# DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement

Yibo Lin, *Member, IEEE*, Zixuan Jiang, *Graduate Student Member, IEEE*, Jiaqi Gu, *Graduate Student Member, IEEE*, Wuxi Li, *Member, IEEE*, Shounak Dhar, *Member, IEEE*, Haoxing Ren, *Senior Member, IEEE*, Brucek Khailany, *Senior Member, IEEE*, and David Z. Pan, *Fellow, IEEE*

***Abstract*—Placement for very large-scale integrated (VLSI) circuits is one of the most important steps for design closure. We propose a novel GPU-accelerated placement framework DREAMPlace, by casting the analytical placement problem equivalently to training a neural network. Implemented on top of a widely adopted deep learning toolkit `PyTorch`, with customized key kernels for wirelength and density computations, DREAMPlace can achieve around 40× speedup in global placement without quality degradation compared to the state-of-the-art multithreaded placer RePlAce. We believe this work shall open up new directions for revisiting classical EDA problems with advancements in AI hardware and software.**

***Index Terms*—Deep learning, GPU acceleration, physical design, VLSI placement.**

## I. Introduction

PLACEMENT is a critical but time-consuming step in the very large-scale integrated (VLSI) design flow. As it determines the locations of standard cells in the physical layout, its quality has significant impacts on the later stages in the flow, such as routing and post-layout optimization. A placement solution also provides relatively accurate estimation to routed wirelength and congestion, which is very valuable in guiding the earlier stages like logic synthesis. Commercial design flows often run core placement engines many times to achieve design closure. As placement involves large-scale numerical optimization, today's placers usually take hours for large designs, thus, slowing down design iterations. Therefore, ultrafast yet high-quality placement is always desired.

Manuscript received September 16, 2019; revised January 9, 2020, April 5, 2020, and June 9, 2020; accepted June 12, 2020. Date of publication June 22, 2020; date of current version March 19, 2021. This work was supported in part by NVIDIA. This article was recommended by Associate Editor I. H.-R. Jiang. *(Corresponding author: Yibo Lin.)*

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

Zixuan Jiang, Jiaqi Gu, and David Z. Pan are with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712 USA.

Wuxi Li is with the FPGA Implementation Software Group, Xilinx Inc., San Jose, CA 95124 USA.

Shounak Dhar is with the Programmable Solutions Group, Intel Corporation, San Jose, CA 95134 USA.

Haoxing Ren and Brucek Khailany are with NVIDIA, Austin, TX 78717 USA.

Digital Object Identifier 10.1109/TCAD.2020.3003843

Analytical placement is the current state-of-the-art for VLSI placement [1]–[15]. It essentially solves a nonlinear optimization problem. Although analytical placement can produce high-quality solutions, it is also known to be relatively slow [11], [13], [14], [16]. Here, we provide a brief introduction to the analytical placement problem. Suppose a circuit is described as a hypergraph $H = (V, E)$, where $V$ denotes the set of vertices (cells) and $E$ denotes the set of hyperedges (nets). Let $\mathbf{x}, \mathbf{y}$ denote the locations of cells. The objective of analytical placement is to determine the locations of cells with wirelength minimized and no overlap in the layout.

Analytical placement can be roughly categorized into quadratic placement and nonlinear placement. Quadratic placement tackles the problem by iterating between an unconstrained wirelength minimization step and a rough legalization (LG) or spreading step [10]–[15]. The wirelength minimization step usually adopts a quadratic wirelength model and minimizes the total wirelength regardless of the overlaps between cells. The rough LG step removes the overlaps based on heuristic approaches without explicit consideration of the wirelength cost. By iterating between these two steps, cells can be gradually spread out. Meanwhile, the wirelength cost is minimized. Nonlinear placement directly solves the placement problem with nonlinear optimization techniques [1]–[9], [17]. It formulates a nonlinear optimization problem with a wirelength objective subjecting to a density constraint. By relaxing the density constraint into the objective, gradient descent-based techniques can be adopted to search for a high-quality solution. In this article, we focus on the nonlinear placement approach, as many commercial tools like Cadence Innovus [18] and Synopsys IC Compiler [19] adopt that.

To accelerate placement, existing parallelization efforts have mostly targeted multithreaded CPUs using partitioning [16], [20], [21]. As the number of threads increases, speedup quickly saturates at around 5× in global placement (GP) with typical quality degradation of 2%–6%. Cong and Zou [22] explored GPU acceleration for analytical placement. They combined clustering and declustering with nonlinear placement optimization. By parallelizing the nonlinear placement part, an average of 15× speedup in GP was reported with less than 1% quality degradation. Lin and Wong [23] proposed GPU acceleration techniques for wirelength gradient computation and area accumulation, but their experiments failed to consider real operations, such as density cost computation,

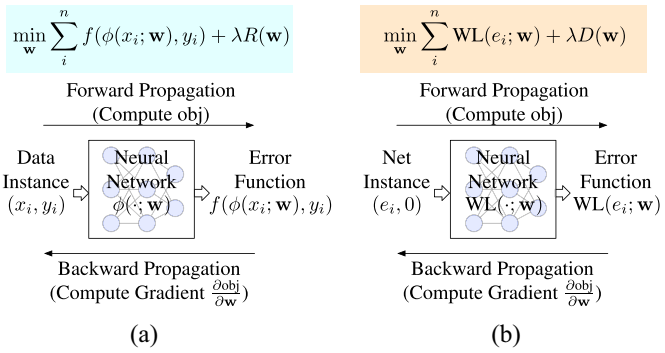



Fig. 1. Analogy between neural network training and analytical placement. (a) Train a network for weights $\mathbf{w}$. (b) Solve a placement for cell locations $\mathbf{w} = (\mathbf{x}, \mathbf{y})$.

and it lacked the validation from real analytical placement flows. In addition, current research on placement is facing challenges in the lack of well-maintained public frameworks and the high development overhead, raising the bar to validate new algorithms systematically.

In this work, we propose *DREAMPlace*, a GPU-accelerated analytical placer developed with deep learning toolkit `PyTorch` [24] by casting an analytical placement problem to training a neural network. DREAMPlace is based on the state-of-the-art analytical placement algorithm ePlace/RePlAce family [6], [8], but the framework is designed in a generic way that is compatible with other analytical placers such as NTUplace [4]. The key contributions are summarized as follows.

1) We take a totally new perspective of making an analogy between placement and deep learning, and build an open-source generic analytical placement framework that runs on both CPU and GPU platforms developed with modern deep learning toolkits.
2) A variety of gradient-descent solvers are provided, such as Nesterov's method, conjugate gradient method, and Adam [25], with the help from deep learning toolkit.
3) We propose efficient GPU implementations of key kernels in analytical placement like wirelength and density computation.
4) We demonstrate around 40× speedup in GP without quality degradation of the entire placement flow over multithreaded RePlAce implementations. More specifically, a design with one million cells finishes in one minute even with LG. The framework maintains nearly linear scalability with industrial designs up to 10-million cells.

The source code is released on Github.[1] To clarify, the casting of placement problem to deep learning problems aims at using the toolkit to solve placement, which is orthogonal to using deep learning models for placement. The remainder of this article is organized as follows. Section II describes the background and motivation. Section III explains the detailed implementation. Section IV demonstrates the results. Section V concludes this article.

[1]https://github.com/limbo018/DREAMPlace

## II. Preliminaries

This section will review the background and motivation.

### A. Analytical Placement

Analytical placement usually consists of three steps: 1) GP; 2) LG; and 3) detailed placement (DP). GP spreads out cells in the layout with a target cost minimized; LG removes the remaining overlaps between cells and aligns cells to placement sites; and DP performs incremental refinement to further improve the quality. Usually, GP is the most time-consuming portion in analytical placement.

GP aims at minimizing the wirelength cost subjecting to density constraints. The formulation can be written as follows:

$$\min_{\mathbf{x},\mathbf{y}} \sum_{e\in E} \text{WL}(e;\mathbf{x},\mathbf{y}) \tag{1a}$$

$$\text{s.t. } d(\mathbf{x},\mathbf{y}) \le d_t \tag{1b}$$

where $\text{WL}(\cdot;\cdot)$ is the wirelength cost function that takes any net instance $e$ and returns the wirelength, $d(\cdot)$ is the density of a location in the layout, and $d_t$ is a given target density. A typical solving approach is to relax the density constraints to the objective as a density penalty [1], [4], [6]

$$\min_{\mathbf{x},\mathbf{y}} \left(\sum_{e\in E} \text{WL}(e;\mathbf{x},\mathbf{y})\right) + \lambda D(\mathbf{x},\mathbf{y}) \tag{2}$$

where $D(\cdot)$ is the density penalty to spread cells out in the layout. The density constraints can be satisfied by gradually increasing the weight of $\lambda$.

### B. Analogy to Deep Learning

As both solving an analytical placement and training a neural network are essentially solving a nonlinear optimization problem, we investigate the underlying similarity between the two problems: 1) the analogy of the wirelength cost to the error of misprediction and 2) that of the density cost to the regularization term. Fig. 1 shows the objective functions of the two problems. In neural network training, each data instance with a feature vector $x_i$ and a label $y_i$ is fed to the network, and the neural network predicts a label $\phi(x_i;\mathbf{w})$. The task for training is to minimize the overall objective over weights $\mathbf{w}$, where the objective consists of the prediction errors for all data instances, and a regularization term $R(\mathbf{w})$ [26]. In the analogy of placement to neural network training, we combine cell locations $(\mathbf{x},\mathbf{y})$ into $\mathbf{w}$ for brevity. Each data instance is replaced with a net instance with a feature vector $e_i$ and a label zero. The neural network then takes a net instance and computes the wirelength cost $\text{WL}(e_i;\mathbf{w})$. Using the absolute error function $f(\hat{y},y) = |\hat{y}-y|$ and noting that wirelength is non-negative, the minimization of prediction errors becomes $\sum_i^n \text{WL}(e_i;\mathbf{w})$. The density cost $D(\mathbf{w})$ corresponds to the regularization term $R(\mathbf{w})$, as it is not related to net instances. With this construction, we find a one-to-one mapping of each component in analytical placement to neural network training, which makes it possible to take advantage of recent developments in deep learning toolkits for implementation. Then, we can solve the placement problem following the neural network

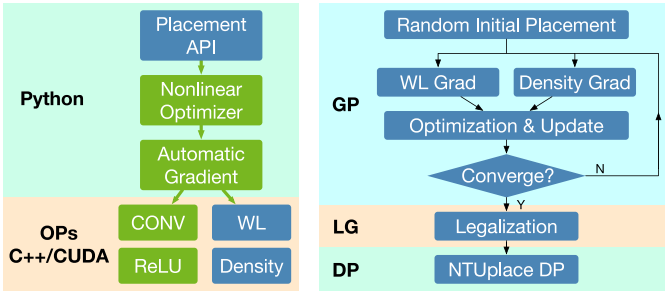



Fig. 2. (a) Software architecture for placement implementation using deep learning toolkits. (b) DREAMPlace flow.

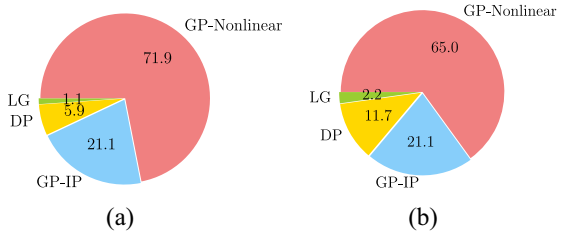



Fig. 3. RePlAce [8] runtime breakdown in percentages on `bigblue4` (2 million cells). (a) One thread. (b) Ten threads.

training procedure, with forward propagation to compute the objective and backward propagation to calculate the gradient.

Deep learning toolkits nowadays consist of three stacks, low-level operators (OPs), automatic gradient derivation, and optimization engines, as shown in Fig. 2(a). Toolkits like `TensorFlow` and `PyTorch` offer mature and efficient implementation of these three stacks with compatibility to both CPU and GPU acceleration. The toolkits also provide convenient APIs to extend the existing set of low-level OPs. Each custom OP requires well defined forward and backward functions for cost and gradient computation. To develop an analytical placement with deep learning toolkits, we only need to implement the custom OPs for wirelength and density cost in C++ and CUDA. Then we can construct a placement framework in Python with very low development overhead and easily incorporate a variety of optimization engines in the toolkit. The placement framework can run on both CPU and GPU platforms. The conventional development of placement engines takes huge efforts in building the entire software stacks with C++. Thus, the bar of designing and validating a new placement algorithm is very high due to the development overhead. Taking advantage of deep learning toolkits, researchers can concentrate on the development of critical parts like low-level OPs and high-level optimization engines.

### C. ePlace/RePlAce Algorithm

ePlace/RePlAce is a state-of-the-art family of GP algorithms that model the layout and netlist as an electrostatic system [6]–[8]. It uses weighted-average wirelength (WA) for wirelength cost originally proposed by [27], [28]

$$\mathrm{WA}_e = \frac{\sum_{i\in e} x_i e^{\frac{x_i}{\gamma}}}{\sum_{i\in e} e^{\frac{x_i}{\gamma}}} - \frac{\sum_{i\in e} x_i e^{-\frac{x_i}{\gamma}}}{\sum_{i\in e} e^{-\frac{x_i}{\gamma}}} \tag{3}$$

where $\gamma$ is a parameter to control the smoothness and accuracy of the approximation to half-perimeter wirelength (HPWL). The smaller $\gamma$ is, the more accurate it is to approximate HPWL, but the less smooth.

Its density penalty is quite different from other analytical placers [1], [3], [4]. With analogy to an electrostatic system, cells are modeled as charges, density penalty is modeled as potential energy, and the density gradient is modeled as the electric field. The electric potential and field distribution can be computed by solving Poisson's equation from the charge density distribution

$$\nabla \cdot \nabla \psi(x,y) = -\rho(x,y) \tag{4a}$$

$$\hat{\mathbf{n}} \cdot \nabla \psi(x,y) = \mathbf{0}, \quad (x,y) \in \partial R \tag{4b}$$

$$\iint_R \rho(x,y) = \iint_R \psi(x,y) = 0 \tag{4c}$$

where $R$ denotes the placement region, $\partial R$ denotes the boundary to the region, $\hat{\mathbf{n}}$ denotes the outer normal vector of the placement region, $\rho$ denotes the charge density, and $\psi$ denotes the electric potential.

The numerical solution of Poisson's equation can be obtained with spectral methods. Given an $M \times M$ grid of bins and $w_u = (2\pi u/M)$ and $w_v = (2\pi v/M)$ with $u = 0, 1, \ldots, M-1, v = 0, 1, \ldots, M-1$, the solution can be computed as follows [6]:

$$a_{u,v} = \frac{1}{M^2} \sum_{x=0}^{M-1} \sum_{y=0}^{M-1} \rho(x,y) \cos(w_u x) \cos(w_v y) \tag{5a}$$

$$\psi_{\mathrm{DCT}}(x,y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v}}{w_u^2 + w_v^2} \cos(w_u x) \cos(w_v y) \tag{5b}$$

$$\xi_{\mathrm{DSCT}}^X(x,y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v} w_u}{w_u^2 + w_v^2} \sin(w_u x) \cos(w_v y) \tag{5c}$$

$$\xi_{\mathrm{DCST}}^Y(x,y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v} w_v}{w_u^2 + w_v^2} \cos(w_u x) \sin(w_v y) \tag{5d}$$

where $\psi_{\mathrm{DCT}}$ denotes the numerical solution of the potential function, and $\xi_{\mathrm{DSCT}}^X$ and $\xi_{\mathrm{DCST}}^Y$ denote the solution of the electric field in horizontal and vertical directions, respectively. Equation (5) requires discrete Cosine transform (DCT) and inverse DCT (IDCT) routines to solve efficiently. The detailed computation is explained in Section III. With the electric field defined for each bin, the density gradient of each cell is the overall force taken by the cell in the system.

After defining wirelength cost and density penalty, RePlAce adopts gradient-descent optimizers, such as Nesterov's method and conjugate gradient method, to solve the optimization problem. RePlAce was implemented with multithreading support [8]. The runtime breakdown for RePlAce [8] is elaborated in Fig. 3. GP including initial placement (GP-IP) and nonlinear optimization (GP-Nonlinear) takes about 90% of the runtime with both single thread and 10 threads. Therefore, accelerating GP is the most effective in reducing the overall runtime.

TABLE I
NOTATIONS

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

## III. DREAMPLACE ALGORITHMS

Our overall placement flow is given in Fig. 2(b). It is slightly different from the typical one that starts from a bound-to-bound initial placement [4], [6]. We observe that starting from a random initial placement also achieves the same quality (<0.04% difference) with significantly less runtime (21.1% in Fig. 3). In initial placement, standard cells are placed in the center of the layout with a small Gaussian noise. In our experiments, the scales of the noise are set to 0.1% of the width and height of the placement region. The kernel GP iterations refer to the loop that involves the computation of wirelength and density gradient, optimization engines, and cell location updating. After the GP converges, LG is performed to remove remaining overlaps and align cells to placement sites. The last step before the output is DP to refine the placement solutions relying on NTUplace3 [4]. The rest of this section will focus on GPU acceleration to the ePlace/RePlAce algorithm [6], [8].

### A. Wirelength Forward and Backward

As RePlAce adopts WA wirelength, we also use it as an example for the GPU acceleration to wirelength forward and backward. Similar insights also apply to other wirelength costs like log-sum-exp (LSE) [29], which is also implemented in the framework. For brevity, we only discuss the equations in the $x$ dimension, as those in the $y$ dimension are similar. The real implementation will separate the computation for $x$ and $y$ into different GPU streams as they are independent.

Direct implementation of WA wirelength defined in (3) may result in numerical overflow, so we convert $e^{(x_i/\gamma)}$ to $e^{[(x_i - \max_{j\in e} x_j)/\gamma]}$, and $e^{-(x_i/\gamma)}$ to $e^{-[(x_i - \min_{j\in e} x_j)/\gamma]}$ in (3), which is an equivalent transformation. With the notations in Table I, the gradient of WA wirelength to a pin location can be written as

$$\frac{\partial \mathrm{WL}_e}{\partial x_i} = \frac{\left(1 + \frac{x_i}{\gamma}\right) b_e^+ - \frac{1}{\gamma} c_e^+}{\left(b_e^+\right)^2} \cdot a_i^+ - \frac{\left(1 - \frac{x_i}{\gamma}\right) b_e^- + \frac{1}{\gamma} c_e^-}{\left(b_e^-\right)^2} \cdot a_i^- . \quad (6)$$

A native parallelization scheme is to allocate one thread for each net. This scheme has also been discussed in [23], which only demonstrated limited speedup because the maximum number of threads to allocate is $|E|$, and the workload for each thread is imbalanced due to the heterogeneity of net degrees.

**Algorithm 1** Wirelength Forward and Backward Atomic [30]

**Require:** A set of nets $E$, a set of pins $P$, and pin locations $x$;
**Ensure:** Wirelength cost and gradient;
**function** FORWARD($E$, $P$, $x$)
  $\mathbf{x}^+ \leftarrow -\infty, \mathbf{x}^- \leftarrow \infty, \mathbf{b}^\pm \leftarrow 0, \mathbf{c}^\pm \leftarrow 0$;
  **for** each thread $0 \le t < |P|$ **do** ▷ $\mathbf{x}^\pm$ kernel
    Define $e$ as the net that pin $t$ belongs to;
    $x_e^+ \xleftarrow{at.} \max(x_e^+, x_t)$; ▷ atomic max
    $x_e^- \xleftarrow{at.} \min(x_e^-, x_t)$; ▷ atomic min
  **end for**
  **for** each thread $0 \le t < |P|$ **do** ▷ $\mathbf{a}^\pm$ kernel
    Define $e$ as the net that pin $t$ belongs to;
    $a_t^\pm \leftarrow e^{\pm\frac{x_t - x_e^\pm}{\gamma}}$;
  **end for**
  **for** each thread $0 \le t < |P|$ **do** ▷ $\mathbf{b}^\pm$ kernel
    Define $e$ as the net that pin $t$ belongs to;
    $b_e^\pm \xleftarrow{at.} b_e^\pm + a_t^\pm$; ▷ atomic add
  **end for**
  **for** each thread $0 \le t < |P|$ **do** ▷ $\mathbf{c}^\pm$ kernel
    Define $e$ as the net that pin $t$ belongs to;
    $c_e^\pm \xleftarrow{at.} c_e^\pm + x_t a_t^\pm$; ▷ atomic add
  **end for**
  **for** each thread $0 \le t < |E|$ **do** ▷ $\mathrm{WL}_e$ kernel
    Define $e$ as $t^{th}$ net in $E$;
    Compute $\mathrm{WL}_e \leftarrow \frac{c_e^+}{b_e^+} - \frac{c_e^-}{b_e^-}$;
  **end for**
  **return** reduce($\sum_{e\in E} \mathrm{WL}_e$), $\mathbf{a}^\pm$, $\mathbf{b}^\pm$, $\mathbf{c}^\pm$;
**end function**
**function** BACKWARD($E$, $P$, $x$, $\mathbf{a}^\pm$, $\mathbf{b}^\pm$, $\mathbf{c}^\pm$)
  **for** each thread $0 \le t < |P|$ **do** ▷ $\frac{\partial \mathrm{WL}_e}{\partial x_t}$ kernel
    Define $e$ as the net that pin $t$ belongs to;
    Compute $\frac{\partial \mathrm{WL}_e}{\partial x_t}$;
  **end for**
  **return** $\{\frac{\partial \mathrm{WL}_e}{\partial x_i}\}$ $\forall i \in P$;
**end function**

Noting that the total number of pins $|P|$ is much larger than $|E|$, we consider the possibility of pin-level parallelization. The dependency graph for WA wirelength forward and backward is elaborated in Fig. 4(a). A straight-forward implementation of this pin-level parallelism is to compute $\mathbf{a}^\pm$, $\mathbf{b}^\pm$, $\mathbf{c}^\pm$ in separate CUDA kernels by using multiple CUDA streams. The computation can be completed in four steps: 1) compute $\mathbf{x}^\pm$; 2) compute and store $\mathbf{a}^\pm$; 3) compute and store $\mathbf{b}^\pm$, $\mathbf{c}^\pm$; and 4) compute $\mathrm{WL}_e$ in forward or $(\partial \mathrm{WL}_e/\partial x_i)$ in backward. Algorithm 1 illustrates this multistream version of pin-level parallel implementation of WA wirelength forward and backward functions. We make all the CUDA kernel functions inline, which should be separate in practice, for brevity. Specifically, computations for an array with different $\pm$ signs, e.g., $\mathbf{x}^+$ and $\mathbf{x}^-$, are separated into different CUDA streams in the implementation. In the algorithm, six kernels are needed. The $\mathbf{x}^\pm$ kernel requires atomic maximum and minimum operations, and the $\mathbf{b}^\pm$, $\mathbf{c}^\pm$ kernels require atomic addition. At the end of the forward function, summation reduction is needed to compute the overall wirelength cost, which is provided by the deep learning toolkit. In our implementation, multiple CUDA

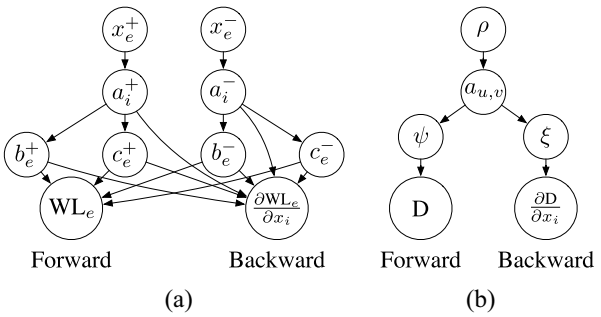



Fig. 4. Forward and backward dependency graph for (a) weighted average wirelength and (b) density computation.

streams are adopted for independent computations, such as $x/y$ directions and positive/negative components.

We observe that Algorithm 1 [30] has several drawbacks: expensive CUDA streams, sequential launches of many kernels, contention, and frequent global memory access. Among these drawbacks, frequent global memory access, especially frequent writing to intermediate variables $\mathbf{x}^{\pm}, \mathbf{a}^{\pm}, \mathbf{b}^{\pm}, \mathbf{c}^{\pm}$, becomes the major runtime bottleneck. In other words, it is memory bounded rather than computation bounded. Thus, we review the natural net-by-net and pin-by-pin approaches again. We discover that the net-by-net strategy has the potential to remove all the intermediate variables by merging the forward and backward functions, as shown in Algorithm 2. Instead of storing $\mathbf{x}^{\pm}, \mathbf{a}^{\pm}, \mathbf{b}^{\pm}, \mathbf{c}^{\pm}$ in global memory, we only create local variables in the kernel function, and directly compute the wirelength for each net and the gradient for each pin. Although variable $\mathbf{a}^{\pm}$ is computed twice, the store instructions only happen to the variables $\mathrm{WL}_e$ and $(\partial \mathrm{WL}_e/\partial x_p)$, which significantly alleviate the memory pressure. The efficiency of the two algorithms is empirically compared in Section IV-B.

For parallel CPU implementation, we adopt the net-by-net strategy and dynamic scheduling for heterogeneous net degrees. We observe that a chunk size of $(|E|/\#\text{threads} \times 16)$ works well for most designs, where $|E|$ is the number of nets in the design.

### B. Density Forward and Backward

Forward and backward of density cost is a computation-intensive procedure. Fig. 4(b) plots the dependency graph for density cost forward and backward. The computation consists of four steps:

1) compute density map $\rho$;
2) compute $a_{u,v}$;
3) compute $\psi$ in forward or $\xi$ in backward;
4) compute D in forward or $(\partial \mathrm{D}/\partial x_i)$ in backward.

We model this computation flow as a dynamic bipartite graph forward and backward process, as shown in Fig. 5. First, density map calculation is modeled as a bipartite graph forward or a special 2-D histogram problem where one cell may update multiple bins [31]. Then the electric potential and field are solved via DCT and other Fourier-related transforms. Finally, the electric force inflicted on each cell is collected from its overlapped bins, which can be modeled as a 2-D gathering problem [31].

**Algorithm 2** Wirelength Forward and Backward Merged

**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 $0 \le t < |E|$ **do** ▷ $\mathrm{WL}_e$, $\frac{\partial \mathrm{WL}_e}{\partial x_p}$ 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
    $\mathrm{WL}_e \leftarrow 0$; ▷ $\mathrm{WL}_e$ 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**

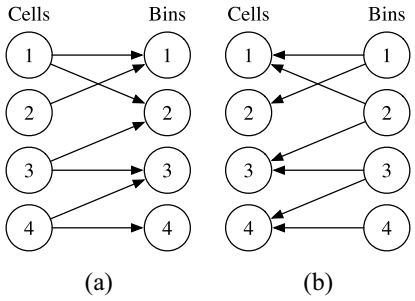



Fig. 5. Computation flow of (a) density map and (b) electric force.

*1) Dynamic Bipartite Graph Forward for Density Map:* Each step of density map computation updates bins based on the overlapping area of corresponding cells. Thus, it can be modeled as a particular 2-D histogram problem or a dynamic bipartite graph forward, as shown in Fig. 5(a). Each edge in the bipartite graph represents an update to the entry of the target bin in the density map, where the edge weight represents the overlapping area of the {cell, bin} pair. The reason why we call it "dynamic" is that, as cells move, edges in the bipartite graph, which indicate overlaps between cells and bins, will change accordingly.

A naive algorithm to parallelize this step is to allocate one GPU thread for each cell and use atomic addition to accumulate the overlapping areas with bins [30]. However, as a cell may cover multiple bins, simply using one GPU thread to update all overlapped bins sequentially will cause load imbalance problem due to the variety in cell sizes. Empirically, the number of bins covered by a cell can vary from ∼10 to ∼1000. This ill-balanced workload within a thread warp introduces a big chunk of idle time and significantly degrades the performance. Therefore, we develop the following techniques to address this issue.

*Sort Cells by Area:* We sort the standard cells by their areas, such that the 32 threads in a warp can process 32 consecutively indexed cells with similar sizes. In this way, the cell-level workloads will be automatically balanced within a warp.

*Update One Cell With Multiple Threads:* We use multiple threads to update a single cell, which can effectively reduce the workload of each thread. Thus, the issue of load imbalance can be further alleviated. An appropriate number of threads need to be selected given that this fine-grained parallelism inevitably introduces some runtime penalty. Specifically, more computational redundancy and memory write contention from atomic operations will happen among threads updating the same cell. We experimentally evaluate different settings of threads. Fig. 6 shows the comparison on the `bigblue4` benchmark. Based on the above results, we empirically adopt $2 \times 2$ threads, i.e., 2 threads for both vertical and horizontal directions. It provides about 20%–30% runtime improvement with both `float32` and `float64`.

For parallel CPU implementation, we adopt the native atomic operations and dynamic scheduling for heterogeneous cell sizes. We set the chunk size to $(|V|/\#\text{threads} \times 16)$, where $|V|$ is the number of cells in the design.

*2) Dynamic Bipartite Graph Backward for Electric Force:* In the electric force computation, each cell receives the forces from the bins it overlaps with. Thus, the computation can be viewed as a 2-D gathering problem or a dynamic bipartite graph backward, as shown in Fig. 5(b). Each edge represents the force from a bin, and the edge weight is the amount of the force. The weight is computed as the product of the overlapping area between the cell and the bin and the electric field at the bin.

A natural strategy to accelerate this step is to allocate one thread for each cell and accumulate the forces sequentially from its overlapping bins [30]. However, considering this computation task shares a similar structure with the density map computation, we borrow the same idea from Section III-B1 by sorting the cells and allocating multiple threads for each cell.

*3) DCT/IDCT for Electric Potential and Field:* The electric potential and field computation in (5) requires fast DCT/IDCT kernels for efficient calculation. The standard DCT/IDCT for 1-D length-$N$ sequence $x$ is

$$\text{DCT}(\{x_n\})_k = \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right) \tag{7a}$$

$$\text{IDCT}(\{x_n\})_k = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi}{N}n\left(k+\frac{1}{2}\right)\right) \tag{7b}$$

where $k = 0, 1, \ldots, N-1$. We further derive IDXST as

$$\text{IDXST}(\{x_n\})_k = \sum_{n=0}^{N-1} x_n \sin\left(\frac{\pi}{N}n\left(k+\frac{1}{2}\right)\right) \tag{8a}$$

$$= (-1)^k \sum_{n=0}^{N-1} x_n (-1)^k \sin\left(\frac{\pi n\left(k+\frac{1}{2}\right)}{N}\right) \tag{8b}$$

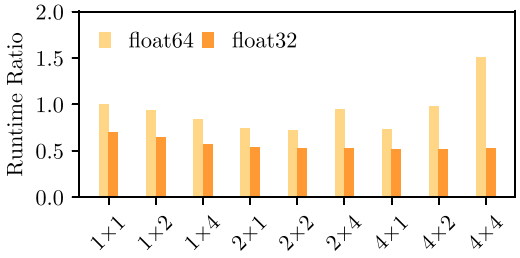



Fig. 6. Comparison of different numbers of threads to update one cell in density forward and backward on `bigblue4`. The numbers are normalized by the runtime of $1 \times 1$ thread with `float64`.

$$= (-1)^k \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi\,(N-n)\left(k+\frac{1}{2}\right)}{N}\right) \tag{8c}$$

$$= (-1)^k \sum_{n=0}^{N-1} x_{N-n} \cos\left(\frac{\pi}{N}n\left(k+\frac{1}{2}\right)\right) \tag{8d}$$

$$= (-1)^k \text{IDCT}(\{x_{N-n}\})_k \tag{8e}$$

where $x_N = 0$. The equality between (8d) and (8e) can be derived by incorporating $x_{N-n}$ into (7b). Given an $M \times M$ density map $\rho$, the electric potential and field can be computed using DCT/IDCT, IDXST routines

$$a_{u,v} = \text{DCT}\big(\text{DCT}(\rho)^{\text{T}}\big)^{\text{T}} \tag{9a}$$

$$\psi_{\text{DCT}} = \text{IDCT}\left(\text{IDCT}\left(\left\{\frac{a_{u,v}}{w_u^2+w_v^2}\right\}\right)^{\text{T}}\right)^{\text{T}} \tag{9b}$$

$$\xi_{\text{DSCT}}^{X} = \text{IDXST}\left(\text{IDCT}\left(\left\{\frac{a_{u,v}w_u}{w_u^2+w_v^2}\right\}\right)^{\text{T}}\right)^{\text{T}} \tag{9c}$$

$$\xi_{\text{DCST}}^{Y} = \text{IDCT}\left(\text{IDXST}\left(\left\{\frac{a_{u,v}w_v}{w_u^2+w_v^2}\right\}\right)^{\text{T}}\right)^{\text{T}} \tag{9d}$$

where $(\cdot)^{\text{T}}$ denotes matrix transposition. The 2-D DCT/IDCT is computed by performing 1-D DCT/IDCT to columns and then rows. We can see all the computations can be broken down into the 1-D DCT/IDCT kernels with proper transformations. Thus, highly optimized DCT/IDCT kernels are critical to the performance.

As the highly optimized fast Fourier transform (FFT) is provided by many deep learning toolkits, we leverage FFT to compute DCT. There are multiple ways to compute DCT using FFT with linear time additional processing. For example, `TensorFlow` adopts the implementation using $2N$-point FFT. We choose the $N$-point FFT implementation [32] and demonstrate better efficiency in the experiments, as shown in Algorithm 3. Due to the symmetric property of FFT for real input sequences, we utilize one-sided real FFT/IFFT to save almost half of the sequence. With additional processing kernels like linear-time reordering and multiplication, DCT/IDCT can be computed with an $N$-point real FFT/IFFT.

In the placement problem, we need to compute 2-D DCT/IDCT. A widely adopted algorithm aforementioned is to perform 1-D DCT/IDCT through the rows and columns sequentially [30]. This row-column DCT algorithm is easy to

**Algorithm 3** DCT/IDCT With $N$-Point FFT

**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
    $x'_t \leftarrow (x_t - jx_{(N-t)})e^{\frac{j\pi t}{2N}}$; ▷ 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**

implement but limited by its two-step procedure, redundant computation, and frequent memory transaction. To achieve better efficiency, we implement 2-D DCT/IDCT directly through 2-D FFT, proven in [32]. Algorithm 4 illustrates the 2-D DCT/IDCT implementation with 2-D preprocessing and post-processing kernels. This implementation eliminates unnecessary computations with a one-time call to 2-D FFT kernels. The pre- and post-processing routines can be fully parallelized. This algorithm is adopted for both GPU and CPU implementations. We evaluate the efficiency of the DCT/IDCT transforms and the density OP in Section IV-B.

### C. Density Weight Updating

We need to update the density weight $\lambda$ in (2) in each iteration to penalize the density cost. RePlAce [8] uses the following equations to update $\lambda$:

$$\mu \leftarrow \begin{cases} \mu_{\max}, & \text{if } p < 0 \\ \max\left(\mu_{\min}, \mu_{\max}^{1-p}\right), & \text{otherwise} \end{cases} \tag{18a}$$

$$\lambda \leftarrow \lambda \cdot \mu \tag{18b}$$

where $\mu_{\min} = 0.95$, $\mu_{\max} = 1.05$, and $p = (\Delta\text{HPWL}/3.5 \times 10^5)$. We follow almost the same scheme with one minor difference. When $p < 0$, we set $\mu \leftarrow \mu_{\max} \cdot \max(0.9999^k, 0.98)$ instead of $\mu_{\max}$, where $k$ is the current iteration. This equation indicates that from iteration 0 to 200, $\mu$ gradually drops from 1.05 to 1.03 and keeps this value afterward, given the previous $\mu_{\max}$ setting. We found that this minor change provides relatively stable convergence in our experiments.

### D. Optimization Engine

ePlace/RePlAce [6], [8] uses Nesterov's method as the gradient-descent solver with a Lipschitz-constant approximation scheme for line search. We implement the same approach in Python leveraging the efficient API provided by the deep learning toolkit. The framework is compatible with other well-known solvers in deep learning toolkits, i.e., various momentum-based gradient descent algorithms like Adam [25] and RMSProp [33], providing additional solver options.

### E. Legalization

We also develop LG as an OP in DREAMPlace. It first follows the Tetris-like procedure similar to NTUplace3 [4]. Then it performs Abacus row-based LG [34]. This step copies the cell locations from GPU to CPU and executes LG purely on CPU because we observe that it only takes several seconds even for million-size designs with a single CPU thread.

### F. Extension to Consider Routability

To optimize routing congestion, we adopt cell inflation to optimize congested regions [35]. We follow a similar scheme to RePlAce [8], which invokes the NCTUgr global router [36] to get the routing overflow map during placement iterations. For each metal layer, we compute the ratio between routing demand and capacity at each routing tile. Then we use the maximum ratio across all layers to compute the inflation ratio for each tile

$$\text{ratio} = \min\left(\left(\max_{\forall l \in L} \frac{\text{demand}_l}{\text{capacity}_l}\right)^{2.5}, 2.5\right) \tag{19}$$

where $L$ is the set of metal layers. The exponent and maximum limits can be adjusted according to the benchmarks. We choose 2.5 in the experiments. After that, we obtain an inflation ratio map. A cell will be inflated according to the inflation ratios of the tiles it overlaps with. If cells inflate too much, there may not be enough total whitespace to digest the area increment. Thus, we limit the area increment to be 10% of the total whitespace area in the layout every time. If the attempted area increment exceeds this ratio, we uniformly scale down the inflation ratio for each cell. During the placement iterations, once the cell overflow drops to 20%, we invoke the global router and perform inflation. The overflow will increase after inflation. Then, the solver is restarted to optimize wirelength and density again. We keep on looping until the total inflation ratio is less than 1% of the total cell area, or we reach a maximum of 5 times of inflation. Starting from the first round of cell inflation, we slow down the density weight updating to make the gradient descent more stable. That is, we update the density weight $\lambda$ every 5 iterations instead of every iteration.

**Algorithm 4** 2-D DCT, 2-D IDCT, IDCT_IDXST, and IDXST_IDCT with $N$-Point 2-D FFT

**Require:** An real $N_1 \times N_2$ matrix $x$; ▷ $N_1$ and $N_2$ can be any positive number

**function** 2D_DCT($x$)

$x' = \texttt{2d\_dct\_preprocess}(x)$ using Equation (10),

$$x'(n_1, n_2) = \begin{cases} x(2n_1, 2n_2), & 0 \leqslant n_1 \leqslant \left\lfloor \frac{N_1-1}{2} \right\rfloor, 0 \leqslant n_2 \leqslant \left\lfloor \frac{N_2-1}{2} \right\rfloor \\ x(2N_1 - 2n_1 - 1, 2n_2), & \left\lfloor \frac{N_1+1}{2} \right\rfloor \leqslant n_1 \leqslant N_1 - 1, 0 \leqslant n_2 \leqslant \left\lfloor \frac{N_2-1}{2} \right\rfloor \\ x(2n_1, 2N_2 - 2n_2 - 1), & 0 \leqslant n_1 \leqslant \left\lfloor \frac{N_1-1}{2} \right\rfloor, \left\lfloor \frac{N_2+1}{2} \right\rfloor \leqslant n_2 \leqslant N_2 - 1 \\ x(2N_1 - 2n_1 - 1, 2N_2 - 2n_2 - 1), & \left\lfloor \frac{N_1+1}{2} \right\rfloor \leqslant n_1 \leqslant N_1 - 1, \left\lfloor \frac{N_2+1}{2} \right\rfloor \leqslant n_2 \leqslant N_2 - 1; \end{cases} \tag{10}$$

$x'' = \texttt{2D\_RFFT}(x')$; ▷ 2D real FFT kernel

**return** $y = \texttt{2d\_dct\_postprocess}(x'')$ using Equation (11),

$$y(n_1, n_2) = 2\Re\left(e^{-\frac{j\pi n_2}{2N_2}}\left(e^{-\frac{j\pi n_1}{2N_1}} x''(n_1, n_2) + e^{\frac{j\pi n_1}{2N_1}} x''(N_1 - n_1, n_2)\right)\right)$$
$$\text{where } x''(N_1, n_2) = x''(n_1, N_2) = 0 \quad \forall n_1, n_2; \tag{11}$$

**end function**

**function** 2D_IDCT($x$)

$x' = \texttt{2d\_idct\_preprocess}(x)$ using Equation (12)

$$x'(n_1, n_2) = e^{-\frac{j\pi n_1}{2N_1}} e^{-\frac{j\pi n_2}{2N_2}} (x(n_1, n_2) - x(N_1 - n_1, N_2 - n_2) - j(x(N_1 - n_1, n_2) + x(n_1, N_2 - n_2))),$$
$$\text{where } \; x(N_1, n_2) = x(n_1, N_2) = 0 \quad \forall n_1, n_2; \tag{12}$$

$x'' = \texttt{2D\_IRFFT}(x')$; ▷ 2D real inverse FFT kernel

**return** $y = \texttt{2d\_idct\_postprocess}(x'') = \texttt{2d\_dct\_preprocess}^{-1}(x'')$ using Equation (13)

$$y(n_1, n_2) = \begin{cases} x''\left(\frac{n_1}{2}, \frac{n_2}{2}\right), & n_1 \text{ is even, } n_2 \text{ iseven} \\ x''\left(N_1 - \frac{n_1+1}{2}, \frac{n_2}{2}\right), & n_1 \text{ is odd, } n_2 \text{ is even} \\ x''\left(\frac{n_1}{2}, N_2 - \frac{n_2+1}{2}\right), & n_1 \text{ is even, } n_2 \text{ is odd} \\ x''\left(N_1 - \frac{n_1+1}{2}, N_2 - \frac{n_2+1}{2}\right), & n_1 \text{ is odd, } n_2 \text{ is odd;} \end{cases} \tag{13}$$

**end function**

**function** IDCT_IDXST($x$)

$x' = \texttt{idct\_idxst\_preprocess}(x)$ using Equation (14)

$$x'(n_1, n_2) = \begin{cases} x(n_1, N_2 - n_2), & n_2 \neq 0, \\ 0, & n_2 = 0; \end{cases} \tag{14}$$

$x'' = \texttt{2D\_IDCT}(x')$;

**return** $y = \texttt{idct\_idxst\_postprocess}(x'')$ using Equation (15)

$$y(n_1, n_2) = (-1)^{n_2} x''(n_1, n_2); \tag{15}$$

**end function**

**function** IDXST_IDCT($x$)

$x' = \texttt{idxst\_idct\_preprocess}(x)$ using Equation (16)

$$x'(n_1, n_2) = \begin{cases} x(N_1 - n_1, n_2), & n_1 \neq 0, \\ 0, & n_1 = 0; \end{cases} \tag{16}$$

$x'' = \texttt{2D\_IDCT}(x')$

**return** $y = \texttt{idxst\_idct\_postprocess}(x'')$ using Equation (17)

$$y(n_1, n_2) = (-1)^{n_1} x''(n_1, n_2); \tag{17}$$

**end function**

### *G. Other Possible Extensions*

The framework is general and can be extended to consider various advanced design objectives and constraints, e.g., timing and fence regions. Timing can be considered by net weighting or additional differentiable timing costs in the objective [29], [37]. Fence regions can be implemented by introducing multiple electric fields, e.g., one for each region, to enable independent spreading between regions.

## IV. Experimental Results

The framework was developed in Python with `PyTorch` for optimizers and API, and C++/CUDA for low-level OPs. The CPU parallelism was implemented with `OpenMP` for wirelength and density OPs. Both the DREAMPlace and the RePlAce [8] programs run on a Linux server with 40-core Intel E5-2698 v4 @ 2.20 GHz and 1 NVIDIA Tesla V100 GPU based on Volta architecture. ISPD 2005

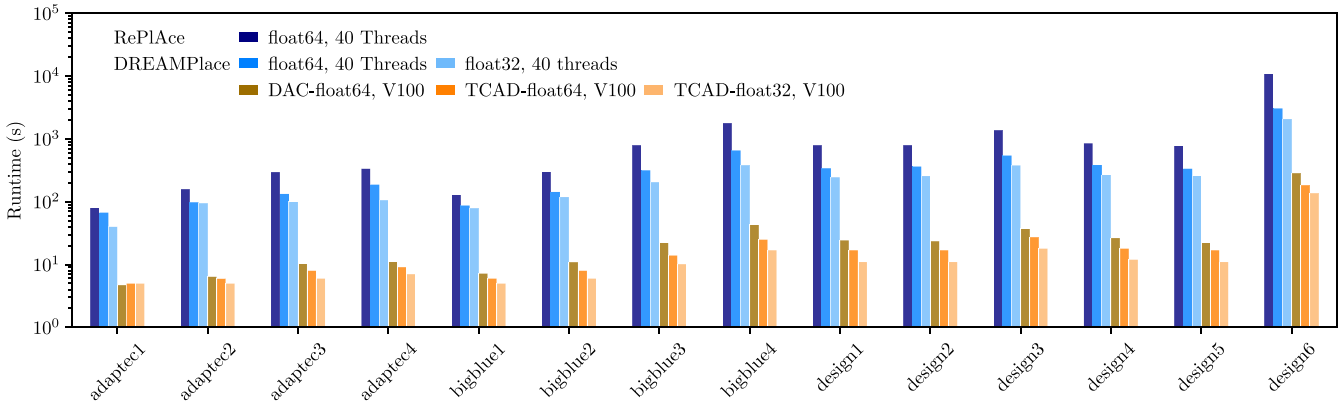



Fig. 7. GP runtime comparison for ISPD2005 and industrial benchmarks between various implementations and precisions. The runtime of `design6` for RePlAce for different number of threads is estimated with the method mentioned in first paragraph of Section IV-A.

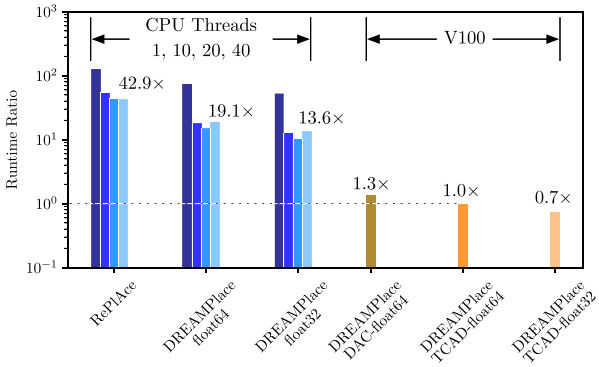



Fig. 8. Average GPU runtime ratio for ISPD2005 and industrial benchmarks with different number of CPU threads. Normalized by the runtime of the TCAD version of DREAMPlace on V100 with `float64`, which is consistent with the ratios in Tables II and III. The normalized ratios for 40 threads and GPUs are annotated for easier comparison.

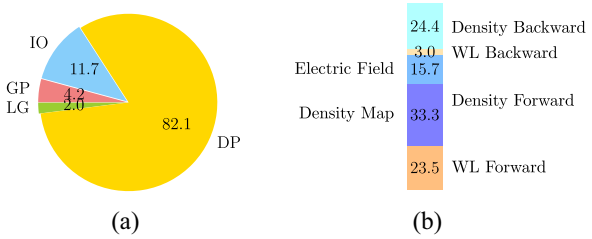



Fig. 9. Runtime breakdown in percentages of DREAMPlace with `float32` on V100 (a) for `bigblue4` and (b) one forward and backward pass in GP.

contest benchmarks [38] and large industrial designs were adopted. We conducted experiments with both double-precision (`float64`) and single-precision (`float32`) floating point numbers on CPU and GPU. We use the same dimensions of bins as RePlAce.

### A. Placement Acceleration

Tables II and III show the HPWL and runtime details on ISPD 2005 and industrial benchmarks. With almost the same solution quality (within 0.3% difference on average), DREAMPlace running on GPU is able to achieve 38× and 47× speedup in GP on the two benchmark suites compared to RePlAce with 40 threads. DREAMPlace running on CPU is also 2× faster than RePlAce with 40 threads in GP. RePlAce [8] crashed on the 10-million-cell industrial benchmark at the 6th iteration for Nesterov's optimization. The potential reason is that the peak memory usage of RePlAce exceeded the maximum memory (64 GB). Before crashing, it took 3396 s for initial placement and on average 7.5 s for

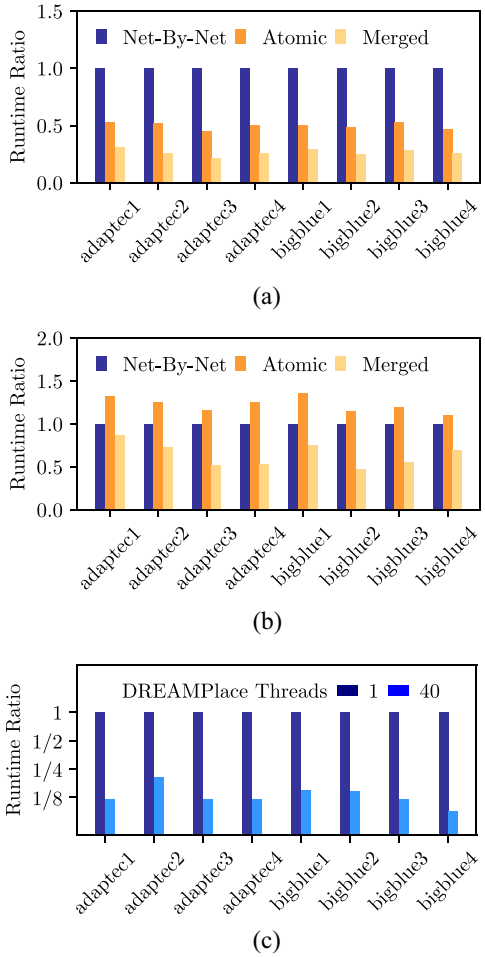



Fig. 10. Wirelength forward and backward with `float32`. (a) GPU runtime comparison of different implementations. (b) CPU runtime comparison of different implementations with 40 threads. (c) CPU runtime comparison of the net-by-net strategy between single thread and 40 threads.

TABLE II
EXPERIMENTAL RESULTS ON ISPD 2005 BENCHMARKS [38] WITH FLOAT64

| Design | #cells | #nets | RePlAce (40 threads) | | | | | DREAMPlace (40 threads) | | | | | | DREAMPlace (V100) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | HPWL | Runtime (s) | | | | HPWL | Runtime (s) | | | | | HPWL | Runtime (s) | | | | |
| | | | | GP | LG | DP | Total | | GP | LG | DP | IO | Total | | GP | LG | DP | IO | Total |
| adaptec1 | 211 | 221 | 73.22 | 80 | 4 | 21 | 112 | 73.22 | 67 | 0.4 | 24 | 4 | 96 | 73.22 | 5 | 0.5 | 25 | 4 | 34 |
| adaptec2 | 255 | 266 | 81.86 | 159 | 7 | 27 | 201 | 82.23 | 98 | 0.5 | 31 | 5 | 134 | 82.22 | 6 | 0.5 | 31 | 5 | 42 |
| adaptec3 | 452 | 467 | 193.34 | 297 | 20 | 48 | 378 | 193.81 | 133 | 1 | 57 | 9 | 201 | 193.72 | 8 | 1 | 57 | 9 | 76 |
| adaptec4 | 496 | 516 | 175.25 | 336 | 20 | 55 | 426 | 173.85 | 187 | 2 | 65 | 10 | 264 | 174.08 | 9 | 2 | 65 | 9 | 85 |
| bigblue1 | 278 | 284 | 89.87 | 130 | 4 | 27 | 170 | 89.40 | 87 | 0.3 | 32 | 5 | 125 | 89.38 | 6 | 0.4 | 31 | 6 | 43 |
| bigblue2 | 558 | 577 | 138.07 | 299 | 22 | 82 | 419 | 136.73 | 143 | 9 | 91 | 10 | 254 | 136.54 | 8 | 9 | 95 | 10 | 123 |
| bigblue3 | 1097 | 1123 | 305.09 | 787 | 41 | 120 | 1030 | 303.89 | 316 | 3 | 142 | 21 | 484 | 303.90 | 14 | 3 | 142 | 20 | 180 |
| bigblue4 | 2177 | 2230 | 743.80 | 1789 | 51 | 299 | 2400 | 743.69 | 655 | 9 | 336 | 45 | 1047 | 743.75 | 25 | 9 | 332 | 45 | 413 |
| ratio | - | - | 1.003 | 38.2 | 10.1 | 0.9 | 4.6 | 1.000 | 18.7 | 0.9 | 1.0 | 1.0 | 2.7 | 1.000 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

TABLE III
EXPERIMENTAL RESULTS ON INDUSTRIAL BENCHMARKS WITH FLOAT64

| Design | #cells | #nets | RePlAce (40 threads) | | | | | DREAMPlace (40 threads) | | | | | | DREAMPlace (V100) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | HPWL | Runtime (s) | | | | HPWL | Runtime (s) | | | | | HPWL | Runtime (s) | | | | |
| | | | | GP | LG | DP | Total | | GP | LG | DP | IO | Total | | GP | LG | DP | IO | Total |
| design1 | 1345 | 1389 | 340.76 | 787 | 39 | 140 | 1039 | 340.64 | 341 | 4 | 173 | 30 | 549 | 340.67 | 17 | 4 | 172 | 30 | 224 |
| design2 | 1306 | 1355 | 274.65 | 793 | 39 | 134 | 1057 | 275.41 | 363 | 4 | 166 | 30 | 564 | 275.36 | 17 | 5 | 167 | 29 | 218 |
| design3 | 2265 | 2276 | 524.36 | 1369 | 74 | 233 | 1777 | 522.68 | 543 | 14 | 299 | 48 | 906 | 522.62 | 27 | 14 | 302 | 48 | 393 |
| design4 | 1525 | 1528 | 454.86 | 857 | 48 | 166 | 1136 | 453.86 | 384 | 8 | 200 | 33 | 626 | 453.83 | 18 | 8 | 202 | 33 | 262 |
| design5 | 1316 | 1364 | 287.46 | 776 | 38 | 138 | 1016 | 287.14 | 335 | 3 | 167 | 29 | 535 | 287.11 | 17 | 4 | 169 | 31 | 221 |
| design6 | 10504 | 10747 | NA | ~10896 | NA | NA | NA | 2360.94 | 3056 | 77 | 1650 | 246 | 5037 | 2358.44 | 181 | 76 | 1666 | 253 | 2184 |
| ratio | - | - | 1.001 | 47.3 | 8.1 | 0.8 | 4.6 | 1.000 | 19.9 | 1.0 | 1.0 | 1.0 | 2.4 | 1.000 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

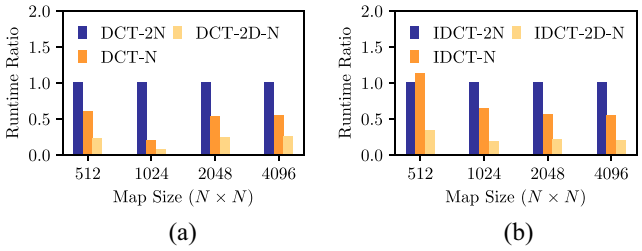



Fig. 11. GPU runtime comparison of (a) DCT and (b) IDCT algorithms with `float32`.

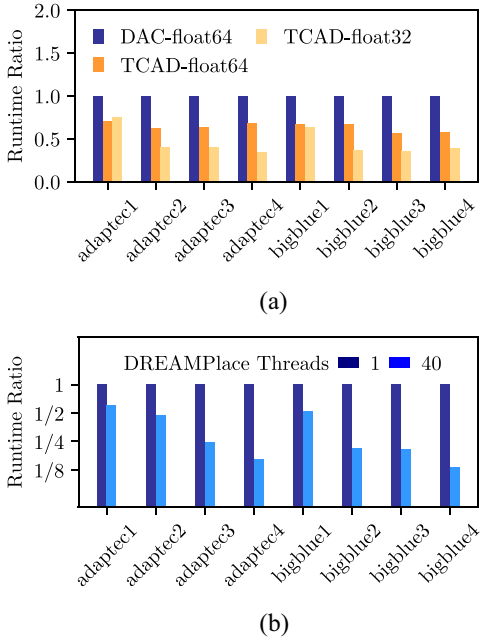



Fig. 12. Density forward and backward comparison. (a) GPU runtime comparison between DAC [30] and this extension. (b) CPU runtime comparison between single thread and 40 threads with `float32`.

each Nesterov iteration. As this benchmark takes 1000 iterations with DREAMPlace, we made a runtime estimation of $3396 + 1000 \times 7.5 \approx 10896$ s. Meanwhile, among all RePlAce runs, initial placement takes 25%–30% of the entire GP time, and solving the nonlinear placement takes around 70% $\sim$ 75%. The LG of DREAMPlace is also around 10$\times$ faster than the NTUplace3 legalizer in the RePlAce flow. As NTUplace3 does the DP for both placers, so the runtime is similar. The speedup for the entire placement flow on GPU is 4.6$\times$, and that on CPU is 2.7$\times$.

Fig. 7 plots the GP runtime comparison between multithreaded DREAMPlace and RePlAce with different precisions and implementations. It can be seen that the parallel CPU version of DREAMPlace is consistently faster than RePlAce. Meanwhile, this TCAD extension further improves the efficiency of the GPU implementations from the DAC version [30] except for the smallest benchmark `adaptec1`. Fig. 8 plots the average runtime ratio for different cases. By switching from `float64` to `float32`, an average speedup of 1.4$\times$ on CPU and 1.3$\times$ on GPU can be achieved, while the quality stays almost the same. Compared with the previous DAC version [30], this extension achieves 1.3$\times$ speedup with `float64` and 1.8$\times$ speedup with `float32`. From Fig. 8, we also observe that the speedup of CPU implementations saturates quickly from single thread to 40 threads. This observation holds for both RePlAce and DREAMPlace. For RePlAce, the best number of threads is 40 with a speedup of 3.2$\times$, while for DREAMPlace, 20 threads provide the best efficiency with a factor of 5.0$\times$.

Fig. 9 draws the runtime breakdown of DREAMPlace on a 2-million-cell design `bigblue4`, where GP and LG only take 6.2% runtime of the entire flow. The runtime of GP and LG is even less than that of file IO for benchmark reading and writing. The majority of the runtime (82%) is taken by

DP, which still relies on the external placer currently. Previous studies [39], [40] have demonstrated more than 6× speedup from GPU acceleration for DP over multithreaded CPU. While DP is not the focus of this article, there is a potential of 18× speedup for the entire placement by future incorporation of GPU-accelerated DP, e.g., $(2400/25 + 9 + 332/6 + 45) \sim 18$ for `bigblue4` according to Table II. On the other hand, within each forward and backward pass of GP, the density-related computation takes longer than wirelength (73.4% versus 26.5%). With efficient DCT/IDCT implementation, the electric field computation is no longer the bottleneck for density forward and backward.

TABLE IV
COMPARISON WITH NATIVE PYTORCH SOLVERS LIKE ADAM [25] AND SGD WITH MOMENTUM WITH FLOAT64 ON GPU

| Design | Nesterov [8] | | Adam | | | SGD Momentum | | |
|---|---|---|---|---|---|---|---|---|
| | HPWL | GP (s) | HPWL | GP (s) | LR Decay | HPWL | GP (s) | LR Decay |
| adaptec1 | 73.22 | 5 | 73.02 | 8 | 0.995 | 73.84 | 8 | 0.993 |
| adaptec2 | 82.22 | 6 | 82.44 | 7 | 0.995 | 83.72 | 9 | 0.993 |
| adaptec3 | 193.72 | 8 | 191.22 | 12 | 0.995 | 198.07 | 12 | 0.993 |
| adaptec4 | 174.08 | 9 | 172.84 | 13 | 0.995 | 175.77 | 14 | 0.993 |
| bigblue1 | 89.38 | 6 | 89.89 | 10 | 0.995 | 89.64 | 9 | 0.993 |
| bigblue2 | 136.54 | 8 | 136.43 | 14 | 0.995 | 137.48 | 14 | 0.993 |
| bigblue3 | 303.90 | 14 | 302.95 | 33 | 0.997 | 312.79 | 24 | 0.995 |
| bigblue4 | 743.75 | 25 | 740.60 | 66 | 0.997 | 744.55 | 57 | 0.995 |
| ratio | 1.000 | 1.000 | 0.997 | 1.781 | | 1.012 | 1.687 | |

### *B. Acceleration of Low-Level Operators*

We further investigate the efficiency of the low-level OPs, e.g., wirelength forward and backward, DCT/IDCT, and density forward and backward. Fig. 10 compares three approaches discussed in Section III-A. "Net-by-Net" denotes the net-level parallelization; "Atomic" denotes the pin-level parallelization with atomic operations in Algorithm 1 [30]; "Merged" denotes the combined forward and backward implementation in Algorithm 2. When using `float32` on GPU, the merged approach achieves 3.7× speedup over the net-by-net one and 1.8× speedup over the atomic one. On CPU, the atomic strategy is 20% slower than the net-by-net strategy with 40 threads, while the merged strategy is over 30% faster. Meanwhile, a promising speedup factor of 7.5× from a single thread to 40 threads can be achieved with the net-by-net strategy.

Fig. 11 compares the 2-D DCT/IDCT implementation using 2*N*-point FFT ("DCT-2N" and "IDCT-2N"), *N*-point FFT ("DCT-N" and "IDCT-N"), and *N*-point 2-D FFT ("DCT-2D-N" and "IDCT-2D-N") [32]. Considering the map sizes in the experiment (from $512 \times 512$ to $4096 \times 4096$) with `float32`, the *N*-point DCT implementation is 2.1× faster [30] and the *N*-point 2-D implementation can be 5.0× faster. For IDCT, the *N*-point implementation achieves 1.3× speedup and the 2-D implementation achieves 4.1× speedup. This result demonstrates the efficiency of Algorithm 4.

As DCT/IDCT is used in the density OP, in Fig. 12, the efficiency of the entire density forward and backward procedure is compared for GPU and CPU implementations. With all the speedup techniques, an average of $1.5\times - 2.1\times$ speedup on GPU can be achieved with the current implementation over the preliminary DAC version [30]. For the parallel CPU implementation, 3.1× runtime reduction can be achieved with 40 threads.

### *C. Comparison With Solvers in `PyTorch`*

As mentioned, DREAMPlace can enable easy adoption of native solvers in `PyTorch`. Here, we compare with the widely used solvers implemented in the toolkit, like Adam [25] and stochastic gradient descent (SGD) with momentum, as shown in Table IV. As these solvers do not have line search, we add simple learning rate decay in each iteration to control the step size of gradient descent with the decay factor shown in the "LR Decay" columns. We use the default configurations for these solvers and report the final HPWL after DP and the runtime for GP in seconds. In our experiments, we find the gradient descent process may be unable to converge if the learning rate is not properly designed. Therefore, we customize the decay factor for each design. It can be seen that Adam can achieve slightly better results than the Nesterov's accelerated gradient decent method (shortened to Nesterov's method for brevity) implemented in RePlAce, while the Nesterov's method converges much faster. Meanwhile, the results for SGD with momentum are about 1.2% worse. As the solvers have many parameters to tune, it is hard to simply conclude that Adam or SGD with momentum is definitely worse than the Nesterov's method with the experiments, but the preliminary results are at least promising enough to worth further exploration. With the DREAMPlace framework, we can investigate new solvers easily by scripting in `PyTorch`.

### *D. Routability-Driven Placement*

To verify the runtime benefits in routability-driven placement, we conducted experiments on the DAC 2012 contest benchmarks [41]. We consider two major metrics for solution quality: 1) "sHPWL" as scaled wirelength and 2) "RC" as routing congestion. In the contest, the RC is defined as a weighted average of overflows in the top 0.5%, 1%, 2%, 5% congested tiles. The minimum value for RC is 100, indicating no overflow. The sHPWL is computed using the following equation [41]:

$$\text{sHPWL} = \text{HPWL} \times (1 + 0.03 \times (\text{RC} - 100)) \tag{20}$$

indicating that unit increase in routing congestion is counted as 3% HPWL overhead.

In this experiment, we obtained the RePlAce binary from Cheng *et al.* [8] to keep consistent experimental settings. Table V shows the solution quality and runtime. As NCTUgr is repeatedly invoked as an external congestion estimator and it only runs on CPU with single-thread, we separate the runtime of GP into two parts: 1) nonlinear optimization ("NL") and 2) global routing ("GR"). NTUplace3 [4] is adopted as the LG and DP for RePlAce, and DP for DREAMPlace. We can see that DREAMPlace with GPU acceleration can provide very similar solution quality, while 20× faster in NL and 9× faster in GP including the runtime of the global router. For the entire placement flow, we can achieve 5× speedup. DREAMPlace also shows compelling efficiency and quality with 40 threads on CPU. We also observe that DREAMPlace invokes the global router less often than RePlAce, leading

TABLE V
EXPERIMENTAL RESULTS ON DAC 2012 BENCHMARKS [41] FOR ROUTABILITY-DRIVEN PLACEMENT

| Design | #nodes | #nets | RePlAce† | | | | | | | DREAMPlace (40 threads) | | | | | | | DREAMPlace (RTX 2080TI) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | sHPWL | RC | Runtime (s) | | | | | sHPWL | RC | Runtime (s) | | | | | sHPWL | RC | Runtime (s) | | | | |
| | | | | | GP | | LG | DP | Total | | | GP | | LG | DP | Total | | | GP | | LG | DP | Total |
| | | | | | NL | GR | | | | | | NL | GR | | | | | | NL | GR | | | |
| SB2 | 1014K | 991K | 62.39 | 102.47 | 6981 | 2168 | 46 | 160 | 9382 | 61.06 | 101.57 | 3953 | 1200 | 30 | 183 | 5390 | 61.20 | 101.76 | 293 | 1215 | 31 | 184 | 1746 |
| SB3 | 920K | 898K | 30.69 | 100.81 | 2354 | 969 | 70 | 149 | 3565 | 30.18 | 100.73 | 3306 | 524 | 16 | 172 | 4040 | 30.18 | 100.65 | 131 | 485 | 16 | 182 | 835 |
| SB6 | 1014K | 1007K | 31.30 | 100.61 | 1874 | 548 | 44 | 144 | 2634 | 30.92 | 100.26 | 1888 | 309 | 27 | 168 | 2414 | 31.00 | 100.33 | 169 | 309 | 27 | 168 | 694 |
| SB7 | 1365K | 1340K | 37.20 | 101.13 | 2068 | 438 | 54 | 201 | 2794 | 36.73 | 100.60 | 963 | 144 | 23 | 234 | 1395 | 36.73 | 100.61 | 78 | 143 | 23 | 233 | 509 |
| SB9 | 847K | 834K | 21.48 | 101.09 | 1866 | 369 | 23 | 148 | 2426 | 21.21 | 100.61 | 677 | 87 | 11 | 170 | 964 | 21.23 | 100.65 | 54 | 83 | 11 | 171 | 337 |
| SB11 | 955K | 936K | 34.28 | 102.65 | 2676 | 549 | 28 | 108 | 3385 | 32.86 | 100.86 | 1218 | 214 | 24 | 125 | 1602 | 32.80 | 100.79 | 150 | 283 | 24 | 125 | 603 |
| SB12 | 1293K | 1293K | 26.69 | 103.02 | 3040 | 441 | 153 | 230 | 3898 | 26.90 | 101.25 | 2767 | 319 | 5 | 278 | 3398 | 26.38 | 100.72 | 171 | 398 | 5 | 278 | 883 |
| SB14 | 635K | 620K | 21.26 | 100.75 | 740 | 188 | 22 | 87 | 1052 | 21.25 | 100.55 | 1067 | 146 | 15 | 104 | 1345 | 21.24 | 100.51 | 65 | 148 | 15 | 108 | 349 |
| SB16 | 699K | 697K | 25.57 | 102.29 | 1669 | 539 | 16 | 91 | 2331 | 25.42 | 101.77 | 649 | 119 | 2 | 105 | 891 | 25.53 | 101.94 | 44 | 115 | 2 | 106 | 283 |
| SB19 | 523K | 512K | 14.21 | 101.05 | 1288 | 257 | 17 | 110 | 1685 | 15.10 | 103.28 | 701 | 108 | 1 | 126 | 948 | 14.67 | 102.73 | 71 | 57 | 1 | 133 | 274 |
| ratio | | | 1.010 | 1.005 | 21.6 | 2.7 | 6.2 | 0.8 | 5.4 | 1.004 | 1.001 | 14.0 | 1.1 | 1.0 | 1.0 | 3.4 | 1.000 | 1.000 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

Both results for RePlAce and DREAMPlace are collected from a Linux machine with two 20-core Intel Xeon Gold 6230 CPUs (40 cores in total) and 1 NVIDIA RTX 2080TI GPU.
† We obtain the binary of RePlAce [8] to keep consistent experimental settings for this benchmark suite. As the RePlAce binary uses `float32` for nonlinear placement, we use the same setting for DREAMPlace in this experiment. The binary also only supports single-thread and the external global router NCTUgr is also single-thread.

to shorter GR time. Meanwhile, for DREAMPlace, GR takes around 70% of the GP time, which is the runtime bottleneck.

## V. CONCLUSION

In this article, we take a new perspective on solving classical analytical placement by casting it into a neural network training problem. Leveraging the deep learning toolkit `PyTorch`, we develop a new open-source placement engine, *DREAMPlace* with GPU acceleration. It achieves around 40× speedup in GP without quality degradation for academic and industrial benchmarks, compared to the state-of-the-art RePlAce running on many threads. We explore different implementations of low-level OPs for forward and backward propagation to boost the overall efficiency.

Furthermore, DREAMPlace is highly extensible to incorporate new algorithms/solvers and new objectives by simply writing high-level programming languages such as `Python`. We plan to further investigate cell inflation for routability and net weighting for timing optimization [29], [35], [37] as well as GPU-accelerated DP. It can also be extended to leverage multi-GPU platforms for further speedup. Meanwhile, we plan to investigate the efficiency of implementations using fixed-point numbers to guarantee run-to-run determinism. As DREAMPlace decouples the high-level algorithmic design and low-level acceleration efforts, we believe this work shall open up new directions for revisiting classical EDA problems.

## ACKNOWLEDGMENT

The authors would like to thank Lutong Wang and Ilgweon Kang from the University of California at San Diego for preparing the RePlAce binary, suggestions on the experimental setups, and verifying the results.

## REFERENCES

[1] A. B. Kahng, S. Reda, and Q. Wang, "Architecture and details of a high quality, large-scale analytical placer," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2005, pp. 891–898.

[2] T. Chan, J. Cong, and K. Sze, "Multilevel generalized force-directed method for circuit placement," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2005, pp. 185–192.

[3] A. B. Kahng and Q. Wang, "A faster implementation of APlace," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2006, pp. 218–220.

[4] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUPlace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1228–1240, Jul. 2008.

[5] M.-K. Hsu *et al.*, "NTUplace4h: A novel routability-driven placement algorithm for hierarchical mixed-size circuit designs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 33, no. 12, pp. 1914–1927, Dec. 2014.

[6] 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.

[7] 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.

[8] C. 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.

[9] Z. Zhu, J. Chen, Z. Peng, W. Zhu, and Y.-W. Chang, "Generalized augmented Lagrangian and its applications to VLSI global placement," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2018, pp. 1–6.

[10] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in *Proc. IEEE/ACM Asia South Pac. Design Autom. Conf. (ASPDAC)*, 2007, pp. 135–140.

[11] X. He, T. Huang, L. Xiao, H. Tian, and E. F. Y. Young, "Ripple: A robust and effective routability-driven placer," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 10, pp. 1546–1556, Oct. 2013.

[12] T. Lin, C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: Placement based on novel rough legalization and refinement," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2013, pp. 357–362.

[13] T. Lin, C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: A high performance mixed-size Wirelengh-driven placer with density constraints," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 3, pp. 447–459, Mar. 2015.

[14] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 1, pp. 50–60, Jan. 2012.

[15] M.-C. Kim, N. Viswanathan, C. J. Alpert, I. L. Markov, and S. Ramji, "MAPLE: multilevel adaptive placement for mixed-size designs," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2012, pp. 193–200.

[16] T. Lin, C. Chu, and G. Wu, "POLAR 3.0: An ultrafast global placement engine," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2015, pp. 520–527.

[17] W. Li, Y. Lin, and D. Z. Pan, "elfPlace: Electrostatics-based placement for large-scale heterogeneous FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2019, pp. 1–8.

[18] *Cadence Innovus*. Accessed: Jun. 1, 2020. [Online]. Available: http://www.cadence.com

[19] *Synopsys IC Compiler*. Accessed: Jun. 1, 2020. [Online]. Available: http://www.synopsys.com

[20] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *Proc. ACM Symp. FPGAs*, 2008, pp. 14–23.

[21] W. Li, M. Li, J. Wang, and D. Z. Pan, "UTPlaceF 3.0: A parallelization framework for modern FPGA global placement," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2017, pp. 922–928.

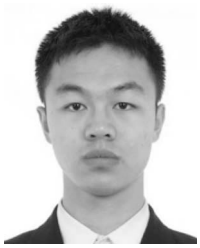
[22] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2009, pp. 681–688.

[23] C.-X. Lin and M. D. Wong, "Accelerate analytical placement with GPU: A generic approach," in *Proc. IEEE/ACM Design Autom. Test Europe (DATE)*, 2018, pp. 1345–1350.
[24] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2019, pp. 8024–8035.
[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–6.
[26] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, vol. 1. Cambridge, MA, USA: MIT Press, 2016.
[27] M.-K. Hsu, Y.-W. Chang, and V. Balabanov, "TSV-aware analytical placement for 3D IC designs," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2011, pp. 664–669.
[28] 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," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, vol. 32, 2013, pp. 497–509.
[29] W. C. Naylor, R. Donelly, and L. Sha, "Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer," U.S. Patent 6 301 693, Oct. 2001.
[30] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled text GPU acceleration for modern VLSI placement," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2019, pp. 1–6.
[31] K. A. Berman and J. Paul, *Fundamentals of Sequential and Parallel Algorithms*, 1st ed. Boston, MA, USA: PWS, 1996.
[32] J. Makhoul, "A fast cosine transform in one and two dimensions," *IEEE Trans. Signal Process.*, vol. SP-28, no. 1, pp. 27–34, Feb. 1980.
[33] F. Zou, L. Shen, Z. Jie, W. Zhang, and W. Liu, "A sufficient condition for convergences of Adam and RMSProp," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 11127–11135.
[34] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Abacus: Fast legalization of standard cell circuits with minimal movement," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2008, pp. 47–53.
[35] T. F. Chan, K. Sze, J. R. Shinnerl, and M. Xie, "mPL6: Enhanced multilevel mixed-size placement with congestion control," in *Modern Circuit Placement*, G. J. Nam, and J. Cong, Eds. Boston, MA, USA: Springer, 2007, pp. 247–288, doi: 10.1007/978-0-387-68739-1_10.
[36] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "NCTU-GR 2.0: multithreaded collision-aware global routing with bounded-length maze routing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 5, pp. 709–722, Mar. 2013.
[37] A. B. Kahng and Q. Wang, "An analytic placer for mixed-size placement and timing-driven placement," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2004, pp. 565–572.
[38] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ISPD2005 placement contest and benchmark suite," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2005, pp. 216–220.
[39] S. Dhar and D. Z. Pan, "GDP: GPU accelerated detailed placement," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2018, pp. 1–7.
[40] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "ABCDPlace: Accelerated batch-based concurrent detailed placement on multi-threaded CPUs and GPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Feb. 4, 2020, doi: 10.1109/TCAD.2020.2971531.
[41] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei, "The DAC 2012 routability-driven placement contest and benchmark suite," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2012, pp. 774–782.

**Yibo Lin** (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiao Tong 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.

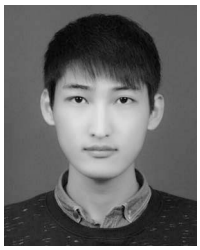
He is currently an Assistant Professor with the Computer Science Department, Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.

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.



**Zixuan Jiang** (Graduate Student Member, IEEE) received the B.E. degree in electronic information engineering from Zhejiang University, Hangzhou, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA.

His current research interests involve machine learning frameworks and applications, and physical design algorithms and implementations.



**Jiaqi Gu** (Graduate Student Member, IEEE) received the B.E. degree in microelectronic science and engineering from Fudan University, Shanghai, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA, under the supervision of Prof. D. Z. Pan.

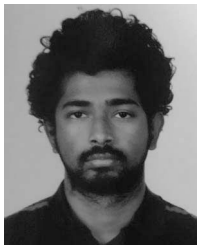
His current research interests include machine learning, algorithm and architecture design, optical neuromorphic computing for AI acceleration, and GPU acceleration for VLSI physical design automation.

Dr. Gu has received the Best Paper Reward at ASP-DAC 2020.

**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.

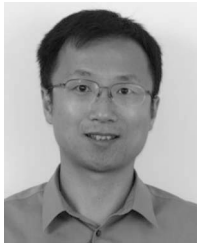
He is currently a Staff Software Engineer with the Vivado Implementation Team, Xilinx, San Jose, CA, USA, where he is primarily working on the physical synthesis field.

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.



**Shounak Dhar** (Member, IEEE) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology Bombay, Mumbai, India, in 2014, and the Ph.D. degree in electrical and computer engineering from the University of Texas at Austin, Austin, TX, USA, in 2019.

He is currently working with Intel Corporation, Santa Clara, CA, USA, on EDA algorithms in Intel's FPGA design implementation tool. His research interests include electronic design automation and hardware acceleration.



**Haoxing Ren** (Senior Member, IEEE) received the B.S./M.S. degrees in electrical engineering from Shanghai Jiao Tong University, Shanghai, China, the M.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, and the Ph.D. degree in computer engineering from the University of Texas at Austin, Austin, TX, USA.

From 2000 to 2006, he was a Software Engineer with IBM Microelectronics, Armonk, NY, USA. From 2007 to 2015, he was a Research Staff Member with IBM T. J. Watson Research Center, Ossining, NY, USA. From 2015 to 2016, he was a Technical Executive with Suzhou PowerCore Technology, Austin. He is currently a Principal Research Scientist with NVIDIA, Austin. He holds over 20 patents and coauthored more than 40 papers including several book chapters in physical design and logic synthesis. His research interests are machine learning applications in design automation and GPU accelerated EDA.

Dr. Ren received many IBM technical achievement rewards including the IBM Corporate Award for his work on improving microprocessor design productivity. He has received the Best Paper Awards at ISPD'13 and DAC'19.

**Brucek Khailany** (Senior Member, IEEE) received the B.S.E. degree in electrical engineering from the University of Michigan at Ann Arbor, Ann Arbor, MI, USA, in 1997, and the Ph.D. degree from Stanford University, Stanford, CA, USA, in 2003.

In 2009, he joined NVIDIA, where he is currently the Director of the ASIC and VLSI Research group. He leads research into innovative design methodologies for integrated circuit development, machine learning (ML) and GPU-assisted electronic design automation algorithms, and energy-efficient ML accelerators. From 2004 to 2009, he was a Co-Founder and the Principal Architect with Stream Processors, Inc., Sunnyvale, CA, USA, where he led research and development activities related to parallel processor architectures. Over 10 years at NVIDIA, he has contributed to many projects in research and product groups spanning computer architecture and VLSI design.

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

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 375 journal articles and refereed conference papers, and is the holder of 8 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, DAC Top 10 Author in Fifth Decade, DAC Prolific Author Award, ASP-DAC Frequently Cited Author Award, 19 Best Paper Awards at premier venues (ISPD 2020, 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, SRC Techcon in 1998, 2007, 2012, and 2015) and 15 additional Best Paper Award finalists, Communications of the ACM Research Highlights in 2014, UT Austin RAISE Faculty Excellence Award in 2014, the Cadence Academic Collaboration Award in 2019, and many international CAD contest awards, among others. He has served as a Senior Associate Editor for the *ACM Transactions on Design Automation of Electronic Systems*, an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART I: REGULAR PAPERS, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: EXPRESS BRIEFS, IEEE DESIGN & TEST, *Science China Information Sciences*, and *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 ASPDAC 2017 Program Chair, the ICCAD 2019 General Chair, the DAC 2014 Tutorial Chair, and the ISPD 2008 General Chair. He is a Fellow of SPIE.