OpenILT: An Open Source Inverse Lithography Technique Framework

(Invited Paper)

Su Zheng, Bei Yu, Martin Wong
Chinese University of Hong Kong

***Abstract*—Semiconductor lithography is a key process for fabricating integrated circuits, but it suffers from various distortions and variations that affect the quality of the printed patterns. Optical proximity correction (OPC) is a technique to improve pattern fidelity and robustness, and inverse lithography technique (ILT) is a promising OPC method that optimizes the mask as an inverse problem of the imaging system. However, ILT is computationally expensive and challenging to implement at a full-chip scale. In this paper, we present OpenILT, an open-source ILT platform that supports the rapid development and evaluation of GPU-accelerated and AI-driven ILT methods. OpenILT provides a modular and flexible framework that integrates various ILT components, such as lithography simulation, objective functions, and evaluation metrics. It also offers a convenient interface to PyTorch, a popular deep learning library, to enable the implementation of GPU-accelerated and AI-driven ILT methods.**

I. Introduction

Semiconductor lithography, an essential process for fabricating integrated circuits (ICs), involves transferring circuit patterns from a mask onto a silicon wafer. This process accounts for around 30% of the total cost of IC manufacturing. As the feature size of ICs has continuously shrunk over the past few decades in accordance with Moore's Law, the lithography proximity effect has become more significant, leading to increased distortions and defects in printed patterns. This phenomenon makes it increasingly difficult to achieve high pattern fidelity and mask printability. Additionally, minor variations in lithography conditions can greatly affect the quality of the printed wafer image. Ensuring the correctness of semiconductor lithography is therefore a critical issue.

Optical proximity correction (OPC) [1]–[9] is a technique used to improve the accuracy and quality of lithography patterns on semiconductor wafers. The first-generation OPC is rule-based OPC [1], which uses pre-computed look-up tables based on width and spacing between features to adjust the patterns on the mask. At advanced technology nodes, OPC evolves to model-based approaches [2] that can leverage extra patterns to improve the resilience to manufacturing variation. As semiconductor technology advances, OPC algorithms face greater challenges in meeting the higher requirements in process window and correctness.

Inverse lithography technique (ILT) [10]–[20] is an important field for optical proximity correction (OPC), treating mask optimization as an inverse problem of the imaging system. It aims at optimizing the carefully designed objective function and adjusting the pixel-wise mask backward. A variety of attempts have been made in ILT to improve both the printed pattern fidelity and the process robustness. It has been explored and developed as the next generation of OPC, promising a solution to challenges of advanced technology such as extreme ultraviolet (EUV).

However, there exist challenges that limit the broad application of ILT. A major reason is that ILT typically consumes a significant amount of runtime, making it challenging to implement ILT at a full-chip scale. To alleviate the problems above, the GPU acceleration of ILT has been adopted in recent works [12], [13]. Deep-learning-based ILT algorithms have also been proposed to reduce the runtime [14]–[20]. Nonetheless, we still lack a general platform that can support the rapid development and evaluation of the ILT method under the designated settings. A convenient interface to a deep learning library is also needed to boost the development of AI for ILT.

In this paper, we present an open-source ILT platform, OpenILT, which aims to facilitate the research on ILT algorithms. OpenILT leverages GPU acceleration and deep learning capabilities to enable efficient and effective ILT algorithm development. With effective acceleration, convenient interfaces, and comprehensive evaluation, OpenILT can make ILT research easier.

II. Preliminaries

Figure 1 shows a typical ILT flow. The parameters to be optimized are transformed into the mask image. The optical projection and photoresist models convert the mask image to the printed image. The cost function minimizes the distance between the printed and target images. The gradient is back-propagated to optimize the parameters. The optimized mask can be obtained after certain optimization iterations.

A. Lithography Simulation Model

The unconstrained parameters $\boldsymbol{P}$ is transformed to the input mask $\boldsymbol{M}$ via a sigmoid function:

$$\boldsymbol{M}(x,y) = \sigma_M(\boldsymbol{P}(x,y)) = \frac{1}{1+\exp\left(-\theta_M \boldsymbol{P}(x,y)\right)}, \quad (1)$$

where θ_M controls the steepness of the sigmoid function. This transformation limits the values of $\boldsymbol{M}$ to the range $[0,1]$.

The lithography simulation process is composed of two components, an optical projection model and a photoresist

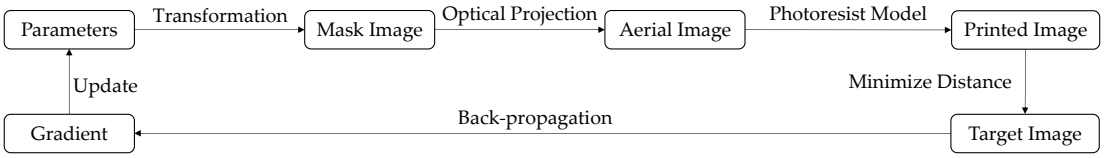


Fig. 1 Typical ILT flow. The forward pass mainly consists of the optical projection model and photoresist model. To minimize the error between the printed and target images, we need to back-propagate the gradient to the parameters.

model. The Hopkins diffraction model [21] is employed to approximate the projection behavior. Mathematically, the aerial image $\boldsymbol{I}$ is obtained by applying a set of optical kernels $\boldsymbol{H}$ to the mask $\boldsymbol{M}$, which can be formulated as:

$$\boldsymbol{I}(x,y) = \sum_{k=1}^{N_h} w_k \left| \boldsymbol{M}(x,y) \otimes \boldsymbol{h}_k(x,y) \right|^2, \tag{2}$$

where "$\otimes$" represents the convolution operation. N_h is the number of optical kernels, which is 24 in our implementation, $\boldsymbol{h}_k$ is the k-th optical kernel in $\boldsymbol{H}$, and w_k is the corresponding weight. After optical projection, the aerial image $\boldsymbol{I}$ is input into the photoresist model. Its intensity threshold I_{th} indicates the exposure level. The printed image $\boldsymbol{Z}$ is computed by the following function:

$$\boldsymbol{Z}(x,y) = \sigma_Z(\boldsymbol{I}(x,y)) = \frac{1}{1+\exp\left(-\theta_Z(\boldsymbol{I}(x,y) - I_{th})\right)}, \tag{3}$$

where θ_Z controls the steepness of the model. Note that after optimization, both $\boldsymbol{M}$ and $\boldsymbol{Z}$ should be binarized with a threshold of 0.5 before evaluation.

B. Evaluation Metrics

a) Square L_2 Error: Given the target image $\boldsymbol{Z}_t$ and the printed image $\boldsymbol{Z}_{nom}$, which represents the image printed via nominal lithography process condition, the square L_2 loss is calculated by $||\boldsymbol{Z}_{nom} - \boldsymbol{Z}_t||_2^2$.

b) Edge placement error (EPE): The EPE violations are counted by sampling a series of points along the contour of the target design. The EPE score is the number of points where the distances between the target and printed images are larger than an EPE constraint th_{EPE}.

c) Process Variation Band (PVB): Under varying lithography conditions, printed images can be different. Given the images printed via the maximum and minimum lithography conditions, $\boldsymbol{Z}_{max}$ and $\boldsymbol{Z}_{min}$, PVB is $||\boldsymbol{Z}_{max} - \boldsymbol{Z}_{min}||_2^2$.

d) Shot Count (#Shots): #Shots [16] counts the rectangles needed to construct the mask. It can evaluate the complexity of an optimized mask.

C. Cost Function

To iteratively improve the mask, ILT algorithms usually adopt cost functions that can optimize the evaluation metrics like L_2 and PVB. For example, it is common to employ the following cost function:

$$\begin{aligned} &L(\boldsymbol{Z}_t, \boldsymbol{Z}_{nom}, \boldsymbol{Z}_{max}, \boldsymbol{Z}_{min}) = \\ &||\boldsymbol{Z}_{nom} - \boldsymbol{Z}_t||_2^2 + ||\boldsymbol{Z}_{max} - \boldsymbol{Z}_t||_2^2 + ||\boldsymbol{Z}_{min} - \boldsymbol{Z}_t||_2^2. \end{aligned} \tag{4}$$

At each iteration, the parameters can be updated by $\frac{\partial L}{\partial \boldsymbol{P}}$.

III. OpenILT Platform

A. Overview

OpenILT provides the following modules:

1) **Lithography simulation**. The optical projection and photoresist models are integrated into a class that implements PyTorch Module interfaces, providing simple but efficient lithography simulation functionality.
2) **Initialization**. Since parameter initialization strongly affects the performance of the ILT process, OpenILT supports different initialization schemes and allows users to develop novel initialization strategies.
3) **Solver**. The ILT task is completed by a solver, which gets the initialized mask, calls lithography simulation, and iteratively updates the parameters. ILT researchers and developers can easily develop their own solvers using user-friendly interfaces.
4) **Evaluation**. Users can evaluate the L_2, PVB, EPE, and #Shots metrics of the optimized masks by calling the simple evaluation functions.

B. Lithography Simulation

A convolution operation in Equation (2) involves two matrices that have large sizes (e.g. 2048×2048), which can be very time-consuming. Thus, the convolution operation is usually implemented by fast Fourier transformation (FFT) to improve efficiency, which can be formulated as:

$$\boldsymbol{h}_k \otimes \boldsymbol{M} = \texttt{IFFT}(\texttt{FFT}(\boldsymbol{h}_k) \odot \texttt{FFT}(\boldsymbol{M})), \tag{5}$$

where `FFT` and `IFFT` represent the FFT and inverse FFT operations, respectively. The notation, $\odot$, represents pointwise multiplication. Computing the optical projection with Equation (5) can significantly reduce the time complexity, since using `FFT` and `IFFT` for $N \times N$ images requires $O(N^2 \log N)$, whereas directly computing the convolution costs $O(N^4)$.

The `LithoSim` class inherits the PyTorch Module class and implements the lithography simulation. Given the mask `M`, the printed images can be obtained by the following code:

```
Znom, Zmax, Zmin = litho(M)
```

`Znom`, `Zmax`, `Zmin` represent the printed images at the nominal, maximum, and minimal process corners, respectively. `litho` is an instance of the `LithoSim` class, whose `forward` function outputs the lithography simulation results.

OpenILT's lithography model comes from ICCAD-13 [23] contest. We implement an exact and a simple `LithoSim`

TABLE I Comparison Between Reproduced and Original Methods

Benchmarks	MOSAIC [11] EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)	Our MOSAIC EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)	LevelSet [12] EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)	Our LevelSet EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)
case1	6	49893	65534	318	8	48896	55028	0.95	4	46032	62693	123	6	45520	57468	2.49
case2	10	50369	48230	256	4	37327	46019	0.95	1	36177	50724	81	1	33571	49680	2.27
case3	59	81007	108608	321	47	81327	86685	0.94	29	71178	100945	214	39	78695	90748	2.26
case4	1	20044	28285	322	2	16409	26358	0.94	0	16345	29831	184	2	18040	27710	2.27
case5	6	44656	58835	315	0	37810	57472	0.94	1	47103	56510	76	2	38226	59035	2.26
case6	1	57375	48739	314	0	36706	52566	0.94	1	46205	51204	65	0	35962	54163	2.27
case7	0	37221	43490	239	2	29520	47598	0.94	0	28609	45056	64	2	30542	48173	2.27
case8	2	19782	22846	258	1	14291	24268	0.94	1	19477	22757	67	1	14252	25043	2.26
case9	6	55399	66331	322	2	47367	64932	0.94	0	52613	64597	63	1	43390	68229	2.26
case10	0	24381	18097	231	0	8950	19871	0.94	0	22415	18769	64	0	8919	20878	2.27
Average	9.1	44012	50899	289	**6.6**	**35860**	**48080**	**0.94**	**3.7**	38615	50309	100	5.4	**34712**	**50113**	**2.29**

TABLE II Comparison Between Reproduced and Original Methods

Benchmarks	GAN-OPC [14] EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)	Our GAN-OPC EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)	MultiLevel [22] EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)	Our MultiLevel EPE	L_2 (nm^2)	PVB (nm^2)	Time (s)
case1	-	55425	58043	-	20	58712	52126	1.13	3	39303	46077	1.42	4	38577	47367	1.03
case2	-	40211	53020	-	1	36669	43861	1.13	0	28986	37626	1.24	1	32104	37572	1.03
case3	-	93090	75644	-	51	85677	68400	1.12	22	66151	68021	1.42	20	64245	72910	1.03
case4	-	22877	26401	-	1	15812	27559	1.13	0	15890	23511	0.72	0	10880	23270	1.03
case5	-	42650	59765	-	5	46249	55090	1.13	0	29138	49987	1.43	0	30454	51915	1.03
case6	-	39776	54878	-	0	37489	51545	1.13	0	30558	44503	1.42	0	30504	46394	1.03
case7	-	22761	49156	-	0	26882	45715	1.14	0	15765	37009	1.43	0	16056	39412	1.03
case8	-	16296	24441	-	0	14654	24076	1.13	0	13943	21503	0.8	0	11560	19991	1.03
case9	-	52157	66492	-	5	51179	61939	1.14	0	36397	55600	1.43	0	36017	58943	1.03
case10	-	9765	21338	-	0	9066	20121	1.12	0	7492	16604	1.42	0	8533	15942	1.03
Average	-	39501	48918	-	**8.3**	**38239**	**45043**	**1.13**	**2.5**	28362	**40044**	1.27	**2.5**	**27893**	41372	**1.03**

classes, which share the same forward pass while having different backward computations. The exact lithography simulator follows the Hopkins diffraction model to compute the gradient. The simple lithography simulator implements the speedup technique in [11] that combines the kernels together in the back-propagation step, utilizing the feature $\sum_k w_k(\boldsymbol{M} \otimes \boldsymbol{h}_k) = \boldsymbol{M} \otimes \sum_k (w_k \boldsymbol{h}_k)$.

C. Initialization

The most common way to initialize the parameters is to use the target image. Given the target image $\boldsymbol{Z}_t$, the parameters $\boldsymbol{P}$ can be initialized by $\boldsymbol{P} = \boldsymbol{Z}_t$ or $\boldsymbol{P} = 2\boldsymbol{Z}_t - 1$.

The following code can easily generate the initial $\boldsymbol{P}$:

```
init = Initializer()
Zt,P = init.run(design,X,Y,dX,dY)
```

`Zt` and `P` denotes the target image and parameters, respectively. `design` contains the target shapes described using rectilinear polygons. `X` and `Y` indicate the size of the image. `dX` and `dY` specify the offset between the coordinates of the polygons and their positions on the target image.

Level-set ILT algorithms [12] employ a different initialization scheme, which is also implemented in OpenILT. The corresponding `Initializer.run` function initializes the value of each pixel by calculating its distance to the nearest edge or corner of the target shapes. Note that the values inside the shapes are negative while the values outside are positive.

D. Solver

Users can implement the transformation from $\boldsymbol{P}$ to $\boldsymbol{M}$ using the sigmoid function provided by PyTorch. The lithography simulation ($\boldsymbol{M} \rightarrow \boldsymbol{Z}$) is done by the `LithoSim` class. After that, the loss function can also be implemented using PyTorch functions. PyTorch optimizers such as SGD and Adam can be utilized to automatically calculate the gradient and update the parameters.

OpenILT provides four examplary solvers: MOSAIC [11], LevelSet [12], MultiLevel [22], GAN-OPC [14]. The first three solvers are traditional ILT algorithms with our GPU acceleration. GAN-OPC is an ILT method based on deep learning, showing that OpenILT can facilitate the development of AI-driven algorithms.

MOSAIC and LevelSet use Equation (4) as the cost function. MultiLevel improves the performance of MOSAIC by employing the average pooling mechanism and multi-resolution ILT scheme. GAN-OPC uses a generative adversarial network (GAN) to give a better initial mask for ILT. We try to reproduce these methods by using a similar number of iterations, learning rate, neural network architecture, etc.

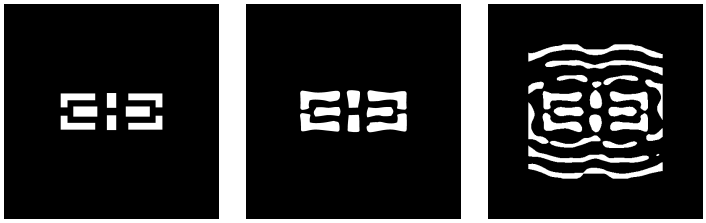
E. Evaluation

OpenILT has three classes for evaluation, `Basic`, `EPEChecker`, and `ShotCounter`. `Basic` calculates the L_2 and PVB metrics. `EPEChecker` and `ShotCounter` estimate the EPE and #Shots, respectively. Each evaluation class uses a `run` function to get the result. The following function can easily evaluate all metrics.

```
l2,pvb,epe,shot = evaluate(M,Zt,litho)
```

F. Implmentation

Given the utilities in OpenILT, we can efficiently reproduce existing methods. Our MOSAIC, LevelSet, MultiLevel, and



(a) (b) (c)

Fig. 2 Examples of (a) target; (b) our LevelSet; (c) our MultiLevel.

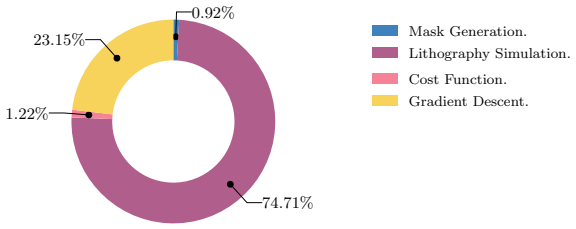


Fig. 3 Runtime analysis.

GAN-OPC only takes 64, 93, 102, and 241 lines of Python code, respectively. The following example shows how to do ILT using MOSAIC solver within a few lines of code.

```
cfg = SimpleCfg()
litho = LithoSim()
solver = SimpleILT(cfg, litho)
design = Design("M1_test1.glp")
Zt,P = PixelInit().run(design)
l2,pvb,P,M = solver.solve(Zt,P)
l2,pvb,epe,shot = evaluate(M,Zt,litho)
```

The seven lines of code load the configuration, get the lithography simulation model, instantiate the MOSAIC solver, read the design, initialize the parameters, optimize the mask, and evaluate the result.

IV. Experiments

A. Reproduction of Existing Methods

TABLE I compares the results of our reproduced MOSAIC/LevelSet algorithms with the results reported in the original papers. With the help of GPU acceleration, our MOSAIC is significantly faster than the original MOSAIC, achieving 321× speedup. Compared to the original GPU-accelerated LevelSet algorithm [12], our implementation has 44× speedup. TABLE II presents the comparison of GAN-OPC and MultiLevel. For these two methods, our reproduced versions can also achieve comparable performance as the original ones. Fig. 2 presents some examples of our reproduced results.

B. Runtime Analysis

Fig. 3 presents the runtime analysis of our MOSAIC. Lithography simulation is the most time-consuming process, which is caused by the FFT and IFFT operations. The gradient descent process also consumes a considerable amount of time. According to the analysis, future development of lithography simulation acceleration is promising to boost the speed of ILT algorithms.

V. Conclusion

OpenILT is an open-source platform for inverse lithography technology (ILT) research. It has a comprehensive and flexible ecosystem of libraries that enable the efficient development and evaluation of ILT algorithm. The platform is implemented with PyTorch, which enables easy GPU acceleration and deep-learning integration. It is available at https://github.com/OpenOPC/OpenILT.

References

[1] J.-S. Park *et al.*, "An efficient rule-based opc approach using a drc tool for 0.18/spl mu/m asic," in *Proc. ISQED*, 2000, pp. 81–85.

[2] A. Awad, A. Takahashi, S. Tanaka, and C. Kodama, "A fast process variation and pattern fidelity aware mask optimization algorithm," in *Proc. ICCAD*, 2014, pp. 238–245.

[3] J. Ou *et al.*, "Directed self-assembly based cut mask optimization for unidirectional design," in *Proc. GLSVLSI*, 2015, pp. 83–86.

[4] Y. Ma, X. Zeng, and B. Yu, "Methodologies for layout decomposition and mask optimization: A systematic review," in *Proc. VLSI-SoC*, 2017.

[5] Y. Ma, J.-R. Gao, J. Kuang, J. Miao, and B. Yu, "A unified framework for simultaneous layout decomposition and mask optimization," in *Proc. ICCAD*, 2017, pp. 81–88.

[6] W. Zhong *et al.*, "Deep learning-driven simultaneous layout decomposition and mask optimization," in *Proc. DAC*, 2020.

[7] H. Yang *et al.*, "VLSI mask optimization: From shallow to deep learning," *Integration, the VLSI Journal*, vol. 77, pp. 96–103, 2021.

[8] W. Zhao *et al.*, "AdaOPC: A self-adaptive mask optimization framework for real design patterns," in *Proc. ICCAD*, 2022.

[9] Z. Yu, P. Liao, Y. Ma, B. Yu, and M. D. Wong, "CTM-SRAF: Continuous transmission mask-based constraint-aware sub resolution assist feature generation," *IEEE TCAD*, 2023.

[10] Y. Liu, D. Abrams, L. Pang, and A. Moore, "Inverse lithography technology principles in practice: Unintuitive patterns," in *BACUS Symposium on Photomask Technology*, vol. 5992, 2005, pp. 886–893.

[11] J.-R. Gao, X. Xu, B. Yu, and D. Z. Pan, "MOSAIC: Mask optimizing solution with process window aware inverse correction," in *Proc. DAC*, 2014.

[12] Z. Yu, G. Chen, Y. Ma, and B. Yu, "A gpu-enabled level set method for mask optimization," in *Proc. DATE*, 2021, pp. 1835–1838.

[13] ——, "A GPU-enabled level-set method for mask optimization," *IEEE TCAD*, vol. 42, no. 2, pp. 594–605, 2022.

[14] H. Yang, S. Li, Y. Ma, B. Yu, and E. F. Young, "GAN-OPC: Mask optimization with lithography-guided generative adversarial nets," in *Proc. DAC*, 2018.

[15] H. Yang *et al.*, "GAN-OPC: Mask optimization with lithography-guided generative adversarial nets," *IEEE TCAD*, vol. 39, no. 10, pp. 2822–2834, 2019.

[16] B. Jiang *et al.*, "Neural-ILT: Migrating ILT to neural networks for mask printability and complexity co-optimization," in *Proc. ICCAD*, 2020.

[17] B. Jiang, L. Liu, Y. Ma, B. Yu, and E. F. Young, "Neural-ILT 2.0: Migrating ilt to domain-specific and multitask-enabled neural network," *IEEE TCAD*, vol. 41, no. 8, pp. 2671–2684, 2021.

[18] G. Chen, W. Chen, Y. Ma, H. Yang, and B. Yu, "DAMO: Deep agile mask optimization for full chip scale," in *Proc. ICCAD*, 2020.

[19] G. Chen *et al.*, "DAMO: Deep agile mask optimization for full-chip scale," *IEEE TCAD*, vol. 41, no. 9, pp. 3118–3131, 2022.

[20] G. Chen, Z. Yu, H. Liu, Y. Ma, and B. Yu, "DevelSet: Deep neural level set for instant mask optimization," in *Proc. ICCAD*, 2021.

[21] H. H. Hopkins, "The concept of partial coherence in optics," in *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, vol. 208, no. 1093. The Royal Society London, 1951, pp. 263–277.

[22] S. Sun, F. Yang, B. Yu, L. Shang, and X. Zeng, "Efficient ilt via multi-level lithography simulation," in *Proc. DAC*, 2023.

[23] S. Banerjee, Z. Li, and S. R. Nassif, "ICCAD-2013 CAD contest in mask optimization and benchmark suite," in *Proc. ICCAD*, 2013, pp. 271–274.