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

Yibo Lin
ECE Department, UT Austin
yibolin@utexas.edu

Shounak Dhar
ECE Department, UT Austin
shounak.dhar@utexas.edu

Wuxi Li
ECE Department, UT Austin
wuxi.li@utexas.edu

Haoxing Ren
Nvidia, Inc., Austin
haoxingr@nvidia.com

Brucek Khailany
Nvidia, Inc., Austin
bkhailany@nvidia.com

David Z. Pan
ECE Department, UT Austin
dpan@ece.utexas.edu

ABSTRACT

Placement for very-large-scale integrated (VLSI) circuits is one of the most important steps for design closure. This paper proposes 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 over 30× speedup in global placement without quality degradation compared to the state-of-the-art multi-threaded placer RePlAce. We believe this work shall open up new directions for revisiting classical EDA problems with advancement in AI hardware and software.

1 INTRODUCTION

Placement is a critical but time-consuming step in the 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. The commercial design flows often run core placement engines many times to achieve design closure. As placement involves large-scale numerical optimization, current placement usually takes hours for large designs, thus, slowing down design iterations. Therefore, ultra-fast yet high quality placement is always in strong demand.

Analytical placement is the current state-of-the-art for VLSI placement [1–7]. It essentially solves a nonlinear optimization problem. Although analytical placement can produce high-quality solutions, it is also known to be relatively slow [8–11]. 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.

To accelerate placement, existing parallelization efforts have mostly targeted multi-threaded CPUs using partitioning [11–13]. As the number of threads increases, speedup quickly saturates at around 5× in global placement with typical quality degradation of 2-6%. Cong et al. explored GPU acceleration for analytical placement [14]. They combined clustering and declustering with nonlinear placement optimization. By parallelizing the nonlinear placement part, an average of 15× speedup in global placement was reported with less than 1% quality degradation. Lin et al. proposed GPU acceleration techniques for wirelength gradient computation and area accumulation [15], but their experiment failed to consider real operations such as density cost computation 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 systematically validate new algorithms.

In this work, we propose *DREAMPlace*, a GPU-accelerated analytical placer developed with deep learning toolkit `PyTorch` 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 [4, 6], but the framework is designed in a generic way that is compatible with other analytical placers such as NTUplace [2]. The key contributions are summarized as follows.

- We take a totally new perspective of making 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.
- We propose efficient GPU implementations of key kernels in analytical placement like wirelength and density computation.
- We demonstrate over 30× speedup in global placement and legalization without quality degradation of the entire placement flow over multi-threaded CPU implementations. More specifically, a design with one million cells finishes in one minute. 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 rest of the paper is organized as follows. Section 2 describes the background and motivation; Section 3 explains the detailed implementation; Section 4 demonstrates the results; Section 5 concludes the paper.

2 PRELIMINARIES

This section will review the background and motivation.

2.1 Analytical Placement

Analytical placement usually consists of three steps: global placement (GP), legalization (LG), and detailed placement (DP). Global placement spreads out cells in the layout with a target cost minimized; legalization removes the remaining overlaps between cells and aligns cells to placement sites; detailed placement performs incremental refinement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317803

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

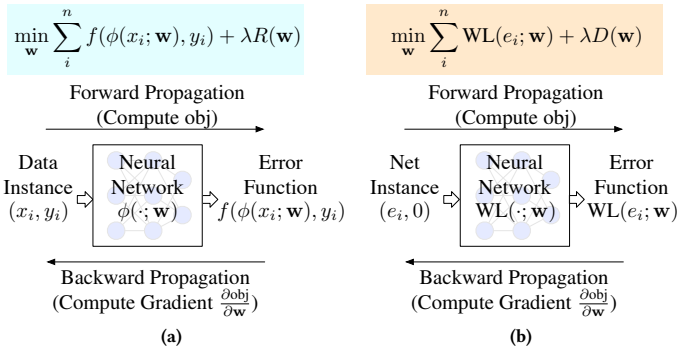

Figure 1: Analogy between neural network training and analytical placement. (a) Train a network for weights $\mathbf{w}$. (b) Solve a placement for cell locations $\mathbf{w} = (\mathbf{x}, \mathbf{y})$.

to further improve the quality. Usually, global placement is the most time-consuming portion in analytical placement.

Global placement aims at minimizing the wirelength cost subjecting to density constraints. A typical solving approach is to relax the density constraints to the objective as a density penalty [2, 4, 16],

$$\min \quad (\sum_{e \in E} \text{WL}(e; \mathbf{x}, \mathbf{y})) + \lambda D(\mathbf{x}, \mathbf{y}), \tag{1}$$

where $\text{WL}(\cdot;\cdot)$ is the wirelength cost function that takes any net instance e and returns the wirelength, and $D(\cdot)$ is the density penalty to spread cells out in the layout. The density constraints can be satisfied by gradually increasing the weight λ.

2.2 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: the analogy of the wirelength cost to the error of misprediction and that of the density cost to the regularization term. Figure 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 $R(\mathbf{w})$ [17]. 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 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 Figure 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 easy APIs to extend the existing set of low-level operators. Each custom operator requires well defined forward and backward functions

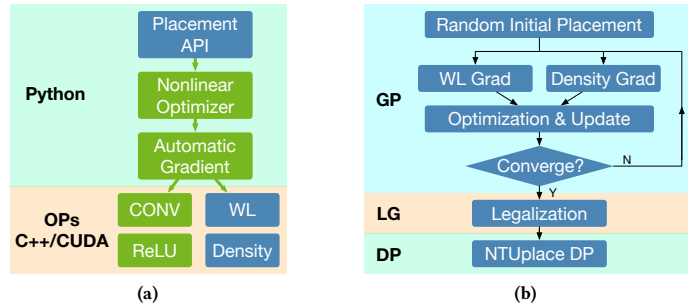

Figure 2: (a) Software architecture for placement implementation using deep learning toolkits. (b) DREAMPlace flow.

for cost and gradient computation. To develop an analytical placement with deep learning toolkits, we only need to implement the custom operators 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. 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 operators and high-level optimization engines.

2.3 The ePlace/RePlAce Algorithm

ePlace/RePlAce is a state-of-the-art family of global placement algorithms that model the layout and netlist as an electrostatic system [4–6]. It uses weighted-average wirelength (WA) for wirelength cost originally proposed by [18, 19],

$$\text{WA}_e = \frac{\sum_{i \in e} x_i e^{\frac{x_i}{\gamma}}}{\sum_{i \in e} e^{\frac{x_i}{\gamma}}} - \frac{\sum_{i \in e} x_i e^{-\frac{x_i}{\gamma}}}{\sum_{i \in e} e^{-\frac{x_i}{\gamma}}}, \tag{2}$$

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

Its density penalty is quite different from other analytical placers [2, 16, 20]. With analogy to an electrostatic system, cells are modeled as charges, density penalty is modeled as potential energy, and density gradient is modeled as the electric field. The numerical solution of the electric potential and field distribution after solving a Poisson's equation can be obtained with 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 follows [4],

$$a_{u,v} = \frac{1}{M^2} \sum_{x=0}^{M-1} \sum_{y=0}^{M-1} \rho(x, y) \cos(w_u x) \cos(w_v y), \tag{3a}$$

$$\psi_{\text{DCT}}(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v}}{w_u^2 + w_v^2} \cos(w_u x) \cos(w_v y), \tag{3b}$$

$$\xi_{\text{DSCT}}^X(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v} w_u}{w_u^2 + w_v^2} \sin(w_u x) \cos(w_v y), \tag{3c}$$

$$\xi_{\text{DCST}}^Y(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} \frac{a_{u,v} w_v}{w_u^2 + w_v^2} \cos(w_u x) \sin(w_v y), \tag{3d}$$

where ψ_{DCT} denotes the potential function, and ξ_{DSCT}^X and ξ_{DCST}^Y denote the electric field in horizontal and vertical directions respectively. Equation (3) requires Discrete Cosine Transform (DCT) and Inverse Discrete Cosine Transform (IDCT) routines to solve efficiently. The detailed computation is explained in Section 3. With the electric field

Table 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^-	$\sum_{i \in e} x_i a_i^-, \forall e \in E$
$\mathbf{x}^+$	$\{x_e^+\}, \forall e \in E$	$\mathbf{x}^-$	$\{x_e^-\}, \forall e \in E$
$\mathbf{a}^+$	$\{a_i^+\}, \forall i \in P$	$\mathbf{a}^-$	$\{a_i^-\}, \forall i \in P$
$\mathbf{b}^+$	$\{b_e^+\}, \forall e \in E$	$\mathbf{b}^-$	$\{b_e^-\}, \forall e \in E$
$\mathbf{c}^+$	$\{c_e^+\}, \forall e \in E$	$\mathbf{c}^-$	$\{c_e^-\}, \forall e \in E$

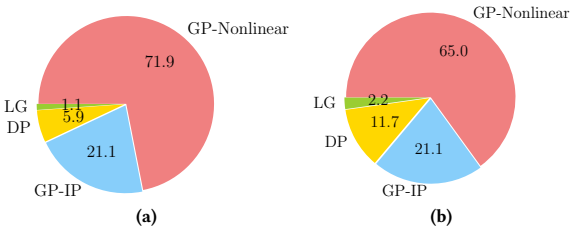

Figure 3: RePlAce [6] runtime breakdown in percentages on `bigblue4` (2 million cells). (a) 1 thread; (b) 10 threads.

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 multi-threading support [6]. The runtime breakdown for RePlAce [6] is elaborated in Figure 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.

3 THE DREAMPLACE ALGORITHMS

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

3.1 Wirelength Forward and Backward

As RePlAce adopts WA wirelength, we use WA 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) [21], which is also implemented in the framework. For brevity, we only discuss the equations in the x direction, as those in the y direction 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 Equation (2) may result in numerical overflow, so we convert $e^{\frac{x_i}{\gamma}}$ to $e^{\frac{x_i - \max_{j \in e} x_j}{\gamma}}$,

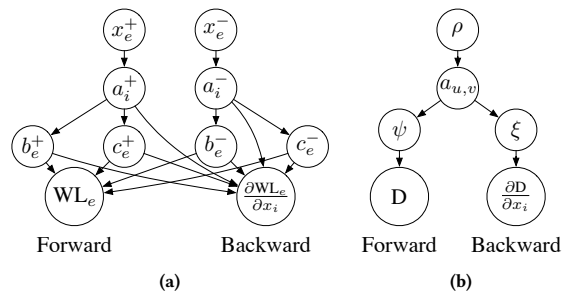

Figure 4: Forward and backward dependency graph for (a) weighted average wirelength and (b) density computation.

and $e^{-\frac{x_i}{\gamma}}$ to $e^{-\frac{x_i - \min_{j \in e} x_j}{\gamma}}$ in Equation (2), which is an equivalent transformation. With the notations in Table 1, the gradient of WA wirelength to a pin location can be written as,

$$\frac{\partial \mathrm{WL}_e}{\partial x_i} = \frac{(1 + \frac{x_i}{\gamma}) b_e^+ - \frac{1}{\gamma} c_e^+}{(b_e^+)^2} \cdot a_i^+ - \frac{(1 - \frac{x_i}{\gamma}) b_e^- + \frac{1}{\gamma} c_e^-}{(b_e^-)^2} \cdot a_i^- . \quad (4)$$

A naive parallelization scheme is to allocate one thread for each net. This scheme has also been discussed in [15], which only demonstrated limited speedup because the maximum number of threads to allocate is $|E|$ and the workload for each thread is ill-balanced due to the heterogeneity of net degrees.

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 Figure 4(a). The computation can be completed in four steps: 1) compute $\mathbf{x}^{\pm}$; 2) compute and store $\mathbf{a}^{\pm}$; 3) compute and store $\mathbf{b}^{\pm}, \mathbf{c}^{\pm}$; 4) compute WL_e in forward or $\frac{\partial \mathrm{WL}_e}{\partial x_i}$ in backward. Algorithm 1 illustrates the pin-level parallel implementation of WA wirelength forward and backward functions. We inline all the CUDA kernel functions, which should be separate in practice, for brevity. Furthermore, although all the computation for an array with different ± signs, e.g., $\mathbf{x}^+$ and $\mathbf{x}^-$, is written together, they 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.

Although [15] suggests to use sparse matrix-vector multiplication provided by CUDA for the $\mathbf{b}^{\pm}, \mathbf{c}^{\pm}$ kernels (line 10-15 in Algorithm 1). we find that the atomic addition is faster, as matrix-vector multiplication is a more general task. A detailed comparison will be made in the experimental results.

3.2 Density Forward and Backward

Backward and forward of density cost is another bulk computation. Figure 4(b) plots the dependency graph for density cost forward and backward. The computation consists of four steps: 1) density map computation; 2) $a_{u,v}$; 3) ψ in forward or ξ in backward; 4) D in forward or $\frac{\partial \mathrm{D}}{\partial x_i}$ in backwrad. Steps 2 and 3 are the bulk computation parts, since DCT/IDCT is required for potential and field computation.

3.2.1 DCT/IDCT for Electric Potential and Field. The electric potential and field computation in Equation (3) requires fast DCT/IDCT kernels for efficient calculation. The standard DCT/IDCT for one-dimensional

Algorithm 1 Wirelength Forward and Backward by Pin

Input: A set of nets E, a set of pins P, and pin locations x;
Output: 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
  for each thread $0 \le t < |P|$ **do** ▷ $\mathbf{a}^\pm$ kernel
    Define e as the net that pin t belongs to;
    $a_t^\pm \leftarrow e^{\pm \frac{x_t - x_e^\pm}{\gamma}}$;
  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
  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
  for each thread $0 \le t < |E|$ **do** ▷ WL_e kernel
    Define e as t^{th} net in E;
    Compute $\mathrm{WL}_e \leftarrow \frac{c_e^+}{b_e^+} - \frac{c_e^-}{b_e^-}$;
  return reduce$(\sum_{e \in E} \mathrm{WL}_e), \mathbf{a}^\pm, \mathbf{b}^\pm, \mathbf{c}^\pm$;
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 $\frac{\partial \mathrm{WL}_e}{\partial x_t}$;
  return $\{\frac{\partial \mathrm{WL}_e}{\partial x_i}\}, \forall i \in P$;

Algorithm 2 DCT/IDCT with N-Point FFT

Input: An even-length real sequence x;
Output: An even-length transformed real sequence y;

function `DCT`(x)
  $N \leftarrow |x|$;
  for each thread $0 \le t < N$ **do** ▷ Reorder kernel
    if $t < \frac{N}{2}$ **then**
      $x'_t \leftarrow x_{2t}$;
    else
      $x'_t \leftarrow x_{2(N-t)-1}$;
  $x'' \leftarrow$ `RFFT`(x'); ▷ One-sided real FFT kernel
  for each thread $0 \le t < N$ **do** ▷ $e^{-\frac{j\pi t}{2N}}$ kernel
    if $t \le \frac{N}{2}$ **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
  return y;
function `IDCT`(x)
  $N \leftarrow |x|$;
  for each thread $0 \le t < \frac{N}{2} + 1$ **do** ▷ Complex kernel
    $x'_t \leftarrow (x_t - j x_{(N-t)}) e^{\frac{j\pi t}{2N}}$; ▷ let $x_N \leftarrow 0$
  $x'' \leftarrow$ `IRFFT`(x'); ▷ One-sided real IFFT kernel;
  for each thread $0 \le t < N$ **do** ▷ Reverse kernel
    if $t \mod 2 == 0$ **then**
      $y_t \leftarrow \frac{N}{4} x''_{\frac{t}{2}}$;
    else
      $y_t \leftarrow \frac{N}{4} x''_{(N - \frac{t+1}{2})}$;
  return y;

(1D) length-N sequence x is,

$$\mathrm{DCT}(\{x_n\})_k = \sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi}{N}(n + \frac{1}{2})k\right), \tag{5a}$$

$$\mathrm{IDCT}(\{x_n\})_k = \frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi}{N} n(k + \frac{1}{2})\right), \tag{5b}$$

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

$$\mathrm{IDXST}(\{x_n\})_k = \sum_{n=0}^{N-1} x_n \sin\left(\frac{\pi}{N} n(k + \frac{1}{2})\right), \tag{6a}$$

$$= (-1)^k \mathrm{IDCT}(\{x_{N-n}\})_k, \tag{6b}$$

where $x_N = 0$. A detailed derivation is omitted due to page limit. Given an $M \times M$ density map ρ, the electric potential and field can be computed using DCT/IDCT, IDXST routines.

$$a_{u,v} = \mathrm{DCT}(\mathrm{DCT}(\rho)^T)^T, \tag{7a}$$

$$\psi_{\mathrm{DCT}} = \mathrm{IDCT}(\mathrm{IDCT}(\{\frac{a_{u,v}}{w_u^2 + w_v^2}\})^T)^T, \tag{7b}$$

$$\xi_{\mathrm{DSCT}}^X = \mathrm{IDXST}(\mathrm{IDCT}(\{\frac{a_{u,v} w_u}{w_u^2 + w_v^2}\})^T)^T, \tag{7c}$$

$$\xi_{\mathrm{DCST}}^Y = \mathrm{IDCT}(\mathrm{IDXST}(\{\frac{a_{u,v} w_v}{w_u^2 + w_v^2}\})^T)^T, \tag{7d}$$

where $(\cdot)^T$ denotes matrix transposition. The two-dimensional (2D) DCT/IDCT is computed by performing 1D DCT/IDCT to columns and then rows. We can see all the computations break down to the 1D 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 [22] and demonstrate better efficiency in the experiments. Algorithm 2 illustrates the FFT implementation with pre-processing and post-processing kernels. Due to the symmetric property of FFT for real 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.

3.3 Optimization Engine

ePlace/RePlAce [4, 6] 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, e.g., gradient descent with momentum, Adam [23], providing additional solver options.

3.4 Legalization

We also develop legalization as an operator in DREAMPlace. It first follows the Tetris-like procedure similar to NTUplace3 [2]. Then it performs Abacus row-based legalization [24]. This step copies the cell locations from GPU to CPU and executes legalization purely on CPU

because we observe that it only takes several seconds even for million-size designs with a single CPU thread.

4 EXPERIMENTAL RESULTS

The framework was developed in Python with `PyTorch` for optimizers and API, and C++/CUDA for low-level operators. The DREAMPlace program runs on a Linux server with 40-core Intel E5-2698 v4 @ 2.20GHz and 1 NVIDIA Tesla V100 GPU based on Volta architecture. Only a single CPU thread was used by DREAMPlace for launching work to the GPU in the experiments. RePlAce [6] runs on a Linux server with 24-core 3.0 GHz Intel Xeon machine with 64GB memory allocated. ISPD 2005 contest benchmarks [25] and large industrial designs were adopted. Currently, we only collected experimental results with double-precision floating point numbers on GPU, so additional easy speedup is possible by switching to single-precision.

4.1 Placement Acceleration

Table 2 and Table 3 show the HPWL and runtime details on ISPD 2005 and industrial benchmarks. With almost the same solution quality (within 0.2% difference on average), DREAMPlace is able to achieve 35× and 43× speedup in GP on the two benchmark suites compared to RePlAce with 40 threads. RePlAce [6] crashed on the 10-million-cell industrial benchmark at the 9th iteration for Nesterov's optimization. The potential reason is that the peak memory usage of RePlAce exceeded the maximum memory (64GB). Before crashing, it took 119 seconds for initial placement and on average 9 seconds for each Nesterov iteration. As this benchmark takes 1000 iterations with DREAMPlace, we made an estimation to the runtime lower bound $119 + 1000 * 9 \approx 9100$ seconds. The LG of DREAMPlace is also around 10× 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 is 5×.

Figure 5 plots the GP runtime comparison between DREAMPlace and multi-threaded RePlAce. The speedup of RePlAce saturates quickly from single thread to 40 threads, i.e., maximum around 2.5×. Figure 5(b) and Figure 5(c) present the scaling curves with increasing number of cells. DREAMPlace shows both good stability and nearly linear scalability with problem sizes. Figure 6 draws the runtime breakdown of DREAMPlace on a 2-million-cell design `bigblue4`, where GP and LG only take 11.7% runtime of the entire flow. The runtime of GP and LG is even comparable to that of file IO for benchmark reading and writing. The majority of the runtime (76%) is taken by DP, which still relies on the external placer currently. A previous study [26] has demonstrated around 6× speedup from GPU acceleration for DP over multi-threaded CPU. While DP is not the focus of this paper, there is a potential of 15× speedup for the entire placement by future incorporation of GPU-accelerated DP, e.g., $\frac{2463}{43+9+336/6+54} = 15.2$ for `bigblue4`. On the other hand, within each forward and backward pass of GP, the density-related computation takes longer than wirelength (67.8% v.s. 32.2%). With efficient DCT/IDCT implementation, the electric field computation is no longer the bottleneck for density forward and backward.

4.2 Acceleration of Low-Level Operators

We further investigate the efficiency of our implementations of the low-level operators, e.g., wirelength forward and backward, DCT/IDCT for density forward and backward. Figure 7 compares three approaches discussed in Section 3.1. "Net-by-Net" denotes the net-level parallelization; "Sparse" denotes the sparse matrix-vector multiplication approach [15]; "Atomic" denotes the pin-level parallelization with atomic operations in Algorithm 1. The atomic approach achieves 1.9× speedup over the net-by-net one, 1.4× speedup over the sparse one.

Figure 8 compares the 2D DCT/IDCT implementation using $2N$-point FFT ("DCT-2N" and "IDCT-2N") and N-point FFT ("DCT-N" and "IDCT-N") [22]. Considering the map sizes in the experiment (from 512×512 to 4096×4096), the N-point implementation is 1.4× faster. Although the runtime gap is decreasing, the N-point implementation still shows superior efficiency with typical map sizes for a wide range of designs.

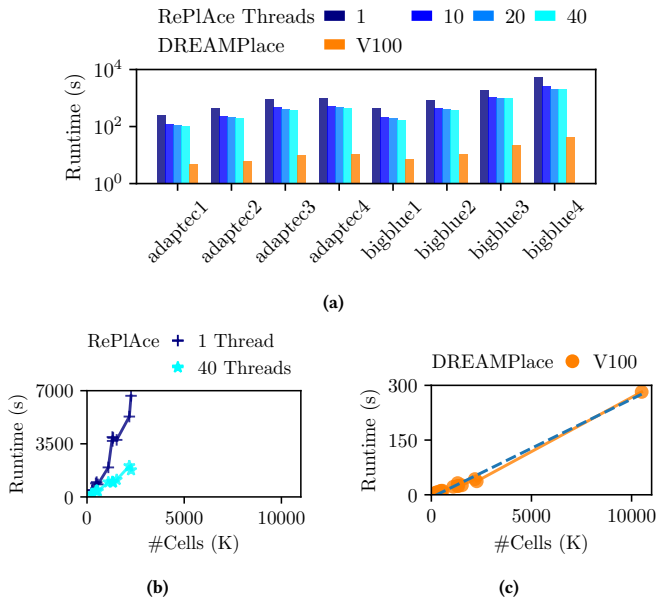

Figure 5: (a) GP runtime for ISPD 2005 benchmarks. (b) GP runtime scaling for RePlAce on ISPD and industrial benchmarks. The points for the 10-million-cell benchmark `design6` are omitted due to crash. (c) GP runtime scaling for DREAMPlace.

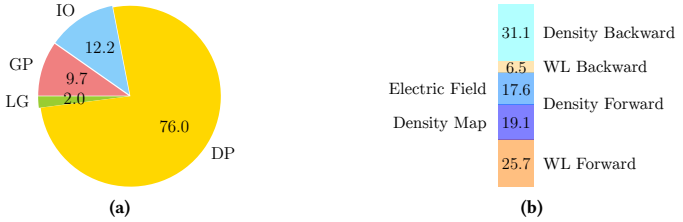

Figure 6: Runtime breakdown in percentages of DREAMPlace (a) on `bigblue4` and (b) one forward and backward pass in GP.

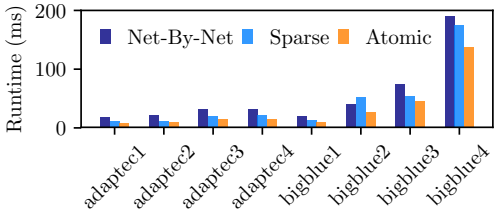

Figure 7: GPU runtime comparison of different implementation of wirelength forward and backward.

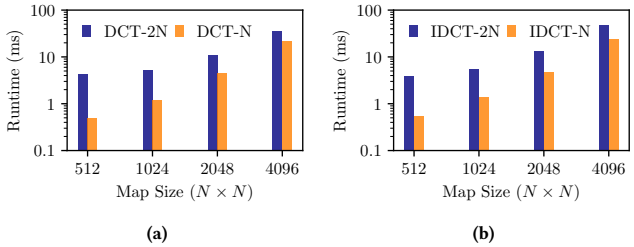

Figure 8: GPU runtime comparison of DCT/IDCT algorithms.

Table 2: Experimental results on ISPD 2005 benchmarks [25].

Design	#cells	#nets	RePlAce [6] (40 Threads)					DREAMPlace					
			HPWL	GP (s)	LG (s)	DP (s)	Total (s)	HPWL	GP (s)	LG (s)	DP (s)	IO (s)	Total (s)
adaptec1	211K	221K	73.26	106	5	24	139	73.30	5	0.5	25	5	37
adaptec2	255K	266K	81.87	194	7	30	236	82.19	6	0.5	32	6	47
adaptec3	452K	467K	193.20	382	22	53	466	194.12	10	1	58	10	82
adaptec4	496K	516K	175.23	434	21	61	524	174.43	11	2	65	11	92
bigblue1	278K	284K	89.85	168	3	30	206	89.43	7	0.3	35	6	51
bigblue2	558K	577K	138.09	383	22	90	505	136.69	11	9	90	12	125
bigblue3	1097K	1123K	304.83	967	46	138	1171	303.99	22	3	145	24	198
bigblue4	2177K	2230K	743.73	2037	56	329	2463	743.75	43	9	336	54	446
ratio	-	-	1.002	34.8	10.6	0.9	5.0	1.000	1.0	1.0	1.0	-	1.0

Table 3: Experimental results on industrial benchmarks.

Design	#cells	#nets	RePlAce [6] (40 Threads)					DREAMPlace					
			HPWL	GP (s)	LG (s)	DP (s)	Total (s)	HPWL	GP (s)	LG (s)	DP (s)	IO (s)	Total (s)
design1	1345K	1389K	340.42	974	46	155	1216	340.87	24	4	179	34	244
design2	1306K	1355K	275.46	1001	44	152	1238	275.76	24	5	180	32	244
design3	2265K	2276K	524.35	1767	84	257	2189	522.79	37	14	305	55	414
design4	1525K	1528K	455.22	1130	55	187	1424	454.38	26	8	207	37	281
design5	1316K	1364K	287.24	955	46	153	1193	288.41	22	4	186	33	248
design6	10504K	10747K	NA	>9100	NA	NA	NA	2356.88	282	73	1681	276	2323
ratio	-	-	1.000	43.1	9.5	0.9	5.0	1.000	1.0	1.0	1.0	-	1.0

5 CONCLUSION

In this paper, we take a totally new perspective of solving classical analytical placement by casting it into a neural network training problem. Leveraging the deep learning toolkit `PyTorch`, thus with small software development effort, we develop a new open-source placement engine, *DREAMPlace* with GPU acceleration. It achieves over 30× speedup in global placement and legalization 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 operators 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 [21, 27, 28] as well as GPU-accelerated detailed placement. It can also be extended to leverage multi-GPU platforms for further speedup. As DREAMPlace decouples the high-level algorithmic design and low-level acceleration efforts, it significantly reduces the development and maintenance overhead. We believe this work shall open up new directions for revisiting classical EDA problems.

ACKNOWLEDGE

This project is supported in part by NVIDIA, Inc.

REFERENCES

[1] T. Chan, J. Cong, and K. Sze, "Multilevel generalized force-directed method for circuit placement," in *Proc. ISPD*. ACM, 2005, pp. 185–192.

[2] 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 TCAD*, vol. 27, no. 7, pp. 1228–1240, 2008.

[3] M.-K. Hsu, Y.-F. Chen, C.-C. Huang, S. Chou, T.-H. Lin, T.-C. Chen, and Y.-W. Chang, "NTUplace4h: A novel routability-driven placement algorithm for hierarchical mixed-size circuit designs," *IEEE TCAD*, vol. 33, no. 12, pp. 1914–1927, 2014.

[4] J. Lu, P. Chen, C.-C. Chang, L. Sha, D. J.-H. Huang, C.-C. Teng, and C.-K. Cheng, "ePlace: Electrostatics-based placement using fast fourier transform and nesterov's method," *ACM TODAES*, vol. 20, no. 2, p. 17, 2015.

[5] J. Lu, H. Zhuang, P. Chen, H. Chang, C.-C. Chang, Y.-C. Wong, L. Sha, D. Huang, Y. Luo, C.-C. Teng *et al.*, "ePlace-MS: Electrostatics-based placement for mixed-size circuits," *IEEE TCAD*, vol. 34, no. 5, pp. 685–698, 2015.

[6] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE TCAD*, 2018.

[7] Z. Zhu, J. Chen, Z. Peng, W. Zhu, and Y.-W. Chang, "Generalized augmented lagrangian and its applications to vlsi global placement," in *Proc. DAC*. IEEE, 2018, pp. 1–6.

[8] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," *IEEE TCAD*, vol. 31, no. 1, pp. 50–60, 2012.

[9] X. He, T. Huang, L. Xiao, H. Tian, and E. F. Y. Young, "Ripple: A robust and effective routability-driven placer," *IEEE TCAD*, vol. 32, no. 10, pp. 1546–1556, 2013.

[10] 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 TCAD*, vol. 34, no. 3, pp. 447–459, 2015.

[11] T. Lin, C. Chu, and G. Wu, "Polar 3.0: An ultrafast global placement engine," in *Proc. ICCAD*. IEEE, 2015, pp. 520–527.

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

[13] W. Li, M. Li, J. Wang, and D. Z. Pan, "UTPlaceF 3.0: A parallelization framework for modern FPGA global placement," in *Proc. ICCAD*. IEEE, 2017, pp. 922–928.

[14] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Proc. ICCAD*. ACM, 2009, pp. 681–688.

[15] C.-X. Lin and M. D. Wong, "Accelerate analytical placement with gpu: A generic approach," in *Proc. DATE*. IEEE, 2018, pp. 1345–1350.

[16] A. B. Kahng, S. Reda, and Q. Wang, "Architecture and details of a high quality, large-scale analytical placer," in *Proc. ICCAD*. IEEE, 2005, pp. 891–898.

[17] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[18] M.-K. Hsu, Y.-W. Chang, and V. Balabanov, "Tsv-aware analytical placement for 3d ic designs," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 664–669.

[19] M.-K. Hsu, V. Balabanov, and Y.-W. Chang, "Tsv-aware analytical placement for 3-d ic designs based on a novel weighted-average wirelength model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 4, pp. 497–509, 2013.

[20] A. B. Kahng and Q. Wang, "A faster implementation of APlace," in *Proc. ISPD*. ACM, 2006, pp. 218–220.

[21] 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," Oct. 9 2001, US Patent 6,301,693.

[22] J. Makhoul, "A fast cosine transform in one and two dimensions," *IEEE Transactions on Signal Processing*, vol. 28, no. 1, pp. 27–34, 1980.

[23] D. Kinga and J. B. Adam, "A method for stochastic optimization," in *Proc. ICLR*, 2015.

[24] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Abacus: fast legalization of standard cell circuits with minimal movement," in *Proc. ISPD*, 2008, pp. 47–53.

[25] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ispd2005 placement contest and benchmark suite," in *Proc. ISPD*. ACM, 2005, pp. 216–220.

[26] S. Dhar and D. Z. Pan, "GDP: GPU accelerated detailed placement," in *Proc. HPEC*, Sept 2018.

[27] T. F. Chan, K. Sze, J. R. Shinnerl, and M. Xie, "Mpl6: Enhanced multilevel mixed-size placement with congestion control," in *Modern Circuit Placement*. Springer, 2007.

[28] A. B. Kahng and Q. Wang, "An analytic placer for mixed-size placement and timing-driven placement," in *Proc. ICCAD*. IEEE, 2004, pp. 565–572.