Xplace: An Extremely Fast and Extensible Global Placement Framework

Lixin Liu
CSE Department, CUHK
lxliu@cse.cuhk.edu.hk

Bangqi Fu
CSE Department, CUHK
bqfu21@cse.cuhk.edu.hk

Martin D.F. Wong
CSE Department, CUHK
mdfwong@cuhk.edu.hk

Evangeline F.Y. Young
CSE Department, CUHK
fyyoung@cse.cuhk.edu.hk

ABSTRACT

Placement serves as a fundamental step in VLSI physical design. Recently, GPU-based global placer DREAMPlace[1] demonstrated its superiority over CPU-based global placers. In this work, we develop an extremely fast GPU accelerated global placer Xplace which achieves around 2x speedup with better solution quality compared to DREAMPlace. We also plug a novel Fourier neural network into Xplace as an extension to further improve the solution quality. We believe this work not only proposes a new, fast, extensible placement framework but also illustrates a possibility to incorporate a neural network component into a GPU accelerated analytical placer.

1 INTRODUCTION

Placement serves as a fundamental step in VLSI physical design due to its strong correlation between the solution quality of placement and the circuit's PPA (power, performance and area). Meanwhile, modern circuits contain millions of standard cells, which highly increases the computational complexity of the placement problem and brings huge challenges to the leading-edge global placers.

To tackle the aforementioned problem, various kinds of CPU based analytical global placers have been proposed over the past decades. Quadratic placers represent wirelength with a quadratic function and mitigate cell density by cell inflation[2, 3], rough legalization[4], force-directed method[5], etc. Although quadratic placers show fast run time to converge, their solution qualities are limited by the low modeling order of the wirelength. Non-linear placers approximate wirelength by a smooth version of the half-perimeter wirelength (HPWL) metric and model cell density by bell-shaped function [6, 7], Poisson's equation[8–10], etc. Different from quadratic placers, non-linear placers produce higher solution quality while the running time overhead is huge.

With the rapid development of GPU's computational power, GPU acceleration becomes an important direction to pursue to handle large scale problem with parallelism. In global placement, the work [11] accelerated multi-level analytical placer mPL[12] with GPU by parallelizing the computation of the wirelength function as well as the spreading force and achieved around 15× speedup. The work [13] explored the idea of utilizing sparse matrix multiplications to compute wirelength and adopting a flattening technique for area computation. However, the maximum wirelength degradation in [11] is larger than 5% and the work [13] does not report their solution quality in details.

Recently, DREAMPlace[1, 14] implemented the approach of ePlace[8] on GPU by casting the placement problem as a neural network training problem and demonstrated the superiority of GPU accelerated global placers. DREAMPlace not only produced the state-of-the-art solution quality and performance but also provided an open-source analytical placement framework for researchers to further develop. However, it focuses on accelerating the wirelength and density operators on GPU while lacking a more general operator-level optimization.

DREAMPlace brought more than 40x speedup on average for large benchmarks, and it is a big challenge to further improve on its performance. In this work, we develop Xplace, a new, fast and extensible GPU accelerated global placement framework built on top of PyTorch[15], to consider factors at operator-level optimization. Xplace not only achieves better performance and quality than DREAMPlace but also shows high extensiblity to incorporate neural network into analytical placer. Our key contributions are summarized as follows.

- Efficiency: with operator combination, extraction, reduction and skipping, Xplace achieves around 3x speedup per GP iteration compared to the state-of-the-art global placer DREAMPlace. A placement-stage-aware parameter scheduling technique is also proposed to improve the solution quality. Experimental results show that Xplace achieves around 2x speedup with better solution quality compared to DREAMPlace.
- Extensiblity: we plug into Xplace a novel Fourier neural network as an extension. The neural network serves as a global guidance for placement. Experimental results not only show that the proposed framework can further improve the solution quality but also illustrates a possibility of adopting neural guidance in analytical global placement.

2 PRELIMINARIES

Given a placement circuit $G = (V, E)$, V represents the set of cells and E denotes the set of nets. Let $p = \{(x_1, y_1), ..., (x_N, y_N)\} \in \mathbb{R}^{N\times 2}$ denote the 2D positions of the cells, where N is the number of cells. The placement region R is uniformly split into an $M \times M$ grid B. The objective of global placement is to minimize the total HPWL of all the nets while satisfying the cell density constraint, which is formulated as,

This research was partially supported by ACCESS - AI Chip Center for Emerging Smart Systems, Hong Kong SAR.

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530485

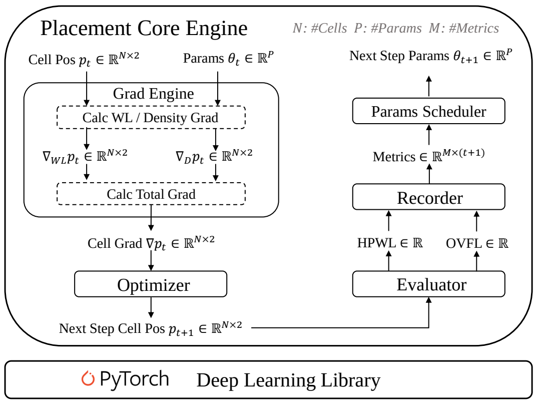

Placement Core Engine
N: #Cells P: #Params M: #Metrics
Cell Pos $p_t \in \mathbb{R}^{N\times2}$
Params $\theta_t \in \mathbb{R}^P$
Grad Engine
Calc WL / Density Grad
$\nabla_{WL} p_t \in \mathbb{R}^{N\times2}$
$\nabla_D p_t \in \mathbb{R}^{N\times2}$
Calc Total Grad
Cell Grad $\nabla p_t \in \mathbb{R}^{N\times2}$
Optimizer
Next Step Cell Pos $p_{t+1} \in \mathbb{R}^{N\times2}$
Next Step Params $\theta_{t+1} \in \mathbb{R}^P$
Params Scheduler
Metrics $\in \mathbb{R}^{M\times(t+1)}$
Recorder
HPWL $\in \mathbb{R}$
OVFL $\in \mathbb{R}$
Evaluator
PyTorch
Deep Learning Library

Figure 1: Overview of Xplace

$$\min_p HPWL(p) = \min_p \sum_{e\in E} HPWL_e(p) \tag{1a}$$

$$\text{s.t.} \quad D_b \le D_t, \forall b \in B \tag{1b}$$

where D_b and D_t denote bin b's cell density and the benchmark-given target density respectively. The HPWL of net e is defined as follows:

$$HPWL_e(p) = (\max_{i\in e} x_i - \min_{i\in e} x_i) + (\max_{i\in e} y_i - \min_{i\in e} y_i). \tag{2}$$

Analytical global placement reformulates the objective with a smooth approximation of $HPWL$ and a density penalty:

$$\min_p \sum_{e\in E} WL_e(p) + \lambda D(p) \tag{3}$$

where the wirelength $WL_e(p) = WL_e(x) + WL_e(y)$ is modeled as the weighted average (WA) wirelength with a coefficient γ,

$$WL_e(x) = \frac{\sum_{i\in e} x_i e^{x_i/\gamma}}{\sum_{i\in e} e^{x_i/\gamma}} - \frac{\sum_{i\in e} x_i e^{-x_i/\gamma}}{\sum_{i\in e} e^{-x_i/\gamma}} \tag{4}$$

and similarly for $WL_e(y)$. A smaller γ leads to more accurate approximation of HPWL. The parameter λ controls the weight of the cell density penalty. A typical placement flow starts with a small λ and gradually increase it to remove overlaps.

In [8] on which Xplace is based, the cell density is modeled as an electrostatic system denoted as:

$$\begin{cases} \nabla \cdot \nabla\psi(x,y) = -\rho(x,y), \\ \hat{\mathbf{n}} \cdot \nabla\psi(x,y) = 0, (x,y) \in \partial R, \\ \iint_R \rho(x,y) = \iint_R \psi(x,y) = 0, \end{cases} \tag{5}$$

where ∂R is the boundary, $\rho(x,y)$ is the electron density map and $\psi(x,y)$ is the potential distribution. The numerical solution of the density gradients are obtained by discrete cosine transformation in [8].

3 PROPOSED FRAMEWORK

In this section, we will discuss the design of Xplace, a fast and extensible GPU accelerated global placer. Our Xplace framework is shown in Figure 1. Xplace is built on top of PyTorch and contains a placement core engine. Inside the core engine, the gradient engine takes cell position and placement parameters as input to compute the cell gradient. Next, the optimizer utilizes the computed gradient to update the cell position. The evaluator evaluates the placement solution and the recorder records the placement metrics like HPWL and overflow. Finally, the scheduler decides how to modify the parameters and whether to stop the global placement. It is worth noting that all these parts are designed as independent modules in Xplace so that one can easily extend Xplace by applying new scheduling techniques, new gradient functions, new placement metrics and so on.

Section 3.1 will discuss several important technical details that enable Xplace running very efficiently on GPU. We propose operator-level optimization techniques to achieve effective parallelization. Section 3.2 will discuss a placement-stage-aware parameters scheduling to improve the solution quality. Section 3.3 will extend Xplace with a Fourier neural operator to show its high extensibility for future development.

3.1 Operator-Level Optimization

3.1.1 Wirelength Operator Combination. Weighted average (WA) wirelength is used as the wirelength objective in many analytical global placers. In Xplace, we adopt the WA wirelength in Equation (4) as our wirelength objective and update the cell position based on the guidance of the WA gradient. To avoid numerical overflow, a numerically stable version of the WA wirelength is given in Equation (6) which needs the minimum and the maximum position among all pins in a net.

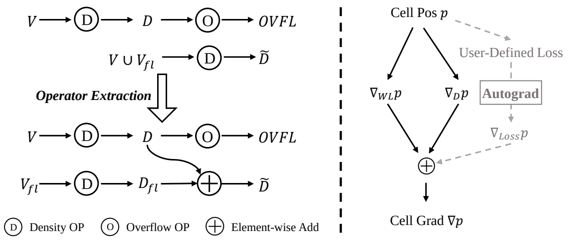
$$WL_e(x) = \frac{\sum_{i\in e} x_i e^{\frac{x_i - \max_{j\in e} x_j}{\gamma}}}{\sum_{i\in e} e^{\frac{x_i - \max_{j\in e} x_j}{\gamma}}} - \frac{\sum_{i\in e} x_i e^{\frac{\min_{j\in e} x_j - x_i}{\gamma}}}{\sum_{i\in e} e^{\frac{\min_{j\in e} x_j - x_i}{\gamma}}} \tag{6}$$

A wirelength objective-and-gradient merging method, proposed in [1], computes both the WA wirelength and the WA gradient within a single GPU thread to mitigate memory bounded problems. Since both the HPWL function and the stable WA wirelength function need the minimum and maximum cell positions in a net, we further modify this merging method by combining the three operators with heavy wirelength-related workload, WA wirelength, WA gradient and HPWL, into one operator to avoid redundant computation of the minimum and maximum function. The proposed operator combination technique can significantly reduce the total GPU execution time.

3.1.2 Density Operator Extraction. Density objective is one of the most computationally intensive operators in global placement. Similar to [1, 13], we implemented a GPU accelerated area accumulation operator to compute the cell density map and apply PyTorch built-in rfft2/irfft2 operators to derive the numerical solution for the electrostatic system in Equation (5). We also implemented a GPU accelerated version of the overflow ratio operator $OVFL$, whose CPU version is applied in [6, 16], to measure the evenness of cell distribution. The overflow ratio is described as,

$$OVFL = \frac{\sum_{b\in B} \max(D_b - D_t, 0) A_b}{\sum_{i\in V_{mov}} A_i} \tag{7}$$

where A_b and A_i denote the area for bin b and cell i; D_b is bin b's cell density; D_t is the target density, and V_{mov} is the set of movable cells. Concretely, each bin b's cell density D_b in the cell density



(a) Operator Extraction (b) Cell Gradient Computation

Figure 2: Illustration for the operator extraction technique and the cell gradient computation scheme.

map $D \in \mathbb{R}^{M \times M}$ is defined as,

$$D_b = \frac{\sum_{i \in V} A_i \cap A_b}{A_b}, \ \forall b \in B \tag{8}$$

where $A_i \cap A_b$ defines the overlap area between cell i and bin b. Similar to [16, 17], we insert filler cells inside the electrostatic system to handle whitespace and prevent the density objective to overly spread the cells. Therefore the total density map $\tilde{D} \in \mathbb{R}^{M \times M}$ used for solving the electrostatic system is formulated as follows,

$$\tilde{D}_b = \frac{\sum_{i \in V \cup V_{fl}} A_i \cap A_b}{A_b} = D_b + \frac{\sum_{i \in V_{fl}} A_i \cap A_b}{A_b}, \ \forall b \in B \tag{9}$$

where V_{fl} denotes the set of filler cells. We denote the filler density map as $D_{fl} \in \mathbb{R}^{M \times M}$, where $D_{fl,b} = (\sum_{i \in V_{fl}} A_i \cap A_b)/A_b, \ \forall b \in B$. Then, the matrix form of Equation (9) is formulated as

$$\tilde{D} = D + D_{fl} \tag{10}$$

We observe that both Equation (8) and Equation (10) contain the computation of cell density map D. Due to the heavy load of the density map operator, performing a common sub-operator extraction in Equation (10) will naturally boost the performance. As shown in Figure 2(a), we first compute the cell density map D and the filler density map D_{fl} separately. We then adopt the element-wise add operator to compute the total density map $\tilde{D}$ and apply the overflow operator to calculate $OVFL$. Note that overflow ratio computation is needed for updating parameters in each GP iteration. The proposed sub-operator extraction technique will reduce the total computation time of the cell density map D and achieve a visible improvement in the total GPU execution time.

3.1.3 Operator Reduction. PyTorch[15] is a well-known deep learning library that provides a lot of built-in differentiable operators (e.g. element-wise addition, matrix multiplication, convolution, etc.). In forward propagation, users can apply the provided/user-defined operators to construct a neural network or a gradient-based optimization. In backward propagation, an automatic differentiation (autograd) engine is invoked to compute derivatives automatically. Although PyTorch makes development convenient, there are technical details that need to be carefully considered when building a global placer using PyTorch.

In PyTorch, the execution of each operator will perform a kernel launching step on CPU before executing the core CUDA kernel on GPU. Not only that the forward propagation will execute operators but also the backward propagation, driven by the autograd engine, need to run gradient operators. However, the kernel launching overheads of these operators may even be much larger than their

Algorithm 1 Placement-Stage-Aware Parameters Scheduling

```
1: $\gamma \leftarrow \gamma_0$                                   ▷ wirelength coefficient
2: $\lambda \leftarrow \lambda_0$                                   ▷ density weight
3: while $iteration$ < ITER and NOT Convergence do
4:     if $0.5 < \omega < 0.95$ and $iteration \% 3 \neq 0$ then
5:         SKIP_UPDATE
6:     else
7:         $\gamma \leftarrow \gamma \times coef(overflow)$
8:         $\lambda \leftarrow \lambda \times \mu(\Delta hpwl)$       ▷ Both $\gamma$ and $\lambda$ are derived from [10]
9:     $\omega \leftarrow \frac{\lambda|\mathbf{H_D}|}{|\mathbf{H_W}|+\lambda|\mathbf{H_D}|}$
```

GPU execution overheads when their computation workloads are small. Except for the heavily load operators (e.g. for wirelength and density computations) that are related to the netlist size and the die area, the kernel launching overheads of most other placement operators are much larger than their GPU execution overheads. In this case, the more operators being executed, the larger the total kernel launching overhead there will be. If the total kernel launching overhead dominates the GPU execution time, the speedup will be limited. To this end, we propose a series of techniques to reduce the number of operators to mitigate the problem.

The first technique is to avoid invoking the heavy autograd engine. Since the number of forward operators are almost the same as that in the backward, invoking the heavy autograd engine will almost double the number of operators and bring large kernel launching overhead on CPU. To resolve this problem, we directly derive the numerical solutions of the wirelength gradient and the density gradient and assign a weighted accumulated gradient to each cell. This step can reduce the total kernel launching time and boost the performance significantly. It is worth noting that avoiding invoking the autograd engine will not affect our framework's extensibility since PyTorch also supports invoking the autograd engine for user-defined loss function and accumulating the separately computed numerical gradient with the backward gradient of the user defined loss function as illustrated in Figure 2(b).

Besides, the PyTorch in-place operators, which directly manipulate on the tensor memory without memory copying, are used as much as possible. This technique will naturally avoid redundant copying. Finally, as frequent synchronization will interrupt the GPU pipeline and slow down the total run time, we reorder the operators that need synchronization to the end of the execution queue in each GP iteration so that the negative effects of synchronization can be alleviated.

3.1.4 Operator Skipping. It is observed that the ratio between density gradient and wirelength gradient $r = \frac{\lambda|\nabla D_{x,y}|}{|\nabla Wl_{x,y}|}$ is ultra-small in the early placement stage. Based on the observation, we propose an early-stage operator skipping technique to further boost the performance. When $(r < 0.01) \wedge (iteration < 100)$, the density gradient operator will only be executed once per 20 iterations.

3.2 Placement-Stage-Aware Parameters Scheduling

A precondition matrix of $\tilde{\mathbf{H}}^{-1} = (\tilde{\mathbf{H_W}} + \lambda\tilde{\mathbf{H_D}})^{-1}$ is applied to the optimization objective for reducing the solution complexity and accelerating convergence[16]. $\mathbf{H_W} = \text{diag}(|S_1|, |S_2|, ..., |S_N|)$ and $\mathbf{H_D} = \text{diag}(A_1, A_2, ..., A_N)$, where $S_i = \{e\}_i$ is the set of nets containing cell$_i$. $|S_i|$ is the number of nets connecting cell$_i$ and A_i is the area of cell$_i$. We further introduce the precondition weighted ratio $\omega = \frac{\lambda|\mathbf{H_D}|}{|\mathbf{H_W}|+\lambda|\mathbf{H_D}|}$ in the range [0,1] to measure the placement stage.

Through our investigation, when $\omega < 0.05$, the optimization objective is wirelength-dominated and cells are driven to the position with minimum wirelength. As ω rapidly grows from 0.05 to 0.95, cells are spreading over the whole map and the overlap ratio significantly decreases. At the end, cells are forced to a final position with minimum local penalty at the final stage when $\omega > 0.95$. In the intermediate stage when $0.5 < \omega < 0.95$, we slow down the parameter update to once per 3 iterations to fully exploit the optimization space as in Algorithm 1.

3.3 Extending the Framework via Neural Enhancement

In this sub-section, we show Xplace's high extensibility by incorporating a deep neural network into its optimization process. As Xplace has a similar architecture to a normal neural network of PyTorch, it is natural and easy to embed a neural network as an extension. Here, we propose a neural-plugged-in framework to explore the possibility of learning-based frameworks.

3.3.1 Electric Field Prediction using Neural Networks. Neural networks have shown great potential on mapping between dynamic systems defined by partial-differential-equations. Previous works of image-to-image mapping tasks are usually conducted in spatial domain and are accurate on training data-sets yet when given a new pattern of instance, they do not perform well. Recently, convolution operation in frequency domain is discovered to be more powerful on generalizing dynamic systems determined by a family of partial differential equations[18]. Such Fourier-Neural-Operator (FNO) is capable of learning the universal solution of a dynamic system with only limited training data.

In our placement problem, the function of electric field Equation (5) solved by Poisson's equation can be modeled as a dynamic system mapping from electron distribution to electric field, in which electron distribution is the 2-D density map of a placement and electric field is the moving force on x and y-axis.

As illustrated in Figure 3, our model is a two-path convolution network consists of a spatial-domain path (blue) to extract the explicit information of a specific feature-map and a frequency-domain path (orange) to generalize the global information of a continuous dynamic system. In order to transfer the model to multi-resolution, the input density map D is concatenated with mesh-grid index $M_x(x, y) = \frac{x}{X}$ and $M_y(x, y) = \frac{y}{Y}$, where X and Y are map sizes. The input map $I = \{D; M_x; M_y\}$ is firstly lifted to multi-channel by a fully-connected layer, denoted as $I_m = FC(I)$, and then fed into two paths.

In the Fourier path, the spatial map is transformed into the frequency space after Fast-Fourier-Transform (FFT) function $\mathcal{F}$. A low-pass-filter (LPF) L preserves a number of lower frequency components and then a linear transformation $\mathcal{W}$ is applied to the filtered map. After applying an Inverse-Fast-Fourier-Transform

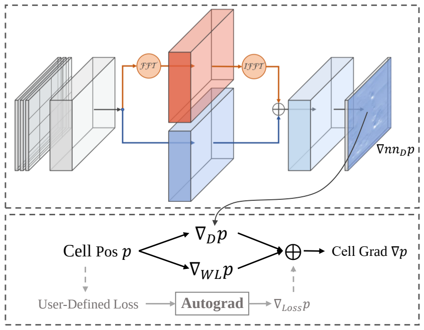


Figure 3: Neural-network plugged-in placement extension.

(IFFT) function $\mathcal{F}^{-1}$, the frequency map is transformed back to the spatial domain,

$$Freq_{layer}(I_m) = \mathcal{F}^{-1}\Big(\mathcal{W}^T \cdot L(\mathcal{F}(I_m))\Big) \tag{11}$$

In the spatial path, a simple pixel-wise convolution layer is operated on the feature map. Maps from two paths are added followed with a nonlinear activation function *GELU*. The above process of a FNO is described as

$$O(I_m) = GELU\Big(Conv_{2D}(I_m) + Freq_{layer}(I_m)\Big) \tag{12}$$

We then get the output after a down-sampling fully-connected layer $FC^{-1}\Big(O(I_m)\Big)$ and a loss function of L_2 is used for back-propagation:

$$L_2(\mathbf{x_i}, f(\mathbf{x_i}; \theta)) = ||f(\mathbf{x_i}, \theta) - \mathbf{y_i}||_2/||\mathbf{y_i}||_2 \tag{13}$$

where $\mathbf{x}_i, \mathbf{y}_i, f$ and θ are the ith input, label, network and network parameters. The label and prediction are normalized for evaluation.

In this model, we do not need to collect the ground-truth training data from real placement benchmarks. Rather, we can generate randomly distributed density maps and compute the numerical solution of the corresponding electric fields which will be used as labels for training.

Since we do a pixel-wise convolution on the spatial maps, the resolution of the input maps will not affect the convolution results. Moreover, in Fourier space, low frequency components describe the global information, while high frequency components describe the explicit local information. That means, a low-resolution image and a high-resolution image will share similar low frequency components, with differences in high frequency components only. As we only preserve a certain number of lower frequency components, our model is resolution-independent. The model can be trained on low-resolution data and extended to high-resolution, which can significantly improve the adaptability of the model.

The electric fields on both the x and y direction share the same partial-differential function, with only a difference in the direction. Therefore the model can be trained on only one direction and still be workable on the other direction by simply flipping the input map, further improving the generalization of this model.

As the trained model has good generalization in global view, we insert the nn-predicted density gradient $\nabla_{nn}D_{x,y}$ into the early stage of placement to help push cells around. With the ω defined in Section 3.2, a smooth function $\sigma(\omega) = 1 - 1/(1 - 5e^{\omega/0.05-0.5})$ is used to weighted-sum with the numerical solution of the density gradient,

$$\nabla' D_{x,y} = (1-\sigma)\nabla D_{x,y} + \sigma \nabla_{nn} D_{x,y} \tag{14}$$

Empirically, function σ has a similar shape as $|\frac{\nabla WL_{x,y}}{\nabla D_{x,y}}|$ with a delay of T iterations (T varies among different benchmarks). $\nabla_{nn} D_{x,y}$ will dominate when $|\frac{\nabla WL_{x,y}}{\nabla D_{x,y}}|$ drops. When σ drops as well, $\nabla D_{x,y}$ takes effect for fine-grained placement.

4 EXPERIMENTAL RESULTS

The Xplace framework is developed with PyTorch and CUDA, and all the experiments are conducted on a Linux machine with 2.90GHz Intel Xeon CPU and a single Nvidia RTX 3090 GPU. We test our GPU accelerate global placer on the ISPD 2005 contest benchmarks[19] and the ISPD 2015 contest benchmarks[20] with fence-region constraints removed. DREAMPlace[1] is a global placement framework accelerated by GPU and shows the state-of-the-art solution quality and performance compared to previous CPU-based global placers. Therefore we compare our Xplace with DREAMPlace on the ISPD 2005 and ISPD 2015 contest benchmarks.

In Section 4.1, we empirically test Xplace on the ISPD 2005 as well as the ISPD 2015 contest benchmarks and compare Xplace's performance and quality with DREAMPlace. In Section 4.2, we conduct ablation studies to show the efficiency of our purposed operator-level optimization techniques mentioned in Section 3.1. In Section 4.3, a novel well-designed neural-network is plugged into the framework of Xplace to further improve the quality without invoking large runtime overhead.

4.1 Validation on Contest Benchmarks

We test Xplace on the ISPD 2005 contest benchmarks[19] and the ISPD 2015 contest benchmarks[20]. Statistics of benchmarks are given in Table 1.

On the ISPD 2005 contest benchmarks, we use NTUPlace3[6] to perform legalization and detail placement for both DREAMPlace's and Xplace's GP results. As for the ISPD 2015 contest benchmarks, we use the legalization method in DREAMPlace[1] and the detail placement method in ABCDPlace[21] for both DREAMPlace's and Xplace's GP results. Note that the version of ABCDPlace we used currently does not support *external* placement results in bookshelf[22] format, which are used by the ISPD 2005 contest benchmarks. Therefore we use NTUPlace3[6] instead. For fair comparison on global placement, we first execute the global placement engine in DREAMPlace and invoke the same legalization engine and the same detail placement engine as that used in Xplace.

Quantitative results on the ISPD 2005 contest benchmarks are presented in Table 2. We compare the HPWL, the global placement time and the detail placement time (including legalization time

This work mainly focuses on developing an extremely fast yet neural-network friendly framework. We will handle the fence region constraints in the future.

https://github.com/limbo018/DREAMPlace/tree/fe53b65a23. We observe this version of DREAMPlace's source codes hasn't yet optimized their CUDA kernel launch bound parameters for the NVIDIA Ampere GPU architecture which RTX 3090 is powered by. For fair comparison, we optimize the launch bound parameters in DREAMPlace's source code and report the experimental results based on the modified version.

Table 1: Benchmarks Statistics

Benchmarks	Design	#cells	#nets	Design	#cells	#nets
ISPD 2005	adaptec1	211k	221k	bigblue1	278k	284k
	adaptec2	255k	266k	bigblue2	558k	577k
	adaptec3	452k	467k	bigblue3	1097k	1123k
	adaptec4	496k	516k	bigblue4	2177k	2230k
ISPD 2015	fft_1	35k	33k	des_perf_1	113k	113k
	fft_2	35k	33k	des_perf_a	108k	115k
	fft_a	34k	32k	des_perf_b	113k	113k
	fft_b	34k	32k	edit_dist_a	127k	134k
	matrix_mult_1	160k	159k	matrix_mult_b	146k	152k
	matrix_mult_2	160k	159k	matrix_mult_c	146k	152k
	matrix_mult_a	154k	154k	pci_bridge32_a	30k	34k
	superblue12	1293k	1293k	pci_bridge32_b	29k	33k
	superblue14	634k	620k	superblue11_a	926k	936k
	superblue19	522k	512k	superblue16_a	680k	697k

Table 2: HPWL($\times 10^6$) and runtime (seconds) results on the ISPD 2005 contest benchmarks[19].

Benchmarks	DREAMPlace[1]			Xplace			Xplace-NN		
	HPWL	GP/s	DP/s	HPWL	GP/s	DP/s	HPWL	GP/s	DP/s
adaptec1	72.89	4.15	34.9	72.93	1.35	35.8	**72.84**	2.53	34.5
adaptec2	81.84	3.73	46.2	**81.04**	1.58	45.4	81.17	2.91	45.2
adaptec3	191.68	4.54	88.1	**190.94**	2.38	89.6	191.04	3.47	88.4
adaptec4	173.45	4.90	95.4	172.41	2.85	96.1	**172.38**	4.12	94.0
bigblue1	89.39	4.03	42.3	89.12	1.47	42.1	**89.07**	2.75	41.8
bigblue2	136.57	4.68	129.3	136.56	2.41	127.2	**136.27**	3.56	129.0
bigblue3	302.58	8.05	207.9	301.36	5.49	209.8	**301.26**	7.66	210.6
bigblue4	742.95	13.38	459.7	741.18	11.65	463.1	**740.44**	15.07	465.7
Sum	1791.36	47.46	1103.6	1785.6	29.18	1109.0	1784.47	42.07	1109.2
Ratio	1.003	1.626	0.995	1.000	1.000	1.000	0.999	1.442	1.000

and detail placement time) with DREAMPlace. Experimental results show that Xplace (without NN) achieves around 1.6x GP time speedup over DREAMPlace and produces better solution quality.

Quantitative results on the ISPD 2015 contest benchmarks are presented in Table 4. The ISPD 2015 contest benchmarks are for detailed-routability driven placement. However, the commercial evaluation tool is currently inaccessible. Therefore, similar to [14], we use NTUplace4dr [7] and the embedded NCTUgr[23] to report the solution quality (HPWL) and the global routing (GR) routability (top5 overflow). Note that top5 overflow measures routability by taking the average overflow of the top 5% most congested GR gcells. Experimental results show that Xplace achieves around 3x GP time speedup than DREAMPlace and produces better HPWL with comparable top5 overflow. Considering both benchmarks together, Xplace can achieve around 2x GP runtime speedup over DREAMPlace.

4.2 Ablation Studies

We perform ablation studies of our proposed operator-level optimization techniques to demonstrate their effectiveness. We measure the runtime performance by the per global iteration time. Quantitative results are shown in Table 3. It is worth noting that operator combination, operator extraction and operator skipping techniques mainly boost the runtime performance of the larger cases while the operator reduction technique accelerates the smaller cases. These results also show that Xplace can achieve around 3x per GP iteration time speedup compared with DREAMPlace on average.

4.3 Neural-Enhanced Performance

The proposed model is a light-weight neural-network with $471k$ parameters, which is only 60% of the well-known image-to-image

Table 3: Ablation Studies of the Operator-Level Optimization Techniques on the ISPD 2005 benchmarks[19]. OR, OC, OE, OS refer to operator reduction, operator combination, operator extraction, and operator skipping respectively. Note that Xplace enables all the operator-level optimization techniques in Section 3.1.

Methods	OR	OC	OE	OS	adaptec1	adaptec2	adaptec3	adaptec4	bigblue1	bigblue2	bigblue3	bigblue4	Avg
Ratio	-	-	-	-	234%	194%	136%	124%	198%	140%	123%	121%	159%
	✓	-	-	-	110%	109%	113%	115%	105%	115%	119%	118%	113%
	✓	✓	-	-	107%	107%	107%	108%	104%	108%	113%	112%	108%
	✓	✓	✓	-	104%	102%	104%	104%	102%	104%	106%	105%	104%
Xplace	Ratio				100%	100%	100%	100%	100%	100%	100%	100%	100%
	GP / Iter Time (ms)				1.478	1.671	2.325	2.688	1.572	2.441	4.974	10.018	-
DREAMPlace	Ratio				462%	345%	288%	254%	376%	288%	199%	158%	296%
	GP / Iter Time (ms)				6.832	5.769	6.699	6.840	5.915	7.023	9.904	15.831	-

Table 4: HPWL($\times 10^6$), Top5 overflow, and runtime (seconds) results on the ISPD 2015 contest benchmarks[20]. OVFL-5 stands for top5 overflow. The benchmarks with fence region constraints removed are labeled with †.

Benchmarks	DREAMPlace [14]				Xplace			
	HPWL	OVFL-5	GP/s	DP/s	HPWL	OVFL-5	GP/s	DP/s
des_perf_1	1107.5	65.28	3.71	1.31	**1106.7**	64.35	1.14	1.23
fft_1	411.7	56.19	3.59	0.67	**411.3**	56.34	1.17	0.61
fft_2	**374.0**	47.72	4.28	0.69	374.3	47.49	1.18	0.64
fft_a	627.6	35.12	3.60	0.61	**625.6**	34.7	1.29	0.70
fft_b	**845.7**	51.82	3.57	0.74	846.2	52.02	1.28	0.73
matrix_mult_1	2129.2	81.02	3.71	1.61	**2116.4**	81.69	1.29	1.44
matrix_mult_2	2163.3	77.61	3.97	1.63	**2152.9**	77.95	1.23	1.48
matrix_mult_a	3036.8	48.10	4.04	2.79	**3031.7**	48.34	1.29	3.68
superblue12	25803.0	92.45	8.91	16.37	**25783.8**	93.18	4.64	17.29
superblue14	**23015.5**	63.56	4.63	11.49	23017.1	64.34	1.60	13.74
superblue19	15633.1	61.82	4.56	8.26	**15544.1**	62.39	1.46	6.60
des_perf_a†	2020.5	53.27	3.66	2.04	**1998.6**	52.32	1.18	1.67
des_perf_b†	**1610.3**	54.65	3.66	1.70	1612.6	53.64	1.27	1.58
edit_dist_a†	4217.9	80.30	3.97	2.31	**4198.7**	80.10	1.45	2.13
matrix_mult_b†	2786.7	44.86	3.82	1.98	**2765.7**	44.98	1.29	1.89
matrix_mult_c†	**2672.9**	42.13	4.07	2.07	2675.2	42.20	1.29	1.89
pci_bridge32_a†	361.8	30.55	3.54	0.82	**356.0**	30.36	1.08	0.69
pci_bridge32_b†	741.1	22.89	6.77	1.04	**714.2**	22.75	1.12	1.04
superblue11_a†	**33411.2**	54.51	5.59	13.33	33528.3	54.78	2.87	12.73
superblue16_a†	25600.9	65.85	4.38	10.60	**25505.1**	65.85	1.91	11.08
Sum	148571	1129.70	88.03	82.06	148364	1129.77	31.03	82.84
Ratio	1.001	1.000	2.837	0.991	1.000	1.000	1.000	1.000

model U-Net [24]. We conduct the training based on ISPD 2005 contest benchmarks with their respective macro layouts. The standard cells are randomly generated at a starting position and are pushed all over the map with only the density objective $D(p)$ for 100 iterations. The density map and electric fields at every iteration are used as training data and labels. The model is tested on real cases collected at every iteration of the ISPD 2005 contest benchmarks placement.

We explore embedding this neural-network into Xplace. The HPWL of Xplace-NN achieves around 1‰ improvement in comparison with the original Xplace on the ISPD 2005 contest benchmarks as shown in Table 2. The results not only illustrate the effectiveness of the proposed network model but also show the high extensibility of our Xplace framework.

5 CONCLUSIONS

In this work, we propose Xplace, a new, fast and extensible GPU accelerated global placement framework. Experimental results on the ISPD 2005 and the ISPD 2015 benchmarks show that Xplace is efficient yet extensible. Future works would include handling additional constraints in placement like routability and fence regions.

REFERENCES

[1] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," *IEEE TCAD*, vol. 40, no. 4, pp. 748–761, 2020.

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

[3] N. K. Darav, A. Kennings, A. F. Tabrizi, D. Westwick, and L. Behjat, "Eh? placer: A high-performance modern technology-driven placer," *ACM TODAES*, vol. 21, no. 3, pp. 1–27, 2016.

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

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

[6] 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.

[7] C.-C. Huang, H.-Y. Lee, B.-Q. Lin, S.-W. Yang, C.-H. Chang, S.-T. Chen, Y.-W. Chang, T.-C. Chen, and I. Bustany, "Ntuplace4dr: a detailed-routing-driven placer for mixed-size circuit designs with technology and region constraints," *IEEE TCAD*, vol. 37, no. 3, pp. 669–681, 2017.

[8] 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.

[9] W. Zhu, Z. Huang, J. Chen, and Y.-W. Chang, "Analytical solution of poisson's equation and its application to vlsi global placement," in *Proc. ICCAD*, 2018.

[10] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE TCAD*, vol. 38, no. 9, pp. 1717–1730, 2018.

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

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

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

[14] J. Gu, Z. Jiang, Y. Lin, and D. Z. Pan, "Dreamplace 3.0: multi-electrostatics based robust vlsi placement with region constraints," in *Proc. ICCAD*, IEEE, 2020.

[15] A. Paszke, S. Gross, and et al., "Pytorch: An imperative style, high-performance deep learning library," in *Proc. NeurIPS*, vol. 32, 2019.

[16] 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, pp. 1–34, 2015.

[17] S. N. Adya, I. L. Markov, and P. G. Villarrubia, "On whitespace and stability in mixed-size placement and physical synthesis," in *Proc. ICCAD*, IEEE, 2003.

[18] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar, "Fourier neural operator for parametric partial differential equations," in *Proc. ICLR*, 2021.

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

[20] I. S. Bustany, D. Chinnery, J. R. Shinnerl, and V. Yutsis, "Ispd 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement," in *Proc. ISPD*, pp. 157–164, ACM, 2015.

[21] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "Abcdplace: Accelerated batch-based concurrent detailed placement on multithreaded cpus and gpus," *IEEE TCAD*, vol. 39, no. 12, pp. 5083–5096, 2020.

[22] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Toward cad-ip reuse: The marco gsrc bookshelf of fundamental cad algorithms," *IEEE Design and Test*, 2002.

[23] 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 TCAD*, vol. 32, no. 5, pp. 709–722, 2013.

[24] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Proc. MICCAI*, pp. 234–241, Springer, 2015.