LSMC Meets GPU Acceleration: Scalable and High-Quality Multi-Row Detailed Placement

†‡Andrew B. Kahng, ‡Jason Liang, ‡Zhiang Wang
†CSE and ‡ECE Departments, UC San Diego, La Jolla, CA, USA.
{abk, jsliang, zhw033}@ucsd.edu

***Abstract*—Detailed placement is a crucial stage in VLSI physical design, optimizing wirelength, timing, routability and power under complex constraints such as edge spacing, site alignment and fence regions. With the aggressive increase of design utilization and adoption of multi-row height cells in advanced technology nodes, existing detailed placers struggle with efficiency and quality, often becoming trapped in local minima. In this work, we develop *GPU-DPO*, a fast and high-quality GPU-accelerated detailed placement framework built on top of the OpenROAD infrastructure, which leverages Large-Step Markov Chain techniques to escape local optima and improve placement quality. *GPU-DPO* is the first GPU-accelerated detailed placer with the capability of fully supporting movable and reorderable multi-row height cells. Experimental results on testcases with varying utilization demonstrate that, in comparison with *DPO* [42] (the default detailed placer in OpenROAD) and *ABCDPlace* [27] (the state-of-the-art GPU-accelerated detailed placer), our approach achieves an average reduction of post-detailed placement half-perimeter wirelength (HPWL) by 1.71% and 3.5% respectively, while consuming similar runtime as *ABCDPlace*.**

I. Introduction

Detailed placement is a critical optimization phase in VLSI, refining cell positions to minimize objectives like wirelength and timing while adhering to strict legality rules [19], [28], [40]. It is frequently reinvoked during backend closure to recover placement quality after incremental changes, making both efficiency and solution quality essential.

Classical detailed placers optimize small cell subsets via techniques such as independent set matching [3], global swap [32], local reordering [32], and row-based refinement [16]. However, sub-10nm scaling, reduced track counts [5], and growing use of multi-row height cells [12], [26], [35] have substantially increased placement complexity, with existing multi-row placers [9], [13], [29], [37], [39] facing scalability and runtime challenges. The GPU-accelerated ABCDPlace [27] achieves speedup but treats multi-row cells as fixed, potentially degrading placement quality. In addition, at high utilization [28], relocating or reordering multi-row cells becomes difficult, causing classical detailed placers to frequently get trapped in local minima that leave room for improvement.

In this work, we propose a novel and efficient open-source GPU-accelerated detailed placement framework that leverages Large-Step Markov Chain (LSMC) [1] to escape local optima, enabling high-quality optimization even in congested, high-utilization designs. Our main contributions are as follows.

- We propose *GPU-DPO*, a fast, high-quality detailed placer that leverages the LSMC approach to improve solution quality in high-utilization designs. It is the first GPU-accelerated placer to fully support *movable and reorderable* multi-row height cells, including intra-row reordering, while handling constraints such as edge spacing, site alignment, and fence regions [40].
- *GPU-DPO* is built on top of the OpenROAD [43] infrastructure with a permissive open-source license, enabling other researchers to readily adapt it for other enhancements.[1]
- Experiments show that across varying utilizations, *GPU-DPO* reduces post-detailed placement half-perimeter wirelength (HPWL) by 1.71% and 3.5% compared to *DPO* [42] (the default detailed placer in OpenROAD) and *ABCDPlace* [27] (the state-of-the-art GPU-accelerated detailed placer), respectively, with runtime comparable to *ABCDPlace* in most cases.
- For extremely high utilization ($\geq$ 80%), *GPU-DPO* achieves significantly better post-detailed placement HPWL compared to *DPO* and *ABCDPlace*, demonstrating the effectiveness of the LSMC-based framework in escaping local minima.

The remaining sections are organized as follows. Section II presents our approach. Section III shows experimental results, and Section IV concludes the paper.

II. Our Approach

The architecture of our proposed *GPU-DPO* framework is illustrated in Figure 1. The input is a legal placement solution (.def file) that contains placed cells (with macros fixed, if any) and fixed IO pins. The output is an enhanced legal placement solution (.def file).[2] *GPU-DPO* consists of two major steps:

- **GPU-Accelerated Detailed Placement (Descent)** (Section II-A): We redesign widely-adopted greedy detailed placement techniques (maximum independent set matching, global swap and local reordering) to handle multi-row height cells, and propose parallel versions for multi-threaded CPUs and GPUs. The proposed GPU-accelerated detailed placement techniques enable multiple refinement passes within a runtime similar to that of sequential detailed placers.
- **Large-Step Markov Chain (LSMC) Booster** (Section II-B): We incorporate the GPU-accelerated detailed placement techniques as part of complex neighborhood move operators within the LSMC framework, enabling *GPU-DPO* to escape local minima and achieve improved placement quality.

We now explain these steps in detail; source code is in [45].

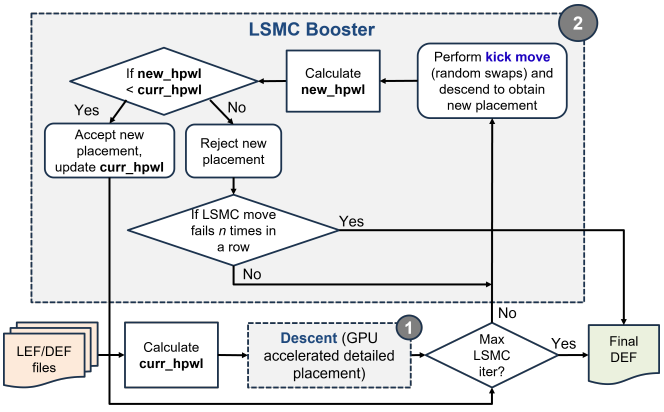

Fig. 1: Overview of the proposed LSMC-based *GPU-DPO* flow.

[1]To support the research community's efforts, we accompany our paper with all runscripts as well as permissively open-sourced code in the GitHub repository [45].

[2]Our detailed placement framework does not modify the netlist.

A. GPU-Accelerated Detailed Placement Kernels

OpenROAD's detailed placement engine implements independent set matching, global swap, local reorder, and flipping. Our detailed placement flow mirrors this sequence of operators but parallelizes the first three, which dominates runtime.

Maximum Independent Set Matching. Independent set matching groups same-height cells that do not share nets, forming move sets that are independent with respect to the HPWL objective. Because no nets are shared, all cell locations in a set can be simultaneously reassigned by solving a small linear assignment problem. Multiple independent sets are processed concurrently, and within each set, the cost of assigning cells to legal sites is evaluated in parallel before computing an optimal one-to-one matching. Multi-row height cells are naturally supported as long as all cells in the set share the same height.

Global Swap. Global swap improves wirelength by exchanging pairs of movable cells. For each cell, we define an optimal destination region using the median bounding box of its incident nets [32]. Specifically, for a cell c, we exclude c from each incident net and compute the bounding box of the remaining pins. The left, right, bottom, and top edges of these boxes across all nets induce a multiset of coordinates. The optimal region for c is then defined by the median x-coordinates and median y- coordinates of the bounding box edges. Swap candidates within this region are evaluated, and the lowest-cost legal swap that reduces HPWL is applied.

Similar to *ABCDPlace*, we adopt a batch-based concurrent global swap strategy to achieve more scalability. We group a batch of B cells and launch GPU threads to concurrently compute candidate regions, evaluate swap costs, and select optimal swaps. Algorithm 1 gives details, denoting the parallel and sequential steps.

Algorithm 1 Concurrent Global Swap

Input: Initial placement $(\mathbf{x}, \mathbf{y})$, netlist $G = (V, N)$, batch size B
Output: Updated placement $(\mathbf{x}^*, \mathbf{y}^*)$

for all cells $c \in V$ **do** ▷ Parallel
  $\mathcal{R}(c) \leftarrow$ ComputeOptimalRegion(c)
end for
for each batch $B_v \subseteq V$ of size B **do**
  $\mathcal{C} \leftarrow$ CollectSwapCandidates($B_v, \mathcal{R}$) ▷ Parallel
  $\mathcal{S} \leftarrow$ EvaluateSwapCosts($\mathcal{C}$) ▷ Parallel
  $\mathcal{B} \leftarrow$ SelectBestSwaps($\mathcal{S}$) ▷ Parallel reduction
  for each swap $s \in \mathcal{B}$ **do** ▷ Sequential conflict resolution
    if s is legal (spacing, alignment, height fits) **then**
      ApplySwap(s)
    end if
  end for
end for

When evaluating multi-row height cells, we require only that the destination span has sufficient rows, is unoccupied, and meets spacing and alignment constraints; no strict size matching between swap pairs is enforced.

Local Reordering. Local reordering finds the optimal permutation of a sequence of consecutive cells within a defined window. Unlike *ABCDPlace*, which restricts reordering to single-row, fixed-size windows, our formulation supports multi-row movement and full reordering of both single- and multi-row height cells. To avoid factorial complexity from full permutation enumeration, we adopt a dynamic programming (DP) formulation inspired by the key idea in [9], which reformulates multi-row cell refinement as a one-dimensional ordering problem and solves it using a dynamic programming (DP) recurrence to explore only legality-preserving placement states. We extend this concept to a GPU-friendly formulation with full support for multi-row height cells and cross-row reordering.

Before delving into details of our DP approach, we introduce several key notations, as follows.

- *Cell ordering* [9]: Given an m-row (window) initial placement, cells in the window are ordered from left to right based on their rightmost boundary, forming a one-dimensional sequence $c_1, c_2, \ldots, c_k$. If two cells share the same rightmost x-coordinate, ties are broken using the y-coordinate of their lower boundary.
- *Site definition*: Let S denote the set of available sites within the window. Site j is denoted as s_j, indexed from the lower-left boundary to the upper-right boundary.
- *Cell-to-site assignment*: A cell c_i is said to be placed at site s_j if and only if the bottom-left corner of c_i aligns with the bottom-left corner of s_j.

In the DP procedure, cells are placed sequentially according to the established order. The DP table entry $dp[i][j]$ denotes the minimum cost solution where the i^{th} cell is assigned to legal site s_j, with i indicating that i cells have been placed. When considering the assignment of cell c_i to site s_j, the algorithm (i) ensures that the site can accommodate the full height of the cell, (ii) verifies vertical site alignment and power/ground rail compatibility, and (iii) confirms that sites in all rows spanned by the cell are unoccupied. We furthermore enforce additional displacement and inter-row spacing constraints. These checks are embedded within the state expansion logic, allowing early pruning of infeasible transitions and preventing the propagation of invalid assignments.

The DP recursion is formulated as follows: for each cell i and each legal site j, the optimal prior placement i of cell $i-1$ is identified, and the DP table is updated by

$$dp[i][j] = \min_{l} \left(dp[i-1][l] + \Delta\mathrm{HPWL}_{i,j} \right) \tag{1}$$

where $\Delta\mathrm{HPWL}_{i,j}$ captures the impact of a cell's placement on bounding box sizes of incident nets. $\Delta\mathrm{HPWL}_{i,j}$ is calculated as summing up the ΔHPWL contribution of all nets incident to c_i, as described in [18].

The detailed algorithm is presented in Algorithm 2. The layout is partitioned into multiple windows, each consisting of m (m = 3 by default) rows, and processed in parallel. Relevant net and site information is cached in shared memory, and DP state transitions are cooperatively computed by multiple threads. Bounding boxes are incrementally updated to evaluate HPWL deltas, and the optimal legal assignment is recovered via backtracking. If the resulting permutation yields a reduction in total wirelength, the new placement is committed.[3]

Algorithm 2 Multi-Row Local Reordering

Input: Initial placement $(\mathbf{x}, \mathbf{y})$, rows R, sites S, window size m
Output: Updated placement $(\mathbf{x}^*, \mathbf{y}^*)$

Partition R into windows $\{w\}$
for all windows $\{w\}$ **do**
  Extract cells $\{c_1, c_2, \ldots, c_i, \ldots, c_k\}$ in window w
  Extract sites $\{s_1, s_2, \ldots, s_j, \ldots, s_l\}$ in window w
  Initialize $dp[0][j] \leftarrow 0$, $dp[i][j] \leftarrow \infty$ **for** $i > 0$ ▷ Parallel
  Sort $\{c_1, c_2, \ldots, c_k\}$ by x-coordinate
  for $i = 1$ to k **do**
    for each site s_j **do**
      if placing c_i at s_j is legal **then**
        Continue
      end if
      */*** Parallel reduction for calculating the minimum cost ***/*
      $dp[i][j] = \min_l \left(dp[i-1][l] + \Delta\mathrm{HPWL}_{i,j} \right)$
    end for
  end for
  $(\mathbf{x}^*, \mathbf{y}^*) \leftarrow \arg\min_j dp[k][j]$ ▷ Parallel reduction
end for
return $(\mathbf{x}^*, \mathbf{y}^*)$

[3]We refer the reader to [45] for the detailed implementation.

B. Large-Step Markov Chain Booster

Classical detailed placement techniques (see Section II-A) often struggle in high-density regions due to limited legal space, with little improvement even after many additional iterations. To overcome this, we incorporate the Large-Step Markov Chain (LSMC) heuristic [1], [7], [31] for escaping local minima into our detailed placement framework. LSMC initially finds a local optimum solution according to some greedy "descent" search (in our case, sequentially applying maximum independent set matching, global swap and local reordering). Its core idea is to perturb the current local optimum via a "kick move" into the starting solution of the next greedy descent. As shown in Figure 1, each LSMC iteration begins in some local optimum solution state, then applies the "kick move" and the descent search to reach a new local optimum. If the new local optimum is better than the previous one, it is adopted as the starting solution for the next iteration. Otherwise, the previous local optimum is retained.[4]

Kick moves are implemented as some number of legal random swaps of cells. The effectiveness of LSMC depends heavily on the size of kick moves [7]. Large kick moves offer more opportunities for escaping poor local minima, but can severely disrupt nearly optimal placements, making recovery during descent search difficult and adversely affecting both runtime and solution quality. Conversely, small kick moves may fail to escape the current "basin of attraction". Our experimental results demonstrate that, with appropriately chosen kick moves and an efficient descent search strategy, LSMC is effective in escaping the poor local minima that are frequently encountered by traditional detailed placers.

Although the LSMC procedure is inherently sequential, the kick moves are computationally lightweight and well suited for CPU execution, while the computationally intensive descent search is parallelized on the GPU. This heterogeneous approach achieves superior placement quality within a runtime similar to that of sequential detailed placers. The detailed workflow is described in Algorithm 3. An early exit mechanism [Lines 17–19] is implemented to terminate the LSMC procedure earlier if no improvement is observed after a predefined number of consecutive iterations F (F = 5 by default).

Algorithm 3 LSMC Booster

Input: Initial cell placement $(\mathbf{x}_0, \mathbf{y}_0)$, kick ratio $k \in (0, 1]$, max iterations T, max failure tolerance F, cost function C
Output: Optimized placement $(\mathbf{x}^*, \mathbf{y}^*)$

1: $\mathbf{x} \leftarrow \mathbf{x}_0$, $\mathbf{y} \leftarrow \mathbf{y}_0$
2: $(\mathbf{x}, \mathbf{y}) \leftarrow$ DESCENT($\mathbf{x}$, $\mathbf{y}$) ▷ Executed on GPU
3: $\mathbf{x}^* \leftarrow \mathbf{x}$, $\mathbf{y}^* \leftarrow \mathbf{y}$ ▷ Initial best solution found
4: $C^* \leftarrow C(\mathbf{x}^*, \mathbf{y}^*)$ ▷ Initial best cost
5: $N \leftarrow$ number of movable cells
6: $f \leftarrow 0$ ▷ Keep track of LSMC failures
7: $n_k \leftarrow \lfloor k \cdot N \rfloor$ ▷ Total kick moves (random cell swaps)
8: **for** $t = 1$ to T **do**
9: $(\mathbf{x}_k, \mathbf{y}_k) \leftarrow$ KICKMOVE($\mathbf{x}^*, \mathbf{y}^*$) using n_k random legal cell swaps ▷ Executed on CPU
10: $(\mathbf{x}_d, \mathbf{y}_d) \leftarrow$ DESCENT($\mathbf{x}_k, \mathbf{y}_k$) ▷ Executed on GPU
11: **if** $C(\mathbf{x}_d, \mathbf{y}_d) < C^*$ **then** ▷ New best local min found
12: $\mathbf{x}^* \leftarrow \mathbf{x}_d, \mathbf{y}^* \leftarrow \mathbf{y}_d$
13: $C^* \leftarrow C(\mathbf{x}^*, \mathbf{y}^*)$
14: $f \leftarrow 0$ ▷ Reset LSMC failures
15: **else**
16: $f \leftarrow f + 1$ ▷ Keep track of LSMC failures
17: **if** $f = F$ **then**
18: **return** $(\mathbf{x}^*, \mathbf{y}^*)$
19: **end if**
20: **end if**
21: **end for**
22: **return** $(\mathbf{x}^*, \mathbf{y}^*)$

III. Experimental Results

GPU-DPO is implemented with C++ and CUDA with a Tcl command line interface on top of the OpenROAD infrastructure. We run all experiments on a Linux server with an AMD Epyc 7742 64-core CPU (128 threads) with 503 GB RAM and an NVIDIA A100-SXM4-80GB GPU. To show the effectiveness of our detailed placer, the following three detailed placers are evaluated and compared:

- *DPO*: Detailed placement is done by *DPO* [42], which is the default detailed placer in the OpenROAD project.
- *ABCDPlace*: Detailed placement is performed by the latest version of *ABCDPlace* [46], which is the state-of-the-art GPU-accelerated detailed placer. In our experiments, we use the default setting in [46].
- *GPU-DPO*: Results are obtained using our detailed placer.

Our experimental flow proceeds as follows. For each testcase, we perform synthesis with Cadence Genus 21.1 and global placement with Cadence Innovus 21.1, producing mixed-height placements subsequently legalized by OpenROAD. We then perform detailed placement using each of the three placers. Post-detailed placement HPWL is reported via OpenROAD, and post-route metrics are obtained from Innovus following post-route optimization [5]. "DP Time" measures kernel runtime, whereas "TAT" represents total runtime, including I/O and GPU overhead in the cases of *ABCDPlace* and *GPU-DPO*.

All experiments use the ASAP7 7nm FinFET PDK [41], supporting multi-row height cells, and three public testcases: AES, JPEG, and Mempool-Group (MP-Group) [43], [44]. Table I summarizes their characteristics. Sections III-A and III-B respectively present post-detailed placement and post-route optimization results, and evaluate QoR across varying utilization. In all experiments, *GPU-DPO* is run with default reorder window size = 3, MIS problem size = 64, and LSMC kick move ratio = 0.10. These were chosen experimentally for an effective balance between HPWL reduction and runtime. In the repository [45], each experiment reported below is mapped to the corresponding runscript(s) used to produce results.

TABLE I: Testcase Specifications

Testcase	#Cells	#Nets	#Multi-row height cells
AES	15347	15975	120
JPEG	61133	63389	1287
MP-Group	2548437	2650624	113

A. Main Results

We first present our main experimental results. Table II compares metrics after detailed placement. Rows represent testcases (utilization) and detailed placement flows, and columns give HPWL (in μm), runtime for detailed placement kernels (DP time, in seconds (s)) and turnaround time for detailed placement (TAT, in s). We observe that *GPU-DPO* consistently outperforms both *DPO* and *ABCDPlace* across all three testcases in terms of post-detailed placement HPWL. On average, *GPU-DPO* achieves 1.71% and 3.5% lower post-detailed placement HPWL compared to *DPO* and *ABCDPlace*, respectively. Notably, for the largest MP-Group testcase (2.5M cells), *GPU-DPO* achieves superior HPWL in just 12% of the runtime required by *DPO*. Although *GPU-DPO* requires more runtime than *ABCDPlace*, an "iso-runtime" comparison demonstrates that allocating additional time to *ABCDPlace* (i.e., running it for five iterations, see Section III-B), does not yield results comparable to those of *GPU-DPO*. This supports our claim in Section II-B that classical detailed placers without the LSMC booster gain negligible improvement even with additional iterations.

[4]Thus, the "large step" in LSMC consists of (kick move + descent). Our version of LSMC may be viewed as zero-temperature annealing in the neighborhood structure induced by this "large step" operator.

[5]Note that we do not benchmark the commercial EDA tool, and no benchmarking should be inferred from our results.

We further examine the post-route wirelength (in μm) and the runtime for detailed router (DR Time, in s). The post-route results are shown in Table III, where NR denotes cases where the tool failed to return a legal placement successfully. On average, *GPU-DPO* achieves 3.1% and 8.5% reduction in routed wirelength compared to *DPO* and *ABCDPlace*, respectively. These results indicate that the detailed placement solutions produced by *GPU-DPO* lead to improved routability and routed wirelength.

TABLE II: QoR Metrics Post-Detailed Placement

Testcase (Util.)	Detailed Placer	HPWL (μm)	DP Time (s)	TAT (s)
AES (0.91)	*DPO*	44823	5	10
	ABCDPlace	45412	**1**	**4**
	GPU-DPO	**44226**	2	4
JPEG (0.72)	*DPO*	96861	34	42
	ABCDPlace	101537	**3**	**10**
	GPU-DPO	**93665**	5	13
MP-Group (0.41)	*DPO*	25089409	1138	1375
	ABCDPlace	NR	NR	NR
	GPU-DPO	**24963574**	**35**	**164**

TABLE III: QoR Metrics After Post-Route Optimization

Testcase (Util.)	Detailed Placer	rWL (μm)	DR Time (s)
AES (0.91)	*DPO*	54434	**123**
	ABCDPlace	59727	125
	GPU-DPO	**52925**	129
JPEG (0.72)	*DPO*	112501	314
	ABCDPlace	116820	330
	GPU-DPO	**109013**	**313**
MP-Group (0.41)	*DPO*	27556757	**482462**
	ABCDPlace	NR	NR
	GPU-DPO	**26854328**	486089

B. *"Solve the Harder Problem": Higher-Utilization Studies*

Next, we systematically evaluate performance of the three detailed placers under conditions of higher placement utilization. We use the AES and JPEG testcases and report both the runtime of the detailed placement kernels (DP Time) and the post-detailed placement HPWL. We sweep utilization across 0.60, 0.70, 0.80, 0.85 and 0.90 for each testcase. Each detailed placer is allowed to perform five iterations of detailed placement operators to generate the final solution. Specifically, this involves five consecutive passes of independent set matching, global swap and local reordering. For *GPU-DPO*, one initial descent pass is followed by four iterations of kick move and descent operations, as described in Algorithm 3. The experimental results are presented in Figure 2 and Figure 3. We observe that our *GPU-DPO* consistently dominates *DPO* and *ABCDPlace* across all utilizations. In the extremely high-utilization case (0.90), *GPU-DPO* achieves approximately 4% lower post-detailed placement HPWL compared to *ABCDPlace*.

We further evaluate the post-route wirelength and the runtime of the detailed router. Experimental results are presented in Tables IV and V: Table IV reports the HPWL after global placement and legalization, and Table V presents the post-detailed placement and post-route metrics. Across all utilizations for both testcases, *GPU-DPO* consistently outperforms both *DPO* and *ABCDPlace* in terms of post-detailed placement HPWL as well as routed wirelength.

TABLE IV: Initial HPWL for Utilization Experiments

Testcase	Util.	Original HPWL (μm)
AES	0.6	50817
	0.8	48685
	0.9	47718
JPEG	0.6	109866
	0.8	108743
	0.9	107384

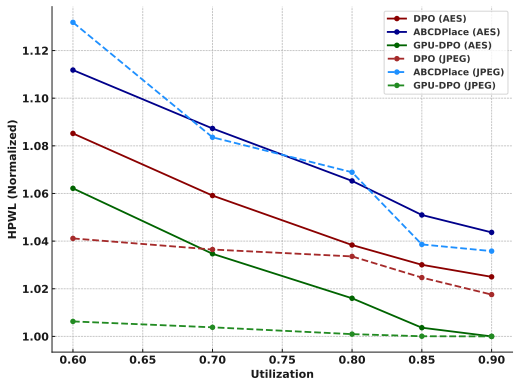

Fig. 2: Post-detailed placement HPWL versus utilization. All values are normalized to the HPWL of *GPU-DPO* for the AES (solid) or JPEG (dashed) testcases at 0.90 utilization.

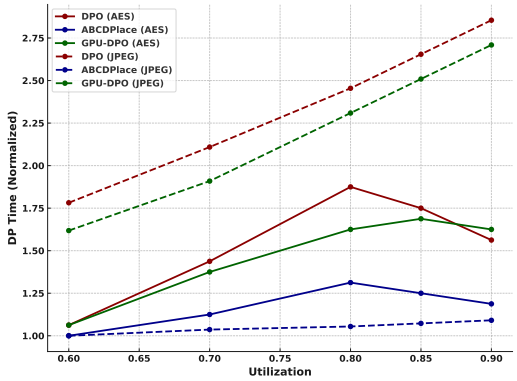

Fig. 3: Runtime for detailed placement kernels (DP Time) versus utilization. All values are normalized to the HPWL of *GPU-DPO* for the AES (solid) or JPEG (dashed) testcases at 0.60 utilization.

TABLE V: Comparisons Across Different Utilizations

Testcase	Detailed Placer	Util.	HPWL (μm)	rWL (μm)	DP Time (s)	DR Time (s)
AES	*DPO*	0.6	49305	58205	17	194
		0.8	47177	56078	30	218
		0.9	46570	55511	25	157
	ABCDPlace	0.6	50514	59194	**16**	180
		0.8	48400	57217	**21**	**177**
		0.9	47418	56180	**19**	**153**
	GPU-DPO	0.6	**48258**	**56706**	17	**135**
		0.8	**46161**	**54544**	26	184
		0.9	**45434**	**53982**	26	167
JPEG	*DPO*	0.6	97235	113009	98	304
		0.8	96530	111843	135	322
		0.9	95034	111282	157	337
	ABCDPlace	0.6	105707	121805	**55**	**291**
		0.8	99830	114853	**58**	**305**
		0.9	96738	111933	**60**	**326**
	GPU-DPO	0.6	**93981**	**109977**	89	298
		0.8	**93483**	**109647**	127	320
		0.9	**93394**	**109383**	149	334

IV. Conclusion

We present *GPU-DPO*, a GPU-accelerated detailed placer integrated into OpenROAD that applies Large-Step Markov Chain techniques. Experiments show that it achieves lower post-placement HPWL than *DPO* and *ABCDPlace*. Ongoing work adds congestion- and pin-aware costs, incremental timing-driven optimization, and advanced constraints such as drain-to-drain separation, minimum implant area, and jog length. OpenROAD integration and open-sourcing position *GPU-DPO* as a foundation for future research on modern, high-quality detailed placers for advanced nodes.

V. Acknowledgments

This work is partially supported by the Samsung AI Center.

REFERENCES

[1] E. B. Baum, "Iterated Descent: A Better Algorithm for Local Search in Combinatorial Optimization Problems", *Unpublished Manuscript*, 1986.

[2] S. Cauley, V. Balakrishnan, Y. C. Hu and C.-K. Koh, "A Parallel Branch-and-cut Approach for Detailed Placement", *ACM Trans. on DAES* 16(2) (2011), pp. 18:1-18:19.

[3] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen and Y.-W. Chang, "NTUplace3: An Analytical Placer for Large-Scale Mixed-Size Designs With Preplaced Blocks and Density Constraints", *IEEE Trans. on CAD* 27(7) (2008), pp. 1228-1240.

[4] W.-K. Chow, J. Kuang, X. He, W. Cai and E. F. Y. Young, "Cell Density-driven Detailed Placement with Displacement Constraint", *Proc. ISPD*, 2014, pp. 3-10.

[5] P. Debacker, K. Han, A. B. Kahng, H. Lee, P. Raghavan and L. Wang, "Vertical M1 Routing-Aware Detailed Placement for Congestion and Wirelength Reduction in Sub-10nm Nodes", *Proc. DAC*, 2017, pp. 1-6.

[6] S. Dhar and D. Pan, "GDP: GPU Accelerated Detailed Placement", *Proc. HPEC*, 2018, pp. 1-7.

[7] A. S. Fukunaga, J. H. Huang and A. B. Kahng, "Large-Step Markov Chain Variants for VLSI Netlist Partitioning", *Proc. ISCAS*, 1996, pp. 496-499.

[8] K. Han, A. B. Kahng and H. Lee, "Scalable Detailed Placement Legalization for Complex Sub-14nm Constraints", *Proc. ICCAD*, 2015, pp. 867-873.

[9] C. Han, A. B. Kahng, L. Wang and B. Xu, "Enhanced Optimal Multi-row Detailed Placement for Neighbor Diffusion Effect Mitigation in Sub-10nm VLSI", *IEEE Trans. on CAD* 38(9) (2019), pp. 1703-1716.

[10] S. Heo, A. B. Kahng, M. Kim, L. Wang and C. Yang, "Detailed Placement for IR Drop Mitigation by Power Staple Insertion in Sub-10nm VLSI", *Proc. DATE*, 2019, pp. 830-835.

[11] I. Hong, A. B. Kahng and B. R. Moon, "Improved Large-Step Markov Chain Variants for the Symmetric TSP", *J. Heuristics* 3(1) (1997), pp. 63-81.

[12] C.-C. Hsu, Y.-C. Chen and M. P.-H. Lin, "In-placement Clock-tree Aware Multi-bit Flip-flop Generation for Power Optimization", *Proc. ICCAD*, 2013, pp. 592-598.

[13] D.-W. Huang, Y.-J. Jiang and S.-Y. Fang, "Spacing Cost-aware Optimal and Efficient Mixed-Cell-Height Detailed Placement for DFM Considerations", *Proc. ICCAD*, 2023, pp. 1-8.

[14] S. W. Hur and J. Lillis, "Mongrel: Hybrid Techniques for Standard Cell Placement", *Proc. ICCAD*, 2000, pp. 165–170.

[15] Z.-W. Jiang, H.-C. Chen, T.-C. Chen and Y.-W. Chang, "Challenges and Solutions in Modern VLSI Placement", *Proc. VLSI-DAT*, 2007, pp. 1–5.

[16] A. B. Kahng, I. L. Markov and S. Reda, "On Legalization of Row-based Placements", *Proc. GLSVLSI*, 2004, pp. 214–219.

[17] A. Kahng, B. Pramanik and M. Woo, "A Hybrid ECO Detailed Placement Flow for Improved Reduction of Dynamic IR Drop", *Proc. GLSVLSI*, 2024, pp. 390-396.

[18] A. B. Kahng, P. Tucker and A. Zelikovsky, "Optimization of Linear Placements for Wirelength Minimization with Free Sites", *Proc. ASP-DAC*, 1999, pp. 241–244.

[19] A. Kennings, N. K. Darav and L. Behjat, "Detailed Placement Accounting for Technology Constraints", *Proc. VLSI-SoC*, 2014, pp. 1-6.

[20] M.-C. Kim, J. Hu, D.-J. Lee and I. L. Markov, "A SimPLR Method for Routability-driven Placement", *Proc. ICCAD*, 2011, pp. 67-73.

[21] S. Li and C. Koh, "Mixed Integer Programming Models for Detailed Placement", *Proc. ISPD*, 2012, pp. 87-94.

[22] S. Li and C. Koh, "MIP-based Detailed Placer for Mixed-size Circuits", *Proc. ISPD*, 2014, pp. 11-18.

[23] D. Lim and H. Park, "Timing-Driven Detailed Placement with Unsupervised Graph Learning", *Proc. DATE*, 2025, pp. 1-7.

[24] Y. Lin, "GPU Acceleration in VLSI Back-end Design: Overview and Case Studies", *Proc. ICCAD*, 2020, pp. 1-4.

[25] T. Lin and C. Chu, "TPL-aware Displacement-driven Detailed Placement Refinement with Coloring Constraints", *Proc. ISPD*, 2015, pp. 75–80.

[26] M. P.-H. Lin, C.-C. Hsu and Y.-T. Chang, "Recent Research in Clock Power Saving with Multi-bit Flip-flops", *Proc. MWSCAS*, 2011, pp. 1-4.

[27] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany and D. Pan, "ABCDPlace: Accelerated Batch-based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs", *IEEE Trans. on CAD* 39(12) (2020), pp. 5083-5096.

[28] Y. Lin, B. Yu and D. Z. Pan, "Detailed Placement in Advanced Technology Nodes: A Survey", *Proc. ICSICT*, 2016, pp. 836-839.

[29] Y. Lin, B. Yu, X. Xu, J.-R. Gao, N. Viswanathan et al., "MrDP: Multiple-row Detailed Placement of Heterogeneous-sized Cells for Advanced Nodes", *Proc. ICCAD*, 2016, pp. 1-8.

[30] Y. Lin, B. Yu, B. Xu and D. Z. Pan, "Triple Patterning Aware Detailed Placement Toward Zero Cross-row Middle-of-line Conflict", *IEEE Trans. on CAD* 36(7) (2017), pp. 1140–1152.

[31] O. Martin, S. Otto and E. W. Felten, "Large-step Markov Chains for the TSP Incorporating Local Search Heuristics", *Operation Research Letters* 11(4) (1992), pp. 219-224.

[32] M. Pan, N. Viswanathan and C. Chu, "An Efficient and Effective Detailed Placement Algorithm", *Proc. ICCAD*, 2005, pp. 48-55.

[33] S. Popovych, H.-H. Lai, C.-M. Wang, Y.-L. Li, W.-H. Liu and T.-C. Wang, "Density-aware Detailed Placement with Instant Legalization", *Proc. DAC*, 2014, pp. 1-6.

[34] H. Ren, D. Z. Pan, C. J. Alpert, G.-J. Nam and P. Villarrubia, "Hippocrates: First-Do-No-Harm Detailed Placement", *Proc. ASP-DAC*, 2007, pp. 141-146.

[35] D. D. Sherlekar, "Cell Architecture for Increasing Transistor Size", U.S. Patent 8631374, Jan. 2014.

[36] J. Vygen, "Algorithms for Detailed Placement of Standard Cells", *Proc. DATE*, 1998, pp. 321–324.

[37] L.-C. Wang and S.-Y. Fang, "Mitigating Layout Dependent Effect-induced Timing Risk in Multi-Row-Height Detailed Placement", *Proc. DATE*, 2023, pp. 1-2.

[38] C.-K. Wang, C.-C. Huang, S. S.-Y. Liu, C.-Y. Chin, S.-T. Hu, W.-C. Wu and H.-M. Chen, "Closing the Gap between Global and Detailed Placement: Techniques for Improving Routability", *Proc. ISPD*, 2015, pp. 149–156.

[39] G. Wu and C. Chu, "Detailed Placement Algorithm for VLSI Design With Double-Row Height Standard Cells", *IEEE Trans. on CAD* 35(9) (2016), pp. 1569-1573.

[40] V. Yutsis, I. S. Bustany, D. Chinnery, J. R. Shinnerl and W.-H. Liu, "ISPD 2014 Benchmarks with Sub-45nm Technology Rules for Detailed-routing Driven Placement", *Proc. ISPD*, 2014, pp. 161–168.

[41] ASAP7 PDK and Cell Libraries. https://github.com/The-OpenROAD-Project/asap7

[42] DPO. https://github.com/The-OpenROAD-Project/OpenROAD/tree/c422dda56f18f479fd707c2362c4d677d76cf043/src/dpl/src/optimization

[43] OpenROAD. https://github.com/The-OpenROAD-Project/OpenROAD

[44] TILOS Macro Placement Benchmarks. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/sept_2025_update

[45] GPU-DPO. https://github.com/ABKGroup/GPU-DPO/tree/main/src/dpl

[46] ABCDPlace implementation. https://github.com/limbo018/DREAMPlace/tree/master/dreamplace/ops