Multiplier Optimization by Exploring Logical Correlations Among Partial Products

Ao Liu[1], Siting Liu[2], Hui Wang[1], Cheng Chen[1], Jie Han[3], Honglan Jiang[1] *
[1]School of Integrated Circuits, Shanghai Jiao Tong University, Shanghai, China
[2]School of Information Science and Technology, ShanghaiTech University, Shanghai, China
[3]Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada
2017la@sjtu.edu.cn; liust@shanghaitech.edu.cn; ihuiwang@sjtu.edu.cn; chencheng_sjtu@sjtu.edu.cn; jhan8@ualberta.ca; honglan@sjtu.edu.cn

Abstract

Small-bit-width multipliers are essential arithmetic components in modern artificial intelligence (AI) workloads and have therefore been extensively optimized by emerging optimization frameworks. However, existing multiplier optimization frameworks usually overlook the logical correlations among partial products (PPs), leading to potential logical redundancy. In this work, the logical correlations among PPs are analyzed, revealing the inherent impossible value combinations (IVCs), which can be used to simplify the full adders (FAs) and half adders (HAs) in the PP accumulation circuit. To fully exploit this characteristic, a reinforcement learning (RL)-based optimization framework, denoted as GRmul, is proposed by modeling the PP accumulation tree as a directed acyclic graph (DAG). The experiments based on open-source synthesis tools and the NanGate 45nm library show that GRmul achieves area-delay product (ADP) reductions of 1.4% to 20.4% compared to state-of-the-art optimization frameworks for the optimization of 4-, 8-, and 12-bit multipliers. Furthermore, evaluated with commercial synthesis tools on a 28nm CMOS technology, 4- to 16-bit multipliers generated by GRmul can outperform the corresponding DesignWare designs.

Keywords

multiplier optimization, partial product, logical correlation, directed acyclic graph, reinforcement learning

1 Introduction

Multiplication is the foundational arithmetic operation in modern artificial intelligence (AI) applications, such as text generation and autonomous driving. In these applications, matrix multiplication typically dominates hardware area and power consumption. With the advancement of quantization techniques, small-bit-width computation has become increasingly prevalent in AI accelerators [2, 5]. Therefore, optimizing small-bit-width multipliers is crucial.

The operation of a combinational multiplier involves three stages, partial product (PP) generation, PP accumulation, and a final carry propagate addition (CPA) [7]. For small-bit-width operands, PPs are typically generated through bitwise AND operations. The resulting PP array is then compressed into two rows during the accumulation stage, followed by a CPA to yield the final product. For operands with relatively large bit widths, radix-4 Booth encoding [12] is commonly employed. As bitwise AND and Booth encoding techniques are well established for PP generation, and the research on CPA largely overlaps with that on general adders, recent studies in multiplier optimization have primarily focused on the PP accumulation stage [22, 24, 26, 27].

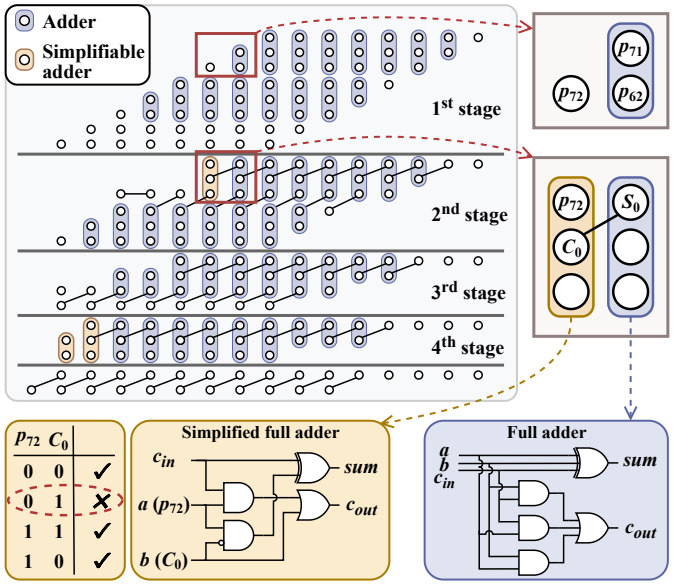


Figure 1: PP accumulator of an unsigned 8-bit Wallace multiplier, where certain adders are simplified by leveraging IVCs.

Generally, full adders (FAs) and half adders (HAs) are the basic building blocks of a PP accumulator. In the rest of this paper, the term 'adder' is used to denote an HA or an FA, unless a carry-propagate adder is explicitly specified. The primary objective in PP accumulator optimization is to find the allocation and interconnection scheme of adders with the best hardware performance. Focusing on this problem, recent studies have explored various optimization paradigms, including integer linear programming (ILP) [25], reinforcement learning (RL) [9, 22, 27], and differentiable optimization [24, 26]. Although these methodologies enhance hardware performance, their failure to fully exploit the logical correlations among PPs leaves substantial potential for further optimization. Specifically, by carefully assigning PPs to the adder inputs, certain adders can be simplified by leveraging the impossible value combinations (IVCs) of PPs.

Figure 1 shows the dot notation of the PP accumulator in an unsigned 8-bit Wallace tree multiplier [21]. By exhaustively traversing the input space of the Wallace multiplier and recording the input patterns of each adder, it can be observed that the input space of certain adders is not fully covered; these adders are referred to as *simplifiable adders*. **This phenomenon arises from the logical correlations among PPs.** Taking the simplifiable FA in the second stage as an example, when its second input $C_0 = 1$, its first input p_{72} must be 1. Here, $p_{ij} = x_i$ AND y_j, where x_i (y_j) is the i-th (j-th) least significant bit of the multiplier input

*Corresponding author: Honglan Jiang.
This work was supported in part by the National Natural Science Foundation of China under Grant 62374108 and in part by the Natural Sciences and Engineering Research Council of Canada under Grant RES0048688, Grant RES0051374, and Grant RES0054326.

ICCAD '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834274

operand. C_0 is the carry-out bit of $p_{71}+p_{62}$, i.e., $C_0 = p_{71}$ AND p_{62}. Thus, $C_0 = 1$ only when $p_{71} = p_{62} = 1$, which implies $x_7 = y_2 = 1$ and thus $p_{72} = 1$. This indicates that the corresponding FA can be simplified by leveraging the "do not care" input cases, as illustrated in the lower-left corner of Figure 1. It should be noted that advanced synthesis tools can typically identify these types of characteristics and implement such adder simplifications; however, they may not restructure the interconnections among adders to create new simplification opportunities. For instance, when p_{72} and C_0 are initially assigned to different FAs, synthesis tools may not remap them to the same FA for logic simplification. We have verified that assigning p_{72} and C_0 to different FAs degrades the hardware performance of the resulting multiplier.

To fully leverage the logical correlations among PPs to optimize the multiplier circuit, this work analyzes the underlying mechanism of the circuit simplification enabled by IVCs of PPs. A directed acyclic graph (DAG)-based model is then proposed to capture the logical correlations among PPs and formulate the multiplier optimization problem. Finally, an RL-based multiplier optimization framework, denoted as GRmul[1], is developed to optimize the allocation and interconnection scheme of HAs and FAs. The contributions of this work are as follows.

- The IVCs arising from the logical correlations among PPs are analyzed. We demonstrate that this characteristic can be used to simplify adders and thus optimize the multiplier circuit.
- A DAG representation of the PP accumulation process is introduced to enable a systematic analysis of the logical correlations among PPs.
- An RL-based framework incorporating the proposed DAG model is developed to explore logical correlations among PPs and enhance the hardware performance of multipliers.
- Experiments show that for 4- to 12-bit multipliers, GRmul can outperform existing optimization frameworks. Moreover, the 4- to 16-bit multipliers generated by GRmul can outperform industrial and manual designs.

2 Background and Motivation

2.1 Multiplier Optimization Methodologies

Focusing on the PP accumulator, several optimization frameworks have been proposed to enhance the hardware performance of multipliers [9, 22, 24–27]. As a pioneer, GOMIL [25] employs ILP to optimize the allocation of HAs and FAs across different accumulation stages. As algorithmic advancements, RL-MUL [27] and ArithTreeRL (ATRL) [9] leverage RL algorithms to achieve superior performance over GOMIL. Wang et al. [22] observed that the action and reward designs in RL-MUL often lead to local optima, and thus developed MUTE by combining RL-MUL with a genetic algorithm. While these methods primarily focus on adder allocation, DOMAC [26] employs a differentiable approach to further optimize the adder interconnection. Moreover, ARITH-DAS (ADAS) [24] introduces a multi-relational DAG model for adder interconnections to enable finer-grained optimization, resulting in state-of-the-art performance.

2.2 Motivation

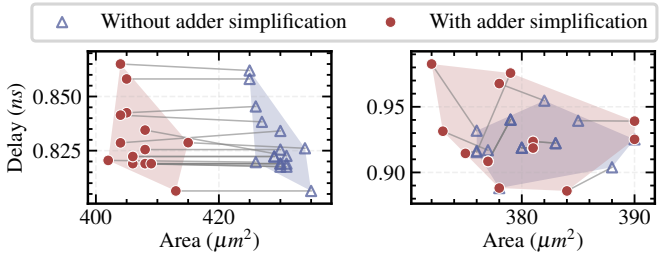
Previous works suffer from two limitations, ❶ absence of mechanisms to handle IVCs, and ❷ lack of PP-correlation-aware optimization.

Limitation 1. Taking the open-source versions of MUTE and ADAS as examples, we statistically analyze the proportion of simplifiable adders in the PP accumulators of unsigned 8-bit multipliers generated by these frameworks. The results are summarized in Table 1, where

[1]GRmul is open-source and available at https://github.com/alliuo/GRmul

Table 1: Statistics of simplifiable adders in PP accumulators.

	MUTE-AND		MUTE-Booth		ADAS-AND		ADAS-Booth	
	HA	FA	HA	FA	HA	FA	HA	FA
Total	6.9	38.8	4.1	30.8	7.0	38.0	6.7	27.3
Simplifiable	0.4	0.6	1.2	6.2	0.0	2.0	0.7	3.3
Proportion (%)	5.4	1.6	28.6	20.0	0.0	5.3	10.4	12.1



(a) Default MUTE synthesis (b) Aggressive synthesis

Figure 2: Changes in area and delay of multipliers resulting from adder simplification (area-targeted optimization).

AND and Booth (radix-4) denote the PP generation methods, and the reported adder counts are averaged over all generated multipliers under the official default configurations. As observed, even these fully optimized multipliers retain adders that can be further simplified, especially in multipliers based on Booth encoding. For MUTE-Booth, we further evaluate the hardware performance improvement achieved by simplifying the simplifiable adders. The adder simplification was performed using Karnaugh map (K-map) minimization, where IVCs are treated as "do not care" cases. Synthesis was conducted under the area-targeted environment of MUTE (NanGate 45nm library, 5ns clock period). As shown in Figure 2(a), adder simplification reduces the area with marginal delay changes. Notably, advanced synthesis tools have inherent optimization capabilities for these simplifiable adders, but their effectiveness is limited. To illustrate this, we replaced the MUTE synthesis script with a more aggressive one that includes logic flattening and global optimization. As shown in Figure 2(b), the performance of some multipliers improves little or even degrades after adder simplification, indicating that the synthesis tool can automatically optimize some simplifiable adders. However, fully simplifying all such adders improves the achievable performance, yielding multipliers with the lowest delay or area. Hence, incorporating adder simplification into multiplier optimization is expected to further improve hardware performance.

Limitation 2. While previous studies such as DOMAC and ADAS have underscored the impact of the adder interconnection scheme on the critical path delay of multipliers, they overlook how the logical correlations among PPs drive the performance variation across different interconnection schemes. Taking the Wallace tree shown in Figure 1 as an example, the column containing p_{72} has six PPs in the second stage. When these PPs are accumulated using two parallel FAs, there are $C(6,3) = 20$ possible assignment schemes. Among these schemes, assigning p_{72} and C_0 to the same FA enables logic simplification, reducing the multiplier area by approximately 0.5%. Existing optimization frameworks typically treat PPs as generic nodes and ignore their logical properties. As a result, even if an interconnection scheme enabling logical simplification is discovered during exploration, the lack of PP correlation features prevents the framework from attributing these performance gains to the underlying logic or generalizing the insights to similar scenarios.

To address these limitations, this work proposes a multiplier optimization framework with two novel features. First, input space coverage checking and circuit simplification for adders are integrated into the

optimization flow. Second, the logical correlations among PPs are incorporated into the optimization problem formulation so that they can be effectively exploited to improve multiplier hardware performance.

3 Methods

This section first presents the methods for detecting IVCs of PPs and simplifying adders. A PP accumulation model is then introduced to formulate the logical correlations among PPs. Finally, the implementation details of the proposed GRmul framework are described.

3.1 Adder Simplification based on IVCs of PPs

For a multiplier with small bit width (e.g. 8 bits), exhaustively traversing its input space is a straightforward approach for collecting all possible PP value combinations. However, for larger bit widths, this approach becomes impractical due to the excessively large input space. To address this issue, we formulate the IVC detection for a given PP set as a Boolean satisfiability problem. Multiplier inputs are modeled as unconstrained Boolean variables, and each PP is represented by a Boolean expression recursively derived from its gate-level logic. For example, the sum bit of an HA is represented as the XOR of the Boolean expressions of its two inputs. We enumerate all possible value combinations of the given PP set and evaluate the feasibility of each combination using the Z3 SMT solver [4]. Specifically, for each combination, constraints are added to fix the selected PPs to the specified values. A SAT result indicates that the combination is reachable under at least one assignment of the multiplier inputs, whereas an UNSAT result confirms that it is an IVC.

As both HAs and FAs are small circuits with only a few logic gates, K-map minimization can be directly applied to derive simpler implementations by treating IVCs as "do not care" cases. However, to automate the simplification process while ensuring optimal hardware performance under a technology library, we adopt a synthesis-based approach. Specifically, the function of a simplifiable adder is described using the Verilog HDL *case* statement, where all "do not care" branches are assigned to 'x.' During synthesis, synthesis tools can automatically map these 'x' values to either '0' or '1' to achieve the most efficient implementation. To validate this synthesis-based simplification, we examined a representative set of commonly occurring simplifiable adders and found that the resulting circuits were consistent with those obtained using K-maps.

3.2 PP Accumulation Model

As illustrated in Figure 1, IVCs for a set of PPs arise from the logical correlations among them. To capture such correlations, the logic path of each PP should be modeled. Based on the computational logic, PPs can be classified into three categories as follows.

- Initial PPs, whose values are computed directly from the multiplier inputs.
- Sum bits of adders, whose values are computed using XOR operations on the adder inputs.
- Carry-out bits of adders, whose values are mainly computed using AND operations on the adder inputs.

Accordingly, as illustrated in Figure 3, we propose a DAG model consisting of three types of nodes and two types of edges to capture the logical correlations among PPs within the PP accumulator. In this model, nodes represent PPs, and edges represent computational relationships directed from the inputs of an adder to its outputs. If multiple nodes are traced back to the same preceding nodes, they share overlapping computational logic, which potentially leads to conflicts in their values. It is noteworthy that certain initial PPs may also be correlated because their computations involve the same multiplier input signals. To handle multipliers with different PP generation approaches uniformly, we represent the relationship between an initial PP and the multiplier inputs as a node feature rather than as edges; further details are given in Section 3.3.3.

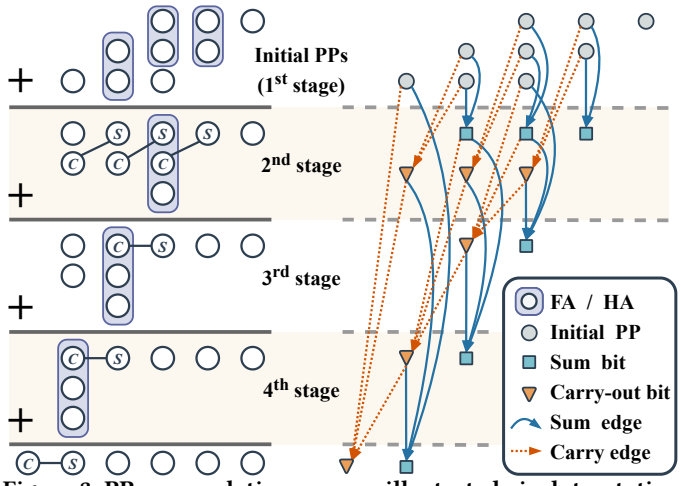


Figure 3: PP accumulation process illustrated via dot notation (left) and DAG model (right). S: sum, C: carry-out.

3.3 Proposed RL-based Framework

3.3.1 Reinforcement Learning Fundamentals. Being good at solving sequential decision-making problems, RL algorithms have been widely applied to circuit design and optimization in recent years [9, 13, 14, 22, 27]. In RL, the *agent* is the decision maker that interacts with an *environment*. Considering an episodic Markov decision process with T steps, at step t ($t = 0, 1, \ldots, T-1$), the agent observes a *state* s_t and selects the *action* a_t based on a *policy* π. The policy gives the conditional probability $\pi(a_t \mid s_t)$ of taking each action in a given state. After step t, the environment provides the next state s_{t+1} and a *reward* r_{t+1} that provides the immediate feedback to the agent. The objective of RL is to find an optimal policy π^* that maximizes the expectation of *return* (i.e., cumulative discounted reward), defined as

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T-1} \gamma^t r_{t+1}\right], \tag{1}$$

where the trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, \ldots, a_{T-1}, r_T, s_T)$ is the time-ordered sequence generated by following the policy π, and $\gamma \in [0, 1]$ is the discount factor that controls the trade-off between immediate and future rewards.

In this work, RL is employed to manage the problem complexity by formulating the multiplier construction as a sequential decision-making process. Specifically, changing the interconnection of an adder affects the logic paths of its output PPs and all downstream PPs, thereby altering whether and how the associated adders can be simplified. Therefore, rather than modifying an existing multiplier, incrementally constructing it from scratch according to the computation order may provide a more systematic approach to exploring better designs.

3.3.2 Overview of GRmul. Figure 4 shows an overview of the proposed GRmul framework. The initial state s_0 corresponds to the PP array before accumulation. Multipliers with various PP generation schemes can be implemented by modifying the initial PP array. The rightmost column containing more than one unaccumulated PP is identified as the target column to be compressed. At each step, the RL agent selects two candidate PPs (i.e., PPs in the target column that have not yet been accumulated) to add. If one of the selected PPs is the sum bit of an HA, the agent further decides whether to replace the original HA with

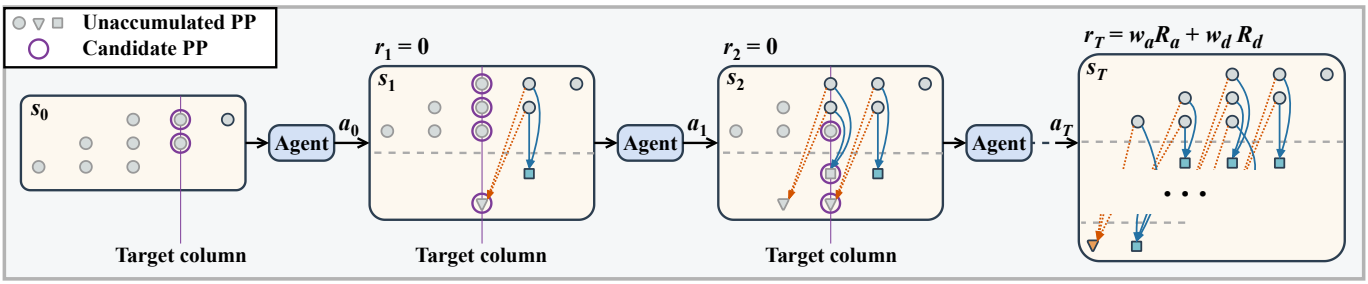


Figure 4: Overview of the GRmul framework.

an FA. The adder implementation is determined based on the IVCs identified for the selected PP set. The process terminates after step $T-1$ in state s_T, when every column in the PP array contains only one PP, indicating that the accumulation is complete. The intermediate rewards r_t for $1 \le t < T$ are set to zero. The terminal reward r_T is computed from the synthesized area and delay of the generated multiplier relative to the corresponding baseline values.

In GRmul, the PP accumulation process proceeds by repeatedly accumulating the rightmost column until every column contains only one PP. This ordering aligns with the fact that carry signals propagate from the least significant bit to the most significant bit. Consequently, the accumulation schemes of the leftward columns do not affect those of the rightward columns, ensuring that no valid accumulation sequence is omitted. Moreover, this PP accumulation process implicitly includes the final CPA stage of the multiplier, implemented using a ripple carry adder (RCA). We optimize these stages jointly because IVCs may also exist in the CPA stage. To prevent the RCA from becoming a timing bottleneck, we propose to identify the longest carry chain that does not involve simplified adders in the final multiplier and implement it using the Verilog HDL addition operator '+.' This enables synthesis tools to optimize the CPA as per specific constraints.

3.3.3 State Representation. The state is represented by the proposed DAG, where each node is encoded by a feature vector consisting of six features, defined as follows.

- *Node type*: It indicates whether the node represents an initial PP, a sum bit, or a carry-out bit.
- *Level*: It records which accumulation stage the PP is located at. The level of the initial PPs is defined as 1, and the output bits generated by accumulating PPs with a maximum level of L are assigned to level $(L+1)$.
- *Weight*: It denotes the column index of the PP in the PP array, where the rightmost column is indexed as 0.
- *In-degree*: The number of incoming edges to the node. A node with an in-degree of 2 is an output bit of an HA, while an in-degree of 3 indicates an output bit of an FA.
- *Out-degree*: The number of outgoing edges from the node. An out-degree of 0 indicates that the PP has not yet been accumulated.
- *Dependent input bit set (DIBS)*: It is the set of input bits of the multiplier that determines the value of the PP. For initial PPs, the DIBS is determined by the PP generation method. For other PPs, the DIBS is the union of the DIBSs of its parent nodes. DIBS is encoded as a binary vector with the length equal to the number of multiplier input bits.

3.3.4 Action Design. The action is represented by an ordered three-element tuple, denoted as $(node_0,\ node_1,\ merge)$, where $node_0$ and $node_1$ correspond to the PPs to be summed. The parameter $merge$ can be 0 or 1, and is valid only when $node_0$ is the sum bit of an HA. An illustrative

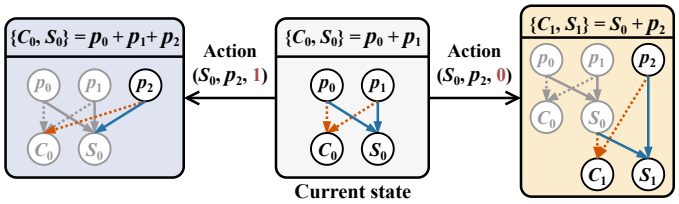


Figure 5: Action examples, where $\{C, S\}$ denotes the concatenation of signals C and S.

example of the *merge* action is provided in Figure 5. S_0 is the sum bit of an HA. The accumulation of S_0 and p_2 is performed by replacing the original HA with an FA when $merge = 1$. Otherwise, S_0 and p_2 are added using another HA.

Based on this action design, the process of using an FA to add three PPs is decomposed into two sequential steps, which reduces the decision complexity at each step. Specifically, given k candidate PPs, the number of possible choices for selecting two or three PPs to add together is

$$n'_a = C(k,2) + C(k,3) = \frac{k^3 - k}{6}, \tag{2}$$

which grows cubically with k. For the action design employed in this work, the selection of an ordered node pair $(node_0,\ node_1)$ is required. This is because merging $node_0$ into $node_1$ and merging $node_1$ into $node_0$ result in different circuit structures when both $node_0$ and $node_1$ are HA sum bits. When the decision for the *merge* parameter is included, the total number of possible choices is given by

$$n_a = 2 \times P(k,2) = 2k(k-1), \tag{3}$$

which grows quadratically with k. It can be readily derived that $n_a < n'_a$ as long as $k > 11$. Additionally, since *merge* is valid only when $node_0$ corresponds to an HA sum bit, certain invalid actions are masked, thereby further reducing the decision space.

Once an action is selected, the corresponding adder implementation is determined based on the IVCs identified for the selected PPs. A delay-aware strategy is then applied to map the selected PPs to the adder inputs. Specifically, the pin-to-pin delays are accumulated along the adder interconnection paths, and the input assignment that minimizes the critical path delay is selected.

3.3.5 Reward Design. The reward is designed as

$$r_t = \begin{cases} 0, & 1 \le t < T \\ w_a R_a + w_d R_d, & t = T \end{cases}, \tag{4}$$

where w_a and w_d denote the weighting coefficients for area and delay, respectively. R_a and R_d are defined as

$$R_a = \frac{area_0 - area}{area_0} \tag{5}$$

and

$$R_d = \frac{delay_0 - delay}{delay_0}, \tag{6}$$

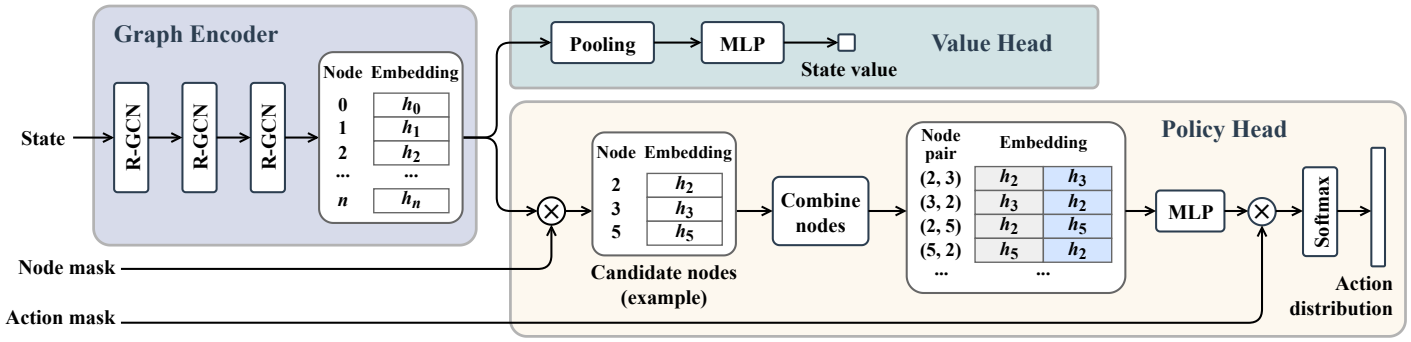


Figure 6: Architecture of the proposed RL agent.

where *area* and *delay* refer to the synthesis results for the area and critical path delay of the multiplier. $area_0$ and $delay_0$ are constant values corresponding to the baseline area and delay, used for normalization.

The zero intermediate reward scheme [13] is adopted to ensure that the optimization objective remains strictly aligned with the global task, i.e., the final synthesis outcomes. Accordingly, the discount factor γ in (1) is set to 1, which makes the cumulative return equal to the terminal reward r_T and invariant to the total number of steps. However, the sparse reward design may introduce training challenges; the corresponding mitigation strategy is introduced in Section 3.3.7.

3.3.6 RL Agent Design. As illustrated in Figure 6, we adopt an actor-critic architecture comprising a shared graph encoder and two task-specific heads, i.e., a policy head and a value head.

The proposed DAG model contains two types of edges (relations) and can be processed by the relational graph convolutional network (R-GCN) [15]. R-GCN is an extension of graph convolutional networks (GCNs) [8] designed for multi-relational graphs. It performs relation-specific message passing by applying distinct weight matrices for each relation and aggregating normalized neighbor contributions. A typical forward-pass update for node i in layer $(l+1)$ of the R-GCN is given by

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right), \tag{7}$$

where $h_i^{(l)}$ is the embedding of node i in layer l, σ is an element-wise activation function (e.g. ReLU), $\mathcal{R}$ is the set of relation types, $\mathcal{N}_i^r$ is the set of neighbors of node i connected via relation $r \in \mathcal{R}$, $c_{i,r}$ is a normalization constant, $W_r^{(l)}$ is the relation-specific weight matrix for relation r in layer l, and $W_0^{(l)}$ is the self-loop weight matrix in layer l.

Three R-GCN layers serve as an encoder that distills the information contained in the state DAG. The initial node representation is the feature vector defined in Section 3.3.3. In each layer, a node updates its representation by summing incoming messages grouped by edge type and a self-loop term, followed by a ReLU nonlinearity, as formulated in (7). The obtained node embeddings are denoted as h_i for $i \in \{1, \ldots, n\}$.

The value head comprises a pooling layer followed by a multi-layer perceptron (MLP). The pooling layer aggregates information across the entire graph to produce a fixed-size summary vector, and the MLP maps this summary to an estimate of the state value.

Only candidate PPs can be selected by the policy head; therefore, a node mask is applied to the node embeddings produced by the graph encoder to filter out non-candidate PP nodes. The embedding for the ordered node pair $(node_0,\ node_1)$ is then constructed by concatenating the embeddings of $node_0$ and $node_1$, as illustrated in Figure 6. Denote the number of node pairs as n_{pair}. These embeddings are passed through an MLP to produce a logit vector of length $2n_{pair}$, representing the action space for $(node_0,\ node_1,\ merge)$. The logit vector is subsequently masked to eliminate illegal actions, i.e., actions in which $node_0$ is not the sum bit of an HA while $merge = 1$. This masking is implemented by replacing the logit value corresponding to each illegal action with a large negative value. Finally, the masked logit vector is normalized with a Softmax function to obtain the action distribution.

3.3.7 RL Agent Training. Let ω, θ_p and ϕ_v denote the parameters of the shared graph encoder, the policy head, and the value head, respectively. We define the complete policy and value parameter sets as $\theta = (\omega, \theta_p)$ and $\phi = (\omega, \phi_v)$. Accordingly, the policy and value functions are written as $\pi_\theta(a \mid s)$ and $V_\phi(s)$.

Given trajectories collected under the old policy $\pi_{\theta_{\text{old}}}$, the probability ratio between the new and old policies is defined as

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\,. \tag{8}$$

Following PPO [17], we update the policy parameters θ by optimizing the clipped surrogate objective

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\left(\rho_t(\theta)\hat{A}_t,\ \text{clip}(\rho_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t\right)\Big], \tag{9}$$

where ε is the clipping threshold, and $\hat{A}_t$ is the estimated advantage at step t, computed using the generalized advantage estimation (GAE) [16]

$$\hat{A}_t = \sum_{k=0}^{T-t-1}(\gamma\lambda)^k\delta_{t+k}\,, \tag{10}$$

where $\lambda \in [0, 1]$ is the decay parameter, and δ_t is the temporal-difference error calculated using the old value function $V_{\phi_{\text{old}}}$

$$\delta_t = r_{t+1} + \gamma V_{\phi_{\text{old}}}(s_{t+1}) - V_{\phi_{\text{old}}}(s_t)\,. \tag{11}$$

The value parameters ϕ are updated by minimizing the mean squared error given by

$$L^{\text{V}}(\phi) = \mathbb{E}_t\left[(V_\phi(s_t) - G_t)^2\right], \tag{12}$$

where $G_t = \hat{A}_t + V_{\phi_{\text{old}}}(s_t)$ is the estimated return.

To encourage exploration, an entropy bonus $S(\pi_\theta(\cdot \mid s_t))$ is added. The overall loss function, minimized with respect to the parameters $(\omega, \theta_p, \phi_v)$, is defined as

$$\mathcal{L}(\omega, \theta_p, \phi_v) = -L^{\text{CLIP}}(\theta) + c_1L^{\text{V}}(\phi) - c_2\mathbb{E}_t\left[S(\pi_\theta(\cdot \mid s_t))\right], \tag{13}$$

where c_1 and c_2 are scalar coefficients that weight the value and entropy terms. Gradients from both the policy and value losses are backpropagated through the shared encoder parameters ω. The parameter updates are performed using the Adam optimizer, with minibatch sampling and global gradient-norm clipping to ensure training stability.

To mitigate the difficulty of learning from sparse reward signals (see (4)), we employ a two-stage schedule for the GAE parameter λ in (10). During the initial training phase, we set $\lambda = 1$, so that G_t reduces

Table 2: Hyperparameter settings for GRmul.

Symbol	Description	Value
w_a, w_d	Coefficients for area and delay in (4)	0.5, 1
ε	The clipping threshold in (9)	0.2
c_1, c_2	The scalar coefficients in (13)	0.5, 0.01
γ	The discount factor in (1), (10), and (11)	1
λ	The decay parameter in (10)	1, 0.95[a]
B	The size of the rollout buffer	$16 \times m^2$
b	The size of the minibatch	$4 \times m^2$
K	The number of epochs	8
α	The learning rate	1×10^{-4}

[a] Adjusted during training.

Table 3: Training time (hours).

Synthesis toolchain	4-bit	8-bit	12-bit	16-bit
Open-source	3.46	9.91	12.20	17.71
Commercial	46.48	48.28	48.35	48.96

to the Monte Carlo return, eliminating its dependence on potentially inaccurate intermediate value estimates. After the initial phase, we set $\lambda = 0.95$ to leverage the learned value function for variance reduction.

4 Experiments

To provide a comprehensive evaluation, we employ two distinct synthesis toolchains for GRmul, i.e., an open-source toolchain for comparison with existing multiplier optimization frameworks, and a commercial toolchain for comparison with industrial multiplier designs.

The open-source toolchain uses Yosys [23], ABC [20], and OpenROAD [1], targeting the NanGate 45nm library [6]. The synthesis settings follow ATRL [9], involving logic flattening and global optimization. The only modification is the removal of the '-fast' option from the ABC logic synthesis command to enable more aggressive optimization. The hyperparameters of GRmul for optimizing m-bit multipliers are summarized in Table 2. They were selected based on empirical performance and convergence on representative cases.

The commercial toolchain uses Synopsys Design Compiler (DC) with a 28nm CMOS technology under typical operating conditions with a temperature of 25°C and a supply voltage of 0.9V. The *compile_ultra* command was used, with area specified as the optimization objective. Accordingly, w_a and w_d in Table 2 are adjusted to 1 and 0.2, respectively.

All experiments were conducted on the Intel Xeon Gold 6226R CPU and the NVIDIA RTX 3090 GPU. Using 10 parallel environment instances, we measured the wall-clock time for 9,000 training episodes, as summarized in Table 3. The synthesis environment has a significant impact on the training time.

4.1 Comparison with Optimization Frameworks

To compare the proposed GRmul with state-of-the-art multiplier optimization frameworks, we consider ATRL [9], MUTE [22], and ADAS [24] as baselines. Similar to GRmul, ATRL is an RL-based framework that constructs PP accumulators from scratch using HAs and FAs. MUTE is a hybrid RL–evolutionary framework, while ADAS leverages differentiable architecture search for interconnection optimization. All parameters, except for the synthesis settings, remained their default values. The optimization objective was set as the mean of the synthesis results obtained under target delays of 5ns and 50ps. As the officially released open-source versions of baselines support only unsigned multipliers, the experiments in this section are restricted to unsigned multipliers[2].

[2]The released ATRL implementation supports only AND-based unsigned multipliers. All baselines were re-evaluated under our unified synthesis settings and optimization objectives; thus, their relative performance may differ from that reported in the original studies.

Table 4: Hardware comparison of multipliers.

Bit width	Type	Area opt.			Delay opt.		
		Area (μm^2)	Delay (ns)	ADP	Area (μm^2)	Delay (ns)	ADP
4	Default	115.0	0.550	63.25	139.0	0.466	64.77
	ATRL [9]	78.00	0.365	28.49	119.0	**0.311**	36.96
	MUTE [22]	78.00	0.383	29.86	92.00	0.357	32.81
	ADAS [24]	73.00	0.421	30.73	87.00	0.364	31.63
	GRmul	**66.00**	**0.357**	**23.58**	**74.00**	0.340	**25.17**
8	Default	389.0	0.958	372.7	423.0	0.831	351.5
	ATRL [9]	365.0	0.764	278.9	433.0	0.651	281.9
	MUTE [22]	375.0	0.837	313.9	423.0	0.706	298.7
	ADAS [24]	373.0	0.750	279.9	435.0	**0.619**	269.0
	GRmul	**361.0**	**0.742**	**267.9**	**410.0**	0.647	**265.3**
12	Default	915.0	1.342	1,228	1,013	1.130	1,145
	ATRL [9]	877.0	**1.055**	925.2	978.0	**0.897**	876.9
	MUTE [22]	817.0	1.089	890.0	878.0	0.936	822.0
	ADAS [24]	799.0	1.141	912.0	862.0	0.950	818.7
	GRmul	**798.0**	1.092	**871.4**	**832.0**	0.970	**807.2**
16	Default	1,667	1.656	2,761	1,754	1.434	2,515
	ATRL [9]	1,619	**1.196**	1,936	1,769	**1.029**	1,821
	MUTE [22]	1,383	1.385	1,915	1,435	1.169	**1,677**
	ADAS [24]	**1,370**	1.395	1,910	**1,433**	1.175	1,683
	GRmul	1,386	1.333	**1,847**	1,441	1.169	1,684

Consistent with the baselines, area and delay are used as the evaluation metrics. Figure 7(a)-(h) compares the resulting unsigned multipliers synthesized under the target delays of 5ns and 50ps, representing area-targeted and delay-targeted optimization scenarios, respectively. As observed, multipliers produced by GRmul largely dominate the Pareto frontier for bit widths smaller than 16, particularly in terms of area for 4- and 8-bit multipliers. This superiority is primarily attributed to the area reduction achieved through adder simplification. The proportion of simplified adders within the multipliers produced by GRmul is shown in Figure 8. Compared with the baselines reported in Table 1, GRmul identified more simplifiable adders in the AND-based 8-bit multiplier, demonstrating its ability to exploit PP correlations for circuit optimization. For 8-bit Booth multipliers, GRmul identified fewer simplifiable adders than MUTE because MUTE pads zeros in the initial PP array, which naturally creates simplifiable adders. As the multiplier bit width increases, the proportion of simplified adders decreases. This is mainly due to the increased complexity of the PP logic path, which involves more signals and thus reduces the potential for value conflicts. Booth multipliers maintain a relatively consistent proportion of simplifiable adders because unsigned Booth encoding inherently introduces logical redundancy due to sign bit extension. However, such straightforward logical simplifications can generally be handled efficiently by synthesis tools. Consequently, the relative advantage of GRmul over other state-of-the-art frameworks diminishes as the bit width increases. For 16-bit multipliers, GRmul generally performs similarly to the baselines. Figure 7(i)–(l) show the results of synthesizing the multipliers under a target delay sweep from 50ps to 2ns. For 12-bit multipliers, the advantage of GRmul becomes less pronounced. This may be because the larger design space introduced by adder simplification makes the optimized designs more specialized for the training targets (5ns and 50ps), reducing their effectiveness under other delay constraints.

Table 4 summarizes the detailed hardware measurements for the multipliers achieving the minimum area-delay product (ADP). The default Verilog HDL multiplication operator '$*$' is included for comparison. Except for the 16-bit multiplier under delay-target optimization, GRmul consistently generates designs with the lowest ADP. ATRL can produce multipliers with relatively low delay because it applies prefix tree optimizations to the CPA stage with delay reduction serving

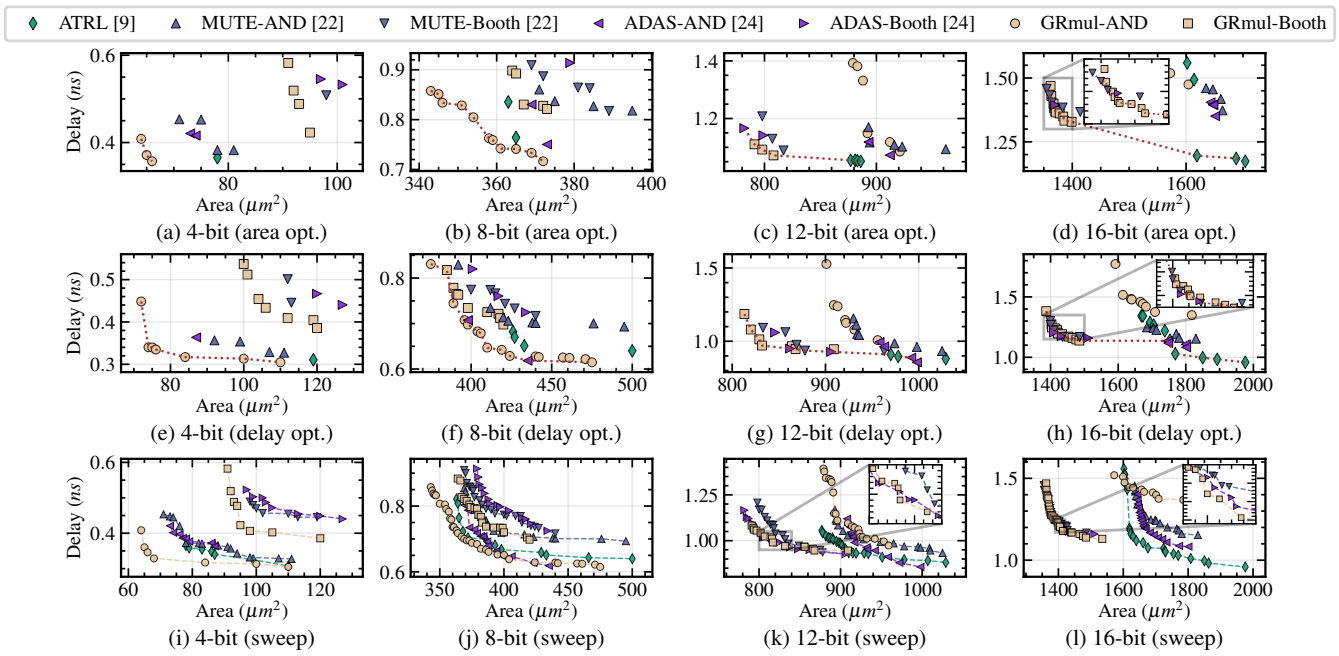


Figure 7: Comparison of area and delay for unsigned multipliers. The Pareto fronts in (a)-(h) are delineated.

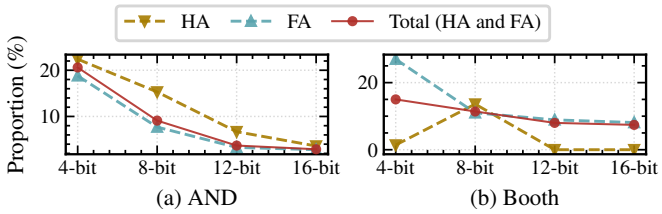


Figure 8: Proportion of simplified adders in multipliers generated by GRmul.

as the primary reward; however, this setting leads to an increase in area. Overall, GRmul achieves an ADP reduction ranging from 24.5% to 62.7% compared to the default multiplier. Compared with state-of-the-art frameworks, the ADP reduction is 1.4% to 20.4% for 4- to 12-bit multipliers, with the smallest and largest gains observed for the 12-bit and 4-bit delay-optimized multipliers, respectively.

To visualize the optimization effect of GRmul, Figure 9 shows the dot notation of an 8-bit multiplier located on the Pareto fronts for both area- and delay-targeted optimizations. The main differences from the Wallace multiplier shown in Figure 1 are summarized as follows.

- PPs are reordered to explore the accumulation sequence that can fully leverage logical correlations.
- Without affecting the critical path, additional IVCs can be created by increasing the number of PP accumulation stages.
- The precedence between HA and FA is critical; in some cases, properly prioritizing the HA can create more IVCs.

Moreover, the critical path of the multiplier varies with the optimization target. This is because the CPA is implemented using the Verilog HDL addition operator '+,' allowing synthesis tools to trade off area and delay by selecting different circuit implementations.

4.2 Comparison with Industrial Designs

To further validate the effectiveness of GRmul, we apply it to multiplier optimization using DC for synthesis. The results are compared with Synopsys DesignWare (DW) multipliers [18] and state-of-the-art

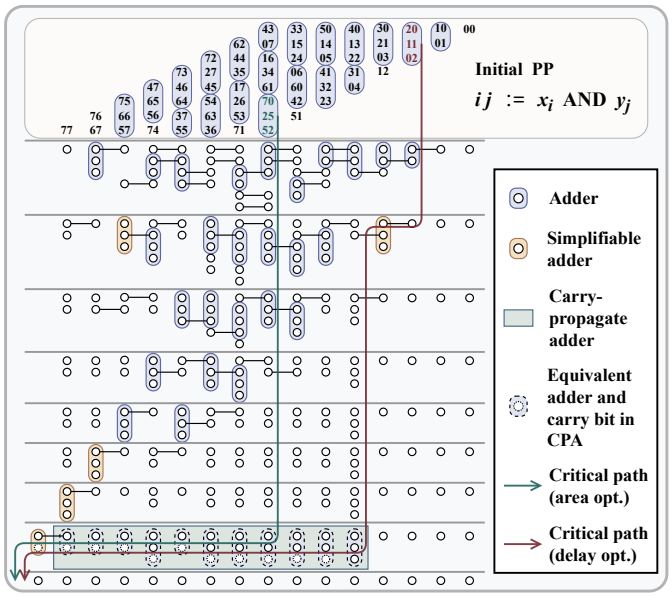


Figure 9: Dot notation of an 8-bit multiplier produced by GRmul.

manual designs, B4G3 [10] and Encode2x2 [11]. DW multipliers are industrial designs that support various configurations. B4G3 uses an enhanced PP generation technique derived from radix-4 Booth encoding. In this comparison, we combined this technique with the DW vector adder [19] to achieve more efficient multipliers. Encode2x2 optimizes 4- and 8-bit multipliers based on the 2×2 PP encoding. As the baseline frameworks in Section 4.1 rely on open-source toolchains, they are excluded here for fairness. To evaluate GRmul across different PP generation schemes, we utilized AND-based PP generation for unsigned 4- and 8-bit multipliers, the modified Baugh–Wooley algorithm [3] for signed 4-bit multipliers, radix-4 Booth encoding for signed 8- and 12-bit multipliers, and B4G3 encoding for all other cases.

During GRmul optimization, the target delay for DC was set to 1.5ns for 16-bit multipliers and 1.0ns for the others. To evaluate robustness

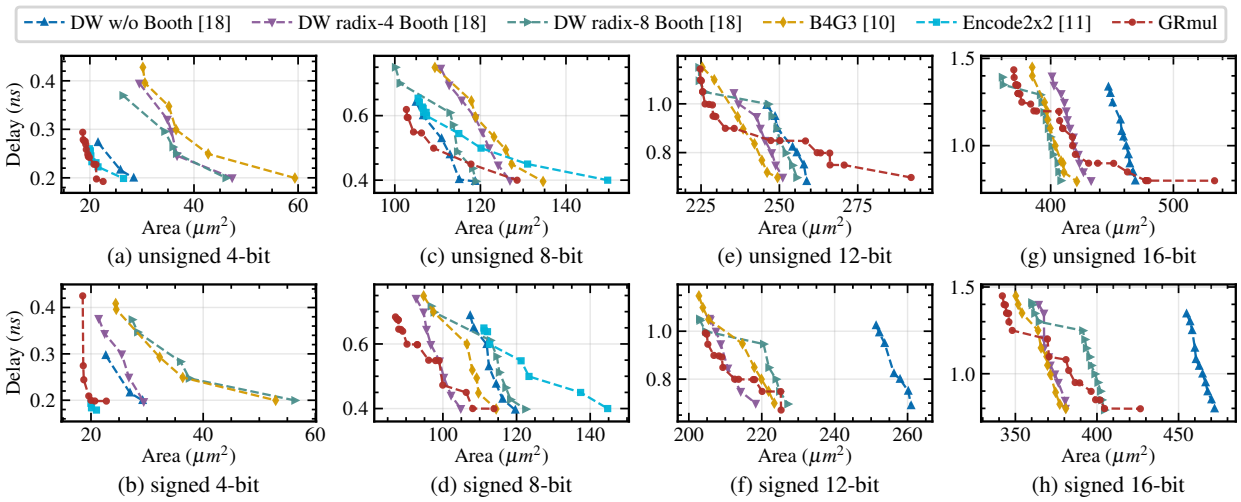


Figure 10: Hardware comparison of DC-synthesized multipliers (28nm CMOS process).

and scalability, the resulting designs were resynthesized across a delay sweep from 0.1ns to 1.5ns. As shown in Figure 10, multipliers produced by GRmul outperform the other designs under relatively relaxed delay constraints (near the initial optimization target), where the area gains from the adder simplification are particularly significant. However, the advantage diminishes under tighter delay constraints. This performance crossover stems from the architectural difference. While the DW generator dynamically reconfigures its microarchitecture to meet varying constraints, GRmul produces a fixed structural configuration highly specialized for the initial optimization targets. Moreover, while Encode2x2 achieves lower delay on signed 4-bit multipliers, its advantage is limited to the 4-bit designs.

4.3 Ablation Study

As two key enhancements of GRmul over baselines, the adder simplification technique and the DAG-based PP accumulation model are evaluated through ablation experiments. First, we removed the adder simplification mechanism from GRmul, yielding a variant denoted as GRmul (w/o simpl.). Second, we integrated the adder simplification mechanism into ATRL [9], resulting in ATRL (w simpl.). All frameworks were trained to convergence to generate unsigned 8-bit multipliers. The experimental settings follow Section 4.1. Figure 11 shows the performance comparison of the resulting multipliers. As observed, disabling adder simplification degrades the performance of GRmul to below that of ATRL, highlighting the critical role of this technique. Without it, the complex problem formulation in GRmul offers no benefit. Conversely, integrating adder simplification into ATRL yields worse results. This is because the introduction of simplifiable adders significantly expands the design space. The relatively simple problem formulation of ATRL struggles to capture the introduced complexity, limiting its ability to converge to an optimal design. Consequently, the proposed DAG model is essential for such an optimization problem.

Moreover, to validate the effectiveness of the agent model and the training scheme in GRmul, we evaluate both the R-GCN and the two-stage training method through ablation experiments. Figure 12 compares the learning curves of different configurations, which were obtained by averaging the results over five random seeds. We compare R-GCN with the standard GCN [8]. The parameters λ_0 and λ_1 specify the λ values (see (10)) used during the first 3,000 and the remaining 6,000 episodes of the training process, respectively. The two best-performing

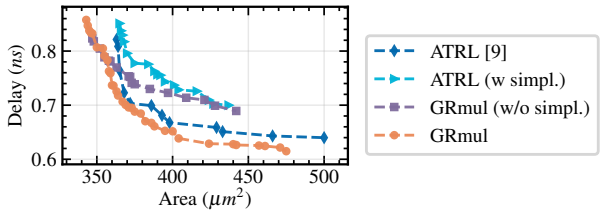


Figure 11: Ablation study for optimization frameworks.

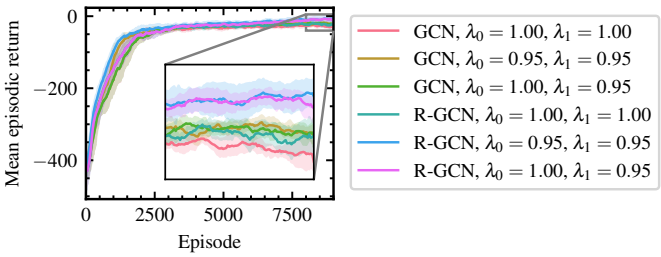


Figure 12: Performance comparison of different RL agent models and λ parameters.

configurations both use R-GCN with $\lambda_1 = 0.95$. Among them, the one with $\lambda_0 = 0.95$ shows slightly better performance but a much wider confidence interval, indicating greater sensitivity to random seeds. One possible explanation is that inaccurate value estimates in the early training stage introduce noise, which may occasionally lead to better solutions but also reduce training stability. Therefore, we select $\lambda_0 = 1.0$ to provide more stable performance across random seeds.

5 Conclusion

This work reveals and analyzes the adder simplification in PP accumulators enabled by IVCs of PPs. These IVCs arise from the logical correlations among PPs and have not been considered in previous multiplier optimization studies. To integrate adder simplification into multiplier optimization, we propose an RL-based framework, GRmul, that employs a DAG-based PP accumulation model to capture and exploit these logical correlations. Experimental evaluations confirm that for multipliers with relatively small bit widths, GRmul outperforms state-of-the-art optimization frameworks. Moreover, GRmul can generate more efficient multipliers than industrial and manual designs, making it well-suited for advanced AI accelerator design.

References

[1] Tutu Ajayi and David Blaauw. 2019. OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain. In *Proceedings of Government Microcircuit Applications and Critical Technology Conference*.

[2] Felix Arnold, Maxence Bouvier, Ryan Amaudruz, Renzo Andri, and Lukas Cavigelli. 2025. Explicit Sign-Magnitude Encoders Enable Power-Efficient Multipliers. *arXiv preprint arXiv:2507.18179* (2025).

[3] Charles R Baugh and Bruce A Wooley. 1973. A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Trans. Comput.* 100, 12 (1973), 1045–1047.

[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[5] Xinkuang Geng, Siting Liu, Jianfei Jiang, Kai Jiang, and Honglan Jiang. 2024. Compact Powers-of-Two: An Efficient Non-Uniform Quantization for Deep Neural Networks. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[6] Nangate Inc. 2008. NanGate FreePDK45 Open Cell Library.

[7] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. 2020. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proc. IEEE* 108, 12 (2020), 2108–2135.

[8] Thomas N Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

[9] Yao Lai, Jinxin Liu, David Z Pan, and Ping Luo. 2024. Scalable and Effective Arithmetic Tree Generation for Adder and Multiplier Designs. *Advances in Neural Information Processing Systems (NeurIPS)* 37 (2024), 139151–139178.

[10] Martin Langhammer, Bogdan Pasca, and Igor Kucherenko. 2024. Multiplier Architecture with a Carry-Based Partial Product Encoding. In *2024 IEEE 31st Symposium on Computer Arithmetic (ARITH)*. IEEE, 104–107.

[11] Ao Liu, Siting Liu, Hui Wang, Qin Wang, Fabrizio Lombardi, Zhigang Mao, and Honglan Jiang. 2025. Low-Power Multiplier Designs by Leveraging Correlations of 2×2 Encoded Partial Products. *IEEE Trans. Comput.* (2025).

[12] Olin L MacSorley. 1961. High-Speed Arithmetic in Binary Computers. *Proceedings of the IRE* 49, 1 (1961), 67–91.

[13] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nova, et al. 2021. A Graph Placement Methodology for Fast Chip Design. *Nature* 594, 7862 (2021), 207–212.

[14] Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. 2021. PrefixRL: Optimization of Parallel Prefix Circuits Using Deep Reinforcement Learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 853–858.

[15] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *European Semantic Web Conference*. Springer, 593–607.

[16] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2015. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438* (2015).

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[18] Synopsys. 2026. DesignWare DW02_mult. https://www.synopsys.com/dw/ipdir.php?c=DW02_mult.

[19] Synopsys. 2026. DesignWare DW02_sum. https://www.synopsys.com/dw/ipdir.php?c=DW02_sum.

[20] Berkeley Logic Synthesis and Verification Group. 2005. ABC: A System for Sequential Synthesis and Verification. https://people.eecs.berkeley.edu/~alanmi/abc/.

[21] Christopher S Wallace. 1964. A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers* 1 (1964), 14–17.

[22] Zhihai Wang, Jie Wang, Xilin Xia, Dongsheng Zuo, Lei Chen, Yuzhe Ma, Jianye Hao, Mingxuan Yuan, and Feng Wu. 2025. Computing Circuits Optimization via Model-Based Circuit Genetic Evolution. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

[23] Clifford Wolf. 2016. Yosys Open Synthesis Suite. https://yosyshq.net/yosys/.

[24] Xilin Xia, Jie Wang, Wanbo Zhang, Zhihai Wang, Mingxuan Yuan, Jianye Hao, and Feng Wu. 2025. High-Performance Arithmetic Circuit Optimization via Differentiable Architecture Search. In *The Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[25] Weihua Xiao, Weikang Qian, and Weiqiang Liu. 2021. GOMIL: Global Optimization of Multiplier by Integer Linear Programming. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 374–379.

[26] Chenhao Xue, Yi Ren, Jinwei Zhou, Kezhi Li, Chen Zhang, Yibo Lin, Lining Zhang, Qiang Xu, and Guangyu Sun. 2025. DOMAC: Differentiable Optimization for High-Speed Multipliers and Multiply-Accumulators. In *2025 International Symposium of Electronics Design Automation (ISEDA)*. IEEE, 250–255.

[27] Dongsheng Zuo, Yikang Ouyang, and Yuzhe Ma. 2023. RL-MUL: Multiplier Design Optimization with Deep Reinforcement Learning. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.