Design and Implementation of Reconfigurable Placement Technique in Soc

Abisha.N[1], Mr.R.Ramesh., M.E [2]
[1]PG scholar
[2]Assistant Professor

***Abstract*—One of the most crucial processes for design closure is placement for very-large-scale integrated (VLSI) circuits. By equating the analytical placement problem to the process of training a neural network, we provide a revolutionary GPU-accelerated placement framework called DREAMPlace. DREAMPlace, which is built on top of the widely used deep learning framework PyTorch, can outperform the state-of-the-art multithreaded placer RePlAce in terms of global placement speed without sacrificing quality by about 40 percent. We think that our effort will pave the way for tackling old EDA issues using modern hardware and software for AI.**

***Index Terms*— GPU, GP, LG, NVIDIA Tesla V100 GPU, Pytorch**

I. INTRODUCTION

In the VLSI design flow, placement is a crucial yet time-consuming step. Its effectiveness greatly affects subsequent steps in the flow, including routing and post-layout optimisation, as it establishes the placements of standard cells in the actual layout. In addition to providing a fairly precise assessment of routed wirelength and congestion, a placement solution is also very helpful in directing earlier stages, such as logic synthesis. To complete a design, commercial design flows frequently use multiple core placement engines. Placement takes hours for complex designs since it requires extensive numerical optimisation, which slows down design iterations. Because of this, extremely quick yet good placement is always preferred.

Analytic placement is the current state of the art for VLSI placement [1]-[15]. It basically solves non-linear optimization problems. Analytical placement can produce high-quality solutions, but it is also known to be relatively slow [11], [13], [14]. Here is a brief introduction to the analytical placement problem. Suppose the circle is described as a hypergraph H = (V, E). where V denotes the set of vertices (cells) and E denotes the set of hyperedges (nets). Let x,y be the cell location. The goal of analytical placement is to minimize route length and place non-overlapping cells in the layout. Analytical placement can be roughly divided into quadratic placement and nonlinear placement. Square placement solves the problem by repeating an unconstrained length minimization step and a crude justification or propagation step [10]–[13]. The wire length minimization step typically uses a quadratic wire length model to minimize the total wire length regardless of overlap between cells.

The coarse-grained legalization step eliminates duplication based on a heuristic approach without explicitly considering the cost of wire length. By repeating these two steps, the cells can be dispersed step by step. At the same time, the cost of cable length is minimized. Nonlinear placement solves the placement problem directly using nonlinear optimization techniques [1]–[9], [12].

It formulates a nonlinear optimization problem with a density-constrained wire length goal. By relaxing the target density constraint, a gradient descent-based approach can be employed. I am looking for a quality solution. This article will focus on nonlinear placement approaches, as many commercial tools such as Cadence Innovus [5] and Synopsys IC Compiler [1] employ nonlinear placement approaches. To speed up placement, existing parallelization efforts are mainly aimed at multithreaded CPUs with partitioning [6], [10], [7]. As the number of threads increases, global placement quickly saturates speed by about a factor of 5, and typically degrades quality by 2-6%. Kong et al. We investigated GPU acceleration for analytical deployment [22].

They combined clustering and declustering with nonlinear placement optimization. By parallelizing the nonlinear placement part, we observed an average speedup of 15x in global placement with less than 1% quality loss. Lynn et al. proposed a GPU-accelerated technique for wire length gradient calculation and area accumulation [23], but their experiments were not considered. Real-world operations such as density cost calculations and validation with real-world analytical placement flows were lacking. Moreover, current deployment research faces challenges due to the lack of well-maintained public frameworks and significant development effort, raising the bar for systematic validation of new algorithms. The key contributions are summarized as follows.

We take a totally new perspective of making an analogy between placement and deep learning, and build an opensource generic analytical placement framework that runs on both CPU and GPU platforms developed with modern deep learning toolkits.

A variety of gradient-descent solvers are provided, such as Nesterov's method, conjugate gradient method, and Adam [25], with the help from deep learning toolkit.

We propose efficient GPU implementations of key kernels in analytical placement like wirelength and density computation.

We demonstrate around 40× speedup in global placement without quality degradation of the entire placement flow over multi-threaded RePlAce implementations. More specifically, a design with one million cells finishes in one minute even with legalization. The framework maintains nearly linear scalability with industrial designs up to 10-million cells.

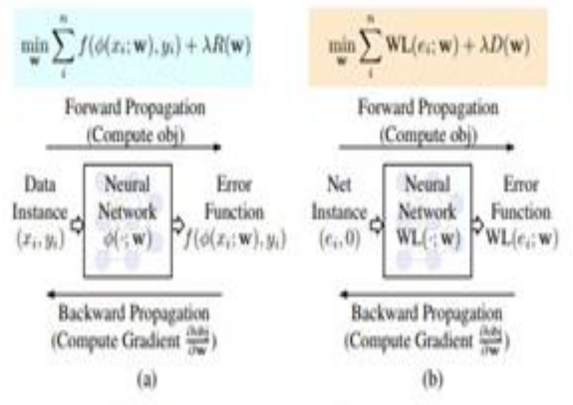
$\min_{\mathbf{w}} \sum_{i}^{n} f(\phi(x_i;\mathbf{w}), y_i) + \lambda R(\mathbf{w})$
Forward Propagation (Compute obj)
Data Instance (x_i, y_i)
Neural Network $\phi(\cdot;\mathbf{w})$
Error Function $f(\phi(x_i;\mathbf{w}), y_i)$
Backward Propagation (Compute Gradient $\frac{\partial obj}{\partial \mathbf{w}}$)
(a)
$\min_{\mathbf{w}} \sum_{i}^{n} \mathrm{WL}(e_i;\mathbf{w}) + \lambda D(\mathbf{w})$
Forward Propagation (Compute obj)
Net Instance $(e_i, 0)$
Neural Network $\mathrm{WL}(\cdot;\mathbf{w})$
Error Function $\mathrm{WL}(e_i;\mathbf{w})$
Backward Propagation (Compute Gradient $\frac{\partial obj}{\partial \mathbf{w}}$)
(b)

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

The source code is published on Github1. To clarify, translating the placement problem into a deep learning problem is aimed at solving placement using a toolkit. This is orthogonal to using deep learning models for deployment. The rest of the work is organized as follows.

Section II describes background and motivation.
Section III describes the detailed implementation.
Section IV presents the results.
Section V completes the work.

2. PRELIMINARIES

Analytical Placement

Analytical deployment typically consists of three steps: Global Housing (GP), Legalization (LG), and Detail Housing (DP). Global placement distributes the cells in the layout while minimizing the target cost. Legalization eliminates any remaining overlap between the two. Align the cell and place the cell in the placement position. Fine alignment performs incremental adjustments to further improve quality. Global deployment is usually the most time-consuming part of analytical deployment. The goal of global deployment is to minimize the cost of density-constrained cable lengths. The formulation can be written as

$$\min_{\mathbf{x},\mathbf{y}} \quad \sum_{e \in E} \mathrm{WL}(e;\mathbf{x},\mathbf{y}), \tag{1a}$$
$$\text{s.t.} \quad d(\mathbf{x},\mathbf{y}) \le d_t, \tag{1b}$$

Analogy to Deep Learning

Both analytical placement solving and neural network training inherently solve nonlinear optimization problems, so let's explore the fundamental similarities between the two problems. Compare the wire length cost to the error of misprediction and the density cost to the regularization term.Figure1 shows the objective functions for the two problems. In training a neural network, each data instance is input to the network with a feature vector xi and a label yi, and the neural network predicts a label φ(xi; w).

The task of training is to minimize the overall goal Beyond the weight w, the target consists of prediction errors.

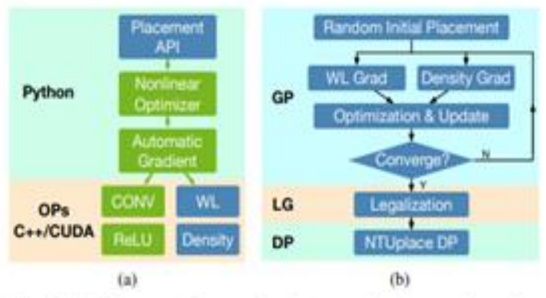

Fig. 2: (a) Software architecture for placement implementation using deep learning toolkits. (b) DREAMPlace flow.

Deep learning toolkits currently consist of three low-level stacks Operator (OP), automatic gradient derivation and optimization The engine shown in Figure 2a. TensorFlow or PyTorch provides mature and efficient implementations of these three. A stack with both CPU and GPU acceleration compatibility. The toolkit also provides convenient APIs for extending existing set. Each custom operator should have a well-defined definition Forward and backward functions for computing costs and gradients. To develop an analytics lab using the deep learning toolkit, All you need to implement is the wire length and wire length custom operator. Density cost for C++ and CUDA. Then you can build your placement A Python framework that requires very little development effort and is easy to build Integrate various optimization engines into your toolkit. The placement framework can run on both CPU and GPU platforms. A lot of effort was required to develop the conventional placement machine When building the entire software stack using C++. So the bar is from the design and validation of new placement algorithms is very high thanks to your development efforts. use deep learning Toolkits allow researchers to focus on developing critical tools Parts such as low-level operators and high-level optimization engines.

The ePlace/RePlAce Algorithm

ePlace/RePlAce is a state-of-the-art global placement algorithm family that models layouts and netlists as electrostatic systems [6]-[8]. The cable length cost originally proposed by [27], [28] uses the weighted average cable length (WA).

$$\mathrm{WA}_e = \frac{\sum_{i\in e} x_i e^{\frac{x_i}{\gamma}}}{\sum_{i\in e} e^{\frac{x_i}{\gamma}}} - \frac{\sum_{i\in e} x_i e^{-\frac{x_i}{\gamma}}}{\sum_{i\in e} e^{-\frac{x_i}{\gamma}}}, \qquad (3)$$

3. THE DREAMPLACE ALGORITHMS

We observe that starting from a random initial placement achieves the same quality (< 0.04% difference) with significantly less runtime (21.1% in Figure 3). In initial placement, standard cells are placed in the center of the layout with a small Gaussian noise. In our experiments, the scales of the noise are set to 0.1% of the width and height of the placement region. The kernel global placement iterations refer to the loop that involves the computation of wirelength and density gradient, optimization engines, and cell location updating. After the global placement converges, legalization is performed to remove remaining overlaps and align cells to placement sites. The last step before the output is detailed placement to refine the placement solutions relying on NTUplace3 [4]. The rest of this section will focus on GPU acceleration to the ePlace/RePlAce algorithm [6], [8].

a) Density Forward and Backward

Forward and backward of density cost is a computation-intensive procedure. Figure 4b plots the dependency graph for density cost forward and backward. The computation consists of four steps:

1) compute density map ρ;
2) compute au,v;
3) compute ψ in forward or ξ in backward;
4) compute D in forward or ∂D ∂xi in backward.

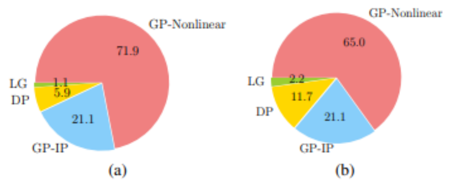

Fig. 3: RePlAce [8] runtime breakdown in percentages on bigblue4 (2 million cells). (a) 1 thread; (b) 10 threads.

We model this computation flow as a dynamic bipartite graph forward and backward process, as shown in Figure 5. First, density map calculation is modeled as a bipartite graph forward or a special 2D histogram problem where one cell may update multiple bins [31]. Then the electric potential and field are solved via DCT and other Fourier-related transforms. Finally, the electric force inflicted on each cell is collected from its overlapped bins, which can be modeled as a 2D gathering problem [31].

1)Dynamic Bipartite Graph Forward for Density Map: Each step of density map computation updates bins based on the overlapping area of corresponding cells.

Thus it can be modelled as a 2D histogram problem or a dynamic bipartite graph forward, as shown in Figure 5a. Each edge in the bipartite graph represents an update to the entry of the target bin in the density map, where the edge weight represents the overlapping area of the {cell, bin} pair. The reason why we call it "dynamic" is that, as cells move, edges in the bipartite graph, which indicate overlaps between cells and bins, will change accordingly. A naive algorithm to parallelize this step is to allocate one GPU thread for each cell and use atomic addition to accumulate the overlapping areas with bins [30]. However, as a cell may cover multiple bins, simply using one GPU thread to update all overlapped bins sequentially will cause load imbalance problem due to theng cells.

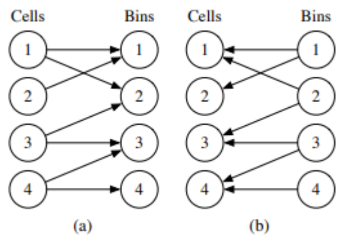

Fig. 5: Computation flow of (a) density map; (b) electric force.

Thus, it can be modeled as a particular variety in cell sizes. Empirically, the number of bins covered by a cell can vary from ∼ 10 to ∼ 1000. This ill-balanced workload within a thread warp introduces a big chunk of idle time and significantly degrades the performance. Therefore, we develop the following techniques to address this issue.

2) Dynamic Bipartite Graph Backward for Electric Force: In the electric force computation, each cell receives the forces from the bins it overlaps with. Thus, the computation can be viewed as a 2D gathering problem or a dynamic bipartite graph backward, as shown in Figure 5b. Each edge represents the force from a bin, and the edge weight is the amount of the force. The weight is computed as the product of the overlapping area between the cell and the bin and the electric field at the bin. A natural strategy to accelerate this step is to allocate one thread for each cell and accumulate the forces sequentially from its overlapping bins [30]. However, considering this computation task shares a similar structure with the density map computation, we borrow the same idea from Section III-B1 by sorting the cells and allocating multiple threads for each cell.

b) Density Weight Updating

We need to update the density weight λ in Equation (2) in each iteration to penalize the density cost. RePlAce [8] uses the following equations to update λ.

$$\mu \leftarrow \begin{cases} \mu_{max}, & \text{if } p < 0; \\ \max(\mu_{min}, \mu_{max}^{1-p}), & \text{otherwise;} \end{cases}, \tag{18a}$$

$$\lambda \leftarrow \lambda \cdot \mu, \tag{18b}$$

c) Optimization Engine

ePlace/RePlAce [6], [8] uses Nesterov's method as the gradientdescent solver with a Lipschitz-constant approximation scheme for line search. We implement the same approach in Python leveraging the efficient API provided by the deep learning toolkit. The framework is compatible with other well-known solvers in deep learning toolkits, i.e., various momentum-based gradient descent algorithms like Adam [13] and RMSProp, providing additional solver options.

d) Legalization

We also develop legalization as an operator in DREAMPlace. It first follows the Tetris-like procedure similar to NTUplace3 [4]. Then it performs Abacus row-based legalization [13]. This step copies the cell locations from GPU to CPU and executes legalization purely on CPU because we observe that it only takes several seconds even for million-size designs with a single CPU thread.

d) Extension to Consider Routability

To optimize routing congestion, we adopt cell inflation to optimize congested regions [14]. We follow a similar scheme to RePlAce [8], which invokes the NCTUgr global router [16] to get the routing overflow map during placement iterations. For each metal layer, we compute the ratio between routing demand and capacity at each routing tile. Then we use the maximum ratio across all layers to compute the inflation ratio for each tile.

4. EXPERIMENTAL RESULTS

The framework was developed in Python using PyTorch for the optimizer and APIs and C++/CUDA for the low-level operators. CPU parallelism was implemented using OpenMP for wire length and density operators. Both DREAMPlace and RePlAce [8] programs run on a Linux server with 40-core Intel E5-2698 v4 @2.20 GHz and one NVIDIA Tesla V100 GPU based on Volta architecture. The ISPD 2005 competition [13] and large-scale industrial design benchmarks were adopted. We ran experiments using both double-precision floating point (float64) and single-precision floating point (float32) on CPU and GPU. Uses the same container dimensions as RePlace.

Placement Acceleration

DREAM Place runs on the CPU and is 2x faster than RePlAce with 40 threads on GP. RePlAce [8] crashed on his 6th iteration of Nesterov's optimization on his 10 million cell industry benchmark.

A possible cause is the maximum memory usage. RePlace exceeded maximum memory (64 GB). Before the crash, it took 3396 seconds to take first place, with Nesterov averaging 7.5 seconds each iteration. This benchmark with DREAMPlace requires 1000 iterations, so the estimated execution time was 3396 + 1000 × 7.5 ≈ 10896 seconds. For all RePlAce runs, the initial placement takes 25-30% of the overall placement time, and the non-linear placement solution takes about 70-75%. DREAMPlace's LG is about 10x faster than NTUplace3 Legalizer in the RePlAce flow. NTUplace3 handles DP for both placers, so runtimes are similar. The overall placement flow speedup is 4.6x for GPU and 2.7x for CPU.

Acceleration of Low-Level Operators

We further investigate the efficiency of the low-level operators, e.g., wirelength forward and backward, DCT/IDCT, and density forward and backward. Figure 10 compares three approaches discussed in Section III-A. “Net-by-Net” denotes the net-level parallelization; “Atomic” denotes the pin-level parallelization with atomic operations in Algorithm 1 [30]; “Merged” denotes the combined forward and backward implementation in Algorithm 2. When using float32 on GPU, the merged approach achieves 3.7× speedup over the netby-net one and 1.8× speedup over the atomic one. On CPU, the atomic strategy is 20% slower than the net-by-net strategy with 40 threads, while the merged strategy is over 30% faster. Meanwhile, a promising speedup factor of 7.5× from a single thread to 40 threads can be achieved with the net-by-net strategy. Figure 11 compares the 2D DCT/IDCT implementation using 2Npoint FFT (“DCT-2N” and “IDCT-2N”), N-point FFT (“DCT-N” and “IDCT-N”), and N-point 2D FFT (“DCT-2D-N” and “IDCT2D-N”) [32]. Considering the map sizes in the experiment (from 512 × 512 to 4096 × 4096) with float32, the N-point DCT implementation is 2.1× faster [30] and the N-point 2D implementation can be 5.0× faster. For IDCT, the N-point implementation achieves 1.3× speedup and the 2D implementation achieves 4.1× speedup. This result demonstrates the efficiency of Algorithm 4. As DCT/IDCT is used in the density operator, in Figure 12, the efficiency of the entire density forward and backward procedure is compared for GPU and CPU implementations. With all the speedup techniques, an average of 1.5 ∼ 2.1× speedup on GPU can be achieved with the current implementation over the preliminary DAC version [30]. For the parallel CPU implementation, 3.1× runtime reduction can be achieved with 40 threads.

Routability-Driven Placement

To validate the run-time benefits of routability-driven placement, we performed an experiment using the DAC 2012 competition benchmark [41].

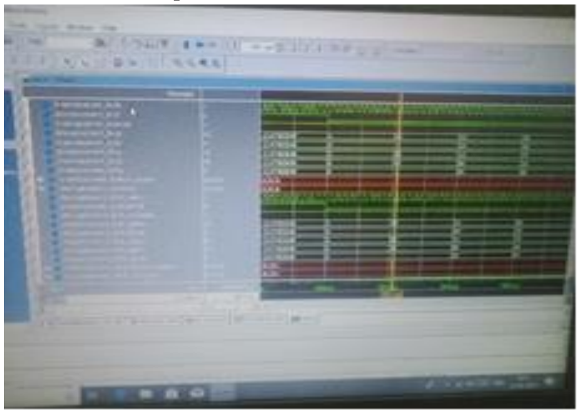

Figure 7.1 Output simulation for a,cb,cc,cd,ce

We Assume values for one module Ca=10, Cb=4, Cc=22, Cd=40, Ce=5., this gives the output results simulation waveform for FPGA module. For solution quality, consider two key metrics: 'sHPWL' is the scaled line length and 'RC' is the routing congestion.

In competition, RC is defined as the weighted average of the overruns in Figure 8 above. Average GPU runtime ratio for ISPD2005 and industry benchmarks with different number of CPU threads. DREAMPlace's TCAD version runtime normalized to V100

Using float64 matches the relationship between Tables II and III. The normalized ratio of 40 threads to GPU is annotated for easy comparison

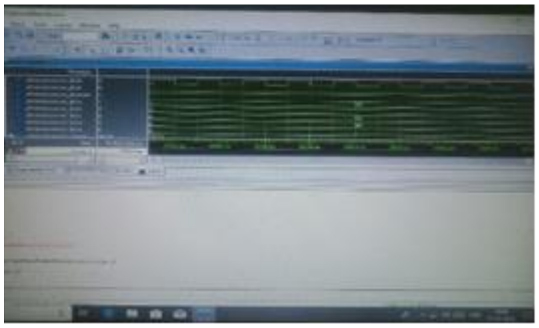
Figure 7.2 Output simulation for module.2 values.

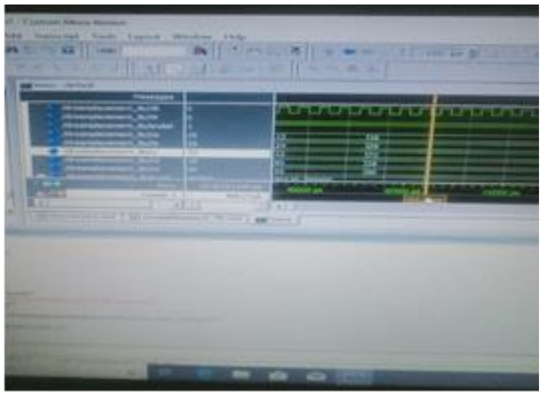
Figure 7.3 For module Ca=16,Cb=29,

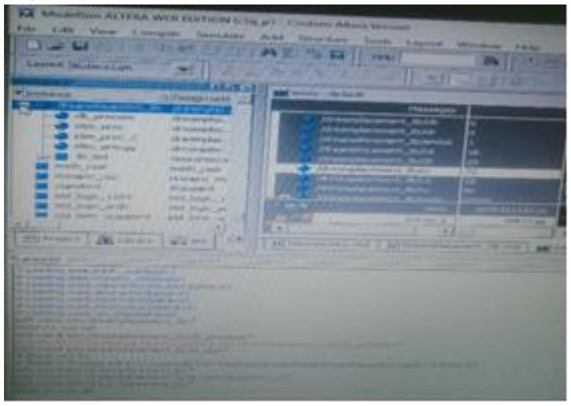
Figure 7.4 Layout process for Cc=72,Cd=18,Ce=50 dream placement

CONCLUSION

By transforming the solution to the traditional analytical placement problem into a neural network training problem, we use a fresh approach in this research. We create the new open-source placement engine DREAMPlace with GPU acceleration using the deep learning framework PyTorch. In comparison to the state-of-the-art RePlAce running on several threads, it provides a speedup of about 40% in global placement without compromising quality for academic and industrial benchmarks. To increase overall efficiency, we investigate various low-level operator implementations for forward and backward propagation. Additionally, DREAMPlace is very extendable, allowing for the simple scripting of high-level programming languages like Python to include additional algorithms/solvers and new objectives. We intend to look at GPU-accelerated detailed cell inflation for routability and net weighting for timing optimisation [29], [35], [37].

For even more performance, it can be expanded to make use of multi-GPU platforms. To ensure run-to-run determinism, we intend to look into the effectiveness of solutions that use fixed point numbers. We anticipate that this approach will open up new directions for addressing traditional EDA challenges because DREAMPlace decouples the high-level algorithmic design from low-level acceleration efforts.

REFERENCE

[1] A. B. Kahng, S. Reda, and Q. Wang, “Architecture and details of a high quality, large-scale analytical placer,” in IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2005, pp. 891–898.

[2] T. Chan, J. Cong, and K. Sze, “Multilevel generalized force-directed method for circuit placement,” in ACM International Symposium on Physical Design (ISPD). ACM, 2005, pp. 185–192.

[3] A. B. Kahng and Q. Wang, “A faster implementation of APlace,” in ACM International Symposium on Physical Design (ISPD). ACM, 2006, pp. 218–220.

[4] 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 (TCAD), vol. 27, no. 7, pp. 1228–1240, 2008.

[5] M.-K. Hsu, Y.-F. Chen, C.-C. Huang, S. Chou, T.-H. Lin, T.-C. Chen, and Y.-W. Chang, “NTUplace4h: A novel routability-driven placement algorithm for hierarchical mixed-size circuit designs,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 33, no. 12, pp. 1914–1927, 2014.

[6] 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.

[7] J. Lu, H. Zhuang, P. Chen, H. Chang, C. Chang, Y. Wong, L. Sha, D. Huang, Y. Luo, C. Teng, and C. Cheng, "ePlace-MS: Electrostatics based placement for mixed-size circuits," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 34, no. 5, pp. 685–698, 2015.

[8] C. 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 (TCAD), vol. 38, no. 9, pp. 1717–1730, 2019.

[9] Z. Zhu, J. Chen, Z. Peng, W. Zhu, and Y.-W. Chang, "Generalized augmented lagrangian and its applications to VLSI global placement," in ACM/IEEE Design Automation Conference (DAC). IEEE, 2018, pp. 1–6.

[10] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC). IEEE, 2007, pp. 135–140.

[11] X. He, T. Huang, L. Xiao, H. Tian, and E. F. Y. Young, "Ripple: A robust and effective routability-driven placer," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 32, no. 10, pp. 1546–1556, 2013.

[12] T. Lin, C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "PO-LAR: placement based on novel rough legalization and refinement," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2013, pp. 357–362.

[13] T. Manochandar and P. K. Diderot, "Classification of Alzheimer's Disease using Neuroimaging Techniques," 2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, India, 2023, pp. 1163-1168, doi: 10.1109/ICICCS56967.2023.10142373.

[14] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 31, no. 1, pp. 50–60, 2012.

[15] M.-C. Kim, N. Viswanathan, C. J. Alpert, I. L. Markov, and S. Ramji, "MAPLE: multilevel adaptive placement for mixed-size designs," in ACM International Symposium on Physical Design (ISPD). IEEE, 2012, pp. 193–200.

[16] T. Lin, C. Chu, and G. Wu, "POLAR 3.0: An ultrafast global placement engine," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2015, pp. 520–527.

[17] W. Li, Y. Lin, and D. Z. Pan, "elfPlace: Electrostatics-based placement for large-scale heterogeneous fpgas," in IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Westminster, CO: IEEE Press, November 2019