RL-MUL: Multiplier Design Optimization with Deep Reinforcement Learning

Dongsheng Zuo
HKUST(GZ)
dzuo721@connect.hkust-gz.edu.cn

Yikang Ouyang
HKUST(GZ)
youyang929@connect.hkust-gz.edu.cn

Yuzhe Ma
HKUST(GZ)
yuzhema@ust.hk

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247941

***Abstract*—Multiplication is a fundamental operation in many applications, and multipliers are widely adopted in various circuits. However, optimizing multipliers is challenging and non-trivial due to the huge design space. In this paper, we propose RL-MUL, a multiplier design optimization framework based on reinforcement learning. Specifically, we utilize matrix and tensor representations for the compressor tree of a multiplier, based on which the convolutional neural networks can be seamlessly incorporated as the agent network. The agent can learn to adjust the multiplier structure based on a Pareto-driven reward which is customized to accommodate the trade-off between area and delay. Experiments are conducted on different bit widths of multipliers. The results demonstrate that the multipliers produced by RL-MUL dominate all baseline designs in terms of both area and delay. The performance gain of RL-MUL is further validated by comparing the area and delay of processing element arrays using multipliers from RL-MUL and baseline approaches.**

I. Introduction

Datapath designs are fundamental building blocks of digital integrated circuits, which have become increasingly critical in modern computing platforms. Particularly, the multiply-accumulate (MAC) operations are heavily used in many applications, including digital signal processing, deep neural networks (DNNs), and image processing. Especially, MAC operations may account for more than 99% of all operations in conventional DNNs, as shown in Fig. 1. On the hardware level, these operations are executed by datapath circuits, among which multipliers contribute a substantial portion of overhead in terms of performance, power, area, and design costs.

Generally, these designs can be completed manually by refining from regular structures, which effectively optimizes area, power, and performance for a particular technology node and application scenario [1]–[4]. However, it is not so flexible considering the required engineering effort. Algorithmic approaches can generate circuit structures based on particular strategies, which mainly contain two categories, namely mathematical programming and heuristic search. Integer linear programming (ILP) has been widely investigated for datapath circuits optimization [5]–[7]. GOMIL [7] proposed an ILP formulation for global multiplier optimization, and the optimization of multipliers on FPGA designs is explored in [6]. In addition, ILP has also been applied for adder trees based on analytical area, power, and timing models [5]. Heuristic search is another approach that attempts to obtain desired circuit structures effectively based on a well-defined representation of the structure [8], [9], where different pruning techniques are introduced to avoid exhaustive searching. Recently design space exploration methodologies have become promising solutions for circuit design and optimization, where various machine learning models are leveraged, including graph learning and Bayesian optimization [10], [11]. All these methods rely on proxy estimation or surrogate models to evaluate a design during the search or optimization.

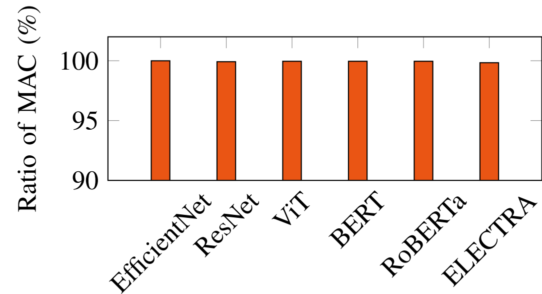

Fig. 1 Ratios of MAC computations in various neural networks.

In contrast to the aforementioned methods, reinforcement learning (RL) can make use of the actual evaluation within the optimization loop. In this paper, we propose an RL-based framework for multiplier design optimization, specifically for compressor tree structures. Recently, RL has been applied to solve different problems [12]–[14] in electronic design automation (EDA). Regarding the circuit structure optimization, PrefixRL [12] proposed to apply deep Q-learning to adder structure exploration. GCN-RL [13] applied the actor-critic algorithm to solve the transistor sizing problem. In this work, we made the first attempt toward multiplier optimization based on reinforcement learning. We propose a matrix and a tensor representation for the compressor tree structures, based on which an RL agent is trained to make adjustments to the structures. The agent can learn to make good moves based on a Pareto-driven reward which is customized to accommodate the trade-off between area and delay. To validate the effectiveness, RL-MUL is applied to design and optimize different bit widths of multipliers, which can outperform various baseline designs from different methods, including legacy designs, evolutionary algorithms, and integer linear programming. The main contributions of this work are as follows:

- We propose RL-MUL, a multiplier optimization framework based on reinforcement learning. To the best of our knowledge, it is the first work to leverage RL for multiplier optimization.
- We present a matrix and a tensor representation for multipliers, based on which conventional deep neural networks

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on December 20,2024 at 07:35:52 UTC from IEEE Xplore. Restrictions apply.

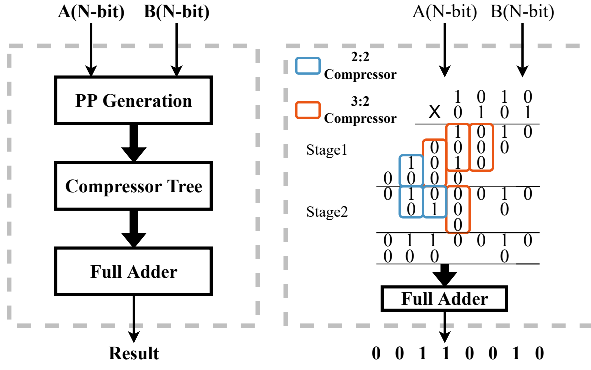

Fig. 2 Multiplier architecture

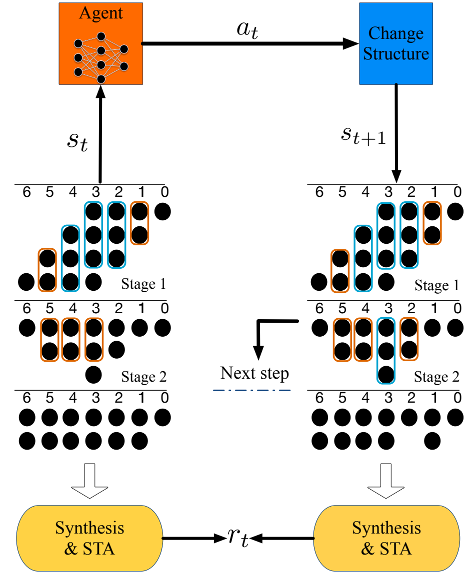

Fig. 3 RL-based multiplier optimization framework

can be seamlessly incorporated as the agent network.

- A Pareto-driven reward is utilized to accommodate the trade-off between the area and delay so that the agent can learn to achieve Pareto-optimal designs.
- Experimental results demonstrate that the multipliers produced by RL-MUL dominate all baseline designs in terms of both area and delay. The performance gain of RL-MUL is further validated by comparing the area and delay of processing element array designs using different multipliers.

II. Preliminary

A. Multiplier Architecture

The multiplier is usually implemented as three main parts: a partial product generator (PPG), a compressor tree (CT), and a carry propagation adder, as shown in Fig. 2. PPG produces partial products (PPs) from the multiplicand and multiplier. The CT is used to compress the PPs to two rows in parallel. After the compression process, an adder is applied to sum up the two rows of PPs to get the final product.

A typical partial product generator is based on AND gate, which uses N^2 AND gates for N bit multiplier to generate parallelogram-shaped PPs. A CT has multiple compression stages to compress the PPs into two rows. The most commonly used compressors include 3:2 compressors and 2:2 compressors, which are implemented using a full adder and a half adder, respectively. For a 3:2 (resp. 2:2) compressor applied at stage i and column j, it takes (resp. 2) partial products from stage i column j as input and passes the *sum* output to stage $i+1$ column j and the *carry out* to stage $i+1$ column $j+1$. Therefore, a 3:2 and a 2:2 compressor reduces 2 and 1 partial products of column j, respectively, and increases one partial product of column $j+1$.

B. Q-Learning

RL involves a set of optimization instances named *state* S, and a set of *actions* A per state. The agent transitions from state s to state s' by performing an action $a \in A$, and receives a *reward* $r(s, a)$ from the RL environment as evaluation. The action selection model is called *policy* π, and the RL agent aims to learn a policy that maximizes accumulative reward.

Q-learning is an RL algorithm that learns the scores of each action a corresponding to the given state s, and the score is called Q-value, which is denoted as $Q(s, a)$. From Bellman equation [15], the Q-value is expected as follows:

$$Q\left(s, a\right) = r\left(s, a\right) + \gamma \max_{a'} Q\left(s', a'\right), \tag{1}$$

where s' indicates the next state, and γ is the discount factor. Therefore, the Q-value is updated by:

$$Q\left(s, a\right) = Q\left(s, a\right) + \alpha \left[r\left(s, a\right) + \gamma \max_{a'} Q\left(s', a'\right) - Q\left(s, a\right)\right], \tag{2}$$

where α is the learning rate.

In this paper, we adopt the deep Q-learning method, which employs a deep neural network as the Q-value approximator. We define the state s as the multiplier structure whose representation will be described in Section III-B. An action a modifies the current structure to a new multiplier structure, i.e., the next state. Reward r is defined as the actual improvement in terms of the area and delay of the multiplier.

III. Proposed Method

A. Overview

Fig. 3 shows our RL-MUL optimization framework. In the RL process, an agent continuously interacts with the environment. At the beginning of every episode, the environment will be at an initial state s_0, which is a Wallace tree structure. At state s_t, the RL-MUL agent selects an action from a set of legal actions based on its policy π in the form of a neural network parameterized by θ, which approximates the Q-value of the actions. The action a_t modifies the multiplier structure to construct a new structure, which is the next state s_{t+1}. Then, the reward r_t is obtained from the EDA tools, based on which the agent will update its neural network model.

B. Multiplier Representation

The RL state space $\mathbb{S}$ consists of all N-bit multiplier structures. It is noted that a primary feature of a multiplier is the number of different compressors in each column, which has a direct impact on its performance after synthesis. Therefore, we use the total number of 3:2 and 2:2 compressors in each column to present the multiplier structure, which can be represented by a matrix

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on December 20,2024 at 07:35:52 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Compressor Assignment

Require: $\mathbf{M}$: Matrix representation
Ensure: $\mathcal{T}$: Tensor representation.
1: **for** $j \leftarrow 1$ *to* $2N$ **do**
2: $\quad i \leftarrow 0$
3: $\quad$ **while** column j exists not assigned comp. **do**
4: $\quad\quad$ Assign 3:2 comp. to stage i column j first
5: $\quad\quad$ Update $t_{ij}^{(0)}$ in $\mathbf{T}^{(0)}$
6: $\quad\quad$ **if** Remaining PPs ≥ 2 **then**
7: $\quad\quad\quad$ Assign 2:2 comp. to stage i column j
8: $\quad\quad\quad$ Update $t_{ij}^{(1)}$ in $\mathbf{T}^{(1)}$
9: $\quad\quad$ **end if**
10: $\quad\quad i \leftarrow i + 1$
11: $\quad$ **end while**
12: **end for**
13: $\mathcal{T}_{0,:,:} \leftarrow \mathbf{T}^{(0)}$
14: $\mathcal{T}_{1,:,:} \leftarrow \mathbf{T}^{(1)}$

$\mathbf{M} \in \mathbb{R}^{2N\times 2}$. The first and second rows of the matrix represent the total number of 3:2 and 2:2 compressors used in each column, respectively. For example, a 4-bit multiplier structure and its matrix representation $\mathbf{M}$ are shown in Fig. 4. To build a complete multiplier structure from matrix representation $\mathbf{M}$, we need to assign the compressors to the corresponding stages. However, the mapping from $\mathbf{M}$ to the structures is not unique since different assignments of compressors in multiple stages may have the same overall number in each column. In order to obtain a unique representation, we extend our matrix representation to a more informative tensor representation, as shown in Fig. 4. We denote $\mathcal{T} \in \mathbb{R}^{K\times 2N\times ST}$ as the tensor representation, where K is the total kinds of compressors used and ST is the number of stages. In this work, 3:2 and 2:2 compressors are used, thus $K = 2$. Note that it can be future extended to support more compressor types. Let $\mathbf{T}^{(0)} = \mathcal{T}_{0,:,:} \in \mathbb{R}^{2N\times ST}$ and $\mathbf{T}^{(1)} = \mathcal{T}_{1,:,:} \in \mathbb{R}^{2N\times ST}$ denote the assignment of 3:2 compressors and 2:2 compressors, respectively. $t_{ij}^{(0)}$ and $t_{ij}^{(1)}$ indicate 3:2 and 2:2 compressor numbers at stage i column j, respectively. Given a matrix $\mathbf{M}$ that contains the information of the overall number of compressors in each column, we can construct the tensor representation $\mathcal{T}$ correspondingly based on an assignment scheme of the compressors in different stages. Since the scheme is deterministic and straightforward, the mapping between $\mathbf{M}$ and $\mathcal{T}$ is unique. Thus we can obtain a unique representation for a multiplier structure. The procedure is presented in Algorithm 1. The assignment method is to assign the compressors from the least significant bit (LSB) columns to the most significant bit (MSB) columns. At column j of stage i, we first assign 3:2 compressors as many as possible (Lines 4 and 5). If there still remain two or more PPs at stage i, we further assign the 2:2 compressors (Lines 6 to 8). Then we move to the next stage until all compressors in column j are assigned.

C. Multiplier Modification

An action a refers to the decision from the RL agent to modify the current multiplier structure. There are four actions for each column, including adding a 2:2 compressor, removing a 2:2 compressor, replacing a 3:2 compressor, and replacing a 2:2 compressor. We denote res_j to present the PP number after compression of column j, which should only be 1 or 2. Adding or removing a 3:2 compressor at column j will cause the res_j to be 0 or 3 and, thus this action is not used. Hence, the action space $|\mathcal{A}| = 2N \times 4 = 8N$. Note that not all actions are valid, e.g., we cannot remove or replace a compressor that even does not exist. For example, there is no 2:2 compressor at column 1 in Fig. 4, thus removing a 2:2 compressor from column 1 is invalid. On the other hand, an action a applied on column j is invalid if a leads the PP numbers after compression to 0 or 3 in column j. For example, if a 2:2 compressor is removed in column 4 in Fig. 4, then res_4 will be three, thus the action is invalid.

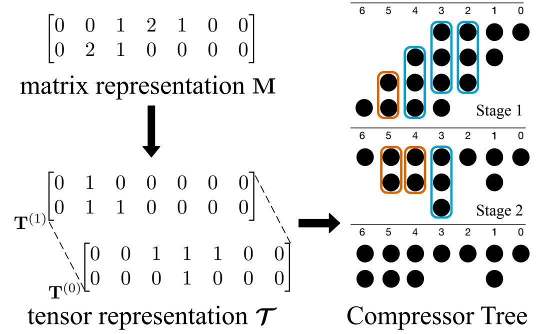

Fig. 4 Structure representation.

For a compressor tree with $2N$ columns, the output of a deep Q-network is a vector that indicates the predicted Q-values:

$$Q(s_t) = [q_{11}, q_{12}, q_{13}, q_{14}, \cdots, q_{2N,1}, q_{2N,2}, q_{2N,3}, q_{2N,4}], \quad (3)$$

where each group of $q_{j1}, q_{j2}, q_{j3}, q_{j4}$ indicates the Q-value of the four actions $a_{j1}, a_{j2}, a_{j3}, a_{j4}$ in column j. To ensure only legal actions can be selected, a mask $\mathbf{m}$ is utilized as the selector to enable valid actions and forbid invalid actions.

$$\mathbf{m} = [m_{10}, m_{11}, m_{12}, m_{13}, \cdots, m_{2N,0}, m_{2N,1}, m_{2N,2}, m_{2N,3}], \quad (4)$$

where each entry is a binary value. If an action a_{ij} is valid, the corresponding entry in m_{ij} is 1. Otherwise, it is 0. In RL-MUL, the final masked Q-value vector is the element-wise multiplication of the mask vector and Q-value vector:

$$Q'(s_t) = Q(s_t) \odot \mathbf{m}. \quad (5)$$

Now the decision is given by

$$a_t = \arg\max_a Q'(s_t, a). \quad (6)$$

Note that only non-zero entries are considered. The action applied to column j changes the number of 3:2 or 2:2 compressors of the current column j, which may cause the number of compressed PPs of subsequent column $j+1$ to become 0 or 3. We use the legalization strategy shown in Algorithm 2 to refine the multiplier structure to ensure the PPs are compressed to 2 rows. It refines from column $j+1$ to the MSB column until all columns have been refined. If there is lack-of-compression we add a 3:2 or replace a 2:2 (Lines 5 to 9), and if there is over-compression, we delete a compressor (Lines 11 to 16). Similar to the assignment procedure, the legalization process is also deterministic. Under state s_t, we can get a new state s_{t+1}

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on December 20,2024 at 07:35:52 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Legalization

Require: Multiplier structure to be legalized; C: action column
Ensure: Legalized multiplier structure
1: **for** $j \leftarrow (C+1)$ *to* $2N$ **do**
2: $res_j \leftarrow$ Get residual PPs after compression
3: **if** $res_j = 1$ or $res_j = 2$ **then**
4: **return** ▷ legalization done
5: **else if** $res_j == 3$ **then**
6: **if** exists 2:2 comp. in column j **then**
7: Replace a 2:2 compressor
8: **else**
9: Add a 3:2 compressor
10: **end if**
11: **else if** $res_j == 0$ **then**
12: **if** exists 2:2 compressor in column j **then**
13: Delete a 2:2 compressor
14: **else**
15: Delete a 3:2 compressor
16: **end if**
17: **end if**
18: **end for**

after performing action a_t to modify the structure along with the legalization.

D. Pareto-driven Reward

In RL-MUL, reward r_t is the improvement on the circuit criteria after applying action a_t at state s_t, including area and delay. Considering the nature of the trade-off between area and delay, a superior multiplier design is always expected to achieve Pareto-optimal in terms of these two dimensions. To do that, we further incorporate a Pareto-driven reward to facilitate that the RL agent can learn to generate Pareto-optimal designs. To train the RL-MUL agent to design Pareto-optimal multipliers, we conduct synthesis flow with multiple design constraints so that the obtained rewards can cover a wide range of scenarios, including area-driven scenario, timing-driven scenario, and area-delay-balance scenario. The total cost is calculated by a weighted sum of area and delay values in different scenarios.

$$cost = \sum_{i=1}^{n} area_i + w \sum_{i=1}^{n} delay_i \tag{7}$$

where $area_i$ and $delay_i$ is the in synthesised metrics with i-th constraint, w is the weight to trade off area and delay. We define our reward r as the difference between s_t and s_{t+1}:

$$r_t = cost_t - cost_{t+1} \tag{8}$$

E. Training Algorithm

We use ResNet-18 [16] as the backbone of Q-Network with the parameters denoted by θ. The state is encoded into the tensor representation $\mathcal{T}$ described in Section III-B and then fed into the Q-network. Algorithm 3 presents the training process of RL-MUL, which is based on the DQN algorithm. Actions a are chosen randomly in warm-up steps (Line 6) and chosen by the policy in future steps (Line 8). At each step t, the agent modifies the multiplier structure s_t to a new structure s_{t+1} and receives reward r_t through the synthesis and timing analysis process.

Algorithm 3 RL-MUL flow

Require: θ_0: Initial Q-network parameters; $\mathbf{M_0}$: initial multiplier structure; γ: discount factor; α: learning rate; T: total training steps; T_B: warm-up steps
Ensure: θ: Q-network parameters
1: Replay buffer $B \leftarrow \{\}$
2: Encode s_0 into $\mathcal{T}$ based on $\mathbf{M_0}$ ▷ Algorithm 1
3: $t \leftarrow 0$
4: **for** $t \leftarrow 0$ to T **do**
5: **if** $t < T_B$ **then**
6: $a_t \leftarrow$ randomly choose from legal actions
7: **else**
8: Get a_t by Equation (6)
9: **end if**
10: Perform a_t to s_t and get s_{t+1}
11: Run EDA tools on s_{t+1} and get r_t ▷ Equation (8)
12: Push (s_t, a_t, r_t, s_{t+1}) to B
13: Sample a batch of transitions from B
14: Update θ by gradient descent ▷ Equations (9) and (10)
15: **end for**

Then a new transition (s_t, a_t, r_t, s_{t+1}) is obtained, and we push it to the replay buffer (Line 12). With the new transition stored, we randomly sample a batch of transitions (s', a', r') from the replay buffer (Line 13). The target Q-value regarding the state-action pair at the sampled step is computed as:

$$y = r' + \gamma \max_{a'} Q(s', a'; \theta), \tag{9}$$

where γ is the discount factor. Based on the expected Q-value y, a gradient of θ can be obtained by:

$$\Delta\theta = \nabla_\theta (y - Q(s, a; \theta))^2. \tag{10}$$

Then the network parameter θ is updated by gradient descent (Line 14).

IV. Experimental Results

A. Setup

RL-MUL is implemented on a Linux machine with a 2.8 GHz AMD EPYC CPU and NVIDIA RTX 3090 GPU. We use EasyMAC [17] for RTL generation. Default adders provided by the synthesis tool are used for the final adder of the generated multiplier. The designs are synthesized by OpenROAD flow [18] with NanGate 45nm Open Cell Library [19]. The number of constraints in Equation (7) is four, and the weight w is 0.25 to trade-off area and delay. OpenSTA [20] is utilized to perform timing analysis. The RL model is implemented with PyTorch. We set γ to 0.8, ϵ to decay from 0.95 to 0.05, and use RMSProp optimizer for the training.

B. Multiplier Results

Since 8-bit and 16-bit multipliers are commonly used, we validate our RL-MUL framework on 8-bit and 16-bit multipliers with both AND-based PPG and modified Booth-encoding (MBE)-based PPG. The baselines include the Wallace tree [1], GOMIL [7] (ILP-based), and the simulated annealing (SA) approach. We train the RL-MUL agent for 3000 steps and run the SA for the same number of steps. We use the open-source C++

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on December 20,2024 at 07:35:52 UTC from IEEE Xplore. Restrictions apply.

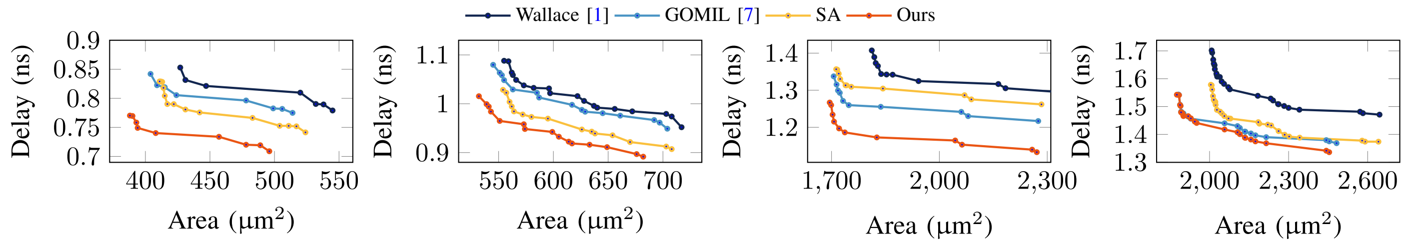

Fig. 5 Pareto-frontiers of the synthesis results on multipliers. From left to right: 8-bit AND-based; 8-bit MBE-based; 16-bit AND-based; 16-bit MBE-based

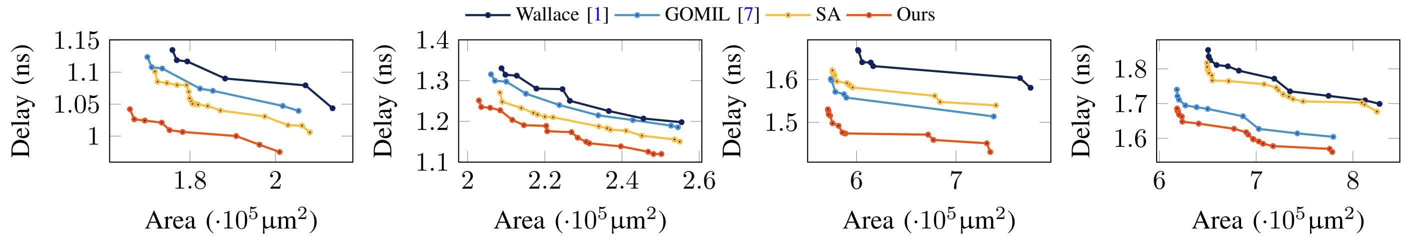

Fig. 6 Pareto-frontiers of the synthesis results on PE arrays. From left to right: 8-bit AND-based; 8-bit MBE-based; 16-bit AND-based; 16-bit MBE-based

code provided by GOMIL [7] to get the multiplier structures, which is also based on NanGate 45nm library.

Synthesizing with different design constraints may generate different netlists for the same RTL design. We synthesize the multipliers obtained from RL-MUL and baseline approaches at target delay ranging from 0.05 ns to 1.2 ns and present the area-delay curve as shown Fig. 5. We can see the multiplier obtained by RL-MUL dominates all baseline designs. TABLE I records the comparison of the minimum area, minimum delay, and area-delay-balance points on the curves. The multipliers obtained from RL-MUL have the least area and delay in all four cases, achieving a maximum reduction of 13.4% on delay under minimum delay constraint and a maximum area saving of 9.1% under minimum area constraint.

Moreover, we utilize the hypervolume [21] to evaluate the quality of the Pareto-frontiers, which refers to the volume fenced by the Pareto-frontier and a reference point in the objective space, as shown in Fig. 7. The hypervolume comparison is shown in Fig. 8(a). Multipliers obtained from RL-MUL produce 37.0% more hypervolume than the GOMIL on average.

We can observe that the margin of improvement over the SA approach is different in 8-bit and 16-bit cases. GOMIL performs better than SA in larger bit width, which may imply that the evolutionary algorithm cannot handle large bit width cases due to the large design space. Nevertheless, RL-MUL consistently performs the best in all cases.

C. Implementation in PE Arrays.

To further validate the performance advantage of RL-MUL and the generated multipliers, we implement the multipliers obtained from all the approaches into large macros. Processing element (PE) arrays are typical datapath designs widely used in DNN accelerators and contain many MAC units. Therefore, we use different multipliers to implement PE arrays to see whether area and timing improvement on the PE arrays can still be obtained. Fig. 6 and Fig. 8(b) show that the multipliers obtained from RL-MUL still dominate all baselines, achieving an average of 34.9% more hypervolume than the ILP approach. TABLE II shows that the RL-MUL can achieve the best performance when synthesis with all three scenarios.

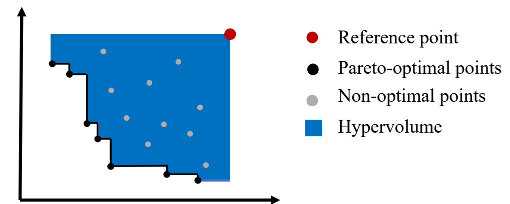

Fig. 7 An illustration of hypervolume. In our problem, a larger hypervolume is better.

V. Conclusion

In this work, we propose RL-MUL, a multiplier optimization framework based on reinforcement learning. We propose an RL agent that learns from the feedback from EDA tools to design Pareto-optimal multipliers. We demonstrate that RL-MUL can design multipliers that Pareto-dominate multipliers produced by existing approaches. In our future work, we plan to extend our RL-based framework to larger datapath modules.

References

[1] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, 1964.

[2] L. Dadda, "Some schemes for fast serial input multipliers," in *1983 IEEE 6th Symposium on Computer Arithmetic (ARITH)*, 1983.

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on December 20,2024 at 07:35:52 UTC from IEEE Xplore. Restrictions apply.

TABLE I Multiplier area and timing comparison.

Preference	Method	8-bit AND Area (μm^2)	8-bit AND Delay (ns)	8-bit MBE Area (μm^2)	8-bit MBE Delay (ns)	16-bit AND Area (μm^2)	16-bit AND Delay (ns)	16-bit MBE Area (μm^2)	16-bit MBE Delay (ns)
Area	Wallace [1]	427	0.8530	555	1.0880	1812	1.4073	2008	1.7016
	GOMIL [7]	404	0.8420	545	1.0797	1706	1.3375	1882	1.5432
	SA	411	0.8291	554	1.0284	1713	1.3566	2005	1.5783
	Ours	**388**	**0.7703**	**532**	**1.0154**	**1695**	**1.2668**	**1876**	**1.5425**
Timing	Wallace [1]	545	0.7791	720	0.9601	2420	1.2672	2645	1.4709
	GOMIL [7]	514	0.7750	706	0.9571	2281	1.2169	2482	1.3684
	SA	524	0.7414	717	0.9153	2288	1.2619	2641	1.3738
	Ours	**496**	**0.7089**	**693**	**0.8998**	**2271**	**1.1330**	**2482**	**1.3361**
Trade-off	Wallace [1]	458	0.8328	637	1.0018	2184	1.3054	2300	1.4954
	GOMIL [7]	435	0.8086	629	0.9837	2061	1.2416	2106	1.4298
	SA	431	0.7808	670	0.9216	1843	1.3045	2440	1.3897
	Ours	**419**	**0.7430**	**617**	**0.9187**	**1755**	**1.1900**	**2093**	**1.4177**

TABLE II PE array area and timing comparison.

Preference	Method	8-bit AND Area (μm^2)	8-bit AND Delay (ns)	8-bit MBE Area (μm^2)	8-bit MBE Delay (ns)	16-bit AND Area (μm^2)	16-bit AND Delay (ns)	16-bit MBE Area (μm^2)	16-bit MBE Delay (ns)
Area	Wallace [1]	175892	1.1347	208782	1.3302	601492	1.6693	650385	1.8543
	GOMIL [7]	170036	1.1237	206058	1.3154	574117	1.6017	618107	1.7403
	SA	171806	1.1010	208374	1.2706	575752	1.6216	649159	1.8174
	Ours	**165950**	**1.0421**	**202926**	**1.2512**	**571257**	**1.5305**	**618767**	**1.6976**
Timing	Wallace [1]	213345	1.0436	258016	1.1988	775001	1.5809	827503	1.6992
	GOMIL [7]	205378	1.0395	254475	1.1856	739591	1.5137	785896	1.6085
	SA	208033	1.0059	257130	1.1587	741361	1.5398	824847	1.6767
	Ours	**200951**	**0.9752**	**250934**	**1.1282**	**734484**	**1.4306**	**785896**	**1.5607**
Trade-off	Wallace [1]	191214	1.1017	236566	1.2254	628322	1.6419	735028	1.7352
	GOMIL [7]	185357	1.0709	221857	1.2703	600947	1.5727	649908	1.6847
	SA	184132	1.0471	224172	1.2101	602582	1.6003	680960	1.7774
	Ours	**178275**	**1.0065**	**218724**	**1.2059**	**585081**	**1.5010**	**624917**	**1.5558**

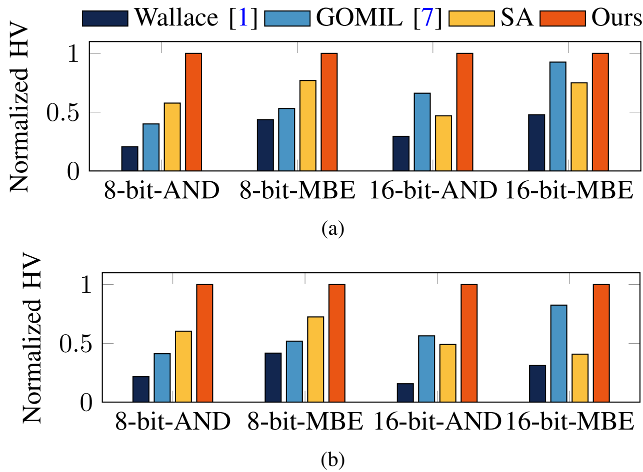

Fig. 8 Pareto-frontiers hypervolume comparison of (a) multipliers; (b) PE arrays.

[3] J. Fadavi-Ardekani, "M*n booth encoded multiplier generator using optimized wallace trees," *IEEE TVLSI*, June 1993.

[4] N. Itoh, Y. Tsukamoto, T. Shibagaki, K. Nii, H. Takata, and H. Makino, "A 32/spl times/24-bit multiplier-accumulator with advanced rectangular styled wallace-tree structure," in *Proc. ISCAS*, 2005.

[5] J. Liu, Y. Zhu, H. Zhu, C.-K. Cheng, and J. Lillis, "Optimum prefix adders in a comprehensive area, timing and power design space," in *Proc. ASPDAC*, 2007.

[6] M. Kumm and P. Zipf, "Pipelined compressor tree optimization using integer linear programming," in *Proc. FPL*, 2014.

[7] W. Xiao, W. Qian, and W. Liu, "Gomil: Global optimization of multiplier by integer linear programming," 2021.

[8] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng, "An algorithmic approach for generic parallel adders," in *Proc. ICCAD*, 2003.

[9] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," in *Proc. DAC*, 2013.

[10] H. Geng, Y. Ma, Q. Xu, J. Miao, S. Roy, and B. Yu, "High-speed adder design space exploration via graph neural processes," *IEEE TCAD*, 2022.

[11] S. Zhang, F. Yang, C. Yan, D. Zhou, and X. Zeng, "An efficient batch-constrained bayesian optimization approach for analog circuit synthesis via multiobjective acquisition ensemble," *IEEE TCAD*, 2021.

[12] R. Roy, J. Raiman, N. Kant, I. Elkin, R. Kirby, M. Siu, S. Oberman, S. Godil, and B. Catanzaro, "Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning," in *Proc. DAC*, 2021.

[13] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han, "Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning," in *Proc. DAC*, 2020.

[14] N. Siddharth, P. Geraldo, H. Corey, T. Yang, K. Brucek, and H. Ren, "Transsizer: A novel transformer-based fast gate sizer," in *Proc. ICCAD*, 2022.

[15] R. Bellman, "Dynamic programming," *Science*, 1966.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016.

[17] J. Zhang, Q. Gao, Y. Guo, B. Shi, and G. Luo, "Easymac: Design exploration-enabled multiplier-accumulator generator using a canonical architectural representation," in *Proc. ASPDAC*, 2022.

[18] T. Ajayi, D. Blaauw, T. Chan, C. Cheng, V. Chhabria, D. Choo, M. Coltella, S. Dobre, R. Dreslinski, M. Fogaça *et al.*, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," *Proc. GOMACTECH*, 2019.

[19] Nangate Inc., "Open Cell Library v2008_10 SP1," 2008. [Online]. Available: http://www.nangate.com/openlibrary/

[20] Parallax Software Inc., "OpenSTA," https://github.com/The-OpenROAD-Project/OpenSTA.

[21] E. Zitzler, D. Brockhoff, and L. Thiele, "The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration," in *International Conference on Evolutionary Multi-Criterion Optimization*, 2007.

Authorized licensed use limited to: The Hong Kong University of Science and Technology (Guangzhou). Downloaded on December 20,2024 at 07:35:52 UTC from IEEE Xplore. Restrictions apply.