}^-$	$\{a_i^-\}, \forall i \in P$
$\mathbf{b}^+$	$\{b_e^+\}, \forall e \in E$	$\mathbf{b}^-$	$\{b_e^-\}, \forall e \in E$
$\mathbf{c}^+$	$\{c_e^+\}, \forall e \in E$	$\mathbf{c}^-$	$\{c_e^-\}, \forall e \in E$

gradient descent on Eq. (9.3). In the following subsections, we introduce the optimized kernels for these operators.

9.3.1 Wirelength

When implementing the wirelength operators, `DREAMPlace` has developed several strategies to remedy the latent numerical problems, to compare the efficiency of different parallelism schemes, to profile the performance bottlenecks, and to further squeeze the GPU performance via operator fusions.

The direct implementation of Eq. (9.5) can potentially cause numerical overflow problems. Thus, they convert $e^{\frac{x_i}{\gamma}}$ to $e^{\frac{x_i - max_{j \in e} x_j}{\gamma}}$ and $e^{-\frac{x_i}{\gamma}}$ to $e^{-\frac{x_i - min_{j \in e} x_j}{\gamma}}$ in an equivalent manner. With the annotations in Table 9.1, the wirelength gradient w.r.t. a pin location can be written as

$$\frac{\partial \mathrm{WL}_e}{\partial x_i} = \frac{(1 + \frac{x_i}{\gamma}) b_e^+ - \frac{1}{\gamma} c_e^+}{(b_e^+)^2} \cdot a_i^+ - \frac{(1 - \frac{x_i}{\gamma}) b_e^- + \frac{1}{\gamma} c_e^-}{(b_e^-)^2} \cdot a_i^-. \tag{9.6}$$

`DREAMPlace` proposes three parallelism schemes and discusses their pros and cons. The first one is the naive *net-by-net* parallelism scheme that allocates one thread for each net. The speedup of this scheme, however, remains limited because the maximum number of allocatable threads is $|E|$, and the heterogeneity of net degrees also leads to imbalanced workload for each thread.

The second one is the *pin-by-pin* scheme. Figure 9.5 illustrates the dependency graph for WA wirelength forward and backward based on Eq. (9.6). Because of the local nature of $\boldsymbol{a}^{\pm}$, $\boldsymbol{b}^{\pm}$, and $\boldsymbol{c}^{\pm}$ at pin level, a straightforward implementation of this pin-level parallelism is to compute $\boldsymbol{a}^{\pm}$, $\boldsymbol{b}^{\pm}$, and $\boldsymbol{c}^{\pm}$ in separate CUDA kernels by using multiple CUDA streams. This method has the probability to outperform the

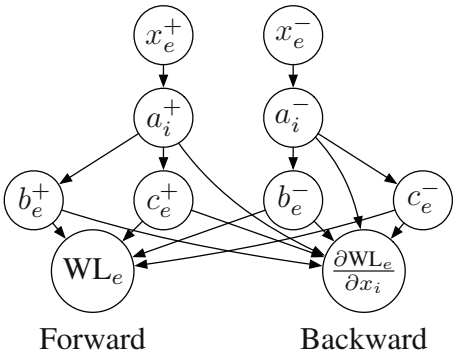


Fig. 9.5 Forward and backward dependency graph for weighted-average wirelength

net-by-net scheme, because the number of pins $|P|$ is much larger than the number of nets $|E|$. However, the pin-level parallelism still has drawbacks such as expensive CUDA streams, frequent kernel synchronization, and heavy global memory access hindering performance improvements.

Based on the elaborated profiling, Lin et al. further point out that the runtime performance is essentially memory bounded. In other words, the major runtime bottleneck lies in frequent writing to intermediate variables $\boldsymbol{x}^{\pm}$, $\boldsymbol{a}^{\pm}$, $\boldsymbol{b}^{\pm}$, and $\boldsymbol{c}^{\pm}$, so they propose the third scheme that can remove all the intermediate variables by merging the forward and backward functions, as shown in Algorithm 9.1. Instead of storing $\boldsymbol{x}^{\pm}$, $\boldsymbol{a}^{\pm}$, $\boldsymbol{b}^{\pm}$, and $\boldsymbol{c}^{\pm}$ in global memory, they only create local variables in the kernel function and directly compute the wirelength for each net and the gradient for each pin. Experimental results show that this *operator fusion* scheme can significantly alleviate the memory pressure and eventually achieve 3.7× speedup over the first scheme and 1.8× speedup over the second scheme with `float32`.

9.3.2 Density Accumulation

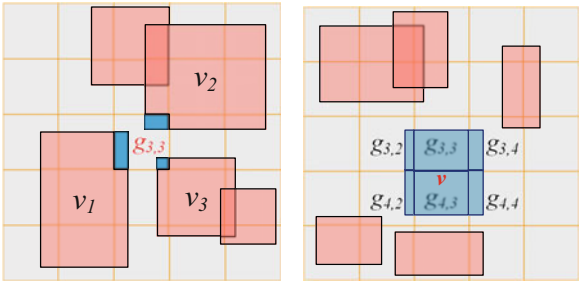
The density operator requires to compute density distribution ρ in the layout (Eq. (9.4)), which is a special case of the density accumulation operation. Density accumulation is a general operation widely used in placement and routing. It can be used for characterizing the density distributions of rectangular shapes on an $M \times N$ grid system. There are two typical variations of density accumulation as shown in Fig. 9.6: the *forward* accumulation for computing the density map inside a set of rectangles and the *backward* one to sum up the weights to the rectangles from a predefined density map. Their definitions are presented as follows.

Problem 1 (Forward Density Accumulation) *Given a set of instances V, an $M \times N$ grid system G, and the weight w_v^{inst} of each instance v, compute the density map ρ^{grid} on the grid system. Density $\rho_{x,y}^{grid}$ for each grid $g_{x,y} \in G$ is defined as follows [27]:*

Algorithm 9.1 Wirelength forward and backward merged [21]

Require: A set of nets E, a set of pins P, and pin locations x;
Ensure: Wirelength cost and gradient;

function FORWARD_BACKWARD(E, P, x)
  for each thread $0 \leq t < |E|$ **do** ▷ WL_e, $\frac{\partial \mathrm{WL}_e}{\partial x_p}$ kernel
    Define e as the net corresponds to thread t;
    $x_e^+ \leftarrow \max_{p \in e} x_p$; ▷ $x_e^\pm$ are local in the kernel
    $x_e^- \leftarrow \min_{p \in e} x_p$;
    $b_e^\pm \leftarrow 0, c_e^\pm \leftarrow 0$; ▷ $b_e^\pm, c_e^\pm$ are local in the kernel
    $\mathrm{WL}_e \leftarrow 0$; ▷ WL_e is in the global memory
    for each pin $p \in e$ **do**
      $a_p^\pm \leftarrow e^{\pm \frac{x_p - x_e^\pm}{\gamma}}$; ▷ $a_p^\pm$ is local in the loop
      $b_e^\pm \leftarrow b_e^\pm + a_p^\pm$;
      $c_e^\pm \leftarrow c_e^\pm + x_p \cdot a_p^\pm$;
    end for
    $\mathrm{WL}_e \leftarrow \frac{\mathbf{c^+}}{\mathbf{b^+}} - \frac{\mathbf{c^-}}{\mathbf{b^-}}$;
    for each pin $p \in e$ **do**
      $a_p^\pm \leftarrow e^{\pm \frac{x_p - x_e^\pm}{\gamma}}$; ▷ Compute $a_p^\pm$ again
      Compute $\frac{\partial \mathrm{WL}_e}{\partial x_p}$; ▷ $\frac{\partial \mathrm{WL}_e}{\partial x_p}$ is in the global memory
    end for
  end for
  return reduce($\{\mathrm{WL}_e\}$), $\{\frac{\partial \mathrm{WL}_e}{\partial x_p}\}, \forall p \in P, e \in E$;
end function



(a) (b)

Fig. 9.6 An illustration on (**a**) forward and (**b**) backward density accumulation [27]

$$\rho_{x,y}^{grid} = \sum_{v \in V} w_v^{inst} \times \frac{OA(Box_v, g_{x,y})}{area_{x,y}^{grid}}, \quad \forall g_{x,y} \in G, \tag{9.7}$$

where w_v^{inst} *is the weight of instance* v, $area_{x,y}^{grid}$ *is the area of the grid* $g_{x,y}$, *and* $OA(Box_v, g_{x,y})$ *is the overlapping area between the bounding box of instance* v *and grid* $g_{x,y}$.

Problem 2 (Backward Density Accumulation) *Given a set of instances* V, *an* $M \times N$ *grid system* G, *and the weight* $w_{x,y}^{grid}$ *of each grid* $g_{x,y}$, *compute the*

density array ρ^{inst} *for all instances. Density* ρ_v^{inst} *for each instance* v *is defined as follows [27]:*

$$\rho_v^{inst} = \sum_{g_{x,y} \in G} w_{x,y}^{grid} \times \frac{OA(Box_v, g_{x,y})}{area_v^{inst}}, \forall v \in V, \tag{9.8}$$

where $w_{x,y}^{grid}$ *is the weight of grid* $g_{x,y}$, $area_v^{inst}$ *is the area of instance* v, *and* $OA(Box_v, g_{x,y})$ *is the overlapping area between the bounding box of instance* v *and grid* $g_{x,y}$.

Density accumulation is widely used in many placers such as `POLAR` [6] and `ePlace` series [32, 35]. A generalized density accumulation with bell-shaped functions is also used in `NTUplace` series [11, 12]. In `DREAMPlace`, density accumulation is used to evaluate and optimize the electric potential term $\Phi(\mathbf{x}, \mathbf{y})$ in Eq. (9.3). Specifically, the cell density is first distributed through a forward accumulation process to yield the density distribution of the grid system $\rho(x, y)$ in Eq. (9.4a). Then after DCT, the computed electric field $\xi_{\text{DSCT}}^X(x, y)$ and $\xi_{\text{DSCT}}^Y(x, y)$ is accumulated by cells through their bounding boxes to yield their moving direction through a backward accumulation process. It has been shown in `DREAMPlace` that the density computation for every iteration of gradient descent takes up to 60% runtime on designs with millions of cells, especially when macro placement is considered where cells have large sizes [21, 27].

Existing works mostly focus on the forward density accumulation through parallelization on CPU [36, 37] and GPU [21, 36]. Lin et al. [36] explores efficient CPU atomic primitives and reproducible GPU kernels to solve the race condition of forward accumulation in parallel computing scenarios. Gessler et al. [37] allocate thread local copies of partial density maps to alleviate synchronization burden while at the cost of larger memory footprint. `DREAMPlace` assigns multiple threads for updating each shape on GPU [21]. All these experiments are based on a standard cell placement flow, where the size of each cell is comparable to the grid size. However, the techniques mentioned before cannot handle much larger rectangles covering many grids, which may come from net bounding boxes in routability modeling or macro placement, as the number of primitive operations is correlated to how many grids a shape covers.

To overcome the challenges in accelerating density accumulation, Guo et al. [27] propose a generic CPU/GPU algorithm for both forward and backward accumulation. They decompose the problem into two phases: a constant-time density collection phase for each instance and a linear-time prefix sum phase. They have a linear runtime complexity that does not depend on the size of the instance bounding boxes.

As both the forward and the backward algorithms in [27] require to compute two-dimensional (2D) prefix sum on a matrix A, we first introduce it as a building block to the algorithms. The result matrix P of the 2D prefix sum has the same dimension as the input matrix A, with element $P_{i,j}$ equals to the sum of all values in A which are above it or on left of it. Each element in matrix P can be written as

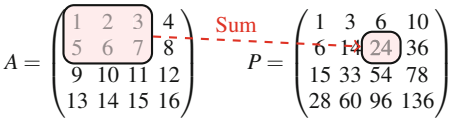


Fig. 9.7 An example of 2D prefix sum on a 4 × 4 matrix [27]

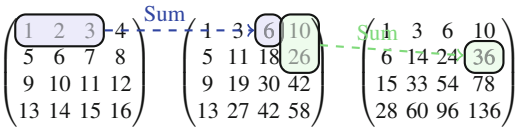


Fig. 9.8 A row sum followed by a column sum is equivalent to a 2D prefix sum [27]

Algorithm 9.2 `compute2DPrefixSum`(A) [27]

```
1: m, n ← A.size;
2: P ← M × N zero matrix
3: for i = 1 to M do
4:     for j = 1 to N do
5:         P_{i,j} ← P_{i,j−1} + A_{i,j};                ▷ Define P_{i,0} = 0
6:     end for
7: end for
8: for j = 1 to N do
9:     for i = 1 to M do
10:        P_{i,j} ← P_{i−1,j} + P_{i,j};                ▷ Define P_{0,j} = 0
11:    end for
12: end for
13: return P;
```

$$P_{i,j} = \sum_{x=1}^{i} \sum_{y=1}^{j} A_{x,y}. \tag{9.9}$$

Figure 9.7 shows an example of 2D prefix sum on a 4 × 4 matrix, where $i = 1, 2, \ldots, M$, $j = 1, 2, \ldots, N$, $A \in \mathcal{R}^{M \times N}$, and $P \in \mathcal{R}^{M \times N}$. Figure 9.8 shows that 2D prefix sum can be computed by performing one-dimensional (1D) prefix sum along rows and then along columns. A detailed algorithm for computing the 2D prefix sum is presented in Algorithm 9.2.

Based on the 2D prefix sum algorithm, the forward and backward accumulation can be made much more efficient by transforming the operations into equivalent forms. Specifically, in a forward accumulation, each instance is transformed into four bottom-right instances (i.e., instances at the bottom-right corner of the grid system), as shown in Fig. 9.9. It is straightforward to deduce the equivalence between the increment/decrements on the 4 bottom-right instances and the increment on the original rectangle. It turns out that the bottom-right instances are easier to handle in forward accumulation. Guo et al. propose an equivalent way to increase the values on a bottom-right subregion of a matrix, as illustrated in Fig. 9.10, by firstly incrementing a single value in a temporary matrix and then computing the 2D

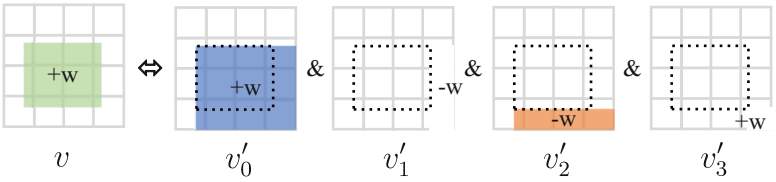


Fig. 9.9 An instance v is decomposed to four bottom-right instances v'_0, v'_1, v'_2, v'_3. The increment w on instance v is equivalent to $v'_0 += w$, $v'_1 -= w$, $v'_2 -= w$, $v'_3 += w$ [27]

Fig. 9.10 If we add a value 1 to $D_{2,2}$ and let P denote the 2D prefix sum of D, we will get that value propagated to the bottom-right region in P

$$D = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad P = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

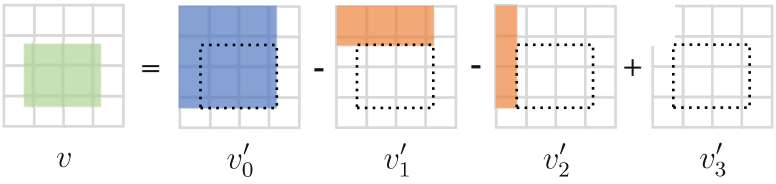


Fig. 9.11 A instance v can be decomposed to four top-left instances: v'_0, v'_1, v'_2, and v'_3 [27]

prefix sum. This method has the advantage that multiple bottom-right increments can be batched to the same temporary matrix leveraging the linearity of the 2D prefix sum. This directly yields a two-phase algorithm consisting of constant-time density collections for instances and a linear-time 2D prefix sum on the resulting matrix. These algorithms do not depend on the sizes of instances as it only needs to increment a constant amount on matrix elements.

The backward accumulation adopts a similar analysis. Different from the forward algorithm, the first step of the backward algorithm is to compute the prefix sum, and then density collection is performed on the prefix sum matrix. Each instance in backward accumulation is decomposed into four independent top-left instances, as shown in Fig. 9.11. Similar ideas can be seen in image processing literature like Crow et al. [38]. The sum of a top-left instance is just the corresponding matrix element in the precomputed prefix sum matrix and can be retrieved in $O(1)$ time.

In placement, bounding boxes in density accumulation may adopt non-integer coordinates. The weights need to be assigned proportional to the proportion of the grids within the bounding box. To deal with such scenarios, they further decompose each top-left or bottom-right instance to four equivalent instances with integer coordinates. For bottom-right instances used in the forward accumulation, this is illustrated in Fig. 9.12, and the top-left instances are processed similarly.

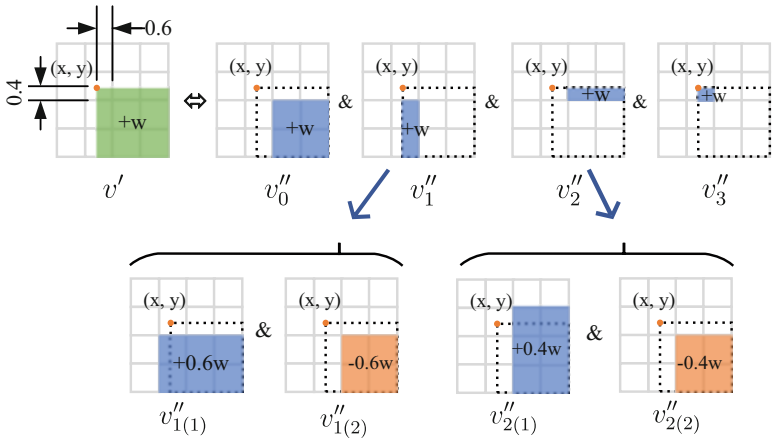


Fig. 9.12 An increment w on a bottom-right instance v' is split into four instances v''_0, v''_1, v''_2, v''_3. Of them, v''_0 is itself a bottom-right submatrix increment; v''_1 and v''_2 can be each split into two bottom-right submatrix increments; v''_3 consists of a single grid and can be handled individually [27]

The accelerated algorithms for the forward and backward accumulation can be parallelized using GPU. The 2D prefix sum is parallelized by first completing a batched 1D prefix sum at all rows, and then another batched 1D prefix sum at columns. The density collection phase of both forward and backward density accumulation consists of computation tasks for different instances, and these tasks are mostly independent so they can be parallelized. However, as one grid may be simultaneously updated by multiple threads in the forward accumulation, this algorithm needs the `atomicAdd` primitive to avoid data race. Guo et al. [27] have implemented these CPU/GPU operators in the `DREAMPlace` framework. According to their experiments, the accelerated algorithms can achieve up to 22× speedup on CPU and up to 64× speedup on GPU compared with naive versions due to balanced workload.

9.3.3 *Discrete Cosine Transformation*

Given the density distribution computed in Sect. 9.3.2, we still need to compute electric potential and fields according to Eq. (9.4), which can be broken down into discrete cosine transformation and its variations like the following [21]:

$$a_{u,v} = \text{DCT}(\text{DCT}(\rho)^T)^T, \tag{9.10a}$$

$$\psi_{\text{DCT}} = \text{IDCT}\left(\text{IDCT}\left(\left\{\frac{a_{u,v}}{w_u^2 + w_v^2}\right\}\right)^T\right)^T, \tag{9.10b}$$

$$\xi^X_{\text{DSCT}} = \text{IDXST}(\text{IDCT}\left(\left\{\frac{a_{u,v}w_u}{w_u^2+w_v^2}\right\}\right)^T)^T, \tag{9.10c}$$

$$\xi^Y_{\text{DCST}} = \text{IDCT}(\text{IDXST}\left(\left\{\frac{a_{u,v}w_v}{w_u^2+w_v^2}\right\}\right)^T)^T, \tag{9.10d}$$

where $(\cdot)^T$ denotes matrix transposition. As 2D DCT, IDCT, and IDXST transformations can be computed by applying the corresponding 1D transformations first to columns and then to rows, respectively, so we illustrate the 1D transformations for simplicity.

1D DCT and IDCT for a length-N sequence x can be written as

$$\text{DCT}(\{x_n\})_k = \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right), \tag{9.11a}$$

$$\text{IDCT}(\{x_n\})_k = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi}{N}n\left(k+\frac{1}{2}\right)\right), \tag{9.11b}$$

where $k = 0, 1, \ldots, N-1$. IDXST can be further derived as

$$\text{IDXST}(\{x_n\})_k = \sum_{n=0}^{N-1} x_n \sin\left(\frac{\pi}{N}n\left(k+\frac{1}{2}\right)\right), \tag{9.12a}$$

$$= (-1)^k \sum_{n=0}^{N-1} x_n(-1)^k \sin\left(\frac{\pi n(k+\frac{1}{2})}{N}\right), \tag{9.12b}$$

$$= (-1)^k \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi(N-n)(k+\frac{1}{2})}{N}\right), \tag{9.12c}$$

$$= (-1)^k \sum_{n=0}^{N-1} x_{N-n} \cos\left(\frac{\pi}{N}n(k+\frac{1}{2})\right), \tag{9.12d}$$

$$= (-1)^k \text{IDCT}(\{x_{N-n}\})_k, \tag{9.12e}$$

where $x_N = 0$. The equality between Eqs. (9.12d) and (9.12e) can be derived by incorporating x_{N-n} into Eq. (9.11b). Since all the computations are broken down into the 1D DCT/IDCT kernels with proper transformations, the performance of DCT/IDCT kernels is critical to the overall efficiency of the entire placement.

DCT and its variations are correlated to fast Fourier transformations (FFT), so we can leverage the highly optimized FFT kernels provided by many deep learning frameworks to compute DCT with linear-time additional processing. Empirically, the most efficient way in the placement problem is to adopt the N-point real

Algorithm 9.3 DCT/IDCT with N-point FFT [21]

Require: An even-length real sequence x;
Ensure: An even-length transformed real sequence y;
function DCT(x)
  $N \leftarrow |x|$;
  for each thread $0 \leq t < N$ **do** ▷ Reorder kernel
    if $t < \frac{N}{2}$ **then**
      $x'_t \leftarrow x_{2t}$;
    else
      $x'_t \leftarrow x_{2(N-t)-1}$;
    end if
  end for
  $x'' \leftarrow \text{RFFT}(x')$; ▷ One-sided real FFT kernel
  for each thread $0 \leq t < N$ **do** ▷ $e^{-\frac{j\pi t}{2N}}$ kernel
    if $t \leq \frac{N}{2}$ **then**
      $y_t \leftarrow \frac{2}{N}\Re(x''_t e^{-\frac{j\pi t}{2N}})$; ▷ get real part
    else
      $y_t \leftarrow \frac{2}{N}\Re(\overline{x''_{(N-t)}} e^{-\frac{j\pi t}{2N}})$; ▷ get real part
    end if
  end for
  return y;
end function
function IDCT(x)
  $N \leftarrow |x|$;
  for each thread $0 \leq t < \frac{N}{2} + 1$ **do** ▷ Complex kernel
    $x'_t \leftarrow (x_t - j x_{(N-t)}) e^{\frac{j\pi t}{2N}}$; ▷ let $x_N \leftarrow 0$
  end for
  $x'' \leftarrow \text{IRFFT}(x')$; ▷ One-sided real IFFT kernel;
  for each thread $0 \leq t < N$ **do** ▷ Reverse kernel
    if $t \mod 2 == 0$ **then**
      $y_t \leftarrow \frac{N}{4} x''_{\frac{t}{2}}$;
    else
      $y_t \leftarrow \frac{N}{4} x''_{(N-\frac{t+1}{2})}$;
    end if
  end for
  return y;
end function

1D FFT/IFFT proposed by Makhoul [39] leveraging the symmetric property of FFT for real input sequences, as shown in Algorithm 9.3. It is reported that this implementation can achieve 2.1× faster than the $2N$-point implementations adopted by `Tensorflow` on DCT with map sizes from 512 × 512 to 4096 × 4096 and `float32`. To further speed up the kernels in practice, `DREAMPlace` also proposes to directly leverage 2D FFT [21] with more complicated pre-processing and post-processing, which can boost the speedup ratio to 5.0× on DCT. This version of implementations can eliminate the redundant computations with a one-time call to 2D FFT kernels. Meanwhile, the pre-processing and the post-processing routines are fully parallelizable.

9.4 Handle Region Constraints

Region constraints are specified by designers to improve performance, isolate voltage regions, leave space for later optimization, reduce power consumption, and so on [40]. Designers can set fence regions that are *member-hard* and *nonmember-hard*. In other words, instances assigned to a fence region have to be placed within the region, while instances not assigned to the region must be placed outside. Figure 9.13 plots an example of fence regions. Note that a fence region can consist of multiple disjoint rectangular subregions in the layout and instances assigned to the region can be placed in any of these subregions. Such a property necessitates global optimization techniques to ensure instances properly distributed to different subregions with minimized wirelength.

The literature has explored various methods to incorporate region constraints into the state-of-the-art placement algorithms. `NTUplace4dr` [12] proposes to handle region constraints with a region-aware clustering during nonlinear placement iterations. `Eh?Placer` [41] and `RippleDR` [42] follow the quadratic placement algorithm, i.e., an upper-bound-lower-bound optimization method with look-ahead rough legalization. They honor the region constraints during the rough legalization. These approaches essentially regard region assignment as a separate step and thus cannot smoothly integrate the constraints into the continuous optimization.

`DREAMPlace 3.0` [25] proposes to handle region constraints by extending the electrostatics-based placement model in `ePlace` series [13, 14] to a multi-electrostatic system. The `ePlace` family casts the placement problem into an energy minimization problem of an electrostatic system, where instances are analogous to electric charges. Minimizing the electric potential energy of the system contributes to the spreading of instances in the layout. Based on such a formulation, the key idea to consider region constraints is as follows. If we construct a separate electrostatic field for each region constraint and minimize the total

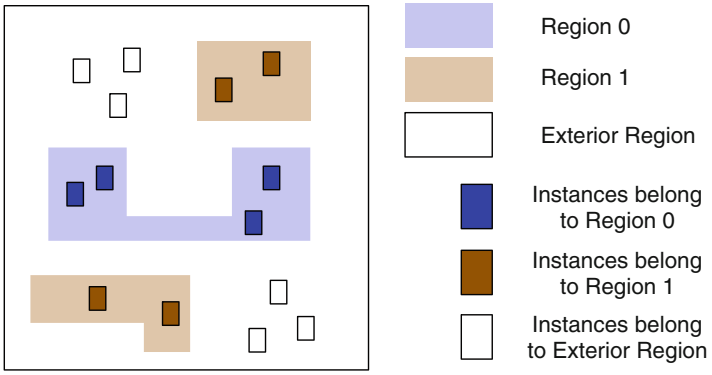


Fig. 9.13 An example of two fence regions in a layout

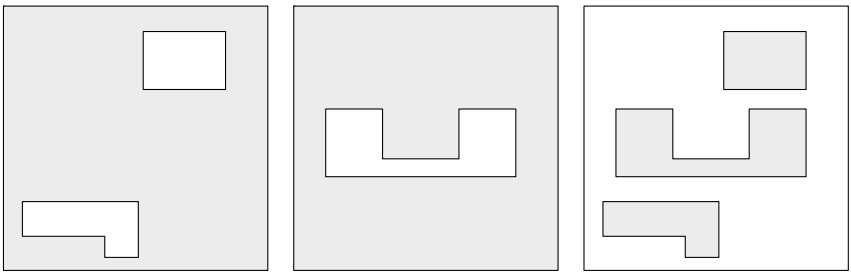


Fig. 9.14 Electric fields for (**a**) region 0, (**b**) region 1, and (**c**) the exterior region in Fig. 9.13. Gray areas denote blockages

potential energy of all fields, then the region constraints can be naturally handled during the placement optimization.

Figure 9.14 provides one example of the multi-field construction for two fence regions. In this case, we need three fields, two for each fence region and one for the exterior region. For each field, we insert blockage to the area outside the fence region such that the electric potential stays high and the related instances only move within the region. Similar settings apply to the exterior region. Based on this setup, we adapt the placement objective in Eq. (9.3) to the following:

$$\min_{\mathbf{x},\mathbf{y}} \quad \sum_{e \in E} \mathrm{WL}(e; \mathbf{x}, \mathbf{y})) + \langle \lambda, \Phi \rangle, \tag{9.13}$$

where the density weight $\lambda \in \mathbb{R}^{K+1}$ is a vector $(\lambda_0, \cdots, \lambda_K)$, K is the number of fence regions, and $\Phi = (\Phi_0, \cdots, \Phi_K)$ is a potential energy vector that considers fence regions into density calculation.

Besides the placement model, `DREAMPlace 3.0` also proposes two techniques to facilitate convergence: a quadratic density penalty to speed up convergence at plateau and an entropy injection method to avoid saddle points, as optimizing a multi-field system challenges the optimizer and often requires more iterations. The placement objective with the quadratic density penalty can be written as the following:

$$\min_{\mathbf{x},\mathbf{y}} \quad \sum_{e \in E} \mathrm{WL}(e; \mathbf{x}, \mathbf{y})) + \langle \lambda, \Phi + \frac{1}{2} \mu \Phi^2 \rangle, \tag{9.14}$$

where μ is the coefficient balancing the first-order and second-order density penalty terms. The basic idea of the quadratic penalty comes from the *augmented Lagrangian relaxation* technique that can avoid ill-conditioning and numerical instabilities [43]. In practice, the quadratic term Φ^2 needs to be normalized by the Φ at iteration 0 [25]. The entropy injection technique inspired by the perturbed

gradient descent method from machine learning [44] helps the optimization to escape from saddle points. By perturbating the movable instances in both x and y directions as the following (only the equation for x direction is shown for brevity; the mechanism for y direction is the same), the optimization can escape from saddle points and speed up the convergence:

$$\hat{\mathbf{x}} = s\left(\mathbf{x} - \frac{\sum_{i \in v} x_i}{|v|}\right) + \frac{\sum_{i \in v} x_i}{|v|} + \Delta\mathbf{x}, \tag{9.15}$$

where $\Delta\mathbf{x}$ is a perturbation vector sampled from a multivariate Gaussian distribution $\Delta\mathbf{x} \sim \mathcal{N}(0, \sigma^2)$. The shrinking factor $s \in (0, 1)$ rolls back the spreading process and increases the density overflow, forcing the optimizer to re-optimize the perturbed wirelength. The above techniques can be easily and efficiently integrated into the `DREAMPlace` framework by only writing Python codes, indicating the flexibility of the methodology.

The experimental results on ISPD 2015 contest benchmarks [40] demonstrate that `DREAMPlace 3.0` can achieve more than 13% HPWL reduction and 11% global routing overflow reduction compared with `NTUplace4dr` and `Eh?Placer` on designs with region constraints.

9.5 Optimize Routability

Routability optimization is required for modern placement engines to achieve high-quality results after routing. In this section, we introduce the typical routability optimization strategy based on instance inflation as well as a recent strategy integrating a machine learning-based routability penalty into the placement objective.

9.5.1 Instance Inflation

A widely adopted technique for routability optimization is instance inflation [14], because routability issues usually come from congested regions. Increasing the area of instances in these regions can leave more space and thus can help resolve routing congestion. `DREAMPlace` [21] adopts this idea and leverages an external global router, `NCTUgr` [45], to evaluate the congestion during placement iterations. Figure 9.15 plots the placement flow for routability optimization based on instance inflation. When the density overflow drops to 20%, it invokes the global router to evaluate the routing congestion map and adjust the areas of instances. For each metal layer, it computes the ratios between routing demand and capacity at each routing tile and chooses the maximum ratio across all layers to compute the ratio:

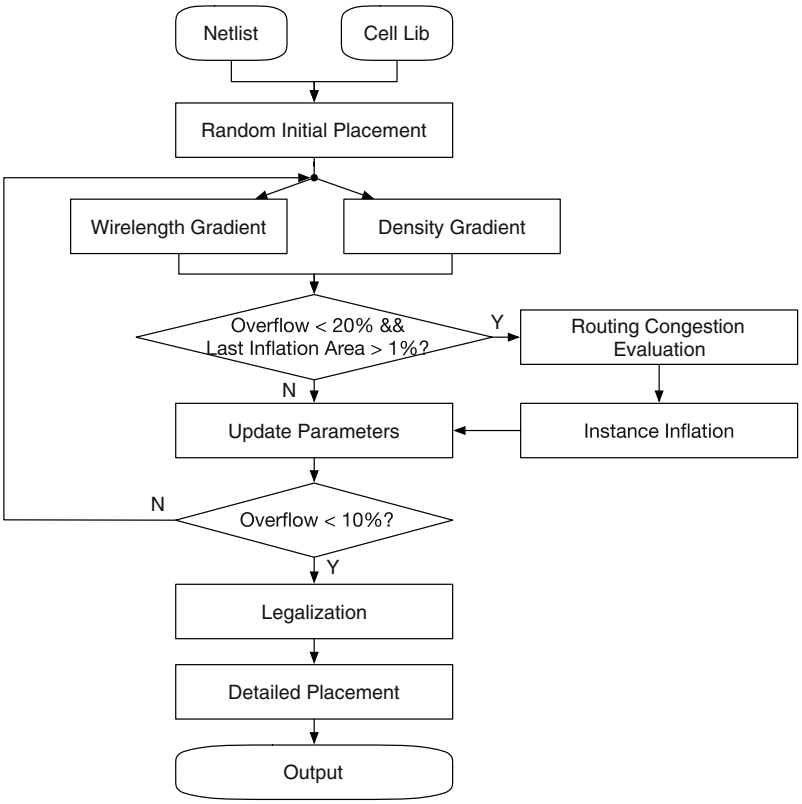


Fig. 9.15 Placement flow for routability optimization based on instance inflation

$$ratio = \min\left(\left(\max_{\forall l \in L} \frac{demand_l}{capacity_l}\right)^{2.5}, 2.5\right), \tag{9.16}$$

where L is the set of metal layers. The exponent and maximum limits can be adjusted according to the benchmarks. The inflation ratios need to be carefully controlled, as there may not be enough whitespace in the layout to digest the inflated areas. The inflation stops when the total increased area is less than 1% of the total instance area.

Experiments on DAC 2012 routability-driven placement contest benchmarks [46] demonstrate that `DREAMPlace` can achieve 10% better scaled HPWL and 0.5% smaller routing congestion compared with `RePlAce` [14]. Meanwhile, with the power of GPU acceleration, `DREAMPlace` is 9× faster than `RePlAce` in global placement and 5× faster in the entire placement flow. It also reports that more than 75% of the runtime in global placement is taken by the external router to obtain congestion maps, which should be optimized in the future.

9.5.2 Deep Learning-Based Optimization

To overcome the runtime issue raised from repeatedly invoking global routing in placement, many studies have explored machine learning-based congestion estimation to improve the efficiency [47–50]. These studies aim at replacing the conventional congestion estimator with machine learning models, while still adopting the instance inflation strategy for routability optimization. Although instance inflation can effectively reduce the density in congested regions, it is an indirect optimization technique and requires careful parameter tuning. Liu et al. [26] propose an explicit routability optimization technique based on the `DREAMPlace` framework leveraging deep learning models. The key idea is to obtain a differentiable congestion estimator using neural networks and then directly integrate the congestion penalty into the nonlinear placement objective in Eq. (9.3) for explicit routability optimization as the following:

$$\min_{\mathbf{x},\mathbf{y}} \quad \left(\sum_{e\in E} \mathrm{WL}(e;\mathbf{x},\mathbf{y})\right) + \lambda\Phi(\mathbf{x},\mathbf{y}) + \eta \mathrm{L}(\mathbf{x},\mathbf{y}), \tag{9.17}$$

where $\mathrm{L}(\cdot)$ is the congestion penalty calculated by the deep learning-based congestion estimator. Due to the differentiability of neural networks, the congestion penalty is compatible with the gradient descent optimizers in placement. Meanwhile, integrating neural networks into the `DREAMPlace` framework is natively supported, since it is developed in the deep learning framework `PyTorch`.

The algorithm can be divided into two phases, routability model training and routability-driven optimization.

Routability model training is to obtain the congestion estimator. They adopt RouteNet [47] as the network architecture for congestion map prediction. RouteNet is a fully convolutional neural network that takes image-like feature maps such as rectangular uniform wire density (RUDY) [51], pin RUDY, and snapshots of macros as input. It outputs a congestion map given the features. They train the network using a dataset with at least 1400 placement solutions from ISPD 2015 contest benchmarks [40] generated by `DREAMPlace`. For brevity, we denote the congestion map prediction task as f_{R}:

$$f_{\mathrm{R}} : \mathbf{M} \subset \mathbb{R}^{M\times N\times 3} \longrightarrow \mathcal{Y} \subset \mathbb{R}^{M\times N}, \tag{9.18}$$

where $\mathbf{M}$ denotes the features and $\mathcal{Y}$ denotes the congestion map.

Routability-driven optimization explicitly minimizes the congestion penalty defined as the mean squared Frobenius norm of congestion map:

$$L(\mathbf{x},\mathbf{y}) = \frac{1}{MN}\|f_{\mathrm{R}}(\mathbf{M})\|_2^2, \tag{9.19}$$

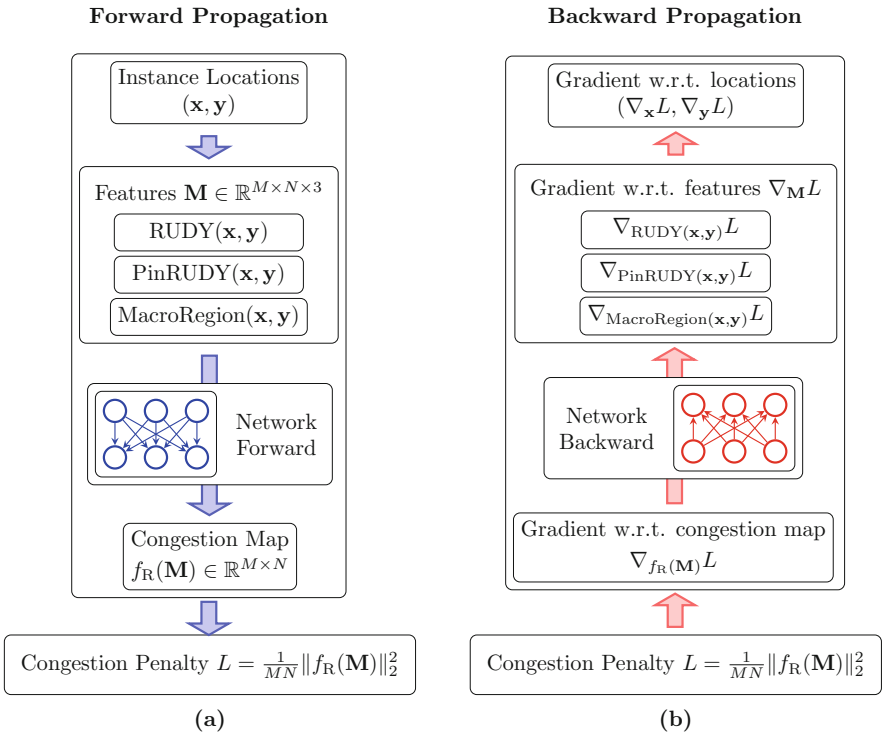


Fig. 9.16 (**a**) Forward and (**b**) Backward propagation for congestion penalty [26]

where the feature map $\mathbf{M}$ is derived from instance locations $\mathbf{x}$, $\mathbf{y}$ using RUDY, pin RUDY, and locations of macros. It is also differentiable to $\mathbf{x}$, $\mathbf{y}$, so we can also follow the procedure of forward and backward propagation to optimize the penalty in placement. Figure 9.16 sketches the forward and backward propagation procedure for the congestion penalty. Different from neural network training where backward propagation computes the gradient to weights, here it computes the gradients to the inputs, i.e., essentially the instance locations. The overall placement flow is shown in Fig. 9.17. The major difference from `DREAMPlace` lies in the congestion gradient computation. Eventually, up to 9.05% reduction in congestion and 5.30% reduction in routed wirelength are reported on ISPD 2015 contest benchmarks compared to the state-of-the-art placer `NTUplace4dr` [12] with more than 20× speedup.

As the routability experiments of [21] and [26] are conducted on different benchmarks suites and different machines, it is difficult to have direct comparison. However, we observe that on million-size designs, both algorithms report similar overall runtime, which is out of expectation. This is because [21] only invokes the external global router for about 6 times, while [26] needs to compute the congestion gradient in every iteration (around 1000 times in total). Although deep learning-based congestion estimator is much faster than the global router, frequent invocation

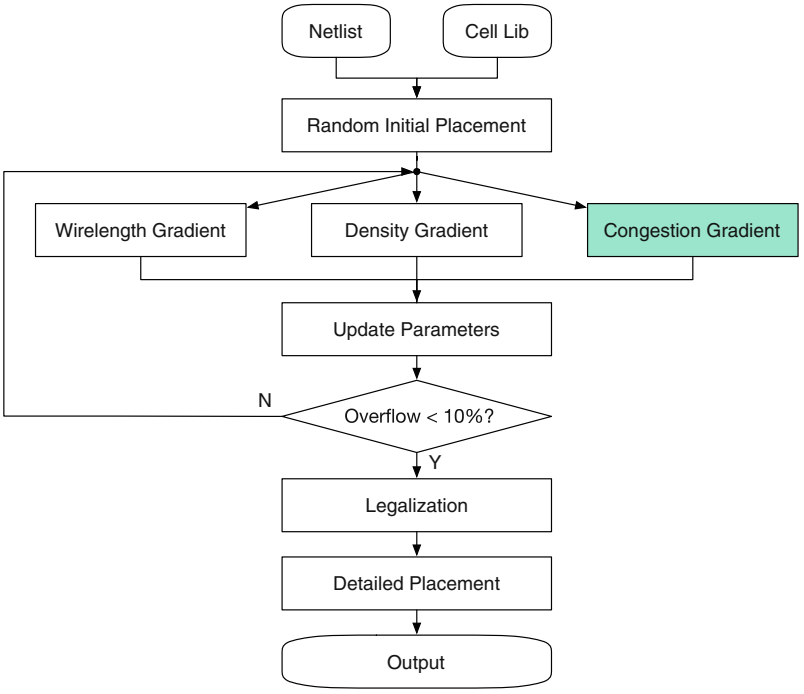


Fig. 9.17 Placement flow for deep learning-based explicit routability optimization

can significantly slow down the placement. There are several directions to reduce the overhead, e.g., simplifying the network architecture or only updating the congestion gradient for a certain amount of iterations, which is worth exploring in the future. Overall speaking, the studies on routability-driven optimization demonstrate the efficiency and flexibility of deep learning frameworks-enabled placement engines and the compatibility with deep learning-assisted optimization.

9.6 Conclusion

In this chapter, we introduce a series of studies leveraging deep learning frameworks to develop placement engines. We summarize the major advantages as follows:

- Efficiency. Developing placement with deep learning framework enables native support to GPU acceleration, which brings significant speedup compared with conventional placement engines on CPU.
- Extensibility. The framework is highly extensible to incorporate additional optimization objectives like routability and natively compatible with machine learning-assisted placement techniques.

- Friendly to beginners. Due to the wide adoption of deep learning techniques, frameworks like `TensorFlow` and `PyTorch` are well-known packages with abundant tutorials, lowering the bar for beginners to explore new ideas.

Despite the current progress, there are still directions worth exploring in the future.

- Timing- and power-driven optimization.
- Integration with other design stages like routing, gate sizing, and clock tree synthesis.
- Application to other scenarios like FPGA placement and routing [52].

As `DREAMPlace` has been open-source, we believe it can stimulate more efforts for placement research and open new directions in both acceleration and optimization.

References

1. Markov, I.L., Hu, J., Kim, M.C.: Progress and challenges in VLSI placement research. Proc. IEEE **103**(11), 1985–2003 (2015)
2. Viswanathan, N., Chu, C.C.: Fastplace: Efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model. IEEE TCAD **24**(5), 722–733 (2005)
3. Viswanathan, N., Pan, M., Chu, C.: FastPlace 3.0: a fast multilevel quadratic placement algorithm with placement congestion control. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC), pp. 135–140. IEEE, Piscataway (2007)
4. Kim, M.C., Lee, D.J., Markov, I.L.: Simpl: An effective placement algorithm. IEEE TCAD **31**(1), 50–60 (2012)
5. He, X., Huang, T., Xiao, L., Tian, H., Young, E.F.Y.: Ripple: a robust and effective routability-driven placer. IEEE TCAD **32**(10), 1546–1556 (2013)
6. Lin, T., Chu, C., Shinnerl, J.R., Bustany, I., Nedelchev, I.: POLAR: a high performance mixed-size wirelengh-driven placer with density constraints. IEEE TCAD **34**(3), 447–459 (2015)
7. Kahng, A.B., Reda, S., Wang, Q.: Architecture and details of a high quality, large-scale analytical placer. In: ICCAD, pp. 891–898. IEEE, Piscataway (2005)
8. Kahng, A.B., Wang, Q.: A faster implementation of APlace. In: ISPD, pp. 218–220. ACM, New York (2006)
9. Chan, T., Cong, J., Sze, K.: Multilevel generalized force-directed method for circuit placement. In: ISPD, pp. 185–192. ACM (2005)
10. Chan, T.F., Sze, K., Shinnerl, J.R., Xie, M.: mPL6: Enhanced multilevel mixed-size placement with congestion control. In: Modern Circuit Placement. Springer, Berlin (2007)
11. Chen, T.C., Jiang, Z.W., Hsu, T.C., Chen, H.C., Chang, Y.W.: Ntuplace3: an analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. IEEE TCAD **27**(7), 1228–1240 (2008)
12. Huang, C., Lee, H., Lin, B., Yang, S., Chang, C., Chen, S., Chang, Y., Chen, T., Bustany, I.: NTUplace4dr: a detailed-routing-driven placer for mixed-size circuit designs with technology and region constraints. IEEE TCAD **37**(3), 669–681 (2018)
13. Lu, J., Zhuang, H., Chen, P., Chang, H., Chang, C.C., Wong, Y.C., Sha, L., Huang, D., Luo, Y., Teng, C.C., et al.: ePlace-MS: electrostatics-based placement for mixed-size circuits. IEEE TCAD **34**(5), 685–698 (2015)
14. Cheng, C.K., Kahng, A.B., Kang, I., Wang, L.: RePlAce: Advancing solution quality and routability validation in global placement. IEEE TCAD (2018)
15. Cadence Innovus. http://www.cadence.com

16. Synopsys IC Compiler. http://www.synopsys.com
17. Ludwin, A., Betz, V., Padalia, K.: High-quality, deterministic parallel placement for FPGAs on commodity hardware. In: FPGA, pp. 14–23. ACM, New York (2008)
18. Lin, T., Chu, C., Wu, G.: Polar 3.0: An ultrafast global placement engine. In: ICCAD, pp. 520–527 (2015)
19. Li, W., Li, M., Wang, J., Pan, D.Z.: Utplacef 3.0: a parallelization framework for modern FPGA global placement. In: ICCAD, pp. 908–914 (2017)
20. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
21. Lin, Y., Jiang, Z., Gu, J., Li, W., Dhar, S., Ren, H., Khailany, B., Pan, D.Z.: Dreamplace: deep learning toolkit-enabled GPU acceleration for modern VLSI placement. IEEE TCAD (2020)
22. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Conference on Neural Information Processing Systems (NIPS), pp. 8024–8035. Curran Associates (2019)
23. Lin, Y., Li, W., Gu, J., Ren, H., Khailany, B., Pan, D.Z.: Abcdplace: accelerated batch-based concurrent detailed placement on multithreaded cpus and GPUs. IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **39**(12), 5083–5096 (2020)
24. Lin, Y., Pan, D.Z., Ren, H., Khailany, B.: Dreamplace 2.0: Open-source GPU-accelerated global and detailed placement for large-scale VLSI designs. In: 2020 China Semiconductor Technology International Conference (CSTIC), pp. 1–4 (2020)
25. Gu, J., Jiang, Z., Lin, Y., Pan, D.Z.: Dreamplace 3.0: multi-electrostatics based robust VLSI placement with region constraints. In: 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1–9 (2020)
26. Liu, S., Sun, Q., Liao, P., Lin, Y., Yu, B.: Global placement with deep learning-enabled explicit routability optimization. In: DATE. Virtual Conference (2021)
27. Guo, Z., Mai, J., Lin, Y.: Ultrafast CPU/GPU kernels for density accumulation in placement. In: DAC. San Francisco (2021)
28. Lin, Y.: GPU acceleration in VLSI back-end design: overview and case studies. In: Proceedings of the 39th International Conference on Computer-Aided Design, ICCAD '20. Association for Computing Machinery, New York (2020)
29. Spindler, P., Schlichtmann, U., Johannes, F.M.: Abacus: Fast legalization of standard cell circuits with minimal movement. In: ISPD, ISPD '08, pp. 47–53. Association for Computing Machinery, New York (2008)
30. Pan, M., Viswanathan, N., Chu, C.: An efficient and effective detailed placement algorithm. In: ICCAD, pp. 48–55 (2005)
31. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: ICLR (Poster) (2015)
32. Cheng, C.K., Kahng, A.B., Kang, I., Wang, L.: Replace: advancing solution quality and routability validation in global placement. IEEE TCAD (2018)
33. Hsu, M.K., Chang, Y.W., Balabanov, V.: TSV-aware analytical placement for 3D IC designs. In: DAC, pp. 664–669. ACM, New York (2011)
34. Hsu, M.K., Balabanov, V., Chang, Y.W.: TSV-aware analytical placement for 3-D IC designs based on a novel weighted-average wirelength model. DAC **32**(4), 497–509 (2013)
35. Lu, J., Chen, P., Chang, C.C., Sha, L., Huang, D.J.H., Teng, C.C., Cheng, C.K.: ePlace: Electrostatics-based placement using fast fourier transform and Nesterov's method. ACM TODAES **20**(2), 17 (2015)
36. Lin, C.X., Wong, M.D.: Accelerate analytical placement with GPU: a generic approach. In: DATE, pp. 1345–1350. IEEE, Piscataway (2018)
37. Gessler, F., Brisk, P., Stojilovič, M.: A shared-memory parallel implementation of the replace global cell placer. In: International Conference on VLSI Design, pp. 78–83. IEEE, Piscataway (2020)
38. Crow, F.C.: Summed-area tables for texture mapping. In: SIGGRAPH '84, pp. 207–212. ACM, New York (1984)
39. Makhoul, J.: A fast cosine transform in one and two dimensions. IEEE Trans. Signal Process. **28**(1), 27–34 (1980)

40. Bustany, I.S., Chinnery, D., Shinnerl, J.R., Yutsis, V.: ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement. In: ISPD, pp. 157–164 (2015)
41. Darav, N.K., Kennings, A., Tabrizi, A.F., Westwick, D., Behjat, L.: Eh?Placer: a high-performance modern technology-driven placer. ACM TODAES **21**(3), 1–27 (2016)
42. Chow, W., Kuang, J., Tu, P., Young, E.F.Y.: Fence-aware detailed-routability driven placement. In: ACM Workshop on System Level Interconnect Prediction (SLIP), pp. 1–7 (2017)
43. Birgin, E.G., Martínez, J.M.: Practical augmented Lagrangian methods for constrained optimization. SIAM (2014)
44. Jin, C., Ge, R., Netrapalli, P., Kakade, S.M., Jordan, M.I.: How to escape saddle points efficiently. In: International Conference on Machine Learning (ICML), pp. 1724–1732. PMLR (2017)
45. Liu, W.H., Li, Y.L., Koh, C.K.: A fast maze-free routing congestion estimator with hybrid unilateral monotonic routing. In: ICCAD, pp. 713–719 (2012)
46. Viswanathan, N., Alpert, C., Sze, C., Li, Z., Wei, Y.: The DAC 2012 routability-driven placement contest and benchmark suite. In: DAC, pp. 774–782. ACM, New York (2012)
47. Xie, Z., Huang, Y.H., Fang, G.Q., Ren, H., Fang, S.Y., Chen, Y., Hu, J.: Routenet: routability prediction for mixed-size designs using convolutional neural network. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8. IEEE, Piscataway (2018)
48. Kirby, R., Godil, S., Roy, R., Catanzaro, B.: Congestionnet: routing congestion prediction using deep graph neural networks. In: 2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC), pp. 217–222. IEEE, Piscataway (2019)
49. Alawieh, M.B., Li, W., Lin, Y., Singhal, L., Iyer, M.A., Pan, D.Z.: High-definition routing congestion prediction for large-scale FPGAs. In: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 26–31. IEEE, Piscataway (2020)
50. Liang, R., Xiang, H., Pandey, D., Reddy, L., Ramji, S., Nam, G.J., Hu, J.: DRC hotspot prediction at sub-10 nm process nodes using customized convolutional network. In: Proceedings of the 2020 International Symposium on Physical Design, pp. 135–142 (2020)
51. Spindler, P., Johannes, F.M.: Fast and accurate routing demand estimation for efficient routability-driven placement. In: DATE, pp. 1226–1231 (2007)
52. Meng, Y., Li, W., Lin, Y., Pan, D.Z.: elfPlace: electrostatics-based placement for large-scale heterogeneous FPGAs. IEEE TCAD (2021)