DREAMPlace 2.0: Open-Source GPU-Accelerated Global and Detailed Placement for Large-Scale VLSI Designs

Yibo Lin[1*], David Z. Pan[2], Haoxing Ren[3], and Brucek Khailany[3]

[1]CS Department, Peking University
[2]ECE Department, The University of Texas at Austin
[3]NVIDIA, Inc.
[*]yibolin@pku.edu.cn

***Abstract*—Modern backend design flow for very-large-scale-integrated (VLSI) circuits consists of many complicated stages and requires long turn-around time. Among these stages, VLSI placement plays a fundamental role in determining the physical locations of standard cells. Due to increasingly large design sizes, placement algorithms usually require long execution time to achieve high-quality solutions. Meanwhile, developing a placer often needs huge coding effort and tedius tuning, raising the bar of further researches. In this work, we present an open-source placement framework, *DREAMPlace 2.0*[1], with deep learning toolkit-enabled GPU acceleration for both global and detailed placement optimization to tackle the issues of efficiency and development overhead.**

I. Introduction

Placement plays critical role to design closure of the VLSI backend flow. It decides the physical locations of standard cells in the layout, which will significantly affect the solution space of the succeeding routing stages.

The placement problem takes a circuit netlist and a standard cell library as input. It needs to place the cells in a fixed-outline layout region with no overlaps between cells and all the design rules satisfied. In modern designs, cells have to be placed in discrete placement sites. The objective of placement includes wirelength, routability, timing, and so on. As the problem is $\mathcal{NP}$-hard [1], [2], placement is often divided into three optimization steps: global placement (GP), legalization (LG), and detailed placement (DP). GP relaxes the discrete cell locations to continuous ones and roughly distributes cells in the layout. LG removes all the overlaps between cells and clean all design rule violations. DP is a refinement step to improve the objective with incremental movement of cells.

As a classic problem, there are many existing efforts. The algorithms for GP can be categorized into quadratic placement and nonlinear placement. Quadratic placement iterates between two phases: an unconstrained quadratic programming phase to minimize wirelength, and a heuristic spreading phase to remove overlaps. Typical quadratic placers include FastPlace [3], Polar [4], [5], Ripple [6], SimPL/ComPlx [7], [8], etc. Nonlinear placement formulates a nonlinear optimization problem and tries to directly solve it with gradient descent methods. There are many nonlinear placers such as mPL6 [9], APlace [10], NTUplace families [11]–[13], ePlace/RePlAce families [14], [15]. Generally speaking, nonlinear placement can achieve better solution quality, while quadratic placement is more efficient.

As GP takes a majority of the runtime in placement, there are also works to accelerate GP algorithms such as POLAR 3.0 [5] and UTPlaceF 3.0 [16] with multi-threading. The acceleration ratio is around $5\times$ with $2-6\%$ quality degradation. Cong et al also explored GPU acceleration for multi-level GP algorithms [17], where $15\times$ speedup was achieved with less than 1% quality degradation.

Besides GP, algorithms for DP are mostly based on heuristic approaches with local searching. Widely used DP algorithms include independent set matching, local reordering, global swap/move, row-based techniques [11], [12], [18]–[20], etc. Dhar et al [21] explored GPU acceleration for a row-based interleaving algorithm.

Despite the previous works, current placement engines suffer from following issues: 1) poor quality-efficiency tradeoff; 3) high development effort. As mentioned, multi-threading on CPU can only achieve limited speedup with high quality gradation [5], [16]. GPU acceleration requires huge development effort and lacks systematic frameworks [17], [21].

To tackle these challenges and stimulate researches on placement, we present an open-source *DREAMPlace 2.0* framework with GPU acceleration enabled by deep learning toolkit `PyTorch` for both global and detailed placement. The major contributions can be summarized as follows.

- We develop the placement algorithms with deep learning toolkit to decouple the algorithmic design and the kernel operator development with reduced coding overhead.
- We present a full placement flow including GP, LG, and DP with both multi-threading and GPU acceleration.
- Experimental results demonstrate that the framework can accelerate the entire placement flow by $14\times$, while matching the state-of-the-art solution quality [11], [15].

Meanwhile, the framework offers an easy way to explore new solvers developed in the deep learning toolkit, e.g., Adam [22], and stochastic gradient descent (SGD) with momentum. The rest of the paper is organized as follows. Section II explains the formulation of placement and the DREAMPlace framework in details. Section III validates the framework with experimental results. Section IV concludes the paper.

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

II. DREAMPlace Framework

A. Placement Problem

In this work, we consider wirelength as the objective.

Problem 1. *Given a netlist, a standard cell library, and a fixed-outline layout, determine the physical locations of movable cells with minimum wirelength.*

In the GP step, we solve a nonlinear placement problem,

$$\begin{aligned} \min_{\mathbf{x},\mathbf{y}} \quad & WL(\mathbf{x},\mathbf{y}), \\ \text{s.t.} \quad & D_i(\mathbf{x},\mathbf{y}) \le t_d, \quad \forall i \in B, \end{aligned} \tag{1}$$

where $\mathbf{x},\mathbf{y}$ denote the coordinates of cells in the layout, which is divided into a set of bins B uniformly. $WL(\cdot)$ denotes the wirelength cost, $D_i(\cdot)$ denotes the density at bin i, and t_d is a given target density. Intuitively, if the density constraints are satisfied at all bins, it indicates that cells have already spread out with very small overlaps. The LG step then legalizes the solution and removes all overlaps between cells. The DP step further refines the solution while maintaining the legality.

B. Software Architecture

As the placement framework is developed with deep learning toolkit, we adopt the same software architecture that separates high-level optimization algorithms with low-level operators, as shown in Figure 1. The algorithms are developed and assembled in `Python`, while low-level operators are highly optimized in `C++/CUDA`.

For GP, we rely on two important operators, wirelength and density, with forward and backward functions to compute the cost and the gradients, respectively. We propose both multi-threading and GPU accelerations for these operators. The gradient descent solvers can be implemented in `Python` with the automatic gradient derivation package in `PyTorch`. Solvers in `PyTorch` like Adam [22] and SGD can be used for optimization. We further develop a custom solver based on Nesterov's method with line search [15], which is among the state-of-the-art optimization techniques for placement.

For LG and DP, each technique is developed as an operator in `C++/CUDA` and assembled in `Python`. For LG, we implement a greedy legalization technique in NTUplace3 [11] and a row-based Abacus algorithm [23] on CPU. For DP, three techniques have been developed, independent set matching, local reordering, and global swap [11], [19]. To accelerate the techniques, we propose parallel variations based on batch execution for multi-threading and GPU.

C. Main Features

As an open-source project, we are actively incorporating new features. With the release of DREAMPlace 2.0, the main features so far are summarized as follows.

- Multi-threading and GPU accelerated GP and DP for weighted wirelength minimization.
- Movable macros supported in GP and LG.
- Bookshelf and LEF/DEF formats supported.

Users can specify whether running on CPU only or GPU only according their machines. In other words, GPU is not mandatory.

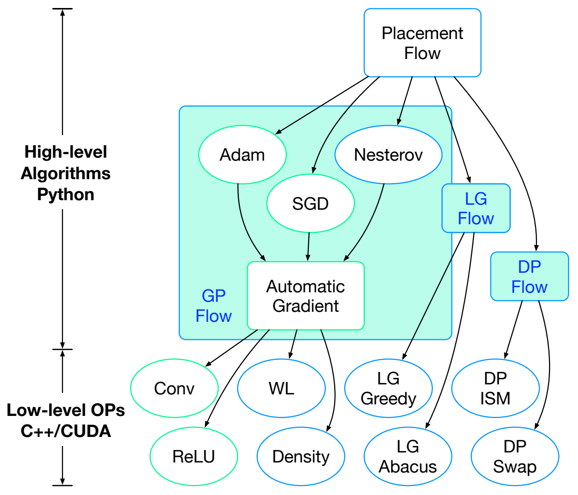

Fig. 1 Software architecture of DREAMPlace. The circles and rectangles with blue boundaries denote the portions we need to implement, while those with green boundaries are offered by `PyTorch`.

D. Global Placement Algorithm

The GP algorithm adopts the state-of-the-art family of placers ePlace/RePlAce, which model the layout as an electrostatic system [14], [15]. It uses the weighted average wirelength for the $WL(\cdot)$ term to approximate the nonsmooth half-perimeter wirelength (HPWL) [24], [25].

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

where γ is a parameter to control the smoothness and accuracy of the approximation.

Its density term $D(\cdot)$ comes from the analogy to an electrostatic system, where cells are modeled as charges, density penalty is modeled as potential energy, and density gradient is modeled as the electric field. By solving Poisson's equation, the electric potential and field distribution can be computed from the charge density distribution.

$$\nabla \cdot \nabla \psi(x,y) = -\rho(x,y), \tag{3a}$$
$$\hat{\mathbf{n}} \cdot \nabla \psi(x,y) = \mathbf{0}, \quad (x,y) \in \partial R, \tag{3b}$$
$$\iint_R \rho(x,y) = \iint_R \psi(x,y) = 0, \tag{3c}$$

where R denotes the placement region, ∂R denotes the boundary to the region, $\hat{\mathbf{n}}$ denotes the outer normal vector of the placement region, ρ denotes the charge density, and ψ denotes the electric potential. To solve the constrained optimization in Equation (1), we relax the constraints into the objective with λ as the Lagrangian multiplier,

$$\min_{\mathbf{x},\mathbf{y}} \quad WL(\mathbf{x},\mathbf{y}) + \lambda D(\mathbf{x},\mathbf{y}), \tag{4}$$

where by gradually increasing the λ, the overlaps between cells can be eliminated.

E. Legalization Algorithm

In the LG step, we first perform macro legalization by ignoring the standard cells. Two macro legalization techniques

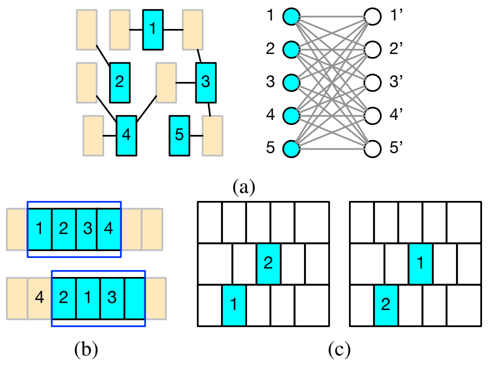

Fig. 2 Three DP techniques: (a) independent set matching; (b) local reordering; (c) global swap.

are developed. The first one is based on greedy spiral search of legal locations for movable macros. Once the overlaps between macros are resolved, we perform min-cost flow based refinement to minimize the displacement following the relative orders between macros [26]. After macro legalization, we fix the movable macros and legalize standard cells based on the Tetris-like approach in NTUplace3 [11], where cells are sorted from left to right and we find spaces to host cells one by one. Once a legal solution is found, we perform row-based Abacus refinement to shift cells for minimum displacement subjecting to the relative orders [23].

F. Detailed Placement Algorithm

In the DP step, three batch-based concurrent detailed placement techniques are proposed: independent set matching, local reordering, and global swap. Figure 2 illustrates the sequential version of the techniques [11], [19]. Independent set matching extracts a small set of cells that are independent (do not share common nets) to each other and perform permutation to the locations with bipartite matching. Local reordering slides a small window within each row and enumerates all the permutations for the best cost. Global swap performs pair-wise swapping of cells to improve the objective. As cells are connected, these techniques can only work on a small set of cells each time, assuming all other cells fixed. Thus, it is difficult to parallelize them.

In DREAMPlace, we modify the algorithms by incorporating batched execution to enable massive parallelization. For independent set matching, instead of extracting independent cells within a small window, we extract a maximal independent set from the entire netlist and partitions the set into many small subsets based on locality. Then, the bipartite matching problems of all the subsets can be solved independently. Be aware that the partitioning is not constrained by any physical window. For local reordering, instead of sliding a window within a row sequentially, we first construct a dependency graph for all rows, where two rows are dependent if any cell in a row is connected with another cell in the other row. With the dependency graph, we can extract independent sets of rows and perform the algorithm on these rows in parallel. For global swap, instead of finding a swap pair of cells one by one, we simultaneously search for swap candidates for a batch of cells and compute the best swapping candidates without considering any conflict. When realizing the swap, we adopt sequential execution with a predetermined order and omit the swap candidates that have conflicts with previous swaps. In this way, we avoid possible data race in parallelization.

III. Experimental Results

The framework was developed in `Python` with `PyTorch` for optimizers and API, and `C++/CUDA` for low-level operators. `OpenMP` was adopted to support multi-threading on CPU. All experimental results are collected from a Linux machine with two Intel 20-core Gold 6230 CPUs @ 2.10GHz (40 cores in total) and one NVIDIA RTX 2080Ti GPU. Due to page limit, we only show the experiments on ISPD 2005 contest benchmarks [27]. More results can be found in [28].

We compare with the state-of-the-art placer RePlAce [15] (use NTUplace3 [11] for legalization and detailed placement) in Table I. Results of 40 threads and GPU with different solvers are reported. Without any quality degradation, DREAMPlace can achieve $1.3\times$ speedup over RePlAce with 40 threads on CPU, and $14\times$ speedup on GPU. The major speedup comes from the GP step. Solvers like Nesterov, Adam and SGD with momentum are also compared, where Nesterov can provide similar solution quality to Adam, but around $2\times$ faster in GP. SGD momentum leads to 1.2% worse wirelength than the other two.

Table II shows the lines of code summarized from `cloc` [29] under Linux. As RePlAce only implemented GP and NTUplace3 is not open-source, we can only compare the lines of code for GP. We can see that DREAMPlace-GP only requires two-third of lines of code compared with RePlAce, while we support both CPU and GPU. It needs to mention that among the 20K lines in DREAMPlace-GP, more than 8K lines are for IO and database construction. Although we count these lines into DREAMPlace-GP for fair comparison with RePlAce, it indicates that the lines for core algorithms are actually even fewer.

IV. Conclusion

In this work, we present *DREAMPlace 2.0*, an open-source placement framework with multi-threading and GPU acceleration enabled by the deep learning toolkit `PyTorch`. Experimental results demonstrate that with GPU acceleration, more than $14\times$ speedup over the state-of-the-art RePlAce using 40 CPU threads can be achieved on the entire placement flow. With the decoupled algorithmic design and kernel operator development, we not only show reduced coding effort, but also easy adoption of new solvers from the deep learning community. Future work includes routability and timing consideration.

Acknowledgement

This is support in part by NVIDIA, Inc.

References

[1] S. Chowdhury, "Analytical approaches to the combinatorial optimization in linear placement problems," *IEEE TCAD*, vol. 8, no. 6, pp. 630–639, 1989.

[2] R. G. Michael and S. J. David, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[3] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in *Proc. ASPDAC*, 2007, pp. 135–140.

TABLE I Comparison on ISPD 2005 Contest Benchmarks with `float32`.

Design	#cells	#nets	RePlAce (40 threads)						DREAMPlace (40 threads)					
			HPWL	Runtime (s)					HPWL	Runtime (s)				
				GP	LG	DP	IO	Total		GP	LG	DP	IO	Total
adaptec1	211K	221K	73.23	89	4	20	5	118	73.19	67	0.6	9	4	84
adaptec2	254K	266K	81.86	175	6	25	7	214	82.11	167	0.5	14	5	190
adaptec3	451K	467K	193.32	342	19	44	13	419	193.22	338	1	18	9	372
adaptec4	495K	516K	175.17	411	18	51	14	494	174.08	274	1	23	9	313
bigblue1	278K	284K	89.86	156	3	25	8	192	89.39	92	0.4	11	5	113
bigblue2	535K	577K	138.14	354	21	72	14	461	136.86	350	8	15	10	390
bigblue3	1093K	1123K	304.94	971	39	110	28	1147	304.05	984	3	46	20	1067
bigblue4	2169K	2230K	743.60	2171	50	274	60	2555	744.11	1361	8	91	44	1536
ratio			1.002	79.478	9.485	9.965	1.018	14.099	1.000	66.023	1.041	3.832	0.725	10.743

Design	DREAMPlace (GPU-Nesterov)						DREAMPlace (GPU-Adam)			DREAMPlace (GPU-SGD)		
	HPWL	Runtime (s)					HPWL	Runtime (s)		HPWL	Runtime (s)	
		GP	LG	DP	IO	Total		GP	Total		GP	Total
adaptec1	73.18	2	0.5	3	8	15	73.04	4	17	73.75	4	17
adaptec2	82.11	3	0.5	4	8	18	82.51	5	19	83.65	5	20
adaptec3	193.33	4	1	6	12	30	190.76	8	32	197.55	8	33
adaptec4	174.08	5	2	6	13	31	172.61	9	35	175.98	10	36
bigblue1	89.36	3	0.4	3	9	18	90.04	6	21	89.57	5	20
bigblue2	136.97	4	8	6	13	38	136.96	9	42	137.84	10	44
bigblue3	304.40	10	3	10	23	59	302.99	25	75	313.28	19	69
bigblue4	744.01	21	8	15	48	122	742.54	57	157	745.03	51	151
ratio	1.000	1.000	1.000	1.000	1.000	1.000	0.998	2.153	1.156	1.012	2.114	1.151

TABLE II Comparison of lines of code using `cloc` [29].

	C/C++ Header	C++	CUDA	Python	Total
RePlAce (GP)	2174	27930	-	-	30104
DREAMPlace-GP	7320	7485	3555	3499	21859
DREAMPlace-LG	1814	863	-	336	3013
DREAMPlace-DP	2404	2390	7150	632	12576

[4] 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.

[5] T. Lin, C. Chu, and G. Wu, "POLAR 3.0: An ultrafast global placement engine," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 520–527.

[6] X. He, T. Huang, W.-K. Chow, J. Kuang, K.-C. Lam, W. Cai, and E. F. Y. Young, "Ripple 2.0: High quality routability-driven placement via global router integration," in *Proc. DAC*, 2013, pp. 152:1–152:6.

[7] 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.

[8] M.-C. Kim and I. L. Markov, "Complx: A competitive primal-dual lagrange optimization for global placement," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 747–755.

[9] 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, pp. 247–288.

[10] A. B. Kahng, S. Reda, and Q. Wang, "Aplace: A high quality, large-scale analytical placer," in *Modern Circuit Placement*. Springer, 2007, pp. 167–192.

[11] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1228–1240, 2008.

[12] 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 3, pp. 669–681, 2017.

[13] W. Zhu, Z. Huang, J. Chen, and Y.-W. Chang, "Analytical solution of poisson's equation and its application to vlsi global placement," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[14] 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 Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 2, p. 17, 2015.

[15] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "RePlAce: Advancing solution quality and routability validation in global placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[16] W. Li, M. Li, J. Wang, and D. Z. Pan, "UTPlaceF 3.0: A parallelization framework for modern fpga global placement," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 922–928.

[17] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *2009 IEEE/ACM International Conference on Computer-Aided Design-Digest of Technical Papers*. IEEE, 2009, pp. 681–688.

[18] S. W. Hur and J. Lillis, "Mongrel: hybrid techniques for standard cell placement," in *Proc. ICCAD*, 2000, pp. 165–170.

[19] M. Pan, N. Viswanathan, and C. Chu, "An efficient and effective detailed placement algorithm," in *Proc. ICCAD*, 2005, pp. 48–55.

[20] W.-K. Chow, J. Kuang, X. He, W. Cai, and E. F. Y. Young, "Cell density-driven detailed placement with displacement constraint," in *Proc. ISPD*, 2014, pp. 3–10.

[21] S. Dhar and D. Z. Pan, "GDP: Gpu accelerated detailed placement," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, 2014.

[23] 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.

[24] 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.

[25] 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.

[26] X. Tang, R. Tian, and M. D. F. Wong, "Optimal redistribution of white space for wire length minimization," in *Proc. ASPDAC*, 2005, pp. 412–417.

[27] 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.

[28] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," in *Proc. DAC*. Las Vegas, NV: ACM, June 2019.

[29] "Aldanial/cloc," https://github.com/AlDanial/cloc.