# RL-MUL 2.0: Multiplier Design Optimization with Parallel Deep Reinforcement Learning and Space Reduction

DONGSHENG ZUO, The Hong Kong University of Science and Technology (Guangzhou)
JIADONG ZHU, The Hong Kong University of Science and Technology (Guangzhou)
YIKANG OUYANG, The Hong Kong University of Science and Technology (Guangzhou)
YUZHE MA, The Hong Kong University of Science and Technology (Guangzhou)

Multiplication is a fundamental operation in many applications, and multipliers are widely adopted in various circuits. However, optimizing multipliers is challenging due to the extensive design space. In this paper, we propose a multiplier design optimization framework based on reinforcement learning. We utilize matrix and tensor representations for the compressor tree of a multiplier, enabling seamless integration of convolutional neural networks as the agent network. The agent optimizes the multiplier structure using a Pareto-driven reward customized to balance area and delay. Furthermore, we enhance the original framework with parallel reinforcement learning and design space pruning techniques and extend its capability to optimize fused multiply-accumulate (MAC) designs. Experiments conducted on different bit widths of multipliers demonstrate that multipliers produced by our approach outperform all baseline designs in terms of area, power, and delay. The performance gain is further validated by comparing the area, power, and delay of processing element arrays using multipliers from our approach and baseline approaches.

**ACM Reference Format:**
Dongsheng Zuo, Jiadong Zhu, Yikang Ouyang, and Yuzhe Ma. 2024. RL-MUL 2.0: Multiplier Design Optimization with Parallel Deep Reinforcement Learning and Space Reduction . *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (December 2024), 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

In the era of rapid advancements in neural networks and streaming media applications, the demand for computational power has intensified. Notably, the multiply-accumulate (MAC) computation can constitute over 99% of operations in standard deep neural networks. At the hardware layer, multipliers and MAC circuits are integral to the architecture of compute-intensive circuits, significantly impacting the system performance, energy consumption, spatial requirements, and design complexities. Therefore, swiftly designing multipliers and MACs that meet metric specifications such as power, performance, and area (PPA) becomes imperative.

Multiplier design optimization at the architecture level is non-trivial due to the huge design space. For an 8-bit multiplier, the design space size is on the order of $10^9$, while for a 16-bit multiplier, it reaches approximately $10^{23}$. This exponential scaling highlights the significant complexity

Dongsheng Zuo and Jiadong Zhu contributed equally to this research.

This work is supported in part by the Department of Education of Guangdong Province (No. 2024KTSCX037), the Guangzhou-HKUST(GZ) Joint Funding Program (No. 2023A03J0155), and the Guangzhou Municipal Science and Technology Project (Municipal Key Laboratory Construction Project, Grant No.2023A03J0013).

Authors' addresses: Dongsheng Zuo, The Hong Kong University of Science and Technology (Guangzhou); Jiadong Zhu, The Hong Kong University of Science and Technology (Guangzhou); Yikang Ouyang, The Hong Kong University of Science and Technology (Guangzhou); Yuzhe Ma, The Hong Kong University of Science and Technology (Guangzhou).


1084-4309/2024/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

involved in effectively exploring and optimizing designs as bit-width increases. The multiplier design is fundamentally segmented into three primary components: a partial product generator (PPG), a compressor tree (CT), and a carry propagation adder (CPA). Among these, the optimization of the Compressor Tree (CT) is pivotal, as it significantly influences the PPA of a multiplier. The architecture of the compressor tree was first introduced in [1], designed for parallel compression (i.e., addition) of partial products in multiplication operations. This innovation has enabled the application of compressor trees in other datapath circuits, such as MACs and vector adders. Conventionally, MAC operations extend the functionality of multipliers by incorporating an accumulator after multiplication, resulting in increased operational delay. In contrast, the merged MAC is proposed, which enables the execution of MAC operations within the multiplication time by integrating the addend directly into the partial products [2]. This approach also allows the optimization methodologies developed for multipliers to be applied to MAC design optimization. Generally, datapath designs, including adders, multipliers, and MACs, can be completed manually. Take multiplier design as an example. The manual design includes Wallace tree structure [1], Dadda tree structure [3], and further optimized designs based on them [4–7], which effectively optimize area, power, and performance for specific technology nodes and applications. The Wallace tree strategically organized the compressor layers [4]. An area-reduced tree is proposed by using a maximum number of 3:2 compressors early and carefully placing 2:2 compressors [6]. Itoh *et al.* [5] proposed an advanced rectangular-styled tree structure, tailored specifically for 32-bit×24-bit multipliers. Optimizations for merged MAC structures have been explored based on the characteristics of multiply-accumulate operations. Basiri *et al.* [8] proposed a high-radix Booth-encoded merged MAC targeting floating-point DSP applications. Their design combines Wallace and Braun tree structures to optimize circuit depth and area, effectively balancing performance and resource usage for floating-point operations. Tung *et al.* [9] proposed a method where the final addition and accumulation of higher significance bits are merged to the partial products of the next multiplication operation. Zhang *et al.* [10] proposed a strategy optimizing pipeline merged MAC. These regular structure-based designs may not always meet the stringent PPA specifications required. To address this, full custom-designed multipliers are developed, which are finely optimized for specific fabrication processes or unique application scenarios [11–13]. However, a significant engineering effort is required to explore the huge design space with manual design, which limits design flexibility and efficiency.

The automatic generation or search methods have provided a more flexible solution to datapath designs. A three-dimensional method for designing the compressor tree was proposed, which utilized an input-to-output delay model [14–16]. Integer linear programming (ILP) is another widely investigated approach for datapath circuit optimization. Xiao *et al.* [17] employed ILP for global optimization of multiplier design by minimizing the total number of compressors in the compressor tree. In addition, ILP has also been applied for exploring adder trees based on analytical area, power, and timing models [18]. However, these works may suffer from the long runtime of the ILP solver as well as the misaligned objective between the modeled PPA metrics and real synthesized metrics. Heuristic search strategies utilize various pruning techniques and avoid exhaustive searches [19–23]. A heuristic is introduced in [21] for the design of compressor trees using generalized parallel counters (GPCs), aiming to optimize the balance between logic utilization and delay. Kumm *et al.* [23] further advanced heuristic method in [21, 22] and combined the heuristic method the with ILP.

Recently, machine learning methodologies have become promising solutions for circuit optimization and design space exploration, where various learning models are leveraged as surrogate models to evaluate designs during the search or optimization process [24–26]. An active learning-based prefix adder exploration framework is proposed in [25], which uses the Gaussian process

regression model to predict the delay and area based on the feature extracted from the prefix tree structure. Geng *et al.* [24] further facilitated automatic feature learning for prefix adder structures and deployed a sequential optimization framework that employs the graph neural process as a surrogate model, which enables a more efficient and effective adder structure exploration. However, the exploration still highly relies on a regression model as a proxy to the real PPA, whose modeling accuracy significantly affects the final results. Contrary to existing approaches, reinforcement learning (RL) integrates actual PPA evaluations directly into its optimization loop, demonstrating its feasibility by efficiently navigating complex design spaces. Recent advancements have seen RL tackle a variety of challenges within electronic design automation (EDA), as evidenced by applications across different domains such as prefix circuit design optimization, analog circuit design optimization and gate sizing [26–28]. Given the complexity of multiplier design and the vastness of its design space, the feasibility of reinforcement learning in multiplier design optimization is underscored. RL addresses this gap by leveraging real synthesized metrics as rewards, allowing optimization of designs that perform better than analytical models after synthesis. In addition, RL algorithms can also be enhanced with parallelism by implementing different environment instances. By utilizing multiple threads, the stability and efficiency of deep reinforcement learning algorithms are enhanced.

The design space of the multiplier is huge to explore. To address this, we have proposed an RL-based framework that is tailored for the optimization of multipliers and merged MACs [29]. However, obtaining a suitable representation of the multiplier structure is non-trivial due to its inherent complexity. To address this, we employ matrix and tensor representations for the compressor tree in a multiplier, enabling seamless integration of neural networks as the agent network in RL. These representations effectively capture the structural characteristics of the multiplier. The agent can learn to make effective decisions by optimizing the trade-off between key performance metrics such as power, performance, and area using a Pareto-driven approach. Furthermore, to exploit the huge design space more efficiently, the proposed framework also features design space pruning and parallel RL agent training for more efficient optimization. To validate the effectiveness of the proposed framework, we applied it to design and optimize multipliers with different bit widths. The experimental results show that our approach outperforms various baseline methods, including legacy designs, evolutionary algorithms, and integer linear programming, in terms of area and delay. Moreover, to validate the effectiveness of the optimized multipliers and MACs, a computation module, e.g., a process element (PE) array, is implemented with the multipliers and MACs generated by the RL agent, and the PPA gets improved accordingly. In summary, the contributions are as follows:

- We propose a multiplier optimization framework based on reinforcement learning, marking the first instance of applying reinforcement learning for this purpose to our knowledge.
- We present a matrix and a tensor representation for multipliers, which enables the seamless integration of deep neural networks as the agent network. A Pareto-driven reward is employed to accommodate the trade-off between the area and delay so that the agent can learn to achieve Pareto-optimal designs.
- To improve search efficiency within this framework, we further enhance the framework with a parallel training methodology to enable faster and more stable training.
- We also broaden the scope of the RL-based multiplier design framework to include fused MAC to validate the applicability.
- Experimental results demonstrate that the multipliers and MACs produced by RL agents dominate all baseline designs in terms of both area and delay. Furthermore, applying the

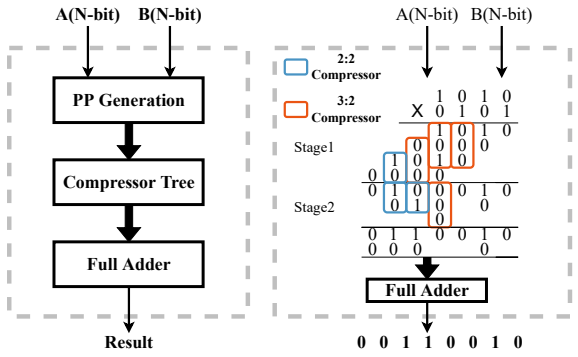


Fig. 1. Multiplier architecture

optimized multipliers and MACs to the implementation of a larger computation module also results in PPA improvement, which validates the effectiveness of the optimized designs.

## 2 PRELIMINARY

### 2.1 Multiplier Architecture

The multiplier typically comprises three primary components: a partial product generator (PPG), a compressor tree (CT), and a carry propagation adder, as shown in Figure 1. PPG generates partial products (PPs) from the multiplicand and multiplier, while the CT compresses these PPs into two parallel rows. Subsequently, an adder is utilized to aggregate these two rows of PPs, culminating in the final product. A typical partial product generator generally employs $N^2$ AND gates for an $N$-bit multiplier. A CT has multiple compression stages to compress the PPs into two rows. Predominantly, there are 3:2 compressors and 2:2 compressors implemented through a full adder and a half adder, respectively. A 3:2 (resp. 2:2) compressor applied at column $j$ of stage $i$ receives 3 (resp. 2) partial products as input from column $j$ of stage $i$, passing the *sum* output to column $j$ of stage $i+1$, and the *carry out* to column $j+1$ of stage $i+1$. Consequently, a 3:2 compressor decreases the partial products of column $j$ by two, while a 2:2 compressor reduces them by one, each incrementing the partial products in column $j+1$ by one.

### 2.2 Q-Learning

RL encompasses a collection of optimization problems referred to as *state* $s$, with a corresponding set of *actions* $A$. An agent transitions from one state $s$ to another state $s'$ by executing an action $a \in A$, consequently receiving a *reward* $r(s,a)$ as an evaluation from the RL environment. The model governing action selection is known as the *policy* $\pi$. The primary objective of the RL agent is to devise a policy that optimizes the cumulative reward.

Q-learning is an RL algorithm that learns the scores of each action $a$ for a given state $s$, and the score is called Q-value, represented by $Q(s,a)$. According to Bellman equation [30], the Q-value is calculated as follows:

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

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

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

where $\alpha$ is the learning rate. In this paper, we utilize the deep Q-learning approach, leveraging a deep neural network to approximate the Q-value. Here, the state $s$ represents the architecture of the multiplier, detailed in Section 3.2. An action $a$ alters the current multiplier architecture into a new one, effectively progressing to the next state. The reward $r$ is quantified by the enhancements in the multiplier's area and delay metrics.

### 2.3 Advantage Actor-Critic

The A2C algorithm [31] addresses the challenges of high variance and unstable learning due to strong correlations between consecutive states by splitting the traditional RL model into two components: an actor that enacts policies and a critic that evaluates these actions. A policy network $\pi(a|s;\theta)$ and a value network $v(s;w)$ are employed, where $\theta$ and $w$ are the parameters of two neural networks, respectively. The synchronous multi-thread coordination method in A2C ensures uniform learning and parameter updates. This synchronization removes the need for different agents in A3C [31], as the single agent with different environment instances suffices, while therefore avoiding updates based on outdated copies, significantly stabilizing the training process and offering sufficient parallelism and effectiveness [32, 33]. The A2C algorithm employs bootstrapped advantage estimates generated by the critic instead of mere state-value approximations to enhance gradient estimation accuracy and learning efficiency [34]. With state $s$ and action $a$, The advantage is defined as:

$$\hat{A}(s,a) = Q_\pi(s,a) - V_\pi(s), \tag{3}$$

where $Q_\pi(s,a)$ indicates the action-value function that estimates the expected reward that can be obtained by taking action $a$ and then following strategy $\pi$ at state $s$, while $V_\pi(s)$ indicates the state-value function that estimates the expected reward that can be obtained in state $s$ if the strategy $\pi$ is followed from that state instead of taking a specific action [32, 35].

For a known transition $(s_t, a_t, r_t, s_{t+1})$, to facilitate calculations of advantage function, we use a value network $v(s_t;w)$ to approximate the state-value function $V_\pi(s_t)$, and estimate $Q_\pi(s_t,a_t)$ through Monte Carlo methods based on the Bellman equation. Consequently, we can approximate (3) as:

$$\hat{A}(s_t,a_t) \approx r_t + \gamma \cdot v(s_{t+1};w) - v(s_t;w), \tag{4}$$

where $\gamma$ is the discount factor.

## 3 PROPOSED METHOD

### 3.1 Overview

As illustrated in the left of Figure 2, our original RL-MUL framework leverages a reinforcement learning approach for multiplier design optimization. An RL agent engages in iterative interactions with its environment from an initial state $s_0$. At any given state $s_t$, the RL agent, guided by a policy $\pi$ derived from the policy network, selects an action $a_t$ from a set of legal actions. This action modifies the current multiplier configuration, leading to a new state $s_{t+1}$. Subsequently, a reward $r_t$ is computed using EDA tools, facilitating the neural network model's update based on the received feedback.

### 3.2 Multiplier Representation

The RL state space, denoted as $\mathcal{S}$, consists of all possible configurations of $N$-bit multipliers. We recognize the count of various compressors in each column as a critical attribute influencing the synthesized performance metrics of the multipliers. Consequently, we characterize the multiplier architecture using the aggregate counts of 3:2 and 2:2 compressors across columns, encapsulated by a matrix $\boldsymbol{M} \in \mathbb{R}^{2N\times 2}$. In this matrix, the first and second rows quantify the total 3:2 and 2:2

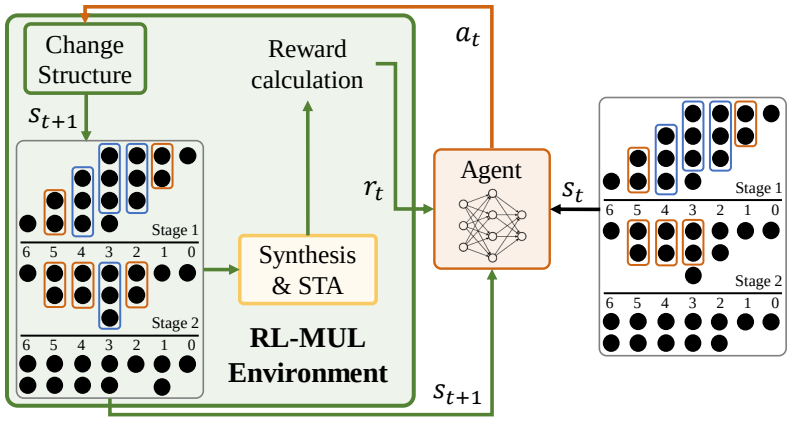


Fig. 2. RL-MUL framework.

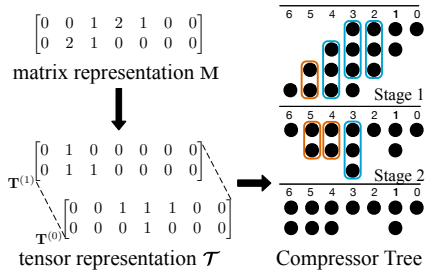


Fig. 3. Structure representation.

compressors in each column, respectively. An illustration of a 4-bit multiplier structure alongside its matrix representation $\boldsymbol{M}$ is provided in Figure 3. To derive a complete multiplier structure from $\boldsymbol{M}$, the compressors are allocated to specific stages. However, the mapping from $\boldsymbol{M}$ to the structures is not unique since different assignments of compressors in multiple stages may have the same overall number in each column. To achieve a distinctive representation, we advance to a tensor representation that offers more informative insights, as illustrated in Figure 3.

We represent the tensor as $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{K\times 2N\times ST}$, with $K$ indicating the total kinds of compressors used and $ST$ the stage count. Specifically, we utilize 3:2 and 2:2 compressors, thus $K = 2$. This framework is designed for potential extension to accommodate more compressor variants. The tensors $\boldsymbol{T}^{(0)} = \boldsymbol{\mathcal{T}}_{0,:,:} \in \mathbb{R}^{2N\times ST}$ and $\boldsymbol{T}^{(1)} = \boldsymbol{\mathcal{T}}_{1,:,:} \in \mathbb{R}^{2N\times ST}$ respectively map the placement of 3:2 and 2:2 compressors. The elements $t_{ij}^{(0)}$ and $t_{ij}^{(1)}$ denote the quantity of 3:2 and 2:2 compressors at the $j$-th column of the $i$-th stage. Given a matrix $\boldsymbol{M}$ that contains the information of the overall number of compressors in each column, we can construct the tensor representation $\boldsymbol{\mathcal{T}}$ correspondingly based on an assignment scheme of the compressors in different stages.

For the assignment process, we employ a deterministic method that assigns compressors from the least to the most significant bit columns, prioritizing 3:2 compressors and then utilizing 2:2 compressors where applicable. This method progresses through stages until all compressors are allocated, as detailed in Algorithm 1. This approach guarantees a unique tensor representation for each multiplier structure, facilitating precise and unambiguous characterizations of the multiplier architecture.

**Algorithm 1** Compressor Assignment

**Require:** $M$: Matrix representation
**Ensure:** $\mathcal{T}$: Tensor representation.

**for** $j \leftarrow 1$ *to* $2N$ **do**
  $i \leftarrow 0$
  **while** column $j$ exists not assigned comp. **do**
    Assign 3:2 comp. to stage $i$ column $j$ first
    Update $t_{ij}^{(0)}$ in $T^{(0)}$
    **if** Remaining PPs $\geq 2$ **then**
      Assign 2:2 comp. to stage $i$ column $j$
      Update $t_{ij}^{(1)}$ in $T^{(1)}$
    **end if**
    $i \leftarrow i + 1$
  **end while**
**end for**
$\mathcal{T}_{0,:,:} \leftarrow T^{(0)}$
$\mathcal{T}_{1,:,:} \leftarrow T^{(1)}$

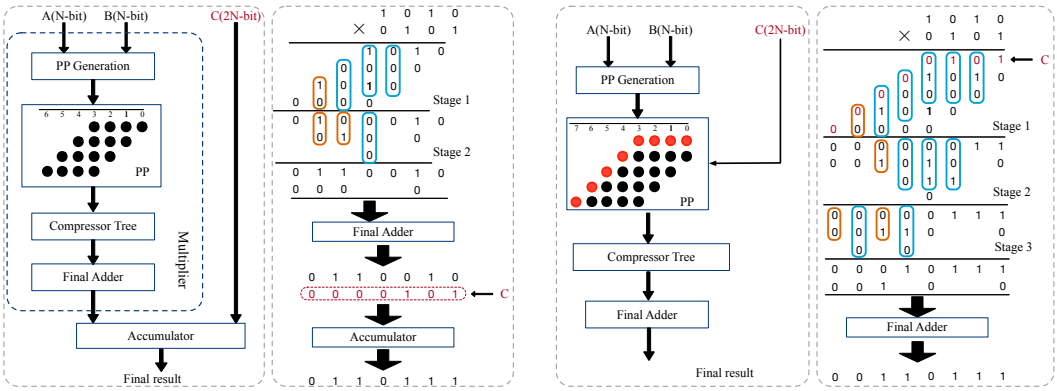


Fig. 4. MAC architectures

### 3.3 Multiplier Modification

In the RL agent's context, an action $a$ signifies the agent's choice to alter the existing structure of the multiplier. The agent can choose from four distinct actions for each column: adding or removing a 2:2 compressor, and replacing a 3:2 or a 2:2 compressor with another type. We denote $res_j$ to present the PP number after compression of column $j$, which should only be 1 or 2. Actions leading to $res_j$ values of 0 or 3, such as adding or removing a 3:2 compressor, are excluded, thereby defining the action space as $|\mathcal{A}| = 2N \times 4 = 8N$. It is important to note that not every action is feasible to yield a legal multiplier structure instantly. For instance, if there is no 2:2 compressor in column 1 as depicted in Figure 3, attempting to remove a 2:2 compressor from this column is considered invalid. Similarly, any action $a$ on column $j$ that results in the partial product (PP) numbers post-compression being either 0 or 3 would be considered invalid. Take another example

from Figure 3. Removing a 2:2 compressor from column 4 would lead to $res_4$ equating to three, thereby invalidating the action.

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

$$Q(s_t) = [q_{11}, q_{12}, q_{13}, q_{14}, \cdots, q_{2N,1}, q_{2N,2}, q_{2N,3}, q_{2N,4}], \tag{5}$$

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 $\boldsymbol{m}$ is utilized as the selector to enable valid actions and forbid invalid actions.

$$\boldsymbol{m} = [m_{10}, m_{11}, m_{12}, m_{13}, \cdots, m_{2N,0}, m_{2N,1}, m_{2N,2}, m_{2N,3}], \tag{6}$$

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 the proposed RL framework, the final masked Q-value vector is the element-wise multiplication of the mask vector and Q-value vector:

$$Q'(s_t) = Q(s_t) \odot \boldsymbol{m}. \tag{7}$$

Now the decision is given by

$$a_t = \arg\max_a Q'(s_t, a). \tag{8}$$

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. This strategy sequentially refines from column $j+1$ to the MSB, addressing under-compression by adding or replacing compressors, and managing over-compression by removing compressors. Similar to the assignment procedure, the legalization process is also deterministic. Under state $s_t$, we can get a new state $s_{t+1}$ after performing action $a_t$ to modify the structure along with the legalization.

### 3.4 Pareto-driven Reward

In our framework, we define the reward, $r_t$, as the improvement in circuit metrics, such as area, delay, and power, achieved by executing an action $a_t$ at state $s_t$. Considering the nature of the trade-off between power, performance, and area (PPA), a superior multiplier design is always expected to achieve Pareto-optimal in terms of these dimensions. To encourage the RL agent to learn to generate Pareto-optimal designs, we introduce a Pareto-driven reward mechanism. This mechanism leverages a synthesis flow under multiple design constraints, enabling the reward to cover a variety of design scenarios: those driven primarily by area, delay, or power, as well as scenarios seeking a trade-off optimization of these three key metrics. The overall cost is calculated as a weighted sum of area, delay, and power, allowing for flexible adjustment of their relative importance in different design scenarios:

$$cost = w_a \sum_{i=1}^{n} area_i + w_d \sum_{i=1}^{n} delay_i + w_p \sum_{i=1}^{n} power_i, \tag{9}$$

where $area_i$, $delay_i$, and $power_i$ are the synthesized metrics under the $i$-th constraint. Since the area, delay, and power values have substantially different ranges, we normalize the area, delay, and power metrics to a consistent scale using Wallace tree implementations. $w_a$, $w_d$ and $w_p$ are the weights to trade off PPA. We define our reward $r$ as the difference between $s_t$ and $s_{t+1}$:

$$r_t = cost_t - cost_{t+1} \tag{10}$$

**Algorithm 2** Legalization

**Require:** Multiplier structure to be legalized; $C$: action column
**Ensure:** Legalized multiplier structure

```
1: for j ← (C + 1) to 2N do
2:     res_j ← 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
```

### 3.5 Training Algorithm

We adopt ResNet-18 [36] as the backbone of Q-Network with the parameters denoted by $\theta$. The state undergoes encoding into a tensor representation $\boldsymbol{\mathcal{T}}$, as detailed in Section 3.2, before being processed by the Q-network. The RL training methodology is outlined in Algorithm 3. Initially, action selections $a$ are randomized during the warm-up phase (Line 6), transitioning to policy-based selections in subsequent steps (Line 8).

Each iteration $t$ leads to the transformation of the multiplier's architecture from $s_t$ to $s_{t+1}$, culminating in a reward $r_t$ derived from synthesis and timing analysis. This process generates a new transition $(s_t, a_t, r_t, s_{t+1})$, which is recorded. Then, the network parameter $\theta$ is updated by gradient descent, and $w$ is also updated in actor-critic methods(Line 14). The target Q-value for each state-action pair within the batch is determined as follows:

$$y = r' + \gamma \max_{a'} Q'(s', a'; \theta), \tag{11}$$

where $\gamma$ is the discount factor. Based on the expected Q-value $y$, a gradient of $\theta$ can be obtained by:

$$\Delta\theta = \nabla_\theta (y - Q'(s, a; \theta))^2. \tag{12}$$

Then, the network parameter $\theta$ is updated by gradient descent(Line 14). By incorporating masked actions in backpropagation, the Q-network learns to assign lower Q-values to seldom-used, invalid actions, minimizing their selection in future iterations.

## 4 RL-MUL 2.0

Optimizing hardware configurations requires an efficient search within a vast design space, particularly in multiplier design, where an increase in bit width exponentially expands the design space.

**Algorithm 3** RL-MUL flow

**Require:** $\boldsymbol{M_0}$: initial multiplier structure; $\gamma$: discount factor; $\alpha$: learning rate; $T$: total training steps; $T_B$: warm-up steps
**Ensure:** $\theta$: Q-network parameters

Replay buffer $B \leftarrow \{\}$
Encode $s_0$ into $\boldsymbol{\mathcal{T}}$ based on $\boldsymbol{M_0}$ ▷ Algorithm 1
$t \leftarrow 0$
**for** $t \leftarrow 0$ to $T$ **do**
    **if** $t < T_B$ **then**
        $a_t \leftarrow$ randomly choose from legal actions
    **else**
        Get $a_t$ by Equation (8)
    **end if**
    Perform $a_t$ to $s_t$ and get $s_{t+1}$
    Run EDA tools on $s_{t+1}$ and get $r_t$ ▷ Equation (10)
    Push $(s_t, a_t, r_t, s_{t+1})$ to $B$
    Sample a batch of transitions from $B$
    Update $\theta$ by gradient descent ▷ Equation (12)
**end for**

Therefore, dealing with this enlarged space effectively becomes crucial, especially for larger designs like MACs, where the DQN algorithm may struggle to achieve optimal results.

In this work, we extend the proposed RL framework to MAC designs, enhancing its application in deep learning acceleration. To tackle the greater challenges of more complex designs, we use parallel algorithms to improve efficiency from two perspectives. Firstly, parallel optimization is always a promising solution in this scenario. Their inherent parallelism not only reliably boosts search efficiency but also fosters a thorough exploration of possible designs, enhancing the likelihood of uncovering optimal or nearly optimal solutions. Secondly, search space pruning condenses the design space by discarding less promising designs. This approach emphasizes exploring viable design areas, thus refining the search process and minimizing computational demands. Eliminating inferior designs early on ensures a more targeted and efficient discovery of superior configurations. In addition, integrating metrics with high correlation can achieve a similar purpose, not only significantly simplifying the optimization process but also enhancing the focus on configurations that genuinely contribute to performance improvements.

### 4.1 Extend to Merged Multiply-Accumulator Architecture

An integral component of many digital signal processing systems and neural network architectures is the multiply accumulator (MAC), which can be a decisive factor in determining the overall performance of many computing-intensive systems. Incorporating compressor trees within MACs offers a pathway to enhance their efficiency. Rather than treating multiplication and accumulation as sequential operations, this approach seamlessly integrates them. By merging the accumulation (addition) directly into the partial product stages of multiplication and conducting partial product compression, we can capitalize on parallelism, thus potentially speeding up the entire MAC operation. We can see that the proposed RL framework can seamlessly support the optimization of fused MAC design.

By integrating the addition into the partial product generation phase, the representation within the RL framework is tweaked to consider the intricacies of the MAC operation. The aim is to train

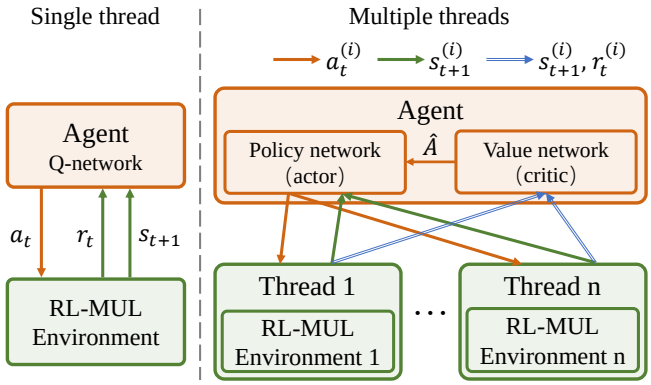


Fig. 5. Comparison of RL algorithm between single-thread and multi-thread implementations.

the RL agent to explore and design optimal MAC structures, utilizing compressor trees for efficient parallel addition. Therefore, the representations in Section 3.2 can be easily extended to MACs by providing "merged" partial products, and the training procedure will be identical. In Section 5, we will demonstrate the effectiveness and superiority of the proposed RL framework for fused MAC design.

### 4.2 Multiple Agents Training

Multiple agents running in parallel are more likely to explore different parts of the environment, promoting more efficient and stable policy training. Therefore, as shown in Figure 5, compared to the single-thread RL algorithm implemented in the proposed RL framework, we further enhance the proposed framework by training multiple agents in parallel, where each agent is handled by a thread. Following the training stability analysis in [31], each agent in our framework is designed with the A2C scheme, where the policy and value networks share the convolution layers of ResNet-18. Specifically, we employ a shared global network parameter across threads, each interacting with its local environment independently, followed by an average of all threads' gradient updates to adjust the global parameter. $n$ parallel threads synchronously process corresponding transitions $(s_t^{(i)}, a_t^{(i)}, r_t^{(i)}, s_{t+1}^{(i)})$. The right side of Figure 2 illustrates the synchronous parallel structure with A2C. At each step, given $n$ is the number of threads, the agent selects $a_t$ including $n$ actions for threads, the $i$-th thread's RL-MUL environment instance receives its corresponding action $a_t^{(i)}$ and transitions to the new state $s_{t+1}^{(i)}$, which is returned with the reward $r_t^{(i)}$ to the agent. Thus, in A2C, each element of the transition tuples $(s_t, a_t, r_t, s_{t+1})$ is an $n$-element vector. Then Equation (5) is transformed into:

$$\pi(\cdot|s_t) = [\pi_{11}, \pi_{12}, \pi_{13}, \pi_{14}, \cdots, \pi_{2N,1}, \pi_{2N,2}, \pi_{2N,3}, \pi_{2N,4}], \tag{13}$$

where each group of $\pi_{j1}, \pi_{j2}, \pi_{j3}, \pi_{j4}$ indicates the probability of the four actions $a_{j1}, a_{j2}, a_{j3}, a_{j4}$ in column $j$. The mask is configured as in Section 3.3, and the final masked probability distribution vector is:

$$\pi'(\cdot|s_t) = \pi(\cdot|s_t) \odot \boldsymbol{m}. \tag{14}$$

Now, the decision is given by

$$a_t \sim \pi'(\cdot|s_t). \tag{15}$$

Algorithm 4 outlines the parallel training and optimization flow in RL-MUL 2.0 [32, 34]. Firstly, $n$ threads (Line 1) are initiated. Then, at each step, a multiplier structure alteration $a_t$ is sampled

**Algorithm 4** RL-MUL 2.0 flow

**Require:** $n$: number of threads; $T$: total training steps; $t_{up}$: update interval
**Ensure:** $\theta$: policy network parameters; $w$: value network parameters
1: Initialize $n$ parallel threads
2: **for** $t \leftarrow 0$ to $T$ **do**
3: Sample a multiplier structure alteration $a_t^{(i)}$ by Equation (15), $\forall i \in \{1, 2, \ldots, n\}$
4: Perform $a_t^{(i)}$ to get structure $s_{t+1}^{(i)}$ and $r_t^{(i)}$, $\forall i \in \{1, 2, \ldots, n\}$
5: **if** $t \mid t_{up}$ **then**
6: Update $\theta$ by gradient ascent ▷ Equation (16)
7: Update $w$ by gradient descent ▷ Equation (19)
8: **end if**
9: **end for**

from Equation (15), which incorporates masks to prevent the selection of actions that lead to invalid multiplier structures (Line 3). Following this, the chosen action is executed to obtain a new structure and its corresponding reward (Line 4). In terms of the model updating, this algorithm employs an $n$-step return approach for faster learning, updating the policy network parameter $\theta$ and the value network parameter $w$ only when the total training steps constitute an integer multiple of the update interval (Line 5). A policy gradient of $\theta$ used to update the policy network can be obtained by:

$$\Delta\theta = \nabla_\theta \log \pi(a_t|s_t;\theta) \cdot \hat{A}(s_t, a_t), \tag{16}$$

where $\pi(a_t|s_t;\theta)$ is the policy network definited by parameter $\theta$ at time $t$, and $\hat{A}$ is the advantage function defined in Equation (4) [35]. Then, the policy network parameter $\theta$ is updated by gradient ascent (Line 6). In addition, the Temporal-Difference (TD) learning [37] aspect of the A2C algorithm guides the value network $v(s_t;w)$ to converge to the TD target $y_t$, defined as:

$$y_t = r_t + \gamma \cdot v(s_{t+1};w), \tag{17}$$

where $\gamma$ represents the discount factor. The TD target combines the real reward $r_t$ after taking the action $a_t$ with the predicted value of the next state $s_{t+1}$, serving as a crucial element in computing the TD error. This error is expressed as:

$$\delta_t = v(s_t;w) - y_t, \tag{18}$$

measuring the discrepancy between the estimated value of the state before taking the action $a_t$ and the TD target. In other words, the TD error reflects the accuracy of the value function prediction. Based on the TD error, a gradient of $w$ used to update the value network can be obtained by:

$$\Delta w = -\nabla_w \frac{(\delta_t)^2}{2} = -\delta_t \cdot \nabla_w v(s_t;w). \tag{19}$$

Then, the value network parameter $w$ is updated by gradient descent (Line 7).

### 4.3 Objective Space Reduction

The goal of multiplier design space exploration is to find designs that are superior in terms of multiple objectives. In Equation (10), a weighted reward is designed such that the agent can acquire a good trade-off among different objectives, while the selection of weights for each objective can impact the final solutions substantially. The more objectives we have, the more effort is required for tuning the weights. Notably, we investigated a correlation between the area and the power of a multiplier. Based on the architectures we have searched for, it is observed that the power and

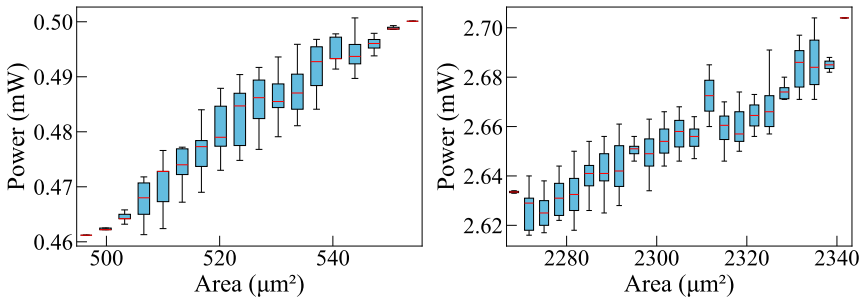


Fig. 6. Correlation between area and power. The upper one is an 8-bit AND-based multiplier, and the lower one is a 16-bit AND-based multiplier.

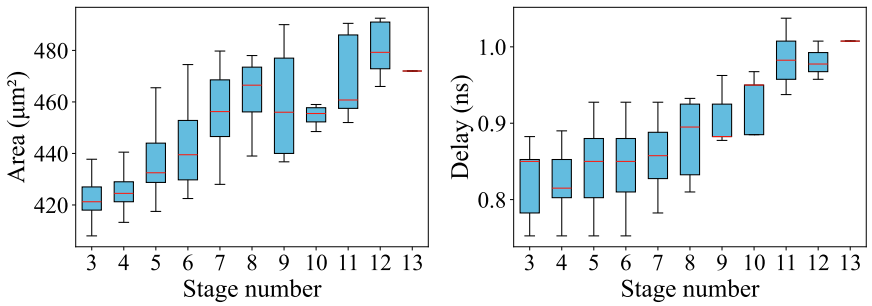


Fig. 7. Correlation between stage number and metrics of 8-bit AND-based multipliers.

area are highly correlated. A correlation between these two factors represented by box plots is illustrated in Figure 6, the upper graph depicts the relationship for 8-bit AND-based multipliers, while the lower plot shows the same for 16-bit AND-based multipliers. The bottom and top boundaries of the box represent the first and third quartiles, respectively, indicating the inter-quartile range (IQR). The median is denoted by the band within the box. The upper whisker represents the maximum value of the data, and the lower whisker represents the minimum value of the data. It can be observed from the trend in Figure 6 that there exists a strong positive correlation between the area and the power, which suggests that the area is a reliable indicator of the power. Consequently, our methodology gives precedence to area and delay as key optimization metrics, which allows Equation (9) to be further reduced to:

$$cost = w_a \sum_{i=1}^{n} area_i + w_d \sum_{i=1}^{n} delay_i \tag{20}$$

### 4.4 Search Space Pruning

Furthermore, another analysis indicates the number of stages of a compressor tree as a significant factor affecting the area and delay of multipliers, as shown in Figure 7. This analysis takes 8-bit AND-based multiplier structures as an example. Notably, there is a positive relationship between stage number and the parameters of area and delay. This suggests that an increase in stage number is associated with a corresponding rise in these metrics. To mitigate this, our framework integrates a strategy to constrain actions that will lead to excessive stage increases, which facilitates a more efficient search and optimization toward desired multiplier structures.

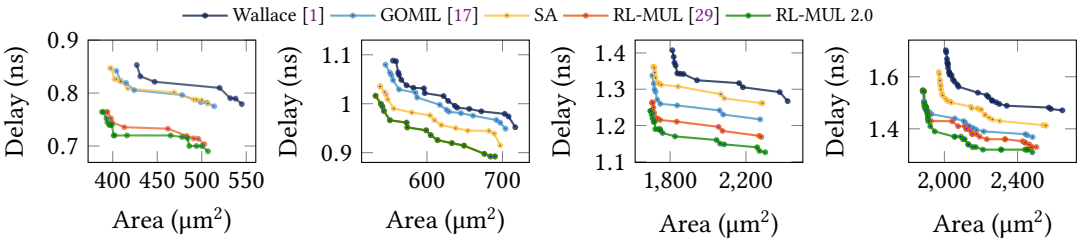


Fig. 8. Pareto-frontiers of the synthesis results on multipliers. From left to right: 8-bit AND-based; 8-bit MBE-based; 16-bit AND-based; 16-bit MBE-based. Note that the Pareto-frontiers of RL-MUL 2.0 and original RL-MUL overlap in the 8-bit MBE-based result.

## 5 EXPERIMENTAL RESULTS

### 5.1 Setup

The proposed framework is implemented on a Linux system powered by a 2.8 GHz AMD EPYC CPU and an NVIDIA RTX 3090 GPU. We use EasyMAC [38] for RTL generation and have extended its capabilities by incorporating Modified Booth Encoding (MBE)-based partial product generation, as well as enhancing support for the RTL generation of merged MAC units. These designs are synthesized using the OpenROAD flow [39] with the NanGate 45*nm* Open Cell Library [40]. OpenSTA[41] is utilized to perform timing analysis. To ensure the functional correctness of the generated multipliers, we first convert RTL into AIGER format using Yosys [42], and use the cec command in ABC[43] to perform logic equivalence verification with a golden implementation of multiplier.

Given the prevalent use of 8-bit and 16-bit multipliers, the RL-MUL 2.0 framework is assessed using both 8-bit and 16-bit multipliers, incorporating AND-based PPG and MBE-based PPG. We compare our approach against established baselines, including the legacy Wallace tree[1], an ILP-based method GOMIL [17], and the simulated annealing (SA) technique. Four delay constraints are configured in Equation (9). The weights $w_a$ and $w_d$ range from 0 to 1, resulting in different optimization preferences towards area or delay. In native RL-MUL implementation, we set $\gamma$ to 0.8, learning rate to 0.0002, $\epsilon$ to decay from 0.95 to 0.05, and employ RMSProp optimizer [44] for the training. In the RL-MUL 2.0 implementation, we employ four synchronization threads and a five-step return. We train the original RL-MUL and RL-MUL 2.0 10,000*s* and run SA for the same amount of time. During this period, RL-MUL performs approximately 450 iterations, while RL-MUL 2.0 performs about 650 iterations, exploring 4 design points in each iteration. Each iteration of RL-MUL takes about 22 seconds, and each iteration of RL-MUL 2.0 takes about 15 seconds. For the ILP approach, solving the 8-bit cases takes approximately 1 *min*, whereas the 16-bit cases require around 16 *h*. Synthesizing under varying design constraints produces different netlists for the same RTL design. We synthesize all the obtained multipliers and MACs across target delays from 0.05 *ns* to 1.2 *ns*. Furthermore, to enhance the demonstration of RL-MUL 2.0's effectiveness and the performance of the resulting designs, we incorporate these multipliers and MACs from all evaluated methods into large macro designs. Processing Element (PE) arrays, commonly utilized in DNN accelerators, consist of numerous MAC units, making them ideal for further evaluating the impact of different multipliers and MACs on area and timing efficiency. By integrating different multipliers and MACs into PE arrays, specifically following a systolic array architecture, we investigate the potential for improvements in both area and timing within these structures.

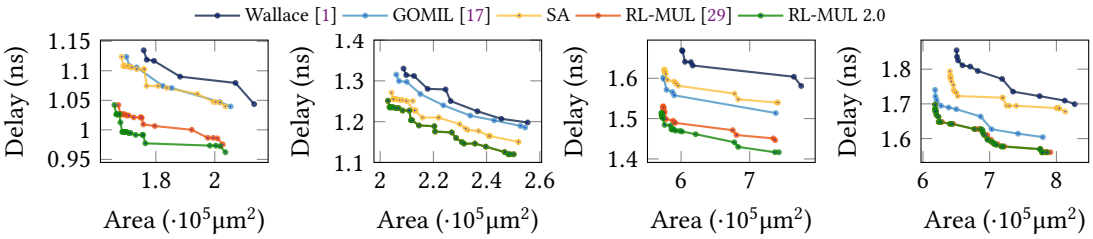


Fig. 9. Pareto-frontiers of the synthesis results on multiplier-implemented PE arrays. From left to right: 8-bit AND-based; 8-bit MBE-based; 16-bit AND-based; 16-bit MBE-based.

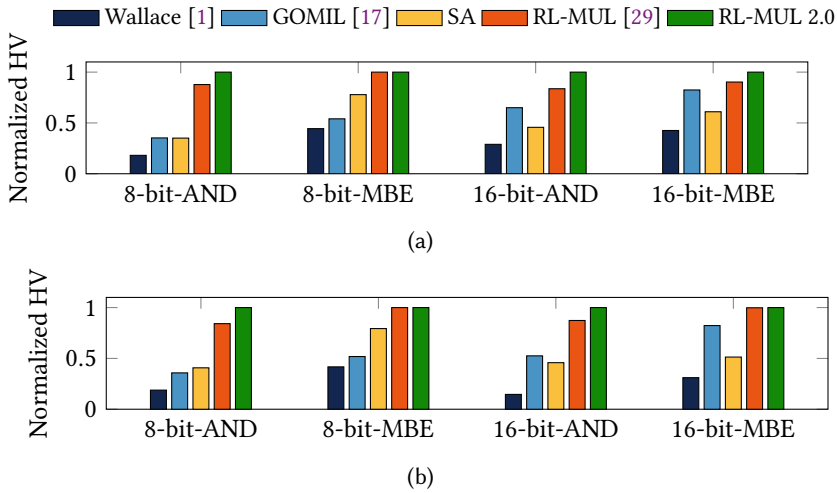


Fig. 10. Pareto-frontiers hypervolume comparison of (a) multipliers and (b) multiplier-implemented PE arrays.

### 5.2 Multiplier Performance Comparison

The resulting area-delay curves for multipliers are illustrated in Figure 8, where the designs derived from the RL-MUL framework outperform all baselines. Detailed statistics of minimum area, delay, and balanced area-delay metrics are presented in Table 1 (the optimal results are marked in bold). In the trade-off scenario, optimal corresponds to the lowest weighted sum of PPA in Equation (10). Through the RL-MUL 2.0 framework, we achieve up to 10.0% area reduction under the minimum area constraint and a 12.5% decrease in delay under the minimum delay constraint. Additionally, the implementation in PE arrays, as shown in Figure 9 and Table 2, indicates similar performance, with up to a 6.0% area reduction and up to 11.5% delay decrease.

The hypervolume [45] measures the volume enclosed by the Pareto frontier and a reference point in the objective space, which is a common metric to evaluate the quality of the Pareto frontiers. Hypervolume comparisons for multipliers presented in Figure 10a show that RL-MUL generates significantly larger hypervolume than GOMIL, with average increases of 85.9%. RL-MUL 2.0 shows an improvement of 11.1% compared to the original RL-MUL. Similarly, for PE arrays constructed with the multiplier, as shown in Figure 10b, the average improvement of RL-MUL 2.0 compared with GOMIL and original RL-MUL is 96.1% and 8.4% respectively.

Table 1. Multiplier area, timing, and power comparison.

| Preference | Method | 8-bit | | | | | | 16-bit | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | AND | | | MBE | | | AND | | | MBE | | |
| | | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) |
| Area | Wallace[1] | 427 | 0.8530 | 0.3513 | 555 | 1.0880 | 0.4975 | 1812 | 1.4073 | 1.962 | 2008 | 1.7016 | 2.187 |
| | GOMIL [17] | 404 | 0.8420 | 0.3352 | 545 | 1.0797 | 0.4833 | 1706 | 1.3375 | 1.855 | 1882 | 1.5432 | 2.013 |
| | SA | 397 | 0.8468 | 0.3317 | 538 | 1.0353 | 0.4845 | 1712 | 1.3619 | 1.860 | 1969 | 1.6184 | 2.133 |
| | **RL-MUL** | 393 | 0.7643 | 0.3261 | **532** | **1.0162** | **0.4752** | 1705 | 1.2633 | 1.855 | 1882 | 1.5478 | 2.016 |
| | **RL-MUL 2.0** | **388** | **0.7643** | **0.3237** | **532** | **1.0162** | **0.4752** | **1696** | **1.2481** | **1.845** | **1881** | **1.5478** | **2.008** |
| Timing | Wallace[1] | 545 | 0.7791 | 0.4977 | 720 | 0.9601 | 0.7054 | 2420 | 1.2672 | 2.822 | 2645 | 1.4709 | 3.032 |
| | GOMIL [17] | 514 | 0.7750 | 0.4726 | 706 | 0.9571 | 0.6836 | 2281 | 1.2169 | 2.629 | 2482 | 1.3684 | 2.791 |
| | SA | 507 | 0.7800 | 0.4656 | 697 | 0.9147 | 0.6886 | 2280 | 1.2616 | 2.619 | 2551 | 1.4125 | 2.893 |
| | **RL-MUL** | 503 | 0.7033 | 0.4650 | **690** | **0.8922** | **0.6736** | 2281 | 1.1684 | 2.638 | 2475 | 1.3318 | 2.780 |
| | **RL-MUL 2.0** | **507** | **0.6931** | **0.4670** | **690** | **0.8922** | **0.6736** | **2302** | **1.1263** | **2.658** | **2481** | **1.3085** | **2.791** |
| Trade-off | Wallace [1] | 458 | 0.8328 | 0.3820 | 637 | 1.0018 | 0.5900 | 2184 | 1.3054 | 2.562 | 2300 | 1.4954 | 2.537 |
| | GOMIL[17] | 435 | 0.8086 | 0.3634 | 629 | 0.9837 | 0.5824 | 2061 | 1.2416 | 2.382 | 2106 | 1.4298 | 2.328 |
| | SA | 402 | 0.8265 | 0.3366 | 556 | 0.9901 | 0.5163 | 1738 | 1.3161 | 1.907 | 2016 | 1.5071 | 2.232 |
| | **RL-MUL** | 399 | 0.7451 | 0.3345 | **551** | **0.9662** | **0.5081** | 1731 | 1.2192 | 1.903 | 1927 | 1.4339 | 2.140 |
| | **RL-MUL 2.0** | **401** | **0.7252** | **0.3360** | **551** | **0.9662** | **0.5081** | **1722** | **1.1875** | **1.887** | **1947** | **1.3923** | **2.148** |

Table 2. PE array (multiplier) area, timing, and power comparison.

| Preference | Method | 8-bit | | | | | | 16-bit | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | AND | | | MBE | | | AND | | | MBE | | |
| | | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) |
| Area | Wallace[1] | 175892 | 1.1347 | 145.14 | 208782 | 1.3302 | 178.67 | 601492 | 1.6693 | 495.67 | 650385 | 1.8543 | 548.16 |
| | GOMIL[17] | 170036 | 1.1237 | 141.11 | 206058 | 1.3154 | 176.36 | 574117 | 1.6017 | 470.92 | 618107 | 1.7403 | 522.13 |
| | SA | 168401 | 1.1237 | 140.08 | 204288 | 1.2711 | 175.17 | 575479 | 1.6216 | 472.59 | 640443 | 1.794 | 536.34 |
| | **RL-MUL** | 167312 | 1.0421 | 138.79 | **202926** | **1.2512** | **173.18** | 573709 | 1.5305 | 471.27 | 618107 | 1.6976 | 520.65 |
| | **RL-MUL 2.0** | **165950** | **1.0421** | **137.65** | **202926** | **1.2512** | **173.18** | **571394** | **1.5148** | **469.23** | **617971** | **1.6976** | **520.57** |
| Timing | Wallace[1] | 213345 | 1.0436 | 175.97 | 258016 | 1.1988 | 220.73 | 775001 | 1.5809 | 639.48 | 827503 | 1.6992 | 692.68 |
| | GOMIL [17] | 205378 | 1.0395 | 169.73 | 254475 | 1.1856 | 217.20 | 739591 | 1.5137 | 610.51 | 785896 | 1.6085 | 659.21 |
| | SA | 203607 | 1.0395 | 168.23 | 251955 | 1.1505 | 214.85 | 739318 | 1.5398 | 608.93 | 813339 | 1.6777 | 681.73 |
| | **RL-MUL** | 202722 | 0.9752 | 167.61 | **250185** | **1.1200** | **213.30** | 737139 | 1.4464 | 606.31 | **778678** | **1.5607** | **652.30** |
| | **RL-MUL 2.0** | **203607** | **0.9621** | **168.52** | **250185** | **1.1200** | **213.30** | **736731** | **1.4166** | **604.51** | **778269** | **1.5607** | **652.30** |
| Trade-off | Wallace[1] | 191214 | 1.1017 | 157.57 | 236566 | 1.2254 | 198.45 | 628322 | 1.6419 | 518.51 | 735028 | 1.7352 | 601.46 |
| | GOMIL[17] | 185357 | 1.0709 | 152.69 | 221857 | 1.2703 | 184.62 | 600947 | 1.5727 | 496.70 | 649908 | 1.6847 | 555.77 |
| | SA | 169014 | 1.1079 | 139.35 | 204901 | 1.2552 | 181.19 | 580110 | 1.5959 | 479.28 | 652019 | 1.7225 | 566.53 |
| | **RL-MUL** | 167925 | 1.0263 | 137.64 | **203539** | **1.2353** | **179.80** | 578339 | 1.4987 | 478.52 | 623691 | 1.6479 | 546.24 |
| | **RL-MUL 2.0** | **168606** | **0.9966** | **138.06** | **203539** | **1.2353** | **179.80** | **576024** | **1.4844** | **475.76** | **623555** | **1.6479** | **545.83** |

### 5.3 MAC Performance Comparison

The curves in Figure 11 for MACs and PE arrays consisting of MAC, along with the detailed comparisons in Table 4, demonstrate that RL-MUL 2.0 designs achieve superior performance compared to baselines. The RL-MUL 2.0 framework leads to up to a 13.4% area reduction under the minimum area constraint and a 15.6% decrease in delay under the minimum delay constraint for MACs. Similarly, PE arrays benefit from up to a 9.6% reduction in area and a 13.1% decrease in delay.

The hypervolume metrics, shown in Figure 12 for MACs and PE arrays implemented by MAC, highlight RL-MUL 2.0's efficiency. RL-MUL 2.0 generates an average of 81.7% more hypervolume than GOMIL for MACs and 80.9% for the arrays. When comparing the performance of RL-MUL 2.0 to the original RL-MUL, there is a 7.0% increase for MACs and a 7.9% increase for arrays.

Regardless of the multiplier cases or the MAC cases, it is observed that the advantage of RL-MUL 2.0 over the SA approach varies between 8-bit and 16-bit configurations. GOMIL outperforms SA in larger bit widths, suggesting that evolutionary algorithms may struggle with large design spaces due to their complexity. The improvement margins over the SA method vary between 8-bit and 16-bit designs, with GOMIL outperforming SA in larger bit widths. This suggests the evolutionary algorithm's limitations in addressing the expansive design space of larger bit widths. Additionally, the ILP-based method GOMIL simplifies the cost function by focusing solely on the area as the optimization objective. This approach limits its ability to achieve optimization gains in terms of delay consistently. In contrast, our approach employs multi-objective optimization, allowing us to attain Pareto-optimal results across both area and delay, demonstrating RL-MUL 2.0's consistent superiority across evaluations.

Table 3. MUL and MAC area, timing, and power comparison (commercial synthesis tool).

| Preference | Method | MUL | | | | | | MAC | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 8-bit | | | 16-bit | | | 8-bit | | | 16-bit | | |
| | | Area ($\mu m^2$) | Delay (ns) | Power (uW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (uW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) |
| Area | Wallace[1] | 332.5000 | 1.5710 | 312.2091 | 1612.2260 | 2.4991 | 2.0378 | 442.6240 | 1.7983 | 473.6569 | 1846.3060 | 2.5995 | 2.4445 |
| | GOMIL [17] | 326.4040 | 1.4703 | 313.3043 | 1570.2830 | 2.4098 | 1.9783 | 393.9460 | 1.6543 | 420.6916 | 1700.0060 | 2.5103 | 2.2788 |
| | **RL-MUL** | 322.6040 | 1.4578 | 196.7360 | 1550.5620 | 2.4004 | 1.9553 | 387.0640 | 1.6527 | 419.2456 | 1689.4040 | 2.5087 | 2.1896 |
| | **RL-MUL 2.0** | **320.5080** | **1.4367** | **196.3859** | **1538.0440** | **2.3857** | **1.9247** | **379.3560** | **1.6432** | **414.6432** | **1640.3560** | **2.4818** | **2.1752** |
| Timing | Wallace[1] | 464.1700 | 1.1124 | 399.9932 | 1904.5600 | 2.3413 | 2.2190 | 576.1560 | 1.2860 | 585.1385 | 2117.3600 | 2.4768 | 2.6243 |
| | GOMIL [17] | 440.7400 | 1.0789 | 378.6793 | 1770.2700 | 2.2812 | 2.1567 | 530.4040 | 1.2164 | 536.2910 | 1970.7940 | 2.4253 | 2.4654 |
| | **RL-MUL** | 442.7080 | 1.0698 | 380.8763 | 1750.5080 | 2.2723 | 2.1367 | 531.8530 | 1.2167 | 533.7445 | 1972.6300 | 2.3464 | 2.4578 |
| | **RL-MUL 2.0** | **428.7080** | **1.0567** | **366.7378** | **1744.7000** | **2.2660** | **2.1335** | **530.2460** | **1.2158** | **531.4654** | **1969.0400** | **2.339** | **2.4563** |
| Trade-off | Wallace[1] | 443.9540 | 1.1084 | 388.6333 | 1776.8800 | 2.3172 | 2.1315 | 554.0780 | 1.3003 | 562.4348 | 1979.8380 | 2.5050 | 2.5145 |
| | GOMIL [17] | 431.4280 | 1.1024 | 378.2456 | 1690.3600 | 2.2419 | 2.0643 | 514.1780 | 1.2153 | 531.8048 | 1846.5720 | 2.4177 | 2.3804 |
| | **RL-MUL** | 428.3760 | 1.0910 | 372.3638 | 1678.0340 | 2.2406 | 2.0541 | 509.3560 | 1.2015 | 527.8372 | 1816.3460 | 2.3935 | 2.3729 |
| | **RL-MUL 2.0** | **420.4060** | **1.0860** | **362.3837** | **1669.4000** | **2.2387** | **2.0452** | **504.5040** | **1.1893** | **521.3543** | **1807.3040** | **2.3912** | **2.3679** |

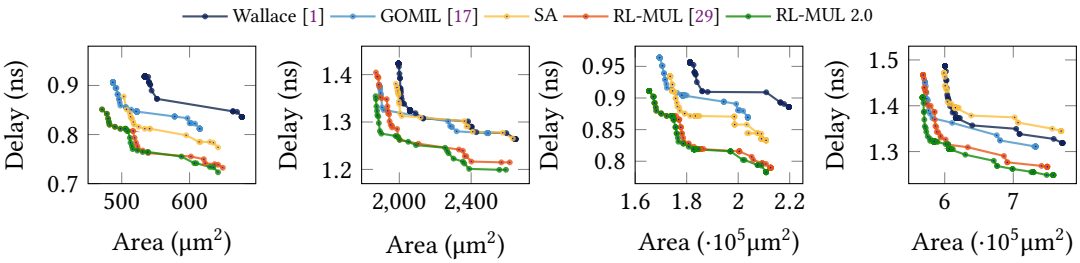


Fig. 11. Pareto-frontiers of the synthesis results on MACs and MAC-implemented PE arrays. From left to right: 8-bit MAC; 16-bit MAC; 8-bit MAC-implemented PE arrays; 16-bit MAC-implemented PE arrays.

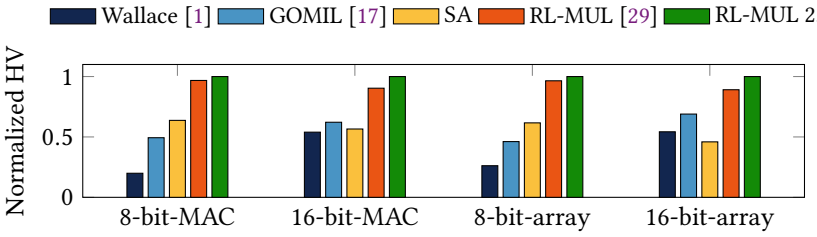


Fig. 12. Pareto-frontiers hypervolume comparison of MACs and MAC-implemented PE arrays.

Table 4. MAC and PE array (MAC) area, timing, and power comparison.

| Preference | Method | MAC | | | | | | PE-MAC | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 8-bit | | | 16-bit | | | 8-bit | | | 16-bit | | |
| | | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) | Area ($\mu m^2$) | Delay (ns) | Power (mW) |
| Area | Wallace[1] | 534 | 0.9182 | 0.5212 | 1995 | 1.4234 | 2.313 | 181340 | 0.9561 | 167.32 | 600471 | 1.4868 | 553.84 |
| | GOMIL [17] | 487 | 0.9063 | 0.4674 | 1889 | 1.3787 | 2.167 | 169491 | 0.9638 | 156.65 | 571053 | 1.451 | 529.78 |
| | SA | 503 | 0.8775 | 0.4846 | 1981 | 1.3807 | 2.289 | 173577 | 0.9342 | 160.37 | 598836 | 1.4714 | 537.73 |
| | **RL-MUL** | **471** | **0.8511** | **0.4584** | 1870 | 1.4046 | 2.169 | 165405 | 0.9112 | 154.41 | 568465 | 1.4673 | 523.36 |
| | **RL-MUL 2.0** | **471** | **0.8511** | **0.4584** | **1868** | **1.3545** | **2.158** | **165405** | **0.9112** | **154.38** | **567784** | **1.4178** | **522.58** |
| Timing | Wallace[1] | 677 | 0.8359 | 0.7392 | 2646 | 1.264 | 3.291 | 219678 | 0.8856 | 202.61 | 771664 | 1.3187 | 710.85 |
| | GOMIL [17] | 615 | 0.8119 | 0.6472 | 2494 | 1.2766 | 3.039 | 203743 | 0.8693 | 188.57 | 730670 | 1.3109 | 675.11 |
| | SA | 642 | 0.7737 | 0.6865 | 2632 | 1.2652 | 3.282 | 210825 | 0.8331 | 195.18 | 769893 | 1.3448 | 694.90 |
| | **RL-MUL** | 649 | 0.7324 | 0.6983 | 2568 | 1.2149 | 3.110 | 212596 | 0.7897 | 197.23 | 749533 | 1.2668 | 677.48 |
| | **RL-MUL 2.0** | **642** | **0.7231** | **0.6948** | **2594** | **1.1992** | **3.192** | **210825** | **0.7827** | **195.18** | **758385** | **1.2487** | **683.39** |
| Trade-off | Wallace [1] | 552 | 0.8727 | 0.5450 | 2060 | 1.3248 | 2.432 | 186038 | 0.9107 | 173.17 | 611502 | 1.3851 | 563.23 |
| | GOMIL[17] | 498 | 0.859 | 0.4857 | 2486 | 1.2766 | 3.014 | 172283 | 0.9161 | 160.37 | 582085 | 1.3729 | 535.26 |
| | SA | 518 | 0.8202 | 0.5043 | 2005 | 1.3181 | 2.335 | 177322 | 0.8799 | 166.99 | 604148 | 1.4084 | 545.24 |
| | **RL-MUL** | **482** | **0.8202** | **0.4774** | 1946 | 1.3016 | 2.329 | 168197 | 0.8799 | 158.36 | 588486 | 1.3429 | 529.68 |
| | **RL-MUL 2.0** | **482** | **0.8202** | **0.4774** | **2002** | **1.2625** | **2.350** | **176777** | **0.8309** | **166.89** | **578816** | **1.3233** | **529.38** |

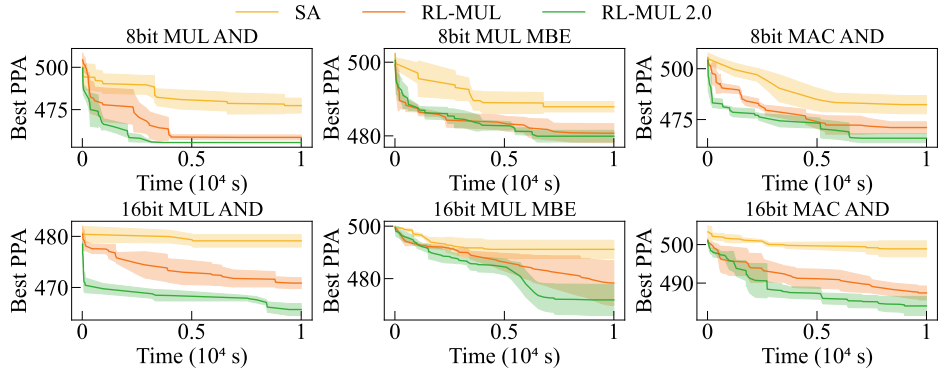


Fig. 13. Optimization trajectories for different methods illustrate the mean PPA ± standard error. The shaded areas represent the standard deviation of the PPA values.

### 5.4 Efficient and Stable Training

We conducted six experiments, each repeated three times, on the original RL-MUL, RL-MUL 2.0, and SA algorithms, with a fixed PPA weight across two bit-widths. These experiments are categorized into three groups: one focusing on AND-based MUL operations, another on MUL operations employing Booth encoding, and a third on MAC operations. The mean PPA values are represented by a solid line, with the standard deviation depicted as the surrounding shadow in Figure 13. Across all datasets, our RL methods consistently demonstrate superior performance, significantly outperforming SA. Particularly, the RL-MUL 2.0 demonstrates superior results and a more stable and efficient training process. Furthermore, it is observed that there exists a gap between the variance shadow caused by independent repetitions of RL-MUL and RL-MUL 2.0 experiments, especially in the 16-bit designs. So it implies that even if the DQN in RL-MUL were to support parallel agents within the 10,000s runtime, the best results achieved are still not as good as RL-MUL 2.0.

When comparing efficiency, runtime serves as a key metric, reflecting the algorithm's ability to explore the design space within a given time. Parallel processing allows RL-MUL 2.0 to utilize computational power to accelerate exploration. Notably, for the 8-bit MUL AND case, RL-MUL 2.0 achieves the optimal PPA value reached by RL-MUL in an average of 2,124 seconds. Similarly, RL-MUL 2.0 reaches this level in 6,275 seconds for the 8-bit MUL MBE case, 5,166 seconds for the 8-bit MAC AND case, 136 seconds for the 16-bit MUL AND case, 5,985 seconds for the 16-bit MUL MBE case, and 4,823 seconds for the 16-bit MAC AND case. These results indicate that RL-MUL 2.0 requires significantly less time to achieve the same performance level as RL-MUL, demonstrating the efficiency of its parallel training approach.

In addition to OpenROAD flow and OpenSTA, we conducted a cross-check synthesis using Synopsys Design Compiler [46] to validate the multipliers and MACs from our RL-MUL framework. The results, shown in Table 3, demonstrate that RL-MUL-designed multipliers consistently outperform baseline multipliers, achieving superior area, delay, and power metrics under commercial EDA tools, thereby confirming the effectiveness and robustness of our designs across synthesis environments.

## 6 CONCLUSION

In this research, we introduce a novel framework for optimizing multipliers through reinforcement learning. The framework utilizes an RL agent that adapts based on EDA tool feedback to engineer

multipliers achieving Pareto optimality. We demonstrate that multipliers and MACs designed by RL can Pareto-dominate multipliers that are produced by existing approaches. The obtained optimized multiplier and MACs can be further applied in the implementation of a larger module, such as a PE array. Looking ahead, we aim to broaden the application of our RL methodology to encompass more extensive datapath components, enhancing the scope and impact of our optimization efforts.

## REFERENCES


[1] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14–17, 1964.

[2] P. Stelling and V. Oklobdzija, "Implementing multiply-accumulate operation in multiplication time," in *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*, 1997, pp. 99–106.

[3] L. Dadda, "Some schemes for fast serial input multipliers," in *1983 IEEE 6th Symposium on Computer Arithmetic (ARITH)*, 1983, pp. 52–59.

[4] J. Fadavi-Ardekani, "M*n booth encoded multiplier generator using optimized wallace trees," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 1, no. 2, pp. 120–125, June 1993.

[5] 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 *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2005, pp. 73–76 Vol. 1.

[6] K. Bickerstaff, M. Schulte, and E. Swartzlander, "Reduced area multipliers," in *Proceedings of International Conference on Application Specific Array Processors (ASAP '93)*, 1993, pp. 478–489.

[7] X.-V. Luu, T.-T. Hoang, T.-T. Bui, and A.-V. Dinh-Duc, "A high-speed unsigned 32-bit multiplier based on booth-encoder and wallace-tree modifications," in *2014 International Conference on Advanced Technologies for Communications (ATC 2014)*, 2014, pp. 739–744.

[8] M. A. Basiri M and N. M. Sk, "An efficient hardware-based higher radix floating point mac design," *ACM Trans. Des. Autom. Electron. Syst.*, 2014.

[9] C.-W. Tung and S.-H. Huang, "A high-performance multiply-accumulate unit by integrating additions and accumulations into partial product reduction process," *IEEE Access*, vol. 8, pp. 87 367–87 377, 2020.

[10] S. Zhang, J. Gu, S. Yin, L. Liu, and S. Wei, "A multiple-precision multiply and accumulation design with multiply-add merged strategy for ai accelerating," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2021, pp. 229–234.

[11] N. Shavit, I. Stanger, R. Taco, M. Lanuzza, and A. Fish, "A 0.8-v, 1.54-pj/940-mhz dual-mode logic-based 16×16-b booth multiplier in 16-nm finfet," *IEEE Solid-State Circuits Letters*, vol. 3, pp. 314–317, 2020.

[12] D. Jangalwa, M. Nagabushanam, and M. C. Parameshwara, "Design and analysis of 8-bit multiplier for low power vlsi applications," in *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*, 2022, pp. 1–5.

[13] D. B. R, D. Shylu Sam, M. G, D. Jayanthi, S. I. J, S. Babafakruddin, and S. E. G. V, "Design of low power pass transistor logic based adders for multiplier in 90nm cmos process," in *2023 4th International Conference on Signal Processing and Communication (ICSPC)*, 2023, pp. 206–210.

[14] V. Oklobdzija, D. Villeger, and S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 294–306, 1996.

[15] C. Martel, V. Oklobdzija, R. Ravi, and P. Stelling, "Design strategies for optimal multiplier circuits," in *Proceedings of the 12th Symposium on Computer Arithmetic*, 1995, pp. 42–49.

[16] P. Stelling, C. Martel, V. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 273–285, 1998.

[17] W. Xiao, W. Qian, and W. Liu, "Gomil: Global optimization of multiplier by integer linear programming," pp. 374–379, 2021.

[18] 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 *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2007, pp. 609–615.

[19] J. Liu, S. Zhou, H. Zhu, and C.-K. Cheng, "An algorithmic approach for generic parallel adders," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2003, pp. 734–740.

[20] 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 *ACM/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–8.

[21] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Efficient synthesis of compressor trees on fpgas," in *2008 Asia and South Pacific Design Automation Conference*, 2008, pp. 138–143.

[22] H. Parandeh-Afshar, A. Neogy, P. Brisk, and P. Ienne, "Compressor tree synthesis on commercial high-performance fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, dec 2011.

[23] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for fpgas," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.
[24] H. Geng, Y. Ma, Q. Xu, J. Miao, S. Roy, and B. Yu, "High-speed adder design space exploration via graph neural processes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 8, pp. 2657–2670, 2022.
[25] Y. Ma, S. Roy, J. Miao, J. Chen, and B. Yu, "Cross-layer optimization for high speed adders: A pareto driven machine learning approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 38, no. 12, pp. 2298–2311, 2019.
[26] 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 *ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 853–858.
[27] 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 *ACM/IEEE Design Automation Conference (DAC)*, 2020.
[28] N. Siddharth, P. Geraldo, H. Corey, T. Yang, K. Brucek, and H. Ren, "Transsizer: A novel transformer-based fast gate sizer," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022.
[29] D. Zuo, Y. Ouyang, and Y. Ma, "Rl-mul: Multiplier design optimization with deep reinforcement learning," in *ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.
[30] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.
[31] V. Mnih, A. P. Badia *et al.*, "Asynchronous Methods for Deep Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, vol. 48, 2016, pp. 1928–1937.
[32] P. Winder, *Reinforcement learning*. O'Reilly Media, 2020.
[33] H.-C. Jang, Y.-C. Huang, and H.-A. Chiu, "A study on the effectiveness of a2c and a3c reinforcement learning in parking space search in urban areas problem," in *International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2020, pp. 567–571.
[34] M. Sewak, *Deep reinforcement learning*. Springer, 2019.
[35] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.
[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
[37] J. Peng and R. J. Williams, "Incremental multi-step Q-learning," in *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 226–232.
[38] J. Zhang, Q. Gao, Y. Guo, B. Shi, and G. Luo, "Easymac: Design exploration-enabled multiplier-accumulator generator using a canonical architectural representation," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2022, pp. 647–653.
[39] 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*, pp. 1105–1110, 2019.
[40] Nangate Inc., "Open Cell Library v2008_10 SP1," 2008. [Online]. Available: http://www.nangate.com/openlibrary/
[41] Parallax Software Inc., "OpenSTA," https://github.com/The-OpenROAD-Project/OpenSTA.
[42] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.
[43] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/~alanmi/abc/.
[44] Geoff Hinton, "RMSProp," https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
[45] 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*. Springer, 2007, pp. 862–876.
[46] Synopsys, Inc., "Design Compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.