Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation

Cheng-Hsiang Chiu
University of Wisconsin-Madison
Madison, Wisconsin, USA
chenghsiang.chiu@wisc.edu

Chedi Morchdi
Texas A&M University
College Station, Texas, USA
chedi.morchdi@tamu.edu

Chih-Chun Chang
University of Wisconsin-Madison
Madison, Wisconsin, USA
chih-chun.chang@wisc.edu

Cunxi Yu
University of Maryland
College Park, Maryland, USA
cunxiyu@umd.edu

Yi Zhou
Texas A&M University
College Station, Texas, USA
yizhou_tamu@tamu.edu

Tsung-Wei Huang
University of Wisconsin-Madison
Madison, Wisconsin, USA
tsung-wei.huang@wisc.edu

Abstract*—CUDA Graph has shown potential in recent GPU-accelerated statistical static timing analysis (SSTA) propagation applications. By representing dependent SSTA tasks as a task graph and reusing the execution flow, CUDA Graph eliminates repetitive kernel launch overhead and improves task asynchrony. This enables more efficient scheduling of SSTA propagation tasks across logic gates. However, application-given CUDA graphs are often suboptimal, as they focus on capturing circuit structures while overlooking GPU resource availability and scheduling constraints. Unfortunately, the latter heavily relies on the CUDA Graph runtime, which is essentially a black box. To tackle this challenge, we propose a Reinforcement Learning (RL)-based framework that optimizes CUDA graphs by learning to restructure SSTA graphs through interactions with the CUDA Graph runtime. Specifically, we formulate graph restructuring as a node-level adjustment problem and solve it by dynamically appending auxiliary edges to the graph during RL decision-making. To enable more informed decisions for our RL agent, we leverage Graph Neural Networks (GNNs) to encode both the graph structure and the application needs. Compared to the original application-given CUDA graph, our optimized CUDA graph can achieve up to a 12% runtime improvement.

I. Introduction

Statistical static timing analysis (SSTA) is a critical step in Electronic Design Automation (EDA) as it enables more accurate delay estimation than traditional static timing analysis (STA) by modeling on-chip process variation (OCV) as random variables [1], [2]. For example, [3] uses numerical integration to estimate circuit yield by exploring device parameter combinations, while [4] models gate delays as random variables and propagates rise and fall arrival time statistically through the timing graph. As design complexity continues to grow, manufacturing variations have introduced a broad range of OCVs that SSTA algorithms must evaluate during propagation. Despite the daunting computational cost, many computations are structurally independent across gates, transitions, and variation dimensions, revealing substantial opportunities for *data parallelism*. This parallelism makes SSTA propagation well-suited for GPU acceleration, which has emerged as a promising solution to meet its growing performance demands [5]–[9].

However, conventional GPU execution models (e.g., CUDA streams) face significant challenges in efficiently scheduling SSTA workloads. In practice, SSTA workloads involve repeated propagation over the circuit graph across different inputs and statistical values [1]. This leads to frequent kernel launches, which can accumulate to expensive synchronization costs when kernels are iteratively offloaded through one or more CUDA streams. To address this challenge, the recent state-of-the-art [5] leverages *CUDA Graph* to model SSTA propagation as a *GPU task graph*. Specifically, instead of launching kernels individually, CUDA Graph allows the execution flow to be constructed once and replayed multiple times with minimal CPU intervention, eliminating redundant kernel launches and reducing synchronization costs. This particularly benefits many SSTA propagation algorithms, where similar computational patterns are repeatedly executed across different timing scenarios [5].

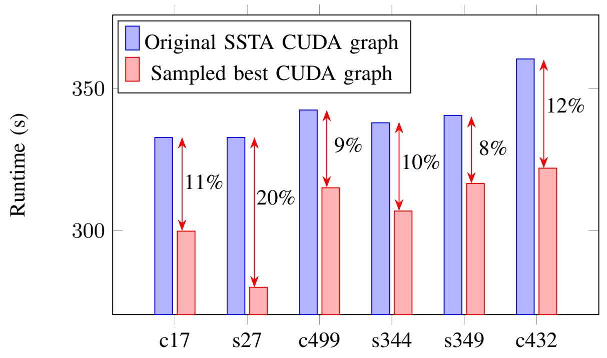


Fig. 1: The original SSTA CUDA graphs leave at least 8% to 20% performance on the table, with the baseline derived from the minimum of 10,000 sampled graphs.

Despite the runtime improvement of CUDA Graph on SSTA workloads [5], application-given CUDA graphs are often suboptimal. For instance, in Figure 1, we evaluate six SSTA benchmarks by generating multiple variants of the application's original CUDA graph. Each sampled graph is created by randomly inserting a small number of auxiliary edges. When comparing runtime performance, we observe that the original graph was 11% slower on *c17* and 20% slower on *s27* than the best-performing sampled graph. This highlights the potential for graph restructuring to enhance execution efficiency. A key factor behind this performance gap is that the application's CUDA graphs prioritize capturing circuit structure but overlook GPU resource availability and scheduling constraints. This oversight often results in resource contention (e.g., multiple tasks competing for limited GPU resources) and reduced task parallelism, as tasks are forced to wait instead of executing concurrently. Unfortunately, the CUDA Graph runtime operates as a black box, with scheduling details hidden as proprietary information. This constraint makes it challenging to design a general-purpose heuristic that can optimize SSTA CUDA graphs across different GPU environments.

Despite the black-box challenge, this problem is particularly well-suited for Reinforcement Learning (RL) [10]–[13], as RL can efficiently explore the complex graph search space and adapt to hidden scheduling behaviors through interactions with the CUDA Graph runtime. For instance, existing work [12] uses RL to adaptively optimize task scheduling for resource efficiency, while DRAS [13] leverages RL to automatically learn and converge to optimal scheduling policies in HPC clusters. Inspired by the success of RL-based schedulers in adaptively optimizing decisions under complex constraints and dynamic runtime conditions, we propose an RL-based framework to optimize CUDA Graph scheduling for SSTA propagation workloads. We summarize our technical contributions as follows:

- We formulate the CUDA Graph scheduling on SSTA propagation workloads as a node-level adjustment problem. With this problem formulation, we transform a complex scheduling challenge into a learnable problem that adjusts node levels in the application's CUDA graph.
- We leverage Graph Neural Networks (GNNs) to capture structural information from both the application workload and the CUDA graph, enabling informed decisions and enhancing the scheduling optimization process.
- We introduce an RL-based framework to solve the node-level adjustment problem. By interacting with the CUDA Graph runtime, the RL agent adaptively learns to restructure the graph, ultimately generating an optimized CUDA graph for improved scheduling performance.

We have evaluated our framework on a set of industrial SSTA benchmarks [5]. For an application-given CUDA graph (i.e., original input CUDA graph), which mainly considers circuit graph structures, our framework generates an optimized CUDA graph by learning to restructure the original input CUDA graph through interactions with the CUDA Graph runtime, achieving up to 12% runtime improvement over the application-given CUDA graphs. Notably, our framework requires no changes to application-level algorithms, but instead restructures the given CUDA graph to guide the CUDA runtime toward better scheduling performance.

II. Scheduling SSTA Propagation Graph on GPU

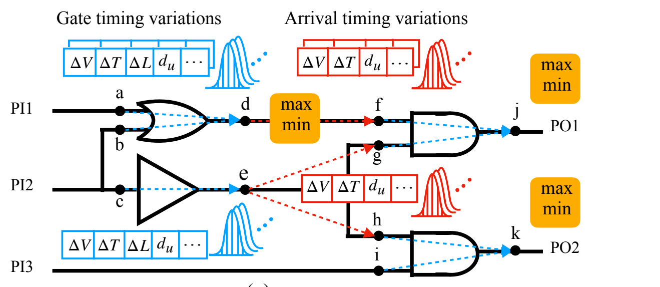
In this paper, we consider the SSTA propagation workload in [5] as our problem formulation: Given an SSTA propagation graph, as illustrated in Figure 2(a), timing variations for gates and interconnections are modeled with random variables and stored in arrays, including voltage fluctuations (ΔV), temperature shifts (ΔT), channel length deviations (ΔL), and so on. In order to deal with various corner cases, each gate has up to 65536 data points where each point represents one statistical phase.[1] During timing propagation through the circuit, each gate contributes the gate delay to the arrival times at its fan-in edges using statistical min/max operations. Delay computations are repeatedly performed for early and late modes, as well as for rise and fall transitions, across multiple corner cases within the SSTA propagation graph. Due to their structural independence across gates, rise/fall transitions, and variation dimensions, these computations exhibit a high degree of data parallelism and are well-suited for GPU acceleration.

Figure 2(b) illustrates how [5] leverages CUDA Graph to execute an SSTA propagation graph from Figure 2(a). The algorithm (1) models circuit pins as CUDA kernels and timing dependencies as edges to capture the graph structure, and (2) inserts a unique memory copy operation before each kernel to transfer the corresponding statistical data points from an array. After the construction, the algorithm outputs a CUDA graph that represents the SSTA propagation graph. Our goal here is to restructure the CUDA graph provided by the application. Instead of modifying application-level algorithms or kernels, we insert a small set of auxiliary edges to guide the CUDA Graph runtime toward better scheduling performance.

III. Proposed Framework

We propose an RL-based framework to generate an optimized CUDA graph of SSTA workloads through interactions with CUDA Graph runtime. We formulate the generation of an optimized CUDA graph as a node-level adjustment problem, which simplifies the scheduling to an appending of edges in the graph and is easy for the RL agent to learn in the decision-making process. Specifically, we move nodes to new levels through the insertion of auxiliary edges, which results in a new CUDA graph without violating the topology order of the original graph description. Figure 3 shows this formulation.

[1]This problem formulation originates from a real-world challenge faced by our industry partners at a leading EDA company. While proprietary details cannot be disclosed, we abstract the core scheduling difficulties and practical constraints into a high-level formulation.


Gate timing variations
Arrival timing variations
ΔV ΔT ΔL d_u ⋯
ΔV ΔT d_u ⋯
max min
PI1
PI2
PI3
PO1
PO2
a b c d e f g h i j k
(a)

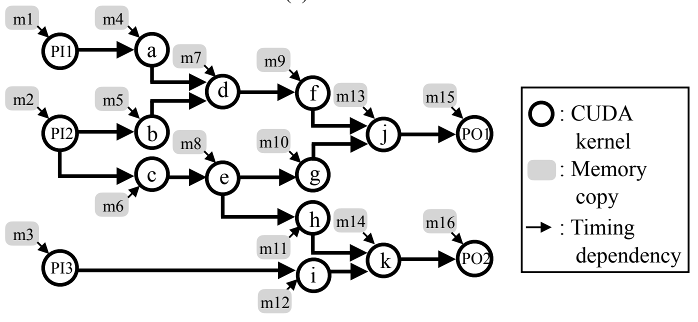

m1 m2 m3 m4 m5 m6 m7 m8 m9 m10 m11 m12 m13 m14 m15 m16
PI1 PI2 PI3 a b c d e f g h i j k PO1 PO2
○ : CUDA kernel
▭ : Memory copy
→ : Timing dependency
(b)

Fig. 2: (a) An SSTA propagation graph. Gate timing (blue) and arrival timing (red) are modeled as random variables in arrays and propagated through the circuit graph using statistical max and min operations. (b) The corresponding CUDA graph of (a). Circles are kernel operations and gray rectangles are memory copy operations.

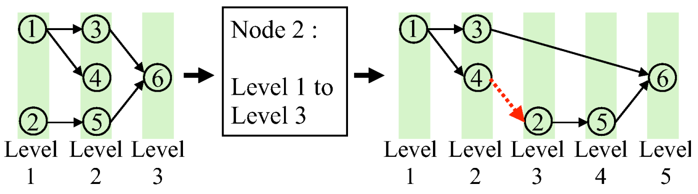


Fig. 3: Illustration of the node-level adjustment formulation. The red edge moves node 2 from level 1 to level 3, resulting in a new graph that potentially alleviates the contention among nodes 3, 4, and 5.

With the node-level adjustment formulation, our RL-based framework comprises two modules. The first module generates a latent *node embedding*, encapsulating both node attributes and graph topology. The second module uses the *node embedding* to adjust node levels and generate a new CUDA graph for the CUDA Graph runtime to execute. The framework learns from the feedback returned by the CUDA Graph runtime to iteratively improve our CUDA graph. Figure 4 illustrates an overview of the proposed framework.

A. Graph Neural Network Module

To generate a better CUDA graph, we need structural information in making the decisions. The information we consider includes both node attributes, which capture the inherent characteristics of each kernel, and the graph topology, which reflects the dependencies and connectivity between kernels. To encode the information, we employ GNN [14] as the first module, as shown in Figure 4. The GNN module comprises three essential components: *adjacency matrix*, which

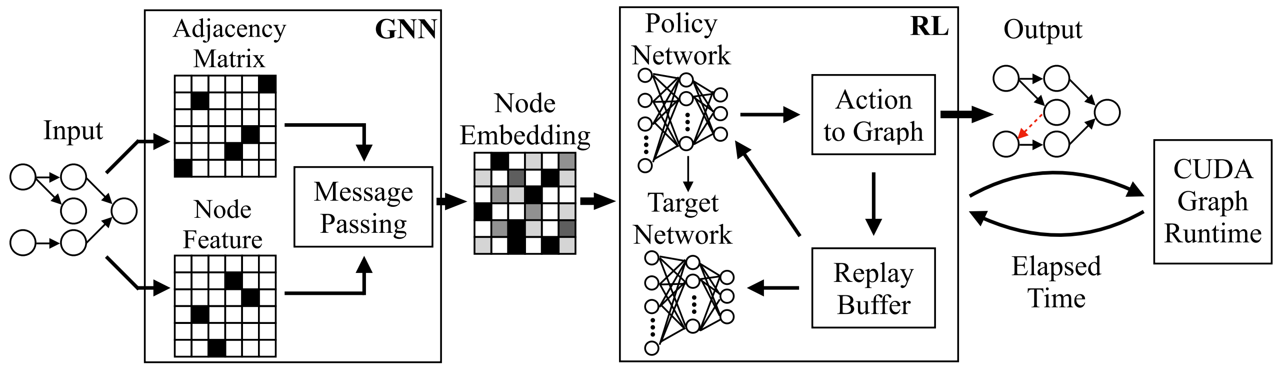
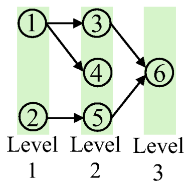


Fig. 4: Overview of the framework. The framework consists of two modules: The first module is GNN and is used to encode the node attributes and graph topology and generate a latent representation *node embedding*. The second module is an RL model that uses the *node embedding* as a state and generates a new CUDA graph for the CUDA Graph runtime to run.

encodes the connectivity between nodes; *node feature*, which represents the initial attributes of each node; and *message passing* mechanism, which enables information propagation across the graph. Using these components, the GNN module effectively captures complex relationships and dependencies within the CUDA graph, providing a rich representation for subsequent decision-making.

1) Adjacency Matrix: The *adjacency matrix* is the first component and is used to define the connectivity structure of the graph and guide the process in the third component *message passing*. The matrix represents the relationships between nodes. A non-zero entry at position (i, j) indicates the presence of an edge pointing from node i to node j, while a zero entry signifies the absence of such a connection. In the third component *message passing*, the *adjacency matrix* acts as a filter, determining which nodes exchange information with each other.

2) Node Feature: The *node feature* is responsible for incorporating both node attributes and graph topology into a unified representation. Each node in the *input graph* represents a kernel operation (preceded by an implicit memory copy) and is characterized by six distinct elements. Three of these elements represent the node's resource requirements. `Thread counts` indicate the number of threads needed for a kernel execution, `block counts` indicate the number of blocks required, and `memory copy size` represents the data size copied from the CPU to the GPU for that node. The remaining three elements encode the graph topology, capturing both local and global properties of the graph. `Node level` indicates the node's level within the graph, `fanin count` represents the number of incoming edges, and `fanout count` represents the number of outgoing edges. Figure 5 illustrates the node feature for all nodes. To avoid larger-magnitude features from overshadowing others and ensure that all features contribute equally to the learning process, we normalize every element between 0 and 1.



	①	②	③	④	⑤	⑥
Threads	32	32	256	512	64	2048
Blocks	2	2	4	1	16	8
Memory	80	60	60	100	120	200
Level	1	1	2	2	2	3
Fanin	0	0	1	1	1	2
Fanout	2	1	1	0	1	0

Fig. 5: Example of the node feature for all nodes on the left graph.

3) Message Passing: The *message passing*, the third component of our GNN, serves as the fundamental mechanism for encoding *node feature* and generating the latent representation *node embedding*. This process involves iteratively propagating information between neighboring nodes, enabling the network to learn complex relationships within the graph. During each iteration, each node aggregates information from its neighbors' previous representations and combines it with the node's own features. For example, for the graph in Figure 5, node 6 aggregates information from both nodes 3 and 5. This aggregation process allows information to flow between nodes across the graph. By repeating this message passing procedure multiple times, we can capture increasingly long-range dependencies, allowing the network to learn sophisticated representations that reflect the global structure of the graph. In this framework, we employ a `two-hop` message passing approach, meaning every node propagates its information to nodes two hops away (e.g., node 2 propagates information to node 6 in Figure 5). We utilize a popular graph convolutional network (GCN) [15] as the underlying network architecture because we do not observe significant runtime difference in our evaluations using other architectures, such as GraphSAGE [16].

B. Reinforcement Learning Module

We leverage an RL agent as the second module to learn to generate an optimized CUDA graph through interactions with the CUDA Graph runtime. The RL agent receives *node embedding*, which encapsulates structural information, from the GNN module. Based on these embeddings, the agent suggests actions, which correspond to node-level adjustments, ultimately resulting in a new CUDA graph. After the CUDA Graph runtime executes the new graph, the agent receives the execution time as feedback. We model this learning process using RL's four fundamental components:

- State (s): A representation of the current situation of the environment that the agent perceives, which is the *node embedding* in the work.
- Action (a): A choice made by the agent that influences the environment, which is the level adjustment of nodes in this work. As each node resides at a specific level, the agent adjusts a target node from its current level to a new level. The action space is limited to three elements: $\{0, 1, 2\}$. Action 0 denotes that the target node remains at its current level. Action 1 denotes a transition to the next level (current level plus one) and Action 2 denotes a transition to the level two steps away (current level plus two). We intentionally limit the action space to avoid higher-level adjustments, as they tend to serialize CUDA Graph execution and degrade performance.
- State transition: The change in the environment's state that occurs as a result of the agent taking an action, including the changes of level, fanin and fanout.
- Reward (r): A signal that CUDA Graph runtime provides to the RL agent after the agent takes an action in a particular state. In this work, we focus on minimizing the execution time (ET) of a CUDA graph. Therefore, we design the reward function to reflect this objective:

$$\text{reward} = -(ET^{\text{after}} - ET^{\text{before}}), \quad (1)$$

where ET^{before} and ET^{after} denote the normalized execution times before and after the action, respectively. Note that minimizing ET is equivalent to maximizing the reward.

To solve the RL problem, we leverage the Deep Q-learning (DQN) algorithm [17]. We do not choose traditional methods [18], such as value iterations, because they are computationally intractable in high-dimensional environments. DQN solves the problem by employing deep neural networks to approximate the `Q-function`, which represents the expected cumulative reward for taking an action (a) in a given state (s), denoted as $Q(s, a)$. The Q-function effectively maps state-action pairs to their corresponding Q-values, enabling the agent to make informed decisions. Next, we discuss how to adapt the DQN algorithm to suggest an action and obtain a CUDA graph with the components, *policy network*, *target network*, *replay buffer*, and *action to graph*, as shown in Figure 4.

1) Policy and Target Network: The *policy network*, a fully connected neural network, maps a state (represented by *node embedding*) to an action, which corresponds to node-level adjustments. The *policy network* has a layered architecture comprising 3072 neurons in the input layer, 256 neurons in the second layer, 64 neurons in the third layer, and 3 neurons in the output layer. The network receives the *node embedding*, forwarded by the GNN module, as its input state. Then the network processes this state information through the two hidden layers. Finally, the network outputs a set of three Q-values. Each Q-value corresponds to the expected reward associated with a specific action that adjusts the level of a target node in the graph. The first Q-value corresponds to action 0, which does not adjust the target node's level. The second Q-value corresponds to action 1, which increments the current level by 1, and the third Q-value corresponds to action 2, which increments the current level by 2. To balance exploration and exploitation during the learning process, we employ the ϵ-greedy strategy [18] in determining the action. This strategy allows the agent to explore new actions with probability ϵ and exploit the learned policy with probability $1 - \epsilon$.

In addition to the *policy network*, we incorporate a separate network, the *target network*, which plays a crucial role in stabilizing the training process. The *target network* maintains an identical architecture to the *policy network*, but its weights are updated less frequently. This separation is essential for mitigating the overestimation bias [19], a common issue in DQN that can lead to unstable training. We periodically copy the weights from the *policy network* to the *target network* using a soft update mechanism [20], expressed in the equation:

$$\theta' = \tau\theta + (1 - \tau)\theta', \tag{2}$$

where θ' denotes the parameters of the *target network*, θ denotes the parameters of the *policy network*, and τ denotes the update rate of the *target network*. This soft update approach allows for a gradual and controlled transfer of learned information from the *policy network* to the *target network*. We use the *target network* to evaluate the expected Q-value of the action suggested by the *policy network*. This evaluation provides an unbiased estimate of the Q-value for the next state, crucial for accurate learning. The expected Q-value is calculated using the equation:

$$Q^{expected} = r + \gamma \max_{a} Q(s', a), \tag{3}$$

where γ represents the discount factor, reflecting the importance of future rewards, and s' denotes the state after the action is executed, reflecting the updated environment.

During the training phase, we employ the Huber loss function [21] to address the challenge of outliers in noisy Q-value estimates, which are common in RL. Our objective is to minimize the Huber loss. The mathematical definition of the Huber loss is shown below:

$$\mathcal{L} = \begin{cases} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(\frac{1}{2}\delta^2\right) & \text{if } |\delta| \leq 1, \\ \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(|\delta| - \frac{1}{2}\right) & \text{otherwise.} \end{cases}$$

where

$$\delta = Q(s, a) - Q^{expected}. \tag{4}$$

Once we calculate the loss, we leverage backpropagation to propagate the error signal through the entire model. To improve convergence, we stop propagating the error signal to the GNN model after 10 epochs in our evaluation. Finally, we utilize the Adam optimizer [22].

2) Action to Graph: The *action-to-graph* component serves as the link between the *policy network*'s output and the physical modification of the CUDA graph. It performs a transformation that converts the action proposed by the *policy network* into a concrete change within the graph, that is, the appending of a new auxiliary edge. Each action, taking values of 0, 1, or 2, corresponds to a level adjustment for a given target node. These adjustments are : (1) maintaining the current level (action 0), (2) incrementing the level by one (action 1), or (3) incrementing the level by two (action 2). When action 0 is suggested by the *policy network*, it implies that no change is needed for the target node's level. However, when actions 1 or 2 are suggested, they necessitate the addition of an auxiliary edge to facilitate the target node's transition to the new level.

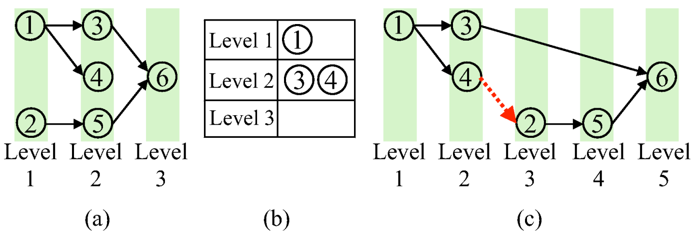


Fig. 6: Illustration of using bucket list to convert an action to an edge. (a) A graph with 5 edges and 6 nodes within 3 layers. (b) A bucket list for node 2. (c) A new graph after moving node 2 from level 1 to level 3. The red dash edge is the auxiliary edge.

To add a new auxiliary edge for a target node, a straightforward approach is to connect the target node at its new level to a randomly selected node at the immediately preceding level. For example, as illustrated in Figure 6(a), an edge from node 1 to node 2 facilitates node 2's transition from level 1 to level 2. However, this design can introduce cycles within the graph, particularly when a successor node is involved in the random selection process. Consider Figure 6(a), if node 5, which is a successor of node 2 and randomly selected from level 2, is connected to node 2 to transition node 2 to a new level (level 3), a cycle is immediately introduced. Therefore, we require a mechanism to explicitly exclude successors of a target node from the random selection process, ensuring that the graph remains acyclic.

To prevent cycles while introducing an auxiliary edge for a target node, we construct a bucket list for each node. Each bucket list is an associative container that stores key-value pairs. The key represents a level, and the value is a set of nodes located at that level. Crucially, this bucket list excludes all the successor nodes of the target node. For instance, as illustrated in Figure 6(b), the bucket list for node 2 in Figure 6(a) contains two entries: {level 1: node 1} and {level 2: {nodes 3, 4}}. Notably, this bucket list intentionally omits node 2's successors, nodes 5 and 6. This design enables a straightforward selection of a non-successor node, such as node 4 at level 2, to add a new edge to node 2. This edge addition transitions node 2 from level 1 to level 3, as shown in Figure 6(c). With this design, we can efficiently convert an action suggested by the *policy network* into a new edge without introducing any cycles into the resulting CUDA graph.

3) Replay Buffer: Replay buffer serves as a memory mechanism in the RL module for a stable and effective learning. We store a collection of transitions which consists of the current state, the suggested action, the received reward, and the next state. By storing these transitions, the

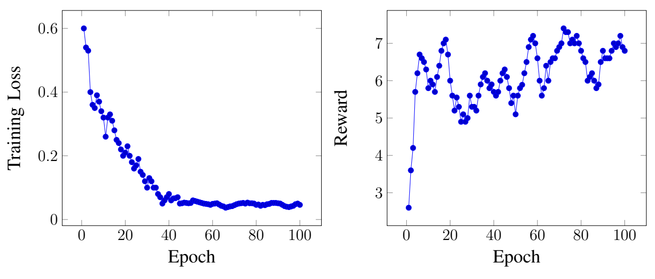


Fig. 7: Training loss and rewards achieved by the RL policy.

agent is able to learn from past interactions. During training, the agent randomly samples batches from the replay buffer, which breaks the correlation between consecutive transitions and enables more efficient learning.

IV. Experimental Results

We implemented our framework using C++17 and CUDA 12.2 and compiled the program with the nvcc compiler on a host compiler of g++11.4 with -std=c++17 and -O3 enabled. We used Pytorch to train and test the model. We ran all the experiments on a Ubuntu Linux 22.04 machine with 20 Intel i5-13500 CPU cores at 4.8 GHz and 128 GB RAM, and an Nvidia RTX A4000 GPU.

A. Benchmarks and Baseline

We evaluated the runtime performance on 12 circuit graphs derived from the [5]. Each circuit graph represents an SSTA propagation graph, as detailed in Section II. The statistical timing quantities of varying batch size B for each pin were sampled from a normal distribution. Table I presents the statistics of the 12 circuit graphs used in the evaluation, with the default batch size B set to 64, meaning that we calculated a pin's 64 data points concurrently per iteration. To finish 65536 data points, each benchmark requires 1024 iterations. We used the application's original CUDA graph as the baseline, which also represents the GPU-accelerated solution provided by [5], [23].

B. Training

We trained our model on a synthetic circuit derived from [5]. To ensure generalization, the training and testing phases used different circuit instances. We used the following hyper-parameters: target network update rate of 0.005, discount factor of 0.99, training iterations of 100, initial ϵ of 1.0, final ϵ of 0.05, ϵ decay rate of 0.997, learning mini batch size of 16, and Adam optimizer learning rate of 0.0001. Figure 7 shows the training error and the rewards achieved by the RL policy. The left plot demonstrates a rapid decay in training loss, indicating effective policy learning. The right plot shows the RL policy converging to rewards between 5 and 7.

C. Overall Performance Comparison

Table I presents the runtime performance of the baseline and our solution. Our solution consistently outperforms the baseline across all benchmarks with batch size 64. For instance, our solution achieves a 12% runtime improvement over the baseline for *s27*. Similarly, for *c2670*, we observe a 3.6%. The reason is the following. The baseline exhibits excessive task parallelism, which, while seemingly beneficial, can introduce resource contention that ultimately degrades runtime performance. Our framework appends edges within the CUDA graph to slightly reduce task parallelism in favor of improved resource utilization. This controlled parallelism aims to achieve a balance between task execution and scheduling management, leading to overall improved performance compared to the baseline. Note that the runtime improvement may appear small for some benchmarks (e.g., 3% on *usb_phy*), but in practice, the gain can accumulate to hours as SSTA applications must run many iterations across different input values

TABLE I: Runtime comparison and circuit statistics of the benchmarks. The batch size in this table is 64. T^B and T^O denote the runtime of the baseline and ours, respectively. Δt denotes the runtime difference between the baseline and ours. *Impr.* denotes the runtime improvement of our CUDA graph over the baseline. $\|E\|'$ denotes the number of edges in the new CUDA graph generated by our framework.

Benchmark	$\|V\|$	$\|E\|$	T^B (s)	T^O (s)	Δt (s)	Impr.	$\|E\|'$
c17	75	78	332.81	306.19	26.61	8%	89
s27	81	87	332.84	292.89	39.95	12%	95
c17_2	150	156	333.47	314.8	18.67	5.6%	162
s27_2	243	261	335.8	308.6	27.2	8.1%	271
c432	483	619	360.51	330.23	30.28	8.4%	633
c499	604	742	342.54	320.62	21.92	6.4%	760
s344	526	625	338	317.72	20.28	6%	640
s349	550	649	340.63	325.64	14.99	4.4%	664
c2670	1365	1665	437.76	422	15.76	3.6%	1685
s1196	1854	2344	494.74	474.95	19.79	4%	2366
s1494	2292	2925	539.96	524.84	15.12	2.8%	2950
usb_phy	2447	2999	540.8	524.58	16.22	3%	3036
Average					22.23 s	6%	

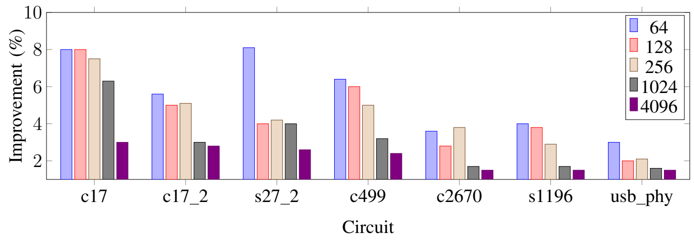


Fig. 8: Plot of runtime improvement of our framework over the baseline on seven benchmarks with five different batch sizes.

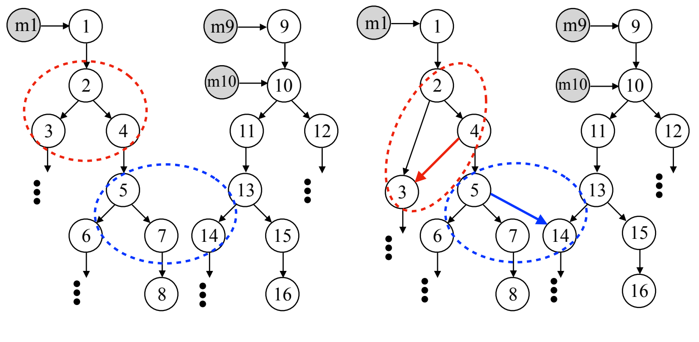
and configurations [5]. More importantly, this runtime gain comes at little cost to developers, as our framework requires no changes to application-level algorithms but simply restructures the CUDA graph to guide the runtime toward better scheduling performance.

These results highlight that batch size is a critical factor in maximizing the benefits of our optimized CUDA graph. We observe that the runtime improvement becomes smaller at a larger batch size. For example, in Figure 8, when running *s27_2*, the improvement decreases from 8.1% with batch size 64 to 2.6% with batch size 4096. We attribute this result to the increased computational load associated with larger batch sizes. Apparently, as kernel computation begins to dominate overall runtime, the relative benefit of improved scheduling diminishes. However, we should also notice that larger batch sizes incur higher GPU memory usage, which in turn limits the maximum circuit size that can be processed on a single GPU.

Our RL algorithm always brings positive benefits, as it does not modify any application-level algorithms, but restructures the given CUDA graph to guide the CUDA runtime toward better scheduling performance. These edges facilitate improved scheduling by adapting the application's needs (i.e., application-level information) to the changing environment on GPU. However, excessive edge additions can potentially serialize CUDA Graph execution, leading to performance degradation. Our framework strikes a balance by minimizing the number of auxiliary edges added. In Table I, our framework consistently introduces a small number of auxiliary edges. For instance, in the *usb_phy* circuit with batch size 64, our CUDA graph includes only 37 additional edges. Figure 9 visualizes the application's original CUDA graph and the optimized CUDA graph from our RL algorithm. Our framework added two edges to reduce scheduling overhead caused by excessive task parallelism in the application's CUDA graphs.

D. Quality of Result

In addition to the runtime comparison, we evaluate the *Quality of Result* (QoR) by judging how close each CUDA graph is to



(a) Original CUDA graph (b) RL-optimized CUDA graph

Fig. 9: Partial *c17* CUDA graph visualization: (a) application's original, (b) our optimized CUDA graph. Blue/red dashed cycles indicate changes due to blue/red edge additions. White circles denote kernels and gray denote memory copies.

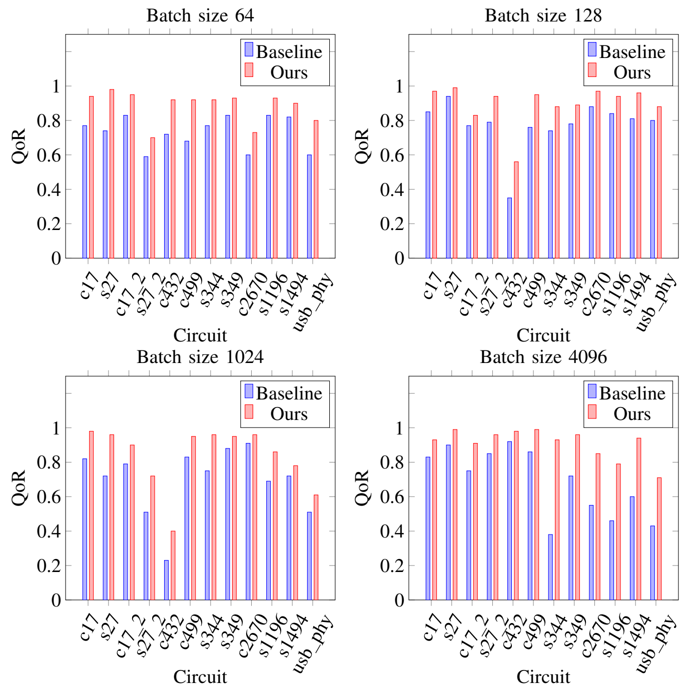


Fig. 10: QoR between the application-given and our optimized CUDA graphs. A value closer to 1 indicates better QoR.

the potentially best result. Specifically, we generated 10,000 distinct CUDA graphs per benchmark by randomly appending auxiliary edges to the application's original CUDA graph. Then, from these 10,000 sampled graphs, we extracted the minimum and maximum runtimes and normalized them to [0, 1] to establish the potentially best and worst performance bounds. We can express QoR as follows,

$$QoR = 1 - \left(\frac{T - T^{min}}{T^{max} - T^{min}}\right), \quad (5)$$

where T^{min} and T^{max} denote the sampled minimum and maximum runtime, respectively. In Figure 10, our optimized graph consistently exhibits a higher QoR compared to the baseline. For example, on the *c2670* benchmark with batch size 128, our solution achieves a normalized QoR of 0.96, compared to 0.9 for the baseline. Furthermore, the majority of our optimized CUDA graphs have QoR over 0.8, indicating a close proximity to the best sample, whereas the baseline typically falls between 0.6 and 0.8. This result highlights the effectiveness of our approach in consistently generating higher-quality CUDA graphs.

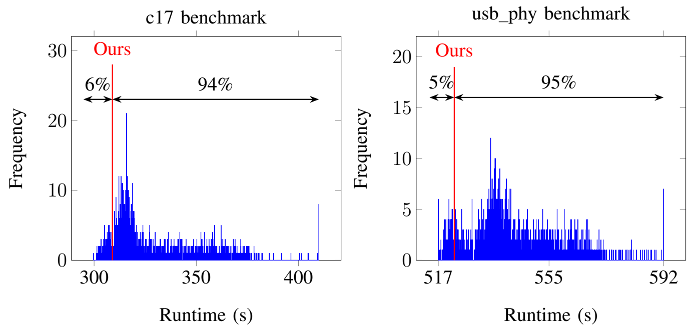


Fig. 11: Histogram of the random edge insertion approach on *c17* and *usb_phy*. 2000 different CUDA graphs were generated by randomly appending the same amounts of auxiliary edges as our solution to the application's original CUDA graph. Only a small portion ($\sim$ 5%) of the 2000 CUDA graphs perform better runtime performance as our optimized CUDA graph (indicated by the vertical red line).

E. Comparison with Random Edge Insertion

To further validate the effectiveness of our framework, we implemented a method that randomly inserts the same quantity of auxiliary edges as ours into the original CUDA graph. The goal is to show that blindly inserting edges without a learning-guided process yields limited or even negative performance improvement. As shown in Figure 11, we randomly inserted 11 and 37 edges into the application's *c17* and *usb_phy* CUDA graphs, ran this experiment 2000 times, and recorded the runtime for each run. We notice that only a small fraction of CUDA graphs can achieve performance comparable to our RL-based solution. For example, on the *usb_phy* benchmark, fewer than 5% of CUDA graphs match the performance of our RL-optimized CUDA graph. We attribute this finding to the fact that our framework learns to identify beneficial edge insertions by interacting with the CUDA Graph runtime and adapting to its hidden scheduling mechanisms. In contrast, the non-learning approach inserts edges randomly without leveraging system feedback. While random insertion may occasionally yield good performance after many attempts, most resulting CUDA graphs fail to deliver decent improvements.

V. Conclusions

We have proposed a reinforcement learning-based framework to optimize CUDA Graph scheduling on SSTA propagation workloads. We have formulated the CUDA Graph scheduling as a node-level adjustment problem. To solve the problem formulation, we have leveraged a graph neural network to encode structural information and employed a deep Q learning algorithm to interact with CUDA Graph runtime and generate an optimized CUDA graph. Compared to the baseline, we achieved up to 12% runtime improvement. Notably, this performance improvement is almost free, as our framework requires no changes to application-level algorithms, but simply restructures the application's given CUDA graph to guide the CUDA Graph runtime toward better scheduling performance. In the future, we plan to extend our algorithms to other applications powered by task graph parallelism [7]–[9], [11], [12], [23]–[91].

Acknowledgment

This project is supported by NSF grants 2134223, 2237830, 2235276, 2349144, 2349143, 2349582, 2349141, 2349670, and 2403134. We appreciate reviewers' feedback to help us improve this paper.

References

[1] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer, “Statistical timing analysis: From basic principles to state of the art,” 2008.

[2] C. Forzan and D. Pandini, “Statistical static timing analysis: A survey,” 2009.

[3] J. Jess, K. Kalafala, S. Naidu, R. Otten, and C. Visweswariah, “Statistical timing for parametric yield prediction of digital integrated circuits,” 2003.

[4] C. Visweswariah, K. Ravindran, K. Kalafala, S. Walker, S. Narayan, D. Beece, J. Piaget, N. Venkateswaran, and J. Hemmett, “First-order incremental block-based statistical timing analysis,” 2006.

[5] C.-C. Chang and T.-W. Huang, “Statistical timing graph scheduling algorithm for gpu computation,” 2025.

[6] K. Gulati and S. Khatri, “Accelerating statistical static timing analysis using graphics processing units,” 2009.

[7] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, “G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis,” in *ACM/IEEE DAC*, 2024.

[8] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, “GPU-accelerated Critical Path Generation with Path Constraints,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[9] Z. Guo, T.-W. Huang, and Y. Lin, “GPU-accelerated Static Timing Analysis,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” in *Arxiv.org*, 2013.

[11] C.-H. Chiu, C. Morchdi, Y. Zhou, B. Zhang, C. Chang, and T.-W. Huang, “Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling,” in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2024.

[12] C. Morchdi, C.-H. Chiu, Y. Zhou, and T.-W. Huang, “A Resource-efficient Task Scheduling System using Reinforcement Learning,” in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024.

[13] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, and M. E. Papka, “Deep reinforcement agent for scheduling in hpc,” in *IEEE IPDPS*, 2021, pp. 807–816.

[14] L. Wu, P. Cui, J. Pei, and L. Zhao, “Graph neural networks,” 2022.

[15] T. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *International Conference on Learning Representations (ICLR)*, 2017.

[16] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation leanring on large graphs,” 2017.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *Arxiv.org/abs/1312.5602*, 2013.

[18] R. S. Sutton and A. G. Barto, “Reinforcement Learning: An Introduction,” in *MIT Press*, 2018.

[19] Q. Lan, Y. Pan, and A. F. andMartha White, “Maxmin q-learning: Controlling the estimation bias of q-learning,” in *The International Conference on Learning Representations (ICLR)*, 2020.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” in *International Conference on Learning Representations*, 2016.

[21] “Huber Loss,” https://en.wikipedia.org/wiki/Huber_loss.

[22] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *International Conference on Learning Representations (ICLR)*, 2014.

[23] D.-L. Lin and T.-W. Huang, “Efficient GPU Computation using Task Graph Parallelism,” in *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2021.

[24] J. Tong, W.-L. Lee, U. Y. Ogras, and T.-W. Huang, “Scalable Code Generation for RTL Simulation of Deep Learning Accelerators with MLIR,” in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2025.

[25] C.-C. Chang and T.-W. Huang, “Statistical Timing Graph Scheduling Algorithm for GPU Computation,” in *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[26] W.-L. Lee, S. Jiang, D.-L. Lin, C. Chang, B. Zhang, Y.-H. Chung, U. Schlichtmann, T.-Y. Ho, , and T.-W. Huang, “iG-kway: Incremental k-way Graph Partitioning on GPU,” in *ACM/IEEE Design Automation Conference (DAC)*, 2025.

[27] Y.-H. Chung, S. Jiang, W. L. Lee, Y. Zhang, H. Ren, T.-Y. Ho, and T.-W. Huang, “SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU,” in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2025.

[28] S. Jiang, Y.-H. Chung, C.-C. Chang, T.-Y. Ho, and T.-W. Huang, “BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram,” in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[29] S. Gener, S. Hassan, L. Chang, C. Chakrabarti, T.-W. Huang, U. Ograss, , and A. Akoglu, “A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems,” in *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*, 2025.

[30] C. Chang, B. Zhang, C.-H. Chiu, D.-L. Lin, Y.-H. Chung, W.-L. Lee, Z. Guo, Y. Lin, and T.-W. Huang, “PathGen: An Efficient Parallel Critical Path Generation Algorithm,” in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[31] B. Zhang, C. Chang, C.-H. Chiu, D.-L. Lin, Y. Sui, C.-C. Chang, Y.-H. Chung, W.-L. Lee, Z. Guo, Y. Lin, and T.-W. Huang, “iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis,” in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[32] W.-L. Lee, D.-L. Lin, C.-H. Chiu, U. Schlichtmann, and T.-W. Huang, “HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner,” in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2025.

[33] C.-C. Chang, B. Zhang, and T.-W. Huang, “GSAP: A GPU-Accelerated Stochastic Graph Partitioner,” in *ACM ICPP*, 2024, p. 565–575.

[34] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang, “HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis,” in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2024.

[35] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei, “FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array,” in *ACM ICPP*, 2024, p. 388–399.

[36] J. Tong, L. Chang, U. Y. Ogras, and T.-W. Huang, “BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism,” in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[37] C. Chang, C.-H. Chiu, B. Zhang, and T.-W. Huang, “Incremental Critical Path Generation for Dynamic Graphs,” in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[38] C.-H. Chiu and T.-W. Huang, “An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads,” in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024.

[39] D.-L. Lin, T.-W. Huang, J. S. Miguel, and U. Ogras, “TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling,” in *International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2024.

[40] W. L. Lee, D.-L. Lin, T.-W. Huang, S. Jiang, T.-Y. Ho, Y. Lin, and B. Yu, “G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner,” in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[41] C. Chang, T.-W. Huang, D.-L. Lin, G. Guo, and S. Lin, “Ink: Efficient Incremental k-Critical Path Generation,” in *ACM/IEEE DAC*, 2024.

[42] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W. L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang, “G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis,” in *ACM/IEEE DAC*, 2024.

[43] T.-W. Huang, B. Zhang, D.-L. Lin, and C.-H. Chiu, “Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System,” in *ACM International Symposium on Physical Design (ISPD)*, 2024.

[44] Z. Guo, T.-W. Huang, J. Zhou, C. Zhuo, Y. Lin, R. Wang, and R. Huang, “Heterogeneous Static Timing Analysis with Advanced Delay Calculator,” in *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2024.

[45] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, “Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms,” in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2023.

[46] C.-C. Chang and T.-W. Huang, “uSAP: An Ultra-Fast Stochastic Graph Partitioner,” in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023.

[47] S. Jiang, T.-W. Huang, and T.-Y. Ho, “GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction,” in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023.

[48] ——, “SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU,” in *ACM International Conference on Parallel Processing (ICPP)*, 2023.

[49] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang, “GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs,” in *ACM/IEEE Design Automation Conference (DAC)*, 2023.

[50] T.-W. Huang, “qTask: Task-parallel Quantum Circuit Simulation with Incrementality,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023.

[51] E. Dzaka, D.-L. Lin, and T.-W. Huang, "Parallel And-Inverter Graph Simulation Using a Task-graph Computing System," in *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*, 2023.

[52] G. Guo, T.-W. Huang, and M. D. F. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2023.

[53] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. Wong, "A GPU-Accelerated Framework for Path-Based Timing Analysis," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[54] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023.

[55] T.-W. Huang and L. Hwang, "Task-parallel Programming with Constrained Parallelism," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

[56] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2022.

[57] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus," in *ACM International Conference on Parallel Processing (ICPP)*, 2022.

[58] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism using Control Taskflow Graph," in *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2022.

[59] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms," in *ACM/IEEE Design Automation Conference (DAC)*, 2022.

[60] T.-W. Huang and Y. Lin, "Concurrent CPU-GPU Task Programming using Modern C++," in *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*, 2022.

[61] K. Zhou, Z. Guo, T.-W. Huang, and Y. Lin, "Efficient Critical Paths Search Algorithm using Mergeable Heap," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022.

[62] D.-L. Lin and T.-W. Huang, "Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.

[63] M. Mower, L. Majors, and T.-W. Huang, "Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs," in *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2021.

[64] T.-W. Huang, "TFProf: Profiling Large Taskflow Programs with Modern D3 and C++," in *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*, 2021.

[65] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021.

[66] Y. Zamani and T.-W. Huang, "A High-Performance Heterogeneous Critical Path Analysis Framework," in *IEEE High-Performance Extreme Computing Conference (HPEC)*, 2021.

[67] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads," in *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*, 2021.

[68] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Path-based Timing Analysis," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[69] Z. Guo, T.-W. Huang, and Y. Lin, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[70] K.-M. Lai, T.-W. Huang, P.-Y. Lee, and T.-Y. Ho, "ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021.

[71] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2021.

[72] T.-W. Huang, C.-X. Lin, and M. Wong, "OpenTimer v2: A Parallel Incremental Timing Analysis Engine," *IEEE Design and Test (DAT)*, 2021.

[73] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022.

[74] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[75] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2022.

[76] C.-X. Lin, T.-W. Huang, and M. Wong, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2020.

[77] D.-L. Lin and T.-W. Huang, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2020.

[78] Z. Guo, T.-W. Huang, and Y. Lin, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020.

[79] T.-W. Huang, "A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020.

[80] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE Design Automation Conference (DAC)*, 2020.

[81] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM Multimedia Conference (MM)*, 2019.

[82] ——, "An Efficient and Composable Parallel Task Programming Library," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2019.

[83] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A General Cache Framework for Efficient Generation of Timing Critical Paths," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[84] T.-W. Huang, C.-X. Lin, , and M. Wong, "Distributed Timing Analysis at Scale," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[85] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Essential Building Blocks for Creating an Open-source EDA Project," in *ACM/IEEE Design Automation Conference (DAC)*, 2019.

[86] T.-W. Huang, C.-X. Lin, and M. Wong, "DtCraft: A High-performance Distributed Execution Engine at Scale," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[87] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "A General-purpose Distributed Programming System using Data-parallel Streams," in *ACM Multimedia Conference (MM)*, 2018.

[88] C.-X. Lin, T.-W. Huang, T. Yu, and M. Wong, "A Distributed Power Grid Analysis Framework from Sequential Stream Graph," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2018.

[89] T.-W. Huang, C.-X. Lin, and M. Wong, "DtCraft: A Distributed Execution Engine for Compute-intensive Applications," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2017.

[90] T.-Y. Lai, T.-W. Huang, , and M. Wong, "Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2017.

[91] T.-W. Huang, M. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A Distributed Timing Analysis Framework for Large Designs," in *IEEE/ACM Design Automation Conference (DAC)*, 2016.