Differentiable-Timing-Driven Global Placement

Zizheng Guo[1,2], Yibo Lin[2*]
[1]School of Computer Science, Peking University [2]School of Integrated Circuits, Peking University
{gzz,yibolin}@pku.edu.cn

ABSTRACT

Placement is critical to the timing closure of the very-large-scale integrated (VLSI) circuit design flow. This paper proposes a differentiable-timing-driven global placement framework inspired by deep neural networks. By establishing the analogy between static timing analysis and neural network propagation, we propose a differentiable timing objective for placement to explicitly optimize timing metrics such as total negative slack (TNS) and worst negative slack (WNS). The framework can achieve at most 32.7% and 59.1% improvements on WNS and TNS respectively compared with the state-of-the-art timing-driven placer, and achieve 1.80× speed-up when both running on GPU.

1 INTRODUCTION

Placement is critical to timing closure in the VLSI design flow. It determines the physical locations of standard cells and thus has significant impacts on later stages like routing, post-routing optimization, and signoff timing verification. With the continuous growth of design complexity, achieving timing closure becomes increasingly challenging due to complicated timing models and expensive design iterations. As accurate timing information cannot be evaluated until post-routing stages, commercial design flows often need to run core placement and routing many times for timing closure, thus slowing down design iterations. Although timing-driven optimization in placement can reduce design iterations, turning on such optimization can slow down the algorithm by hundreds of times due to the large-scale numerical optimization in placement and the long feedback loop for timing evaluation. Therefore, high-performance and efficient timing-driven placement is always desired for design closure.

Placement can be divided into three steps: global placement (GP), legalization (LG), and detailed placement (DP). Global placement determines the rough locations of cells, legalization removes the overlaps between cells, and detailed placement refines the placement with local perturbation. Among these steps, global placement is crucial to the eventual placement quality, as it determines the global distribution of cells. Current state-of-the-art placers are based on analytical global placement algorithms [1–16]. They essentially formulate the core wirelength-driven placement into a nonlinear optimization problem with a wirelength term and a density penalty term in the objective. The wirelength term minimizes the length of nets and the density penalty minimizes the overlaps between cells. Various methods have been proposed to model the density penalty term, such as the NTUplace family [4, 5] and the ePlace family [6–8, 16], while most placers only focus on wirelength minimization without considering timing.

Timing-driven placement aims at optimizing timing metrics such as *total negative slack* (TNS) and *worst negative slack* (WNS). Typical timing-driven placement techniques can be categorized into net-based approaches and path-based approaches. **Net-based** approaches exploit timing analysis tools to obtain the timing information and update the weights of nets according to timing slack or critical paths [17–24]. These approaches do not directly optimize the timing metrics but try to minimize the delays of critical nets through net weighting and weighted wirelength minimization. Such an idea is naturally compatible with existing wirelength-driven placers, so they are widely adopted in global placement. **Path-based** approaches aim at minimizing the delay of critical paths by formulating mathematical programming problems [25–28]. They have precise control over the path delay and hence can achieve high-quality solutions, but they are not scalable with the number of paths. Thus, path-based approaches are usually adopted in detailed placement for local perturbation.

Despite the previous efforts, timing optimization in placement still follows indirect approaches to improve timing metrics, i.e., minimizing delays of specific nets or paths. With the increase of design complexity, such approaches are reaching their limitations for timing optimization. In this work, we propose a differentiable-timing-driven global placement framework inspired by deep neural networks. By establishing the analogy between static timing analysis and deep neural networks, we propose a differentiable timing objective for explicit optimization of timing metrics. The key contributions are summarized as follows.

- We propose a new paradigm for timing-driven placement based on a differentiable timing engine inspired by deep neural networks, which can directly optimize global timing metrics such as TNS and WNS.
- We design a holistic framework for differentiable delay and signal arrival time propagation that is highly extensible to various timing models.
- We develop high-performance GPU-accelerated kernels for the differentiable timing engine to boost the efficiency.
- Compared with the state-of-the-art timing-driven placer [24] based on net weighting, we can achieve at most 32.7% and 59.1% improvements on WNS and TNS, respectively, as well as 1.80× speed-up when both placers run on GPU.

We believe this work shall open up new directions for timing optimization in the VLSI design flow. The rest of the paper is organized as follows. Section 2 introduces the background and motivation; Section 3 explains the detailed algorithms; Section 4 demonstrates the results; Section 5 concludes this paper.

2 PRELIMINARIES

In this section, we introduce basic concepts of STA, nonlinear placement, as well as prior work on timing-driven global placement.

2.1 Static Timing Analysis

Static timing analysis (STA) is the central step for analyzing the circuit timing performance [29]. As shown in Figure 1, STA models the circuit as a directed acyclic graph (DAG) with net arcs and cell arcs indicating the signal propagating directions. Arcs introduce delay and slew when signals propagate through them. The delays are accumulated to the earliest and latest signal arrival times on pins, by performing `min` and `max` operations on the arrival times of fan-in signals. The arrival times are used to model the worst-case timing scenarios and perform setup and hold checks.

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530486

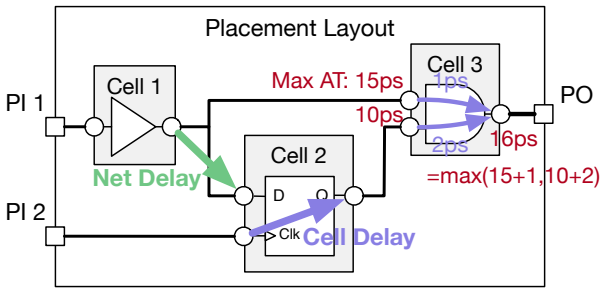

Figure 1: An example placement stage with STA.

Machine Learning	Placement
Train a neural network	Solve global placement
Neural network weights	Cell locations
Dataset	Net instances
Loss function	Wirelength objective
Regularization	Density constraint

Table 1: The analogy between ML training and placement [16].

The worst-case circuit performance is quantified by setup slack and hold slack, which are computed from the arrival times and the required arrival times at timing endpoints.

$$\begin{aligned} slack_{\text{setup}}(p) &= rat_{\text{early}}(p) - at_{\text{late}}(p), \\ slack_{\text{hold}}(p) &= at_{\text{early}}(p) - rat_{\text{late}}(p). \end{aligned} \tag{1}$$

In Equation (1), $slack_{\text{setup}}(p)$ and $slack_{\text{hold}}(p)$ are respectively the setup and hold slack values at timing endpoint p. Generally, a positive slack means that the signal meets the timing constraints, and a negative slack indicates a timing violation.

In timing-driven optimization tasks, our goal is to minimize the absolute value of worst negative slack (WNS), as well as the total negative slack (TNS) among all timing endpoints, as defined in Equation (2).

$$\begin{aligned} WNS_{\text{setup/hold}} &= \min_{\text{endpoint } p} slack_{\text{setup/hold}}(p), \\ TNS_{\text{setup/hold}} &= \sum_{\text{endpoint } p} \min\left(0, slack_{\text{setup/hold}}(p)\right). \end{aligned} \tag{2}$$

2.2 Wirelength-driven Nonlinear Placement

Global placement determines the locations of cells in the placement layout, as illustrated in Figure 1. The cell locations determine the pin locations, which affect the wirelength and delay of the nets. In wirelength-driven global placement, the goal is to minimize the total net wirelength, under the cell density constraint. The density constraint is usually transformed into a density penalty term in the objective function,

$$\min_{\mathbf{x},\mathbf{y}} \sum_{\text{net } e} WL(e;\mathbf{x},\mathbf{y}) + \lambda D(\mathbf{x},\mathbf{y}), \tag{3}$$

where $\mathbf{x}, \mathbf{y}$ are cell locations and λ is the density penalty weight. In analytical placement, this optimization problem is solved by gradient descent on the objective function. The placement engine controls λ by gradually increasing it to encourage cell spreading.

2.3 Timing-driven Placement by Net Weighting

To achieve circuit correctness and performance, it is critical to include timing metrics into the placement objective. The problem formulation for timing-driven global placement is presented below.

Problem (Timing-driven Global Placement)**.** Given a set of cells and a placement layout, determine the cell locations $\mathbf{x}, \mathbf{y}$ to minimize the absolute value of total negative slack and worst negative slack, i.e., $TNS(\mathbf{x},\mathbf{y})$ and $WNS(\mathbf{x},\mathbf{y})$.

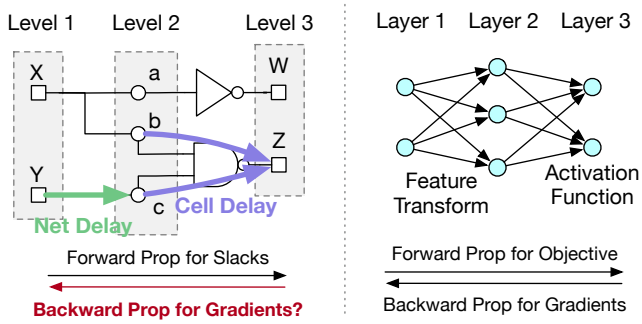

Figure 2: The analogy between an STA engine and a deep neural network marks the potential for differentiable timing.

However, timing is usually handled by an external STA engine due to its complexity compared to simple objectives like wirelength. STA is complex because it evaluates the *global features* of the circuit, instead of concentrating on *local features* such as the wirelength of a single net. In STA, we need to have a whole view of the circuit to find the nets that lie on timing-critical paths, and the length of such paths may exceed 300 [30].

To bridge the gap between the global view of an STA engine and the local view of wirelength objectives, prior works have used a technique called *net weighting*. The idea is to repeatedly invoke an STA engine on the current placement solution, and analyze the timing report to find timing-critical nets. Then, the relative weights of these nets in the wirelength objective are increased by a small amount. The objective is to minimize the weighted sum of net wirelength, as shown below.

$$\min_{\mathbf{x},\mathbf{y}} \sum_{\text{net } e} w_e \cdot WL(e;\mathbf{x},\mathbf{y}) + \lambda D(\mathbf{x},\mathbf{y}). \tag{4}$$

The guidance from an STA engine is then incorporated in the frequent update of net weights. However, net weighting is still an inefficient and artificial approximation to the WNS and TNS metrics.

3 ALGORITHMS

In this section, we propose our timing-driven placement algorithm based on a differentiable STA engine. Section 3.1 presents an inspiration from deep neural networks. Section 3.2 gives the timing objective function with smoothing. Section 3.3 presents an overview of our differentiable timer. Section 3.4 and Section 3.5 present our differentiable wire delay model and delay propagation model. Finally, Section 3.6 presents the overall timing-driven placement flow with GPU acceleration.

3.1 Differentiable Timing Inspired by Deep Neural Networks

In light of the theory and practice development of machine learning (ML), prior work has raised the analogy between the global placement problem in Equation (3) and the training of a machine learning model [16]. The detailed analogy is listed in Table 1. In this context, the wirelength objective can be regarded as a *single-layer* neural network with a receptive field of only 1 hop, as it is only related to the distance between pins in every single net. On the other hand, STA can be regarded as a neural network as deep as the length of the longest path in the circuit, e.g., > 300 layers [30]. As a result, net-weighting-based timing optimization essentially tries to approximate a deep neural network with a shallow one.

In reality, we note that deep neural networks can be efficiently trained with backpropagation (BP) [31]. Backpropagation accumulates and propagates partial derivatives with respect to intermediate variables in the reverse topological order of computation. The result is the gradients to the loss function with respect to all model weights. Inspired by this, we intend to perform a similar backpropagation in STA to obtain the

gradients of slacks with respect to all cell locations, as shown in Figure 2. We call this a *differentiable STA engine*.

In this work, we design and implement such a differentiable timing engine that directly computes the $TNS(\mathbf{x}, \mathbf{y})$ and $WNS(\mathbf{x}, \mathbf{y})$, and their gradients with respect to $\mathbf{x}, \mathbf{y}$. Compared to prior works based on net weighting, our differentiable model enables direct optimization of global timing metrics in placement.

3.2 The Smoothed Objective Function

One key challenge in differentiable STA compared to neural networks is that STA functions are highly non-smooth. This is because the arrival time updates in STA include `max` and `min` operations on the fan-in arrival times. Regardless of the number of fan-in nodes, only 1 of the fan-in node receives a non-zero gradient value. As a result, a direct gradient descent on WNS only updates the single most critical path. Such a pattern is known to introduce oscillation and instability to the optimization process.

To solve this issue, we replace the `max` and `min` operations in STA with their smoothed approximations. Similar smoothing techniques have been widely used in placers to approximate wirelength objectives such as HPWL [4]. Specifically, we replace `max` with Log-Sum-Exp (LSE) smoothing stated as follows,

$$LSE_\gamma(x_1, x_2, ..., x_n) = \gamma \log \left(\sum_{i=1}^{n} \exp \frac{x_i}{\gamma} \right). \tag{5}$$

We transform `min` to the `max` of the inverse value of operands. In Equation (5), γ controls the degree of smoothing. A larger γ makes the result more smooth, at the cost of lower accuracy. We denote the smoothed TNS and WNS objective functions as $TNS_\gamma(\mathbf{x}, \mathbf{y})$ and $WNS_\gamma(\mathbf{x}, \mathbf{y})$, respectively. The smoothed objective function is thus,

$$\min_{\mathbf{x},\mathbf{y}} \sum_{\text{net } e} WL(e; \mathbf{x}, \mathbf{y}) + \lambda D(\mathbf{x}, \mathbf{y}) + t_1 TNS_\gamma(\mathbf{x}, \mathbf{y}) + t_2 WNS_\gamma(\mathbf{x}, \mathbf{y}), \tag{6}$$

where t_1 and t_2 are weights of the TNS and WNS objectives.

3.3 The Overview of Our Differentiable Timer

Figure 3 shows an overview of our differentiable STA engine consisting of the following stages:

(1) We first initialize the pin levels by a topological sorting. This needs to be done only once as logical levels of pins do not change along with the movement of pin locations.
(2) Given a set of pin locations, we compute rectilinear Steiner minimal trees (RSMTs) of nets and apply the Elmore delay model to get net delay, impulse values (i.e. slew components), and net capacitive loads. We split these outputs according to pin levels.
(3) Given the Elmore delay information at each level, we propagate the arrival times and slews level by level.
(4) Given the arrival times (ATs), we compute the required arrival times (RATs) and WNS/TNS based on timing constraints. The slacks are computed by subtracting ATs and RATs.
(5) Finally, we compute the gradient with respect to pin locations by going backward according to the blue edges shown in Figure 3. Specifically, we first compute the gradient of WNS and TNS to pin arrival times. Then, we compute the gradient to the Elmore delay information by back-propagating through pin levels. Finally, we compute the gradient to pin locations by taking the derivative of the Elmore delay model.

3.4 Differentiable Wire Delay Model

3.4.1 Rectilinear Steiner Tree Generation. To compute wire delay with the given pin locations, we should first compute an RSMT for each net as a rough routing result. This is usually accomplished using FLUTE [32] in timing-driven placement works [24, 33, 34]. We note that FLUTE can

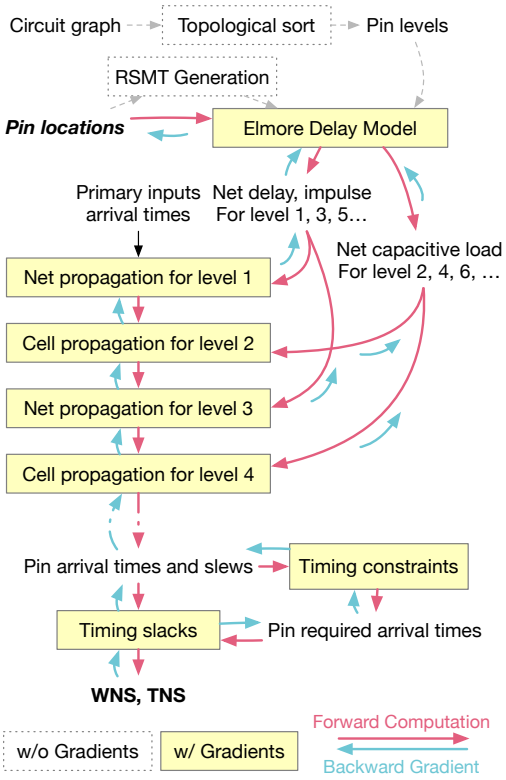

Figure 3: The overall flow of our differentiable timer from pin locations to WNS and TNS, and back to the gradients of pin locations.

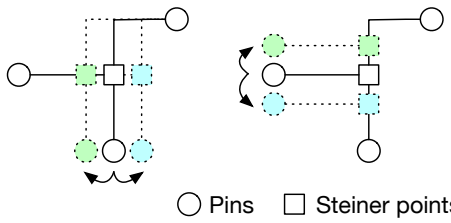

Figure 4: Perturbations on pin position cause Steiner points to move along with its "branches" on the tree.

be replaced by other RSMT generation algorithms in our framework. The problem with these algorithms, including FLUTE, is that they are not differentiable. Specifically, our delay model computes gradients of all points on the Steiner tree, including both pins and router-inserted Steiner points. We have to deal with gradients on Steiner points to consider the routing effect in the backpropagation process.

To solve the problem, we analyze the generation of the Steiner tree under small perturbations to pin locations. As illustrated in Figure 4, we note that the Steiner points move along with the movement of horizontal and vertical tree "branches". As long as the pin displacement is small enough, the positions of Steiner points are determined by pins related to their branches. As a result, we apply the gradients on Steiner points to pins on their branches to ensure their gradients are correctly computed.

3.4.2 Differentiable Elmore Delay Model. Given the net routing tree, we compute the net delay, impulse, and capacitive load using the Elmore delay model. The model computation can be described by the following recursive equations [30],

$$Load(u) = Cap(u) + \sum_{\text{child } v} Load(v), \tag{7a}$$

$$Delay(u) = Delay(fa(u)) + Res(fa(u) \to u) \cdot Load(u), \tag{7b}$$

$$LDelay(u) = Cap(u) \cdot Delay(u) + \sum_{\text{child } v} LDelay(v), \tag{7c}$$

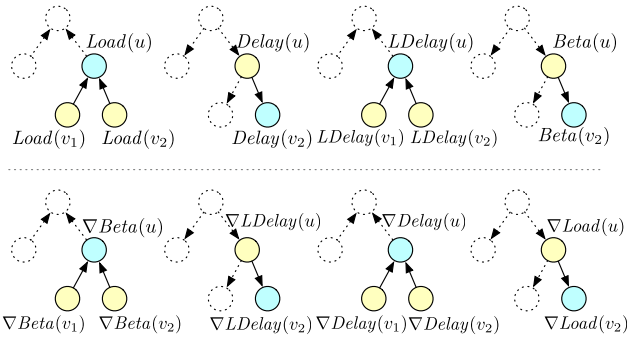

Figure 5: The Elmore dynamic programming passes. The upper 4 passes compute Elmore delay, and the lower 4 passes compute the gradients.

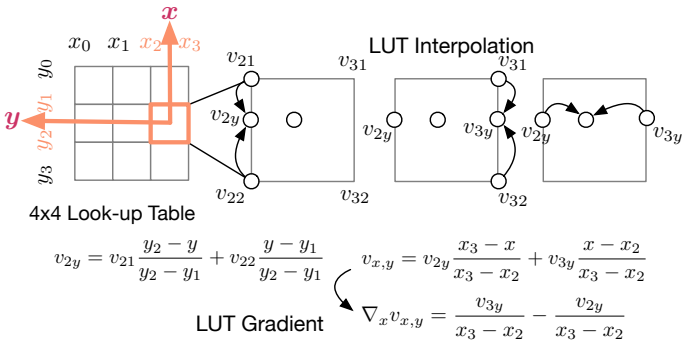

Figure 6: LUT interpolation and gradient computation. A 2D interpolation can be computed with three 1D interpolations.

$$Beta(u) = Beta(fa(u)) + Res(fa(u) \to u) \cdot LDelay(u), \tag{7d}$$

$$Impulse(u) = \sqrt{2 \cdot Beta(u) - Delay^2(u)}, \tag{7e}$$

where $fa(u)$ denotes the parent node of u, and $LDelay(u)$, $Beta(u)$ are intermediate values in slew/impulse computation.

Using basic calculus and the chain rule of partial derivatives, we can write the gradient form of Equation (7) as follows, with ∇F denoting the gradient of the objective function f with respective to F, i.e., $\frac{\partial f}{\partial F}$.

$$\nabla Beta(u) = 2 \cdot \nabla Impulse^2(u) + \sum_{\text{child } v} \nabla Beta(v) \tag{8a}$$

$$\nabla LDelay(u) = Res(fa(u) \to u) \cdot \nabla Beta(u) + \nabla LDelay(fa(u)) \tag{8b}$$

$$\nabla Delay(u) = Cap(u) \cdot \nabla LDelay(u) + \sum_{\text{child } v} \nabla Delay(v) - 2 \cdot Delay(u) \cdot \nabla Impulse^2(u) \tag{8c}$$

$$\nabla Load(u) = Res(fa(u) \to u) \cdot \nabla Delay(u) + \nabla Load(fa(u)) \tag{8d}$$

$$\nabla Cap(u) = \nabla Load(u) + Delay(u) \cdot \nabla LDelay(u) \tag{8e}$$

$$\nabla Res(fa(u) \to u) = Load(u) \cdot \nabla Delay(u) + LDelay(u) \cdot \nabla Beta(u) \tag{8f}$$

By computing according to Equation (8)[1] given the gradients of Elmore delay, impulse, and load, we obtain the gradient to the pin capacitance and edge resistance, which are directly computed from distances between two pins. We then obtain the gradients with respect to pin locations.

In an STA engine, the Elmore delay Equation (7) is implemented by 4 passes of dynamic programming on trees [35], alternating between bottom-up and top-down updates. As shown in Figure 5, we implement Equation (8) by another 4 passes of dynamic programming on trees, in the reverse order of the forward passes.

While we give the derivation of the Elmore delay model as an example, we note that our algorithm is generalizable to other more complex interconnect delay models, such as Π model and coupling effects as long as the model can be written in analytical form like Equation (7).

3.5 Differentiable Delay Propagation

Delay propagation computes the arrival time and slew of pins level by level, as shown in Figure 3. Corresponding to net arcs and cell arcs, there are two types of propagation, for net delay and cell delay respectively. We apply the two propagation types alternately to push forward the arrival time computation. To compute the gradient of propagation, we propagate backward on the pin levels.

3.5.1 Net Delay Propagation. In this step, we compute the arrival time and slew of net sinks given their Elmore delay and impulse. For other wire delay models, we have different specific forms of net delay propagation, but the basic idea remains the same. We denote the net driver as u and a net sink as v, and the net delay propagation rule can be written as follows [30],

$$AT(v) = AT(u) + Delay(v), \tag{9a}$$

$$Slew(v) = \sqrt{Slew^2(u) + Impulse^2(v)}. \tag{9b}$$

We similarly write down the gradients of Equation (9) as follows,

$$\frac{\partial f}{\partial AT(v)} \frac{\partial AT(v)}{\partial AT(u)} = \nabla AT(v) \tag{10a}$$

$$\nabla Delay(v) = \nabla AT(v) \tag{10b}$$

$$\frac{\partial f}{\partial Slew(v)} \frac{\partial Slew(v)}{\partial Slew(u)} = \frac{Slew(u)}{Slew(v)} \cdot \nabla Slew(v) \tag{10c}$$

$$\nabla Impulse^2(v) = \frac{1}{2 \cdot Slew(v)} \cdot \nabla Slew(v) \tag{10d}$$

As there is at most 1 fan-in net arc for each pin, we do not need `max` or `min` operations to aggregate arrival time corners. Instead, we simply follow Equation (9) to compute arrival time and slew, and Equation (10) to compute the gradient components for arrival time and slew from previous levels, as well as Elmore delay and impulse gradients for the current level.

3.5.2 Cell Delay Propagation. The delay and timing constraint of logic cells are characterized by a cell library (`.lib` file) included in the process design kit (PDK). Different from wire delay models, cell delay and slew do not take analytical forms. Instead, they are modeled by point values in the form of a matrix, which we call look-up tables (LUTs). A LUT is an $N \times N$ matrix $v_{ij}, i = 0 \dots N-1, j = 0 \dots N-1$ with $2N$ associated values $x_0 \dots x_{N-1}, y_0 \dots y_{N-1}$, as shown on the left side of Figure 6. A query to the LUT is a 2D point (x, y) and the result is a linear interpolation or extrapolation in a 2×2 submatrix in the LUT where the query point lies.

The widely-used non-linear delay model (NLDM) defines cell delay, slew, and constraints directly using LUTs. We note that other cell delay models such as the current source model also use LUTs to model DC current and other parameters. In NLDM, there are 4 LUTs for each timing arc, named `cell_rise`, `cell_fall`, `rise_transition`, and `fall_transition`. The first two LUTs characterize the cell delay, while the latter two LUTs are for the cell slew. Depending on the function of the logic cell, the timing arcs might be positive unate or negative unate, so one of the two LUTs would be used in certain signal edges. To simplify the discussion, we omit the `rise` and `fall` settings in the following equations.

We denote the cell output pin as v, its related cell input pins as u, and the LSE smoothing hyperparameter as γ. The NLDM delay, slew, and arrival times are defined as follows,

$$Delay_u(v) = LUT_{\text{cell } u \to v}(Slew(u), Load(v)), \tag{11a}$$

$$Slew_u(v) = LUT_{\text{transition } u \to v}(Slew(u), Load(v)), \tag{11b}$$

$$AT(v) = LSE_\gamma^{\text{all inputs } u}\{AT(u) + Delay_u(v)\}, \tag{11c}$$

$$Slew(v) = LSE_\gamma^{\text{all inputs } u}\{Slew_u(v)\}. \tag{11d}$$

[1] In equation (8c)-(8f), We have fixed several mistakes in the proceeding version.

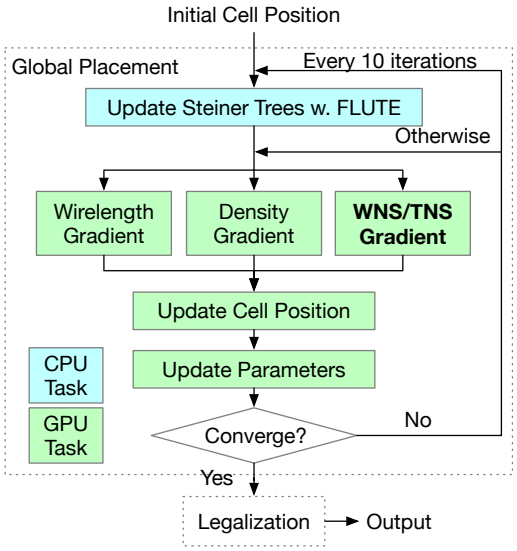

Figure 7: Our GPU-accelerated timing-driven placement flow.

We write down the gradients of Equation (11) as follows,

$$\nabla AT(u) = \nabla_{\text{input } u} LSE_\gamma \{AT(u) + Delay_u(v)\} \cdot \nabla AT(v), \tag{12a}$$

$$\nabla Delay_u(v) = \nabla_{\text{input } u} LSE_\gamma \{AT(u) + Delay_u(v)\} \cdot \nabla AT(v), \tag{12b}$$

$$\nabla Slew_u(v) = \nabla_{\text{input } u} LSE_\gamma \{Slew_u(v)\} \cdot \nabla Slew(v), \tag{12c}$$

$$\begin{aligned} \nabla Slew(u) = {} & \nabla_x LUT_{\text{cell } u\to v}(Slew(u), Load(v)) \cdot \nabla Delay_u(v) + \\ & \nabla_x LUT_{\text{transition } u\to v}(Slew(u), Load(v)) \cdot \nabla Slew_u(v) \end{aligned} \tag{12d}$$

$$\begin{aligned} \nabla Load(v) = {} & \nabla_y LUT_{\text{cell } u\to v}(Slew(u), Load(v)) \cdot \nabla Delay_u(v) + \\ & \nabla_y LUT_{\text{transition } u\to v}(Slew(u), Load(v)) \cdot \nabla Slew_u(v) \end{aligned} \tag{12e}$$

Computing Equation (12) requires the gradient of a LUT query. We compute the LUT gradient by linear interpolation similar to a forward LUT query, as shown in Figure 6. We first find the corresponding LUT cell to the query (x, y). Then, we do two 1D linear interpolations on one dimension. Finally, the gradient of 2D interpolation can be transformed into the gradient of a final 1D interpolation.

3.6 GPU-Accelerated Placement Framework

Global placement is a very time-consuming design step. It consists of hundreds to thousands of iterations that take hours on CPU for a large design. To ensure reasonable placement runtime, there have been GPU-accelerated placement engines with efficient GPU kernels for wirelength and density gradient computation. However, in a timing-driven placement flow, the runtime is dominated by repeated calls to the STA engine [24]. To this end, we propose GPU-accelerated kernels for timing analysis and gradient computation. Our differentiable STA engine is fully GPU-accelerated for both forward computation and backward gradients.

Figure 7 shows our placement flow. The only CPU kernel is the Steiner tree computation with FLUTE, which we accelerate by multi-threading on CPUs. We call FLUTE to update the Steiner tree periodically every 10 iterations and store the Steiner points for use in the next 9 iterations. In these 9 iterations, coordinates of Steiner points are updated according to their related pins (Figure 4), instead of recomputed from scratch. In this way, we maximize the performance of our placement engine with a small loss in the accuracy of gradients.

4 EXPERIMENTAL RESULTS

We implement our differentiable timing engine as C++ and CUDA kernels on top of the open-source placer DREAMPlace [16]. We evaluate the performance of our timing-driven placer on large industrial designs from the ICCAD 2015 contest [33]. The benchmark statistics are listed in Table 2. Our test environment is a 64-bit Linux machine with 24-cores Intel Xeon CPU at 2.20 GHz, 1 NVIDIA Titan RTX GPU, and 256GB RAM. We set t_1, t_2, γ hyperparameters to tune the benchmark performance, where γ is set to around 100, t_1 is set to around 0.01, and t_2 is set to around 0.0001. We start our timing optimization from around the 100th iteration where cells have been initially spread out. We then increase t_1 and t_2 by 1% after each iteration. We compare our work with the original DREAMPlace [16], and the state-of-the-art timing-driven placer [24] based on net weighting.

Table 2: ICCAD 2015 contest benchmark statistics.

Benchmark	#Cells	#Nets	#Pins
superblue1	1209716	1215710	3767494
superblue3	1213253	1224979	3905321
superblue4	795645	802513	2497940
superblue5	1086888	1100825	3246878
superblue7	1931639	1933945	6372094
superblue10	1876103	1898119	5560506
superblue16	981559	999902	3013268
superblue18	768068	771542	2559143

Table 3 shows the overall comparison on WNS, TNS, half-perimeter wirelength (HPWL), and runtime. Our differentiable-timing-driven placer outperforms the state-of-the-art timing-driven placer [24] by a huge amount, setting new records on these benchmarks. The most notable result on the benchmark superblue18 improves WNS and TNS by 32.7% and 59.1%, respectively. On most other benchmarks, 10–30% WNS improvement and 20–40% TNS improvement have been obtained. On average, the WNS and TNS results of [24] are 28.2% and 47.2% worse compared to us.

Meanwhile, our WNS results are consistently better in all benchmarks. Optimizing WNS is much more important than TNS, as WNS represents the global timing bottleneck in the design which is directly related to circuit correctness and performance. However, prior works like [24] have stated the extreme difficulty in optimizing WNS in the global placement stage. In this work, we have overcome that difficulty and brought timing-driven global placement to a new milestone. We ascribe this to our differentiable timing engine that directly computes the gradient of global timing metrics. Note that our timing-driven placer does not degrade wirelength compared to the original DREAMPlace without timing optimizations. As shown in Table 3, we obtain almost identical HPWL results given the same stop criterion on density overflow. In other words, the improvements on WNS and TNS are obtained almost "for free".

Our timing-driven placer also has decent efficiency. We achieve 1.80× speed-up on average compared to the state-of-the-art net weighting placer [24], which also runs on GPU. By adding timing objectives, we use 3.14× the runtime of DREAMPlace compared to 5–6× in [24]. We ascribe this efficiency to our fully GPU-accelerated timing engine with both forward and backward CUDA kernels, and our strategy to reuse Steiner tree results to reduce the number of FLUTE calls.

Figure 8 draws the HPWL, density overflow, WNS, and TNS along with a placement run. The added timing objectives do not influence the HPWL and density optimization and the two curves overlap. However, the timing objectives have improved WNS and TNS in later iterations. This shows the effectiveness of our timing objective gradients in the optimization flow.

5 CONCLUSION

In this paper, we propose a new paradigm for timing-driven global placement based on a differentiable STA engine inspired by deep neural networks. By directly analyzing and optimizing global timing metrics, we improve TNS by 32.7% and WNS by 59.1% at most compared with state-of-the-art timing-driven placers based on net weighting. We also improved the overall efficiency of timing-driven placement by implementing efficient GPU kernels for our differentiable timer. Our work is

Table 3: Comparing WNS, TNS, HPWL and runtime with the original DREAMPlace [16] and the state-of-the-art net-weighting-based timing driven placer [24]. The best results on WNS, TNS and runtime are bold-faced, and the second-best ones are colored brown.

Benchmark	DREAMPlace [16]				Net Weighting [24]				Ours			
	WNS	TNS	HPWL	Runtime	WNS	TNS	HPWL*	Runtime†	WNS	TNS	HPWL	Runtime
superblue1	-18.866	-262.441	422.0	**79.48**	-14.103	-85.032	443.1	471.77	**-10.770**	**-74.854**	423.8	268.31
superblue3	-27.648	-76.644	478.2	**72.96**	-16.434	-54.742	482.4	451.22	**-12.374**	**-39.430**	478.4	266.65
superblue4	-22.041	-290.881	312.0	**52.21**	-12.781	-144.380	335.9	283.64	**-8.492**	**-82.924**	312.2	156.36
superblue5	-48.918	-157.816	488.3	**116.69**	-26.760	**-95.782**	556.2	772.75	**-25.212**	-108.076	488.7	259.26
superblue7	-19.751	-141.548	604.3	**125.57**	**-15.216**	-63.863	604.0	774.32	**-15.216**	**-46.426**	602.1	450.85
superblue10	-26.099	-731.941	935.9	**205.92**	-31.880	-768.748	1036.7	859.28	**-21.974**	**-558.054**	934.4	465.24
superblue16	-17.711	-453.566	435.8	**63.59**	-12.112	-124.181	448.1	335.10	**-10.854**	**-87.026**	485.1	217.65
superblue18	-20.288	-96.756	243.0	**27.55**	-11.871	-47.246	253.6	174.07	**-7.987**	**-19.314**	243.6	156.99
Avg. Ratio	1.897	3.125	0.987	**0.318**	1.282	1.472	1.043	1.807	**1.000**	**1.000**	1.000	1.000

WNS: in ($\times 10^3$)ps. **TNS**: in ($\times 10^5$)ps. **HPWL**: half-perimeter wirelength ($\times 10^6$). *We acquired the DEF result from authors of [24] to evaluate. † The runtime of [24] is scaled to reflect machine difference: compensated runtime = reported runtime in [24] $\times \frac{\text{Our DREAMPlace runtime}}{\text{DREAMPlace runtime in [24]}}$.

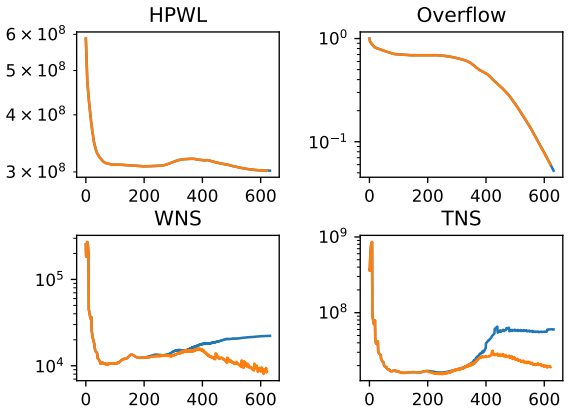

Figure 8: Optimization iterations for benchmark `superblue4`. The blue curve is DREAMPlace, and the orange one is our work.

a general differentiable timing analysis framework that is widely applicable to different STA models. Our future work shall focus on designing dynamic updating strategies for timing weights and preconditioning for timing gradients to achieve even better performance and efficiency.

ACKNOWLEDGE

This work was supported in part by the National Key Research and Development Program of China (No. 2019YFB2205001), the National Science Foundation of China (Grant No. 62034007 and No. 62141404), and the 111 Project (B18001).

REFERENCES

[1] A. B. Kahng, S. Reda, and Q. Wang, "Architecture and details of a high quality, large-scale analytical placer," in *Proc. ICCAD*. IEEE, 2005, pp. 891–898.
[2] T. Chan, J. Cong, and K. Sze, "Multilevel generalized force-directed method for circuit placement," in *Proc. ISPD*. ACM, 2005, pp. 185–192.
[3] A. B. Kahng and Q. Wang, "A faster implementation of APlace," in *Proc. 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 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 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 TODAES*, vol. 20, no. 2, p. 17, 2015.
[7] J. Lu, H. Zhuang, P. Chen, H. Chang, C.-C. Chang, Y.-C. Wong, L. Sha, D. Huang, Y. Luo, C.-C. Teng *et al.*, "ePlace-MS: Electrostatics-based placement for mixed-size circuits," *IEEE TCAD*, vol. 34, no. 5, pp. 685–698, 2015.
[8] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE TCAD*, 2018.
[9] Z. Zhu, J. Chen, Z. Peng, W. Zhu, and Y.-W. Chang, "Generalized augmented lagrangian and its applications to vlsi global placement," in *Proc. 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 *Proc. ASPDAC*, 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 TCAD*, vol. 32, no. 10, pp. 1546–1556, 2013.
[12] T. Lin, C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: placement based on novel rough legalization and refinement," in *Proc. ICCAD*, 2013, pp. 357–362.
[13] ——, "POLAR: A high performance mixed-size wirelengh-driven placer with density constraints," *IEEE TCAD*, vol. 34, no. 3, pp. 447–459, 2015.
[14] 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.
[15] M.-C. Kim, N. Viswanathan, C. J. Alpert, I. L. Markov, and S. Ramji, "MAPLE: multilevel adaptive placement for mixed-size designs," in *Proc. ISPD*, 2012, pp. 193–200.
[16] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," *IEEE TCAD*, 2020.
[17] M. Burstein and M. N. Youssef, "Timing influenced layout design," in *Proc. DAC*. IEEE, 1985, pp. 124–130.
[18] A. E. Dunlop, V. D. Agrawal, D. N. Deutsch, M. Jukl, P. Kozak, and M. Wiesel, "Chip layout optimization using critical path weighting," in *Proc. DAC*. IEEE, 1984, pp. 133–136.
[19] H. Chang, E. Shragowitz, J. Liu, H. Youssef, B. Lu, and S. Sutanthavibul, "Net criticality revisited: An effective method to improve timing in physical design," in *ispd*, 2002, pp. 155–160.
[20] T. Kong, "A novel net weighting algorithm for timing-driven placement," in *Proc. ICCAD*, 2002, pp. 172–176.
[21] B. Halpin, C. R. Chen, and N. Sehgal, "A sensitivity based placer for standard cells," in *Proceedings of the 10th Great Lakes symposium on VLSI*, 2000, pp. 193–196.
[22] T.-Y. Wang, J.-L. Tsai, and C. C.-P. Chen, "Sensitivity guided net weighting for placement driven synthesis," in *Proc. ISPD*, 2004, pp. 124–131.
[23] Z. Xiu and R. A. Rutenbar, "Timing-driven placement by grid-warping," in *Proc. DAC*, 2005, pp. 585–591.
[24] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin, and B. Yu, "DREAMPlace 4.0: Timing-driven global placement with momentum-based net weighting," in *Proc. DATE*, Antwerp, Belgium, March 2022.
[25] A. Chowdhary, K. Rajagopal, S. Venkatesan, T. Cao, V. Tiourin, Y. Parasuram, and B. Halpin, "How accurately can we model timing in a placement engine?" in *Proc. DAC*, 2005, pp. 801–806.
[26] M. A. Jackson and E. S. Kuh, "Performance-driven placement of cell based IC's," in *Proc. DAC*, 1989, pp. 370–375.
[27] W. Swartz and C. Sechen, "Timing driven placement for large standard cell circuits," in *Proc. DAC*, 1995, pp. 211–215.
[28] T. Hamada, C.-K. Cheng, and P. M. Chau, "Prime: A timing-driven placement tool using a piecewise linear resistive network approach," in *Proc. DAC*, 1993, pp. 531–536.
[29] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.
[30] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.
[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[32] C. Chu and Y.-C. Wong, "FLUTE: Fast lookup table based rectilinear steiner minimal tree algorithm for VLSI design," *IEEE TCAD*, vol. 27, no. 1, pp. 70–83, 2008.
[33] M. Kim, J. Hu, J. Li, and N. Viswanathan, "ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite," in *Proc. ICCAD*, 2015, pp. 921–926.
[34] C. Guth, V. Livramento, R. Netto, R. Fonseca, J. L. Güntzel, and L. Santos, "Timing-driven placement based on dynamic net-weighting for efficient slack histogram compression," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, 2015, pp. 141–148.
[35] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proc. ICCAD*. ACM, 2020.