Enhancing Graph Partitioning with Reinforcement Learning-based Initialization

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

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

Wan Luan Lee
Department of ECE
University of Wisconsin-Madison
Madison, Wisconsin, USA
wanluan.lee@wisc.edu

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

Yi Zhou
Department of CSE
Texas A&M University
College Station, Texas, USA
yi.zhou@tamu.edu

***Abstract*—Graph partitioning is fundamental in many computer-aided design (CAD) applications, as it decomposes large problems into more manageable subproblems. State-of-the-art frameworks often count on the Fiduccia-Mattheyses (FM) algorithm to iteratively refine partition quality. However, the solution quality of FM heavily depends on the quality of the initial partition, which is typically generated randomly and can lead to suboptimal results. To address this issue, we propose a reinforcement learning (RL)-based algorithm to generate high-quality initial partitions that improve the final solution quality of the FM algorithm. By designing a reward function that promotes cut size reduction while penalizing partition imbalance, our RL agent learns to produce balanced graph partitions with minimized cut sizes. Evaluated on a set of industrial circuits, our experiments show that running FM on our RL-based initial partitions yields significantly better cut sizes than with random initialization.**

I. Introduction

Graph partitioning is fundamental in many computer-aided design (CAD) applications, as it decomposes large problems into smaller, manageable subproblems while minimizing interconnect between them. For example, static timing analyzers [1], [2] and RTL simulators [3], [4] partition the circuit graph into dependent pieces for multithreading; placement and routing tools [5] partition the graph to implement divide-and-conquer algorithms. As a result, high-performance graph partitioning libraries, such as METIS [6], KaHyPar [7], and G-kway [8], [9], have been developed and widely adopted in both academia and industry.

While existing graph partitioners adopt different strategies (e.g., multilevel [6], [7], data parallelism [8]), nearly all of them count on *Fiduccia-Mattheyses* (FM)-inspired variants [10] to perform greedy refinements. Specifically, for a given initial partition, FM is applied to iteratively refine the partition quality by moving vertices between partitions to locally reduce the cut size. Despite the efficiency, this type of local refinement is highly sensitive to the quality of the initial partition, often leading to suboptimal results if the initial partition is poor. As shown in Figure 1, where we randomly sampled different initial partitions as input to FM, the solution quality in terms of cut size can vary dramatically. For instance, on circuit aes_core, the maximum difference reaches up to 12×. Notably, only a few initial partitions achieve very small cut sizes (below the red line). Although existing partitioners have attempted to address this issue using different heuristics (e.g., random assignment [6], greedy clustering [11]), they often lack a global perspective of the underlying graph structure, leading to only moderate-quality initial partitions for FM.

To address this challenge, we propose a *reinforcement learning* (RL)-based algorithm that generates high-quality initial partitions to guide FM toward better solutions. Our motivation stems from the fact that RL is a powerful framework for iterative decision-making, which aligns well with the refinement steps of FM. Specifically, FM improves a given partition by iteratively moving vertices across partitions, guided by a *gain* value that quantifies the reduction in cut size. Similarly, an RL agent performs *actions* (e.g., moving vertices) and receives a *reward* reflecting the quality of those actions. This alignment between FM's gain-driven local optimization and RL's reward-based learning suggests that RL can serve as a generalizable framework for learning effective partitioning strategies. We summarize our technical contributions as follows:

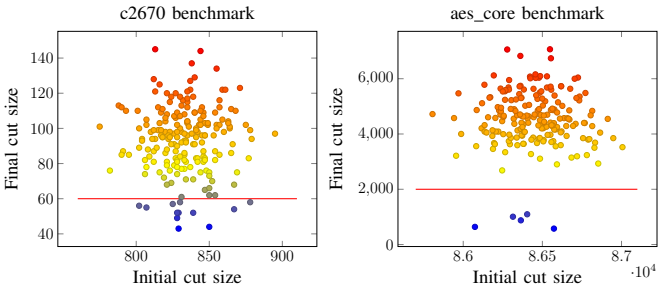

Fig. 1. Different initial partitions can result in significantly varying cut sizes for the FM partitioner.

- We design an RL-based initializer to generate high-quality input partitions for FM. By creating a reward function that promotes cut size reduction and penalizes partition imbalance, our RL agent learns to produce balanced partitions with minimal interconnect. Using these RL-initialized partitions, FM achieves significantly better cut sizes and converges faster than with greedy initializations.
- We design an RL state that captures both local and global graph structure. Locally, it encodes the distribution of partition assignments among neighbors, while globally, it includes partition sizes. This design enables the policy to make high-quality decisions, leading to higher rewards.
- We introduce two strategies to optimize the scalability of RL inference with large graphs, *vectorized balance filtering* and *incremental state computation*. These strategies leverage vectorized operations and efficient data access patterns to significantly improve testing runtime performance.

We have evaluated on a set of large-scale industrial circuits. Compared with baselines, running FM on RL-initialized partitions yields significantly better cut sizes than random or greedy initializations. We plan to open-source our partitioner to benefit both the CAD and machine learning communities in graph partitioning research.

II. Preliminaries

A. Graph Partitioning

Given an undirected graph, $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge $e(u, v) \in E$ connects a pair of vertices (u, v) and has unit weight. We denote $\{p_0, p_1, ..., p_{k-1}\}$ as a k-way partition of V, and use $P(v)$ to indicate the partition to which vertex v belongs. In this work, we focus on the fundamental 2-way partitioning problem as a starting point, laying the groundwork for future extensions to the general k-way partitioning problem.

The goal of graph partitioning is to find a partitioning that minimizes the cut size, defined as the number of edges $e(u, v)$ that connect vertices in different partitions, i.e., $P(u) \neq P(v)$. Cut size is a widely used metric for evaluating partition quality, as it reflects the amount of interconnections between partitions. In addition to minimizing cut size, the partitions must satisfy a balance constraint, which limits the number of vertices in each partition. Specifically, for all partitions p_i, the size must satisfy $|p_i| \leq (1+\delta)\frac{|V|}{k}$, where $|\cdot|$ denotes the cardinality and $0 < \delta \ll 1$ is the imbalance ratio specified by the application.

B. Reinforcement Learning

Reinforcement Learning (RL) is a powerful framework for learning optimal decision-making in dynamic environments [12], [13]. The interaction between the RL agent and the environment is described by the following Markov Decision Process (MDP).

1) Markov Decision Process: MDP is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathsf{P}, r)$, where $\mathcal{S}$ denotes a collection of states that represent possible configurations of the environment, $\mathcal{A}$ denotes a collection of actions that the agent can take, P denotes the state transition kernel that specifies the probability of transitioning from one state to the next given a particular action, and r denotes a reward signal that the agent receives from the environment.

The following equation (1) illustrates the trajectory of MDP.

$$\text{(MDP):}\quad s_0 \overset{\pi(\cdot|s_0)}{\longrightarrow} a_0 \overset{\mathsf{P}(\cdot|s_0,a_0)}{\longrightarrow} (s_1, r_0) \overset{\pi(\cdot|s_1)}{\longrightarrow} a_1 \cdots \tag{1}$$

To elaborate, at each timestep t, the agent observes the current state s_t of the environment, based on which it samples an action a_t following its policy π. After this action is taken, the current state transfers to the next state s_{t+1} following the state transition kernel P, and the agent receives a reward r_t. The agent aims to learn the optimal policy π^* that maximizes the cumulative reward, i.e., $\pi^* = \arg\max_\pi \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi]$, where $\gamma \in (0, 1)$ is a discount factor specifying the importance of future rewards relative to recent rewards.

2) Deep Double Q-Learning Algorithm: To learn the optimal policy, a widely used RL algorithm is *Deep Double Q-learning* [14], which estimates a state-action value function to guide decision-making. Specifically, the value function $Q(s, a)$ represents the expected cumulative reward obtained by taking action a in state s and subsequently following the optimal policy. This function satisfies the modified, well-known Bellman equation:

$$Q(s_t, a_t) = r_t + \gamma \mathbb{E}\left[Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a))\right]. \tag{2}$$

Once the optimal state-action value function is learned, it naturally induces a policy that selects the action associated with the highest value in a certain state s_t, i.e., $\pi(a_t|s_t) = \arg\max_a Q(s_t, a)$. In modern RL, we usually parameterize the state-action value function $Q(s, a)$ using a neural network Q_θ, where θ represents the network parameters. In particular, the network model training process involves the following steps:

- **Collecting MDP data:** At each timestep t, the agent takes an action a_t (using ϵ-greedy strategy), the environment transfers from state s_t to s_{t+1}, and the agent receives a reward signal r_t. We store this transition data (s_t, a_t, r_t, s_{t+1}) over the past N timesteps in a so-called experience replay memory.
- **Sampling data:** To train the Q-network, at each training iteration, we first randomly sample a mini batch of B data samples from the experience replay memory and compute the target $y_T = r_T + \gamma Q_{\theta'}(s_{T+1}, \arg\max_a Q_\theta(s_{T+1}, a))$ for all data $T \in B$ based on Equation 2. Here, $Q_{\theta'}$ denotes the target network whose parameters are periodically copied from θ every K iterations. The purpose is to decouple the original Q-network from the target network in order to allow proper use of backpropagation later.
- **Updating network parameters:** We compute the batch MSE loss $L = \frac{1}{B}\sum_{t\in B}(y_t - Q_\theta(s_t, a_t))^2$ and update the parameters θ via backpropagation.

III. RL-based Graph Partitioning

We formulate graph partitioning as an MDP and leverage RL techniques to learn a good partition policy. Later in the experiments, we observe that such an RL approach can generate balanced partitions with low cut size. Moreover, it can significantly enhance the performance of the FM algorithm when used as an initialization.

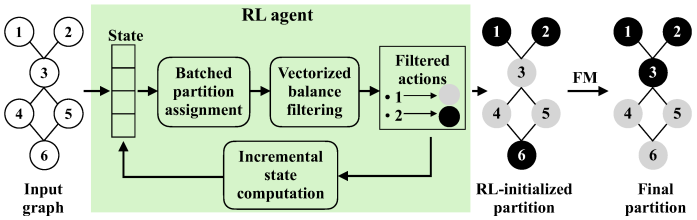

Fig. 2. System overview.

Figure 2 shows an overview of our system. The RL agent begins the partitioning process for the input graph by suggesting partition assignments for batches of vertices, each represented as a state vector. We pass these batched assignments (i.e., actions) through a vectorized balance filtering to enforce partition constraints. We then use the filtered actions to update the state vectors incrementally, recomputing only the affected ones for efficiency. This loop continues until the agent assigns partitions to all vertices, completing one epoch. We repeat this process across multiple epochs to progressively improve the partition quality. Afterward, we refine the RL-initialized partition using the FM algorithm to get a final partition.

A. RL Formulation for Graph Partitioning

We provide an RL formulation for the graph partitioning problem by modeling it as an MDP. Consequently, one can train a policy by following the RL approach described in Section II-B. Consider the case of two partitions. At timestep t, let v_t denote the vertex to be assigned to a partition. Let $N(v_t)$ be the set of neighbors of v_t, and $P(v_t)$ denote the partition to which v_t is currently assigned. We define the MDP as follows:

- **State:** The state s_t associated with vertex v_t is a vector with dimension $2k$ ($k = 2$ partitions in our work): the first k elements represent the number of neighbors v_t has in each partition $j \in \{0, ..., k-1\}$. The other k elements represent the total number of vertices of the current partitions. This state encodes local information about the partition assignments of v_t's neighbors and global information about the balance among the total size of different partitions. For instance, consider the vertex 1 in Figure 3. From a local perspective, it has no neighbors in partition 0 (gray) and one neighbor in partition 1 (black). From a global perspective, partition 0 contains a total of four vertices, while partition 1 contains two. Hence, the state of vertex 1 can be represented by the tuple $\{0, 1, 4, 2\}$, which encodes its local and global partition context.

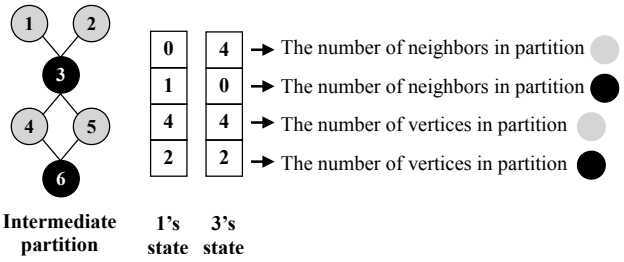

Fig. 3. Illustration of our RL state representation for two partitions before normalization. Partition 0 is shown in gray, and partition 1 is shown in black.

We divide the first k elements of the state by the maximum degree of the graph and divide the other k elements by the total number of vertices $|V|$ to normalize numerical values.

- **Action:** The agent can assign each vertex to one of the k partitions based on the state vector.
- **State transition:** After the agent takes an action and assigns vertex v_t to a new partition, the state vectors of its neighbors will change accordingly. This new state vector is calculated every time we query a new vertex.
- **Reward:** We design the following reward to encourage actions that reduce cut size and balance partitions,

$$r_t = \Delta_{cut} + \alpha \cdot \Delta_{balance}, \tag{3}$$

where $\alpha > 0$ is a hyperparameter, and Δ_{cut}, $\Delta_{balance}$ respectively denote the changes in cut size and partition balance resulting from the action. Here, partition balance is defined as $1 - \max(|p_0|, ..., |p_{k-1}|)\frac{k}{|V|}$, where $\max(|p_0|, ..., |p_{k-1}|)$ denotes the cardinality of the largest partition, $|V|$ the number of vertices in the graph and k the number of partitions. Intuitively, partition balance becomes zero when the partitions are perfectly balanced.

B. Improving RL Inference Throughput

During the testing phase, we apply the trained RL policy to partitioning unseen graphs by processing batches of vertices. Specifically, we sample a batch of vertices and compute their corresponding states, which are then fed into the policy network to generate a batch of predicted actions, i.e., partition assignments, for each vertex in the batch. To ensure that these actions lead to balanced partitions, we introduce a balance filter that sequentially checks whether each action satisfies the balance constraint. This approach, though simple, introduces two major challenges. The first is the balance filter overhead. The balance filter sequentially evaluates each action's impact and discards those that violate the balance constraint. However, this method incurs significant computation time overhead during inference due to the cost of checking each action sequentially. The second is the state computation cost. For each batch of vertices, the corresponding state vectors need to be computed from scratch for every vertex. Performing these computations sequentially in a loop introduces significant computation overhead.

These two challenges significantly degrade runtime performance, particularly on large graphs. To address them, we introduce two optimization strategies: *Vectorized Balance Filtering* (VBF) and *Incremental State Computation* (ISC).

1) Vectorized Balance Filtering: We propose VBF (see Algorithm 1) to efficiently process a batch of actions while enforcing partition balance. To elaborate, let $A \in \{0,1\}^B$ denote the array of model-generated actions for a batch of B vertices. VBF operates as follows:

- Step 1: Based on the batch of actions A, we identify the sets of vertices transitioning from partition 0 to 1 and from 1 to 0, denoted as $T_{0\to1}$ and $T_{1\to0}$, respectively. This step is performed efficiently using vectorized operations.
- Step 2: Next, for partition 1, we compute the net increase in the size as $\Delta = |T_{0\to1}| - |T_{1\to0}|$. If this increase would violate the balance constraint, we flip the actions of $\lceil \frac{\Delta - (L_{\max} - |p_1|)}{2} \rceil$ number of vertices in $T_{0\to1}$ back to 0, keeping them in partition 0. Here, $L_{\max} = (1+\delta)\frac{|V|}{2}$ denotes the maximum allowed size for each partition. The same process is also applied to partition 0.
- Step 3: After applying the necessary action flips, the filtered action array A is used to update the partition assignments, and the partition size counters are updated accordingly.

Algorithm 1 Vectorized Balance Filtering

```
Compute T_{0→1}, T_{1→0}, and Δ = |T_{0→1}| − |T_{1→0}|
if Δ > L_max − |p_1| then
    Flip actions of ⌈(Δ − (L_max − |p_1|))/2⌉ vertices in T_{0→1}
else if −Δ > L_max − |p_0| then
    Flip actions of ⌈(−Δ − (L_max − |p_0|))/2⌉ vertices in T_{1→0}
end if
Update vertex partition assignment with filtered actions
```

To illustrate how VBF works, consider a batch of actions generated by the RL agent. Suppose these actions would move four vertices from partition 0 to 1 and one vertex from 1 to 0 (i.e., $|T_{0\to1}| = 4, |T_{1\to0}| = 1$), while the rest of the vertices remain unchanged, resulting in a net increase of $\Delta = 4 - 1 = 3$ vertices in partition 1. If this violates the balance constraint, VBF restores balance by flipping the actions of a minimal number of vertices in $T_{0\to1}$, as determined by Algorithm 1. In summary, VBF leverages vectorized operations to improve runtime performance compared to the sequential filtering approach.

2) Incremental State Computation: After applying the filtered actions, the state information must be updated. Instead of recalculating the full state vector for each vertex sequentially, we adopt a vectorized strategy that maintains a sparse state matrix for all vertices and incrementally updates their states. The state vector consists of four elements: the first two encode local neighborhood information, while the last two capture global partition balance. Since applying actions only affects the neighbors of the vertices that change partitions, we update the first two elements only for the neighbors of those transitioning vertices. In contrast, the last two elements, representing global partition balance, are updated for all vertices simultaneously. Taking Figure 2 as an example, suppose the filtered actions change the partitions of vertices 1 and 2. Since vertex 3 is a neighbor of both, we update the first two elements of its state vector, representing local neighborhood information. In contrast, the corresponding elements in the state vectors of non-neighboring vertices 4, 5, and 6 remain unchanged. Additionally, we update the last two elements of each state vector, which encode the global partition balance, for all vertices. This selective update mechanism exemplifies how ISC efficiently avoids redundant computations while maintaining accurate state representations.

Specifically, given an array A of filtered actions applied to the corresponding batch of B vertices, we first retrieve the previously computed $T_{0\to1}$ and $T_{1\to0}$. Let $N_{0\to1}$ and $N_{1\to0}$ denote the sets of neighbors of the vertices in $T_{0\to1}$ and $T_{1\to0}$, respectively. These neighbor sets are retrieved in a vectorized fashion using a sparse adjacency matrix. We then use a *bincount* [**?**] operation to count how many times each vertex appears in $N_{0\to1}$ and $N_{1\to0}$. For example, if vertex v_i appears $l_{0\to1}$ and $l_{1\to0}$ times in $N_{0\to1}$ and $N_{1\to0}$, respectively, then this indicates that v_i has $l_{0\to1}$ neighbors moving from partition 0 to partition 1 and $l_{1\to0}$ neighbors moving from partition 1 to partition 0. The first two elements of v_i's state vector s_i should be updated as follows:

$$s_i[0] += \frac{l_{1\to0} - l_{0\to1}}{max\_degree} \quad \text{and} \quad s_i[1] += \frac{l_{0\to1} - l_{1\to0}}{max\_degree}.$$

We divide by the maximum degree of the graph to normalize numerical values. In addition to that, we update the last two elements of all the states for all the vertices as follows:

$$s_i[2] = \frac{|p_0|}{|V|} \quad \text{and} \quad s_i[3] = \frac{|p_1|}{|V|},$$

where $|p_0|$ and $|p_1|$ are the updated partition loads after applying the filtered actions. We divide by the total number of vertices in the graph to normalize numerical values. These updates are done using vectorized operations on the sparse state matrix that contains the state vectors for all the vertices. This significantly improves the runtime performance by removing a large amount of redundant computation compared to the sequential state calculation approach.

IV. Experimental Results

We evaluated the cut size and runtime performance on industrial circuit graphs generated by OpenTimer [2]. These circuit graphs are significantly larger and more challenging to partition than the typical benchmarks used in [6], [7], and have been increasingly adopted by recent graph partitioning research [8], [9], [15]. We trained the RL model using PyTorch on an NVIDIA RTX 4090 GPU, and compiled programs using `g++` 11.4 with `-std=c++20` and `-O3` enabled to partition graphs. We ran all the experiments on a Ubuntu 22.04.3 machine with 20 Intel i7-11700 CPUs at 2.50 GHz and 125 GB RAM. We set the imbalance ratio to $\delta = 3\%$ in the experiment, as this is a commonly used setting in many existing works [8], [16].

A. Baselines

We chose the classical FM algorithm [10] as our 2-way graph partitioner, as it is widely adopted in many existing partitioning frameworks [6]–[9], [15]. FM operates in multiple *passes*, where each pass scans the entire vertex set to identify the best subsequence of vertex moves between partitions. Specifically, FM iteratively selects the vertex whose move yields the highest improvement in cut size (i.e., the highest gain), moves it, and then locks it to prevent repeated moves within the same pass. This process continues until all vertices are locked. Once complete, FM identifies and commits to the subsequence of consecutive moves that yields the maximum total gain.

We chose two popular initialization strategies as the baselines. The first one is random initialization, which partitions the vertex set randomly into two halves. The second one is a BFS-based strategy where, at each iteration, the vertex with the highest connectivity to the current frontier is selected. This process is repeated until all vertices have been assigned to a partition. Due to the simplicity and fast runtime, BFS-based initialization has been widely adopted by many existing partitioners [6], [8], [15].

B. Training and Testing the RL Agent

1) Training phase: To train our RL-based graph partitioner, we implemented the Deep Double Q-learning algorithm [14]. The training graphs are non-overlapping with the testing graphs listed in Table I.

We used the following hyperparameters: training batch size $B = 128$, reward discount factor $\gamma = 0.2$, reward weight $\alpha = 0.7$, experience replay memory size $N = 10$k, target network synchronization period $K = 10$k steps. For the ϵ-greedy strategy used when collecting the MDP data, we initialized $\epsilon = 1.0$ and multiplied it by a factor of $\epsilon\_decay = 0.9999997$ with a lower bound set for ϵ as $\epsilon\_min = 0.01$. We trained for 200 epochs, where each training epoch corresponds to traversing the training graph once. We designed a deep Q-network with an input linear layer, followed by two residual blocks inspired by the ResNet architecture [**?**] and an output linear layer. The Q-network takes as input a state vector of dimension = 4 and outputs a vector of two Q-values corresponding to the two possible actions. Figure 4 illustrates the network architecture. We used the standard Adam optimizer with learning rate $\eta = 0.001$ to update the Q-network parameters θ via backpropagation.

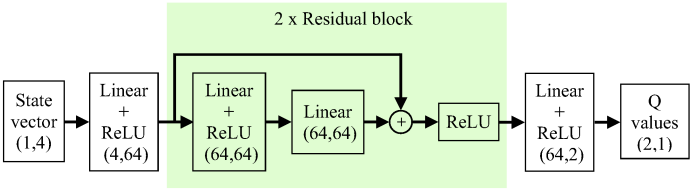

Fig. 4. Illustration of the Q-network architecture. The network takes the state vector as input, propagates the state through a linear layer, two residual blocks, and a linear layer, and outputs a vector of two Q-values corresponding to the two possible actions. The chosen action is the one corresponding to the highest Q-value.

Figure 5 plots the training loss and the training cut size achieved by the RL policy during the training process. From the left figure, it can be seen that the training loss decays quickly, indicating that the learned policy performs well on the training data. The right figure illustrates the progressive reduction in cut size on the training graph throughout the training process. This figure shows that the policy effectively learns to reduce the cut sizes over time.

2) Testing phase: After training the RL agent, we evaluated its performance on unseen test graphs. Given a test graph, we feed batches of vertex states to the trained RL agent and apply the VBF-filtered actions to the corresponding vertices. This process continues until all the vertices in the graph have been traversed, which constitutes one test epoch. We then repeated this for 100 test epochs to obtain a high-quality result.

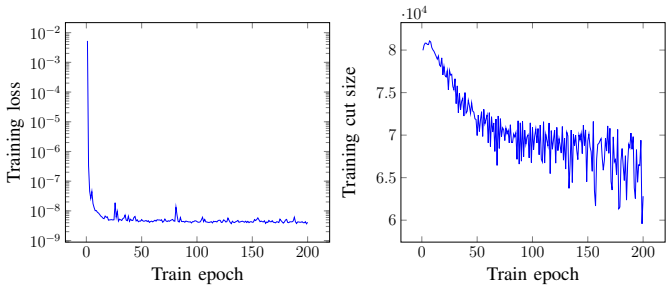

Fig. 5. Left: Decreasing training loss with more train epochs. Right: Decreasing cut size during the training phase. Every train epoch corresponds to traversing all the vertices in the training graph once.

C. Overall Performance Comparison

We compare the cut size and runtime performance of the two baselines (random+FM and BFS+FM) and our proposed approach (RL+FM) across different graphs. We also report the cut size (C_{RL}) obtained by the RL agent (without applying FM). Table I summarizes the results. We report the results corresponding to a test batch size of 1–3% of the graph size as this range yields high-quality results. As shown in Table I, our method outperforms both baselines on most benchmarks in terms of final cut size due to our high-quality initial partitions generated by the RL agent. For example, when partitioning the tv80 benchmark, we achieve a cut size of 169 (starting from an initial cut size of 90,934), significantly lower than the baseline 1's 83,381 (from 763,712) and baseline 2's 78,105 (from 763,603). We attribute the improved cut size to our RL state design, which incentivizes the agent to learn from both global and local graph contexts and optimize long-term rewards. This design enables more informed partitioning decisions, yielding higher-quality initial partitions that guide FM toward superior final cut sizes.

TABLE I

GRAPH STATISTICS, TEST BATCH SIZES, INITIAL AND FINAL CUT SIZES, AND RUNTIME BETWEEN THE BASELINES AND OURS. BASELINE 1 REFERS TO FM USING THE RANDOM INITIALIZATION AND BASELINE 2 USING BFS-BASED INITIALIZATION. C_{RL} DENOTES THE CUT SIZE OBTAINED BY OUR RL AGENT. ALL RUNTIMES (IN SECONDS) INCLUDE BOTH THE INITIALIZATION AND THE FM REFINEMENT. WE RAN 100 TEST EPOCHS FOR OUR RL-BASED INITIALIZATION.

Graph	$\|V\|$	$\|E\|$	Baseline 1 (Random+FM)			Baseline 2 (BFS+FM)			Ours (RL+FM)			
			Initial cut size	Final cut size	Time	Initial cut size	Final cut size	Time	Batch	C_{RL}	Final cut size	Time
c2670	1,365	1,665	846	109	0.033	812	106	**0.019**	32	124	**71**	7.53
c7552	3,802	4,791	2,392	288	0.08	2,363	257	**0.065**	64	334	**230**	10.18
ac97_ctrl	42,438	53,586	27,647	**2,833**	9.49	27,114	3,132	**5.91**	1,024	3,729	3,102	28.55
aes_core	133,502	172,892	88,590	4,645	31.29	86,496	4,265	**21.84**	4,096	11,479	**72**	32.82
des_perf	607,380	774,582	399,048	22,315	471.22	396,658	22,227	627.74	16,384	39,477	**2,203**	**343.99**
tv80	1,090,432	1,477,568	763,712	83,381	406.16	763,603	78,105	629.39	16,384	90,934	**169**	**183.93**
vga_lcd	1,591,236	1,995,464	1,024,728	132,676	1,427.97	1,023,236	134,779	**1,088.26**	16,384	145,265	**73,519**	1,698.24
ac97_ctrl_2	2,037,024	2,572,128	1,327,056	128,688	1,889.18	1,326,859	129,856	2,176.37	32,768	80,009	**4,923**	**1,058.08**
usb_phy	2,505,728	3,070,976	1,615,872	131,783	804.87	1,615,808	130,312	846.86	32,768	235,640	**1,203**	**571.03**

Regarding runtime performance, we observe that the runtime of our method becomes increasingly competitive with, and eventually faster than, the baselines as the graph size (and test batch size) increases. The runtime reported for RL+FM includes the time required for 100 test epochs of the RL agent, as well as the time taken by the FM algorithm initialized from the RL-generated partition. For instance, on the aes_core benchmark, our method took 32.82 seconds compared to the BFS-based baseline 2's 21.84 seconds. Notably, when partitioning the larger des_perf benchmark, our method's runtime was 343.99 seconds, significantly faster than the baseline 2's 627.74 seconds. We attribute this trend to two factors. First, larger graphs tend to require more FM passes to converge, making them more sensitive to the quality of the initial partition. In this scenario, RL-initialized partitions provide a high-quality starting point, which significantly accelerates convergence. Second, our optimizations (VBF and ISC) scale efficiently, offering greater runtime benefits on larger graphs.

D. Impact of Test Batch Sizes on C_{RL}

In the testing phase, we evaluate the impact of varying test batch sizes on the cut size C_{RL} obtained by our RL agent (without applying FM). As shown in Figure 6, increasing the batch size generally improves C_{RL} up to a certain threshold. Beyond this point, however, larger batch sizes lead to a degradation in cut size performance. Initially, increasing the test batch size improves C_{RL} because it reduces variance in the agent's decisions, leading to more stable and consistent partitioning. Larger batches also provide richer global context per decision as each update step aggregates information from more data points or graph parts simultaneously, allowing the agent to better estimate gradients and recognize structural patterns in the graph that help minimize cut size. For instance, for the ac97_ctrl_2 benchmark, the test batch size 32,768 gives the best C_{RL}. However, beyond a certain test batch size threshold, performance degrades, which we attribute to two reasons: First, as more assignments are proposed, the vectorized balance filtering becomes less precise, potentially flipping high-quality actions to enforce partition balance. Second, with large batches, the state vectors are not updated between individual actions, but only after the whole batch is processed. As the test batch size becomes too large, this leads to reduced freshness of state information when making decisions, which degrades the accuracy of the agent's actions. These dynamics create a trade-off, indicating the existence of an optimal test batch size that balances decision stability and responsiveness. Empirically, this optimal size is found to be around 1–3% of the total graph size.

E. Impact of Test Epochs on C_{RL}

Given that our RL agent provides the initial partition for the FM algorithm in our RL+FM method, it is crucial to analyze how the quality of these initial solutions (C_{RL}) evolves over test epochs. Figure 7 presents the improvement of C_{RL} during the testing process. The steady decrease in C_{RL} as the number of test epochs increases indicates the RL agent in providing progressively better starting points for FM refinement. The most significant improvement occurs within the initial 20 test epochs. Following this rapid improvement, the rate of reduction slows, and C_{RL} starts to plateau around test epoch 40, suggesting a convergence in the quality of the RL initialization. We attribute the observed trend to the RL agent's ability to generalize its learned partitioning policy to unseen graphs. This improvement in C_{RL} over test epochs shows the effectiveness of the learned policy, driven by the reward design that explicitly encourages cut size reduction while penalizing partition imbalance.

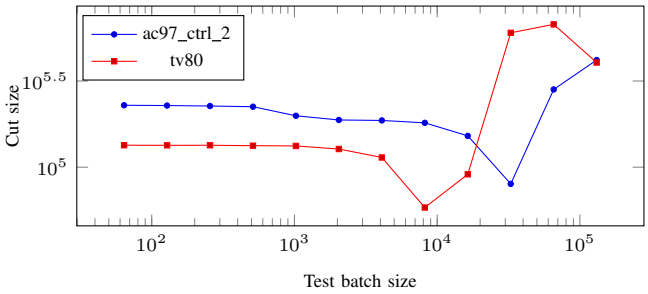

Fig. 6. Effect of varying the RL test batch size on C_{RL}. Test batch sizes between 1–3% of the graph size yield the best C_{RL}.

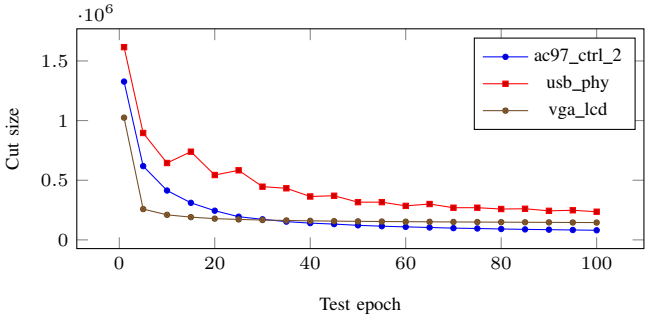

Fig. 7. The improvement of cut sizes obtained by our RL agent (C_{RL}) over test epochs on three benchmarks.

F. Effect of Optimization Strategies on Runtime

To gain a detailed insight of the runtime improvements contributed by our optimization strategies (VBF and ISC) within the RL agent

TABLE II
RUNTIME BREAKDOWN (IN SECONDS) WITH (W/) AND WITHOUT (W/O) THE OPTIMIZATION STRATEGIES FOR INFERENCE.

Graph	State computation		Balance filtering	
	w/	w/o	w/	w/o
c2670	0.18	1.58	6.5	52.71
c7552	0.18	6.28	8.25	119.25
ac97_ctrl	0.28	61.74	19.33	869.62
aes_core	0.06	201.85	23.16	3,980.28
des_perf	0.38	914.11	68.06	17,738.48
tv80	0.37	1,619.4	132.21	33,140.9
vga_lcd	1	2,413.4	203.2	47,578.8
ac97_ctrl_2	0.76	2,990.5	246.24	60,989.2
usb_phy	0.97	3,611.6	270.3	75,844.6

of our RL+FM method, we present a runtime breakdown during the inference phase in Table II. The runtime mainly comprises two components: *state computation time* and *balance filtering time* (which ensures partition size balance on model-generated actions). Table II reveals that ISC and VBF yield significant reductions in both state computation and balance filtering times. Specifically, the state computation time decreases because ISC efficiently updates the state representation only for the affected portions when a vertex changes partition, avoiding a full state recalculation. The balance filtering time is significantly improved because VBF performs the partition size balance checks using vectorized operations. For instance, on the usb_phy benchmark, state computation time decreases from 3,611.6 to 0.97 seconds (3723× speedup), and balance filtering time improves from 75,844.6 to 270.3 seconds (280× speedup). These significant reductions demonstrate the effectiveness of VBF and ISC in accelerating the RL agent's runtime.

G. Comparison of FM Passes

Figure 8 compares the number of passes required by FM to converge between the baselines and our method. Here, the number of passes, representing the iterations FM takes to converge to a stable solution, directly indicates the efficiency of its refinement process, where fewer passes signify faster convergence. Across all benchmarks, our method consistently completes FM refinement in fewer passes than the baselines. For instance, on the usb_phy benchmark, our method requires only 64 passes, compared to 150 for both baselines, which reach the termination condition. This improvement is attributed to the higher quality of our initial partitions, which guide FM toward better solutions more efficiently. These results align with the runtime advantage reported in Table I, further demonstrating the effectiveness of our learning-based initialization.

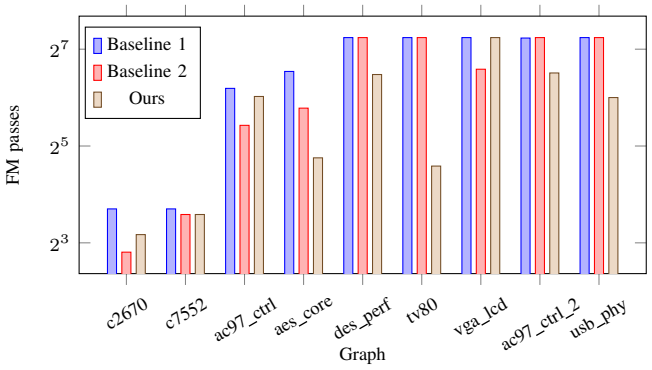

Fig. 8. The number of FM passes between the baselines and ours, with an upper limit of 150 passes to avoid excessive runtime.

TABLE III
DETAILED RUNTIME BREAKDOWN (IN SECONDS) FOR THE BASELINES AND OURS, SHOWING TIMES OF THE INITIALIZATION (INIT.) AND FM. THE INITIALIZATION TIME INCLUDES BOTH PARTITIONING AND THE SETUP OF THE CORRESPONDING DATA STRUCTURES (E.G., BUCKETLIST) FOR FM TO START WITH.

Graph	Random		BFS		Ours	
	Init.	FM	Init.	FM	Init.	FM
c2670	0.02	0.013	0.01	0.009	7.52	0.01
c7552	0.01	0.07	0.01	0.055	10.1	0.08
ac97_ctrl	0.09	9.4	0.09	4.82	20.27	1.28
aes_core	0.29	31	0.29	21.55	23.89	8.93
des_perf	2.22	469	2.24	625.5	70.99	273
tv80	2.16	404	2.19	627.2	132.23	51.7
vga_lcd	4.97	1,423	4.96	1,083.3	207.24	1,491
ac97_ctrl_2	4.18	1,885	4.17	2,172.2	249.58	808.5
usb_phy	4.87	800	4.86	842	274.03	297

H. Detailed Runtime Breakdown

Table III shows the detailed runtime breakdown. Apparently, RL-based initialization incurs higher computational cost than both baselines. Taking c2670 for example, RL-based initialization takes 7.52 seconds to finish, while random- and BFS-based initializations both finish within 1 second. This high cost for RL arises because it processes each vertex through a trained neural network (involving multiple matrix operations and nonlinear transformations) to determine the partition assignment. In addition, the Python-based inference introduces non-negligible overhead compared to C++-based baselines. While RL-based initialization incurs higher computational cost, it significantly accelerates FM convergence and improves final cut sizes due to higher-quality initial partitions. For example, on usb_phy, FM converges in 800, 842, and 297 seconds with random-, BFS-, and RL-based initializations, respectively, while RL achieves over 100× better final cut size (see Table I) compared to both baselines. In this case, the end-to-end runtime of RL (571.03 seconds) is also faster than baselines (804.87 and 846.86 seconds).

V. CONCLUSION

We have proposed a RL-based initialization to generate high-quality initial partitions for the FM algorithm. We have designed a tailored reward function that balances cut size reduction with partition balance. We have introduced a state representation that captures both local and global graph structure. We have developed two scalable RL inference throughput optimizations, VBF and ISC, to improve the runtime efficiency. Our experiments on a set of industrial circuit graphs demonstrate that applying FM to our RL-generated initial partitions results in significantly better cut sizes compared to both random and BFS-based initializations. Our work highlights the potential of data-driven, machine learning methods for tackling the fundamental graph partitioning problem. As future work, we plan to extend our approach to the more general k-way partitioning setting, using recursive bisection as a scalable strategy. We plan to also apply the similar method to optimize task graph parallelism, inspired by our prior research [2]–[4], [8], [9], [12], [13], [15], [17]–[82].

ACKNOWLEDGMENT

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

REFERENCES

[1] T.-W. Huang and M. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 895–902.

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

[3] 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, pp. 1–12.

[4] 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, pp. 1–6.

[5] S. Lin, J. Liu, and M. D. Wong, "Gamer: Gpu accelerated maze routing," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–8.

[6] G. Karypis and V. Kumar, *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, University of Minnesota, 1997, technical Report 97-061.

[7] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag, "Scalable shared-memory hypergraph partitioning," in *23rd Workshop on Algorithm Engineering and Experiments (ALENEX 2021)*. SIAM, 2021, pp. 16–30.

[8] 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, pp. 1–6.

[9] 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.

[10] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference (DAC)*. IEEE Press, 1982, pp. 175–181.

[11] Ü. V. Çatalyürek and C. Aykanat, "Patoh: A multilevel hypergraph partitioning tool, version 3.0," in *Proceedings of the 8th International Symposium on Solving Irregularly Structured Problems in Parallel (Irregular)*. Springer, 1999, pp. 353–354.

[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, pp. 89–95.

[13] 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.

[14] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2016, pp. 2094–2100.

[15] 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.

[16] G. Karypis and V. Kumar, *hMetis: A Hypergraph Partitioning Package*, 1999, university of Minnesota, Technical Report. [Online]. Available: https://www-users.cs.umn.edu/~karypis/metis/hmetis/

[17] C.-H. Chiu, C. Morchdi, C.-C. Chang, C. Yu, Y. Zhou, and T.-W. Huang, "Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation," in *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, 2025.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] C.-C. Chang, B. Zhang, and T.-W. Huang, "GSAP: A GPU-Accelerated Stochastic Graph Partitioner," in *ACM International Conference on Parallel Processing (ICPP)*, 2024, pp. 565–575.

[26] 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, pp. 1–9.

[27] S. Jiang, R. Fu, L. Burgholzer, R. Wille, T.-Y. Ho, and T.-W. Huang, "FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array," in *ACM International Conference on Parallel Processing (ICPP)*, 2024, pp. 388–399.

[28] 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, pp. 789–793.

[29] 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, pp. 771–774.

[30] 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, pp. 766–770.

[31] 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, pp. 151–166.

[32] C. Chang, T.-W. Huang, D.-L. Lin, G. Guo, and S. Lin, "Ink: Efficient Incremental k-Critical Path Generation," in *ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–6.

[33] 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 Design Automation Conference (DAC)*, 2024, pp. 1–6.

[34] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis," in *ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–6.

[35] 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, pp. 51–59.

[36] 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.

[37] 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, pp. 1–8.

[38] C.-C. Chang and T.-W. Huang, "uSAP: An Ultra-Fast Stochastic Graph Partitioner," in *IEEE High-performance and Extreme Computing Conference (HPEC)*, 2023, pp. 1–7.

[39] 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.

[40] ——, "SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU," in *ACM International Conference on Parallel Processing (ICPP)*, 2023, pp. 51–61.

[41] T.-W. Huang, "qTask: Task-parallel Quantum Circuit Simulation with Incrementality," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 746–756.

[42] 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.

[43] 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.

[44] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. Wong, "A GPU-Accelerated Framework for Path-Based Timing Analysis," in *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023, pp. 4219–4232.

[45] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," in *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 2023, pp. 4973–4984.

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

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

[48] 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, pp. 283–284.

[49] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms," in *ACM/IEEE Design Automation Conference (DAC)*, 2022, pp. 1388–1389.

[50] 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, pp. 588–597.

[51] 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, pp. 190–195.

[52] D.-L. Lin and T.-W. Huang, "Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022, pp. 3041–3052.

[53] 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.

[54] 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, pp. 1–6.

[55] 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.

[56] 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, pp. 1–9.

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

[58] 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, pp. 468–479.

[59] 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, pp. 435–450.

[60] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Path-based Timing Analysis," in *IEEE/ACM Design Automation Conference (DAC)*, 2021, pp. 721–726.

[61] 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, pp. 3466–3478.

[62] 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, pp. 278–283.

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

[64] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2022, pp. 1303–1320.

[65] 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.

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

[67] 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, pp. 64–71.

[68] 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.

[69] 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, pp. 1–2.

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

[71] 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, pp. 1–6.

[72] C.-X. Lin, T.-W. Huang, G. Guo, and M. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM Multimedia Conference (MM)*, 2019, pp. 2284–2287.

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

[74] 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.

[75] T.-W. Huang, C.-X. Lin, , and M. Wong, "Distributed Timing Analysis at Scale," in *ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–2.

[76] 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, pp. 1–4.

[77] T.-W. Huang, C.-X. Lin, and M. Wong, "DtCraft: A High-performance Distributed Execution Engine at Scale," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019, pp. 1070–1083.

[78] 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, pp. 1360–1363.

[79] 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, pp. 183–188.

[80] 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, pp. 757–764.

[81] 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, pp. 1–6.

[82] 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, pp. 1–6.