MUL-GPT: TIMING-STRUCTURE-AWARE GENERATIVE OPTIMIZATION FOR MULTIPLIER CIRCUITS

Anonymous authors
Paper under double-blind review

ABSTRACT

Multiplier optimization is a challenging combinatorial optimization problem, as its design space grows rapidly with increasing operand bit width. In recent years, learning-driven methods have provided a promising approach to discovering high-speed and area-efficient circuit structures. However, existing methods typically optimize only a small set of global post-synthesis metrics, such as area and delay, without explicitly modeling the internal timing structure of multipliers or the mechanisms underlying critical path formation. Consequently, they struggle to accurately identify timing bottlenecks and effectively exploit internal timing information to guide structural optimization. To address these limitations, we propose MUL-GPT, an internal-timing-structure-aware generative framework for multiplier optimization. MUL-GPT is a generative pretrained Transformer (GPT)-style generative framework that addresses the internally coupled timing optimization problem of multipliers. It unifies constrained autoregressive compressor-tree generation, arrival-time-conditioned adder optimization, and static timing analysis (STA)-guided local interconnection rewiring, with internal timing information serving as the link across these stages. Experiments show that the best designs found by MUL-GPT consistently outperform state-of-the-art methods across various arithmetic-unit design tasks, reducing the area-delay product (ADP) by up to **10.57%**. These improvements are preserved when the optimized arithmetic units are integrated into large-scale artificial-intelligence (AI)-accelerator circuits, where MUL-GPT achieves an ADP reduction of up to **7.90%**, further demonstrating its scalability and practical applicability.

1 INTRODUCTION

Multipliers are core arithmetic units in processors, graphics processing units, and artificial-intelligence (AI) accelerators (Arora et al., 2023), and their area and delay directly affect system performance and energy efficiency (Behrooz, 2000; Lee & Yoo, 2021). A typical multiplier consists of a partial product generator, a compressor tree, and a final carry-propagate adder (Wallace, 1964; Dadda, 1983). Since its critical path may span all three stages, multiplier performance depends on both individual substructures and their internal timing interactions (Ramkumar & Kittur, 2011). The exponentially growing design space further makes multiplier optimization a challenging combinatorial problem.

In recent years, AI-based methods have advanced multiplier optimization (Wang et al., 2024; Xue et al., 2026) and adder optimization (Ma et al., 2018; Roy et al., 2021). Representative methods, including RL-MUL (Zuo et al., 2023; 2026), ArithTreeRL (Lai et al., 2024), MUTE (Wang et al., 2025), and ARITH-DAS (Xia et al., 2026), explore compressor-tree topology and interconnections through reinforcement learning (Mnih et al., 2015), tree search (Silver et al., 2016), genetic evolution (Holland, 1992), and graph-based prediction (Kipf & Welling, 2017). However, most rely on post-synthesis area and delay as feedback, without explicitly exploiting the internal timing propagation characteristics of multipliers. Such global scalar metrics can evaluate the final quality of a candidate design, but they provide limited insight into the internal timing reasons behind performance differences. UFO-MAC (Zuo et al., 2024) considers such timing information, but its combinatorial search remains inefficient. For multipliers, similar module-level area or delay does not necessarily imply similar internal timing behavior. Therefore, multiplier optimization should not rely solely on global performance metrics, but should account for internal timing structures from three aspects. First, timing propagation inside the compressor tree is determined by carry propagation and level

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

dependencies among compressors. Second, the non-uniform arrival-time profile produced by the compressor tree directly affects critical-path formation in the final adder (Zuo et al., 2024). Third, even with fixed compressor locations, different assignments between source signals and compressor input ports can alter local path delays (Xia et al., 2026).

Motivated by these observations, we propose MUL-GPT, an internal-timing-structure-aware generative framework for multiplier optimization. We formulate compressor-tree construction as a constrained sequence generation task and employ a generative pretrained Transformer (GPT)-style autoregressive model to generate legal compressor trees. Self-attention models dependencies among compression operations, while synthesis feedback guides the model toward improved area-delay performance. The final adder is then generated conditional on the output arrival-time profile of the compressor tree, thereby improving the timing compatibility between the two stages. Finally, we introduce a critical-path-guided local rewiring method, which uses static timing analysis (STA) (Bhasker & Chadha, 2009) to identify critical compressor slices and the Cross-Entropy Method (CEM) (Rubinstein, 1999) to optimize local interconnections. In this way, MUL-GPT performs global structural exploration while explicitly exploiting internal timing information to refine the critical path.

We evaluate our method on representative arithmetic circuits, including multipliers and multiply-accumulate (MAC) units (Chen et al., 2016; Markidis et al., 2018). Results show that our approach consistently outperforms state-of-the-art methods, achieving up to **10.57%** area-delay product (ADP) reduction. When applied to large-scale AI-accelerator circuits (Xia et al., 2026), it further reduces ADP by up to **7.90%**, demonstrating strong scalability and practical effectiveness.

The main contributions are as follows: (1) we reformulate compressor-tree optimization as a constrained sequence generation problem for GPT-style autoregressive models. (2) we introduce a synthesis-driven alternating optimization framework that coordinates compressor-tree generation with arrival-time-conditioned adder generation. (3) we develop a modular critical-path-guided local rewiring method that can be applied as a general post-processing module to further improve performance. (4) we achieve consistent ADP improvements over state-of-the-art methods across multiplier, MAC, and large-scale AI-accelerator circuits.

2 Background

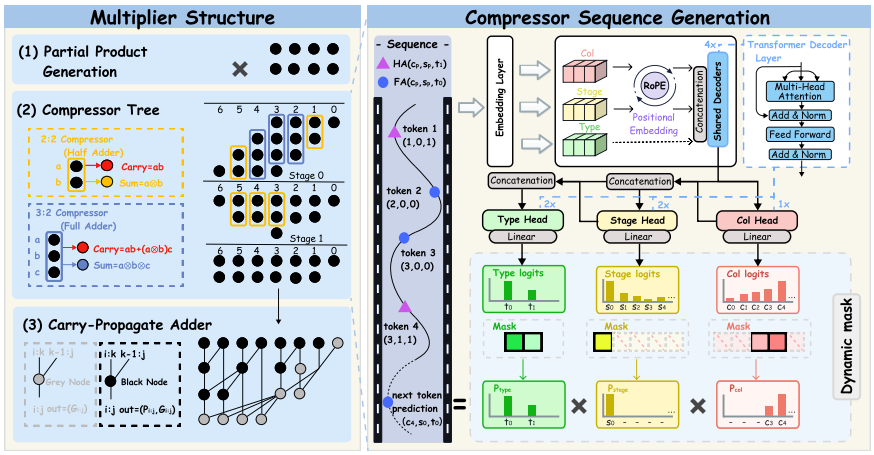
2.1 Multiplier Architecture

Multiplier designs typically comprise three main components: a partial-product generator (PPG), a compressor tree (CT), and a carry-propagate adder (CPA). As shown in Figure 1(a), the PPG generates partial products (PPs) from the multiplier and the multiplicand. The initial PP state is determined by the partial-product encoding scheme. For conventional AND-based PPs, bitwise AND operations between multiplier bits and multiplicand bits generate shifted rows of PPs. This layout is similar to the row-by-row shifted summation in standard long multiplication, resulting in a regular parallelogram-like initial PPs structure. The CT consists of multiple compression stages built from 3:2 and 2:2 compressors, which correspond to full adder (FA) and half adder (HA). A 3:2 compressor (resp. 2:2) located at column c and stage s takes 3 (resp. 2) PPs from column c and stage s as inputs, respectively. It produces two outputs: a sum output propagated to column c at stage $s+1$, and a carry output propagated to column $c+1$ at stage $s+1$. The placement and combination of compressors can change the propagation paths of intermediate signals during compression, thereby significantly influencing the final performance of the multiplier circuit. Finally, the CT reduces the PPs to two parallel rows, which are then summed by the CPA to produce the final result.

2.2 Generative Pretrained Transformer (GPT)

In recent years, Generative Pretrained Transformer (GPT) models have extended beyond natural language processing to structured generation tasks such as biological sequence modeling, material design, and circuit design (Ferruz et al., 2022; Kazeev et al., 2025; Ding et al., 2026). GPT-style models are typically built on decoder-only Transformer architectures and generate sequences autoregressively. Given a sequence $X = (x_1, x_2, \ldots, x_T)$, GPT factorizes its joint probability as

$$P(X) = \prod_{t=1}^{T} P(x_t \mid x_{1:t-1}). \tag{1}$$



(a) (b)

Figure 1: MUL-GPT. (a) Multiplier structure. (b) Compressor sequence generation.

With causal self-attention (Vaswani et al., 2017), each token is predicted from previously generated tokens, enabling step-by-step sequence construction. After pretraining (Radford et al., 2018; Brown et al., 2020), the model can learn structural regularities from valid sequences, and subsequent fine-tuning, such as supervised fine-tuning or reinforcement learning (Stiennon et al., 2020; Ouyang et al., 2022), further aligns the model with task-specific objectives. In this work, we adopt this GPT-style paradigm for compressor-tree generation, where each token represents a compression operation and dynamic legality masks ensure circuit validity.

3 Motivation and Challenges

Existing methods mainly optimize compressor types and placements using area and delay from logic models or synthesis. Although recent studies (Zuo et al., 2024; Xia et al., 2026) have shown that compressor interconnections also affect multiplier performance, existing methods mainly rely on global scalar metrics without explicitly modeling internal timing propagation. Consequently, they have limited ability to locate critical paths and guide further structural optimization based on internal timing bottlenecks. In summary, existing methods have three fundamental limitations: (1) they overlook how critical path propagates across compressors; (2) they lack an explicit mechanism for adapting the final-adder topology to non-uniform compressor-tree arrival times; (3) global interconnect search is costly, whereas optimization should target critical paths.

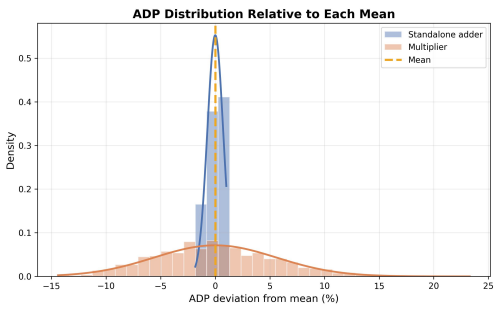


Figure 2: Distribution of ADP deviations relative to the mean ADP of 1,000 sampled standalone adders and complete multipliers.

Specifically, a multiplier contains multiple timing paths, and its overall delay is determined by the longest one, i.e., the critical path. To demonstrate the importance of internal timing paths, we conduct a random experiment. We first generate and select 1,000 adder structures with similar standalone performance, where both delay and area variations are limited to approximately 1%. We then connect these adders to the same fixed compressor tree, obtaining 1,000 complete multipliers for performance comparison. As shown in Figure 2, although these adders exhibit similar standalone delay and area, the ADP of the complete multipliers still varies noticeably after they are connected to the same compressor tree. This indicates that the optimization process should not rely solely on global scalar metrics such as area, delay, or ADP. Even adders with similar standalone performance may have different internal timing paths, which form different timing couplings with the compressor-tree output

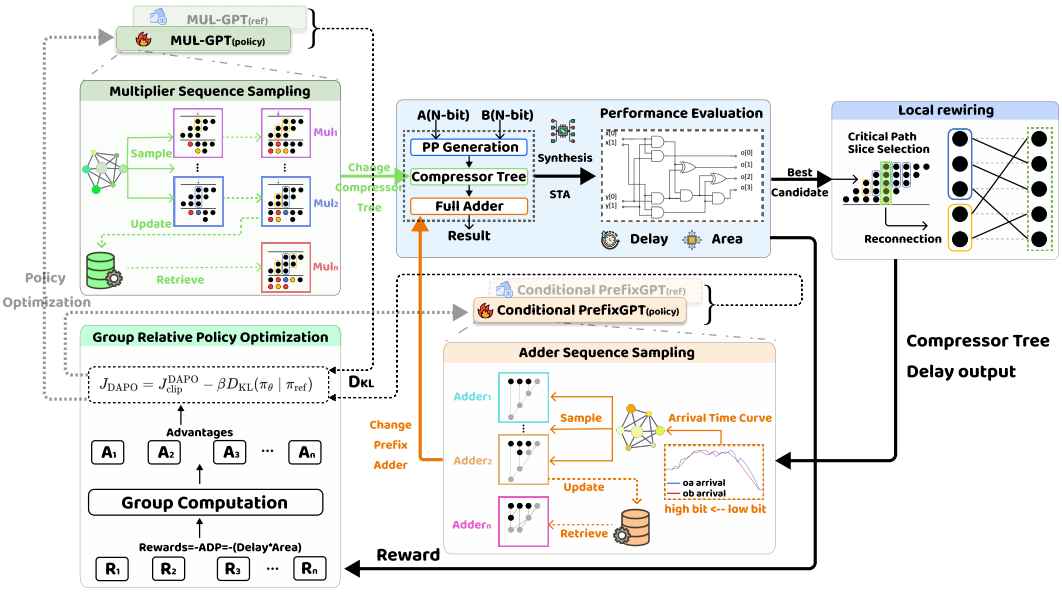


Figure 3: Overview of the proposed MUL-GPT pipeline.

arrival profile and therefore lead to different final delay and ADP. Thus, multiplier optimization should explicitly consider internal timing structures. MUL-GPT combines global structural generation with timing-aware refinement. It models compressor dependencies using self-attention, conditions final-adder generation on compressor-tree arrival times through cross-attention, and rewires STA-identified critical regions. Appendix C further analyzes these design choices.

4 METHOD

We start with an overview of the MUL-GPT framework in Section 4.1. Then, Section 4.2 formalizes the sequence construction rules and dynamic masks for legal compressor-tree generation. Section 4.3 presents the GPT-style generator, including pretraining, fine-tuning based on reinforcement learning, and conditional adder generation guided by arrival times. Finally, Section 4.4 introduces a critical-path-guided CEM for local rewiring.

4.1 Overview of Our Framework

Figure 3 illustrates the overall pipeline of MUL-GPT. First, we develop a fine-grained sequential representation that reformulates compressor-tree optimization as a sequence generation task subject to circuit legality constraints. A GPT-style model is then pretrained on randomly generated legal sequences to learn structurally valid circuit patterns. The generator is then fine-tuned using group-relative Decoupled Clip and Dynamic sAmpling Policy Optimization (DAPO) (Yu et al., 2026), with post-synthesis area and delay serving as feedback to guide the exploration of higher-quality multiplier structures. The compressor tree and final adder are optimized alternately. At each round, the accepted compressor tree with the lowest ADP is paired with its arrival-time profile to generate the conditional adder. Finally, we introduce a critical-path-guided local rewiring method based on the CEM, which iteratively samples candidate rewiring configurations and updates the sampling distribution to refine local interconnections near the critical path, thereby further improving the area-delay trade-off.

4.2 Compression-Operation Sequence

GPT-style models generate sequences autoregressively by predicting each element from the preceding sequence until termination. Thus, both the sequence representation and next-element prediction are critical. A compressor tree is represented as $Z = (z_1, z_2, \ldots, z_T)$, where each token $z_p = (c_p, s_p, t_p)$ specifies the compression column, stage, and compressor type. Here, $t_p \in \{0, 1\}$ denotes the

compressor type, with 0 for an FA and 1 for an HA. The compressor-tree state is represented by $H(s, c)$, the number of uncompressed PPs in column c at stage s. Initially, $H(0, c)$ equals the number of input PPs in column c, and $H(s, c) = 0$ for $s > 0$. The generation objective is to leave at most two remaining PPs in each column, producing two output rows for the final adder.

At each generation step, the set of legal actions is constrained by the current state H. Columns are processed from lower to higher bit positions. If the current column has a height of at least 3, compression must continue within that column. If its height is 2, the generator may either apply an HA or advance to the next column. If its height is below 2, it directly advances to the next column requiring compression. This rule enforces a deterministic column-wise progression during compressor-tree construction. Within the same column, the selected stage indices must be nondecreasing, whereas the generator may restart from a lower stage after moving to a new column. Remaining PPs from lower stages are propagated to the selected target stage, and a compressor action is considered legal only when sufficient PPs are available at the candidate position. An FA requires three PPs, while an HA requires two. Both compressors produce a sum output at (s+1,c) and a carry output at (s+1,c+1). Accordingly, the state is updated as

$$(H(s,c), H(s+1,c), H(s+1,c+1)) \leftarrow (H(s,c)-k,\ H(s+1,c)+1,\ H(s+1,c+1)+1), \quad (2)$$

where $k = 3$ for an FA and $k = 2$ for an HA. All legality constraints are explicitly encoded as dynamic action masks. Invalid actions are excluded during sampling, and only sequences reaching the terminal two-row condition are accepted as valid compressor trees.

4.3 GPT-style Compressor-tree Generation and Conditional Adder Design

Compressor-Tree Generator We formulate multiplier design as two coupled GPT-style structured sequence generation tasks: compressor-tree generation and conditional adder generation. For compressor-tree generation, as shown in Figure 1(b), each token $z_p = (c_p, s_p, t_p)$ denotes the compression column, stage, and compressor type. The three fields are encoded with separate learnable embeddings $\mathbf{E}_p^c$,$\mathbf{E}_p^s$ and $\mathbf{E}_p^t$, where the column embeddings $\hat{\mathbf{E}}_p^c$ and stage embeddings $\hat{\mathbf{E}}_p^s$ are enhanced by Rotary Position Embedding (RoPE) (Su et al., 2024) to capture relative spatial relationships. The resulting token embedding is $\mathbf{T}_p = \hat{\mathbf{E}}_p^c \| \hat{\mathbf{E}}_p^s \| \mathbf{E}_p^t$. Given $\mathbf{T}_{1:k}$, a causal Transformer decoder produces shared hidden states $\mathbf{H}_{1:k}^{\text{share}} = \text{SharedDecoder}(\mathbf{T}_{1:k})$. The next token is generated through a cascaded prediction scheme. The column head first predicts the next compression column from $\mathbf{H}_{1:k}^{\text{col}} = \text{ColumnHead}(\mathbf{H}_{1:k}^{\text{share}})$ as $P(c_{k+1} \mid z_{1:k}) = \text{Softmax}(\mathbf{W}^{\text{col}}\mathbf{h}_k^{\text{col}})$. The stage head then predicts the compression stage using the concatenated normalized shared and column representations, and the type head further predicts the compressor type using the shared, column, and stage representations: $P(s_{k+1} \mid z_{1:k}, c_{k+1}) = \text{Softmax}(\mathbf{W}^{\text{stage}}\mathbf{h}_k^{\text{stage}})$ and $P(t_{k+1} \mid z_{1:k}, c_{k+1}, s_{k+1}) = \text{Softmax}(\mathbf{W}^{\text{type}}\mathbf{h}_k^{\text{type}})$. Thus, the next token is generated as

$$P(z_{k+1} \mid z_{1:k}) = P(c_{k+1} \mid z_{1:k}) P(s_{k+1} \mid z_{1:k}, c_{k+1}) P(t_{k+1} \mid z_{1:k}, c_{k+1}, s_{k+1}). \quad (3)$$

This cascaded formulation explicitly models the dependency among column selection, stage selection, and compressor-type selection. During decoding, legality masks are applied at each prediction stage to ensure that the generated compressor operation satisfies hardware constraints.

Pretraining We construct the pretraining corpus by randomly sampling legal compression operations from the initial partial-product matrix until all columns satisfy the terminal-height constraint. Given a sequence Z, the model autoregressively predicts the next operation from its prefix using three prediction heads $q \in \{c, s, t\}$ for the column, stage, and compressor type. The pretraining objective is $\mathcal{L}_{\text{pre}} = \mathcal{L}_{\text{CE}} + \lambda_{\text{legal}}\mathcal{L}_{\text{legal}} + \lambda_{\text{margin}}\mathcal{L}_{\text{margin}}$. Let y_t^q and $p_t^q(a)$ denote the ground-truth target and legality-masked probability of candidate action a for head q at step t. The cross-entropy loss is

$$\mathcal{L}_{\text{CE}} = -\frac{1}{T}\sum_{t,q} \log p_t^q(y_t^q). \quad (4)$$

In addition to supervising the demonstrated operation, we explicitly supervise the legality of all candidate actions. Let $m_{t,a}^q = 1$ indicate that candidate action a is legal for head q at step t, and let $s_{t,a}^q$ denote the corresponding raw logit. The multi-label legality loss is defined as

$$\mathcal{L}_{\text{legal}} = -\frac{1}{N}\sum_{t,q,a} \left[m_{t,a}^q \log \sigma(s_{t,a}^q) + (1 - m_{t,a}^q) \log(1 - \sigma(s_{t,a}^q)) \right]. \quad (5)$$

Unlike cross-entropy, which ranks actions only within the legal set, this loss separates legal from illegal actions and enforces a margin between the target action and the highest-scoring illegal action:

$$\mathcal{L}_{\text{margin}} = \frac{1}{T}\sum_{t,q}\max\left(0,\ \max_{a:m^q_{t,a}=0} s^q_{t,a} - s^q_{t,y^q_t} + \delta\right). \tag{6}$$

Fine-tuning The pretraining stage enables the model to learn the legal generation rules of compressor-tree sequences, but it does not guarantee that the generated structures achieve optimal circuit performance. To align the generation policy with the area-delay optimization objective, we further introduce fine-tuning based on reinforcement learning. Specifically, each generated sequence Z_i is translated into Verilog and evaluated through logic synthesis and STA to obtain its area and delay. The reward is defined as the negative area-delay product (ADP), $R_i = -\text{Area}(Z_i) \times \text{Delay}(Z_i)$, such that a smaller ADP corresponds to a higher reward. Considering the long-sequence nature, strong structural constraints, and non-differentiable evaluation of multiplier generation, we adopt a DAPO-style (Yu et al., 2026) policy optimization method. At each iteration, the model samples a group of candidate designs and computes the group-relative advantage $A_i = \frac{R_i - \mu_R}{\sigma_R}$, where μ_R and σ_R are the group reward mean and standard deviation (Shao et al., 2024). To improve sample efficiency, we maintain an elite replay pool that stores historically high-reward designs generated within the current optimization round (Mnih et al., 2015). A subset of elite samples is retrieved and mixed with the newly sampled candidates during policy optimization, allowing the model to revisit high-quality structures from earlier iterations. The policy update is weighted by the importance-sampling ratio $r_{i,t}(\theta) = \frac{\pi_\theta(z_{i,t}|z_{i,1:t-1})}{\pi_{\text{old}}(z_{i,t}|z_{i,1:t-1})}$, which measures the probability change of token $z_{i,t}$ under the current policy relative to the old policy (Schulman et al., 2017). We further use a clipped objective to constrain the update magnitude and introduce a KL regularization term with respect to the pretrained reference policy. This helps preserve the legality-aware structural prior while optimizing circuit performance. With these definitions, the overall fine-tuning loss is minimized as follows

$$\mathcal{L}(\theta) = \frac{1}{N_{\text{token}}}\sum_{i=1}^{B}\sum_{t=1}^{T_i}\left[-\min\left(r_{i,t}A_i, \text{clip}(r_{i,t}, 1-\epsilon_{low}, 1+\epsilon_{high})A_i\right) + \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})\right]. \tag{7}$$

Conditional Adder Generator After generating the compressor-tree sequence, we introduce a conditional adder generator to optimize the final addition stage. The generator is built upon the PrefixGPT (Ding et al., 2026) backbone and undergoes a similar pretraining stage. During fine-tuning, however, it differs from an independent adder generator by explicitly conditioning on the timing distribution of the current compressor tree. Specifically, for each selected compressor tree Z_i, we perform synthesis and STA to obtain the arrival-time profiles of the two final operand vectors. These profiles are organized as a conditional feature sequence $\mathbf{X}^{\text{arr}}_{1:n}$, which conditions the generation of a final adder jointly synthesized and evaluated with Z_i. For each bit position i, the corresponding conditional feature is defined as $\mathbf{x}^{\text{arr}}_i = \left[a^{(0)}_i, a^{(1)}_i\right]$, where $a^{(0)}_i$ and $a^{(1)}_i$ denote the arrival times of the two final operand bits at position i, respectively. Thus, $\mathbf{X}^{\text{arr}}_{1:n} = (\mathbf{x}^{\text{arr}}_1, \ldots, \mathbf{x}^{\text{arr}}_n)$. After normalization and projection to the hidden dimension, the arrival features are encoded by a Transformer encoder as $\mathbf{A}_{1:n} = \text{Encoder}(\mathbf{X}^{\text{arr}}_{1:n})$. The conditional adder is represented as a row-column coordinate sequence, where each token $a_p = (r_p, c_p)$ denotes an adder-cell decision. The embedded adder-token sequence $\mathbf{E}_{1:k}$ is then processed by a causal decoder with cross-attention to the encoded timing features, yielding $\mathbf{H}^{\text{add}}_{1:k} = \text{Decoder}(\mathbf{E}_{1:k}, \mathbf{A}_{1:n})$, where $\mathbf{A}_{1:n}$ provides the timing condition for adder generation. Finally, a row head and a column head sequentially predict the row and column coordinates, enabling the generated adder to adapt to the non-uniform arrival distribution and capture the internal timing interactions propagated from the current compressor tree.

4.4 Critical-Path-Guided Interconnect Rewiring

The generated sequence determines compressor placement, but multiple source-to-port assignments may still exist under the same placement, leading to different timing performance. Here, a compressor slice refers to the set of compressor operations at the same compression column and stage. To avoid global search over a large connection space, we perform local rewiring only on the compressor slices traversed by the critical path, thereby directly targeting timing bottlenecks with reduced search overhead. Before local rewiring, the initial assignment is determined using a delay-based heuristic.

Specifically, for each top-ranked candidate, we first perform STA to locate the critical path and the compressor slices traversed by it. We then reassign source signals among the input ports of neighboring compressors within these slices using the Cross-Entropy Method (CEM) (Rubinstein, 1999), which iteratively samples candidate assignments, selects high-performing elites, and updates the sampling distribution toward these elite samples. For each selected slice with m source signals and m input ports, we maintain a connection-score matrix $P \in [0,1]^{m\times m}$, where P_{ij} is an edge-wise score that biases the assignment of source i to port j. A legal candidate assignment is sampled using Gumbel perturbation (Maddison et al., 2017) and the Hungarian algorithm (Kuhn, 1955):

$$S_{ij} = \log(P_{ij} + \varepsilon) + \tau_g G_{ij}, \qquad X = \mathrm{Hungarian}(S). \tag{8}$$

Here,τ_g is the Gumbel temperature controlling sampling randomness, and ε is a small constant used for numerical stability. The Hungarian algorithm returns a maximum-weight bipartite matching from the score matrix S. The resulting $X \in \{0,1\}^{m\times m}$ is a permutation matrix, where $X_{ij} = 1$ indicates that source i is assigned to port j. At each CEM iteration, the sampled rewiring candidates are evaluated through logic synthesis and STA and ranked according to their ADP values. The top ρ_e fraction is selected as the elite set $\mathcal{E}$. The empirical connection-frequency matrix $\widehat{P}$ of the elite samples is used to update the connection-score matrix:

$$\widehat{P} = \frac{1}{|\mathcal{E}|}\sum_{X\in\mathcal{E}} X, \qquad P \leftarrow (1-\alpha)P + \alpha\widehat{P}. \tag{9}$$

Here, α controls the smoothing of the sampling matrix. After multiple iterations, the candidate with the lowest ADP is selected. If the critical path changes, STA is repeated, and optimization continues using the updated critical path until no further ADP improvement is achieved.

5 Experiments

Section 5.1 describes the experimental setup, baselines, and evaluation metrics. We evaluate MUL-GPT across multiplier widths in Section 5.2, assess transfer to large-scale AI-chip macros in Section 5.3, and quantify component contributions through ablation studies in Section 5.4.

5.1 Experimental Settings

Experimental Setup During pretraining, the model is trained for five epochs on 10^6 randomly generated legal compressor-tree sequences using Adam with a learning rate of 10^{-4}. We then fine-tune it using DAPO-based reinforcement learning. At each iteration, $G = 512$ legal multiplier designs are generated, converted to Verilog, and evaluated through logic synthesis and STA with Yosys (Wolf et al., 2013; Brayton & Mishchenko, 2010) and OpenROAD (Ajayi & Blaauw, 2019) with the NanGate 45 nm standard cell library (Nangate Inc., 2008). We perform three rounds of alternating optimization, with 70 iterations per stage. The first compressor-tree stage uses a behavioral adder (c=a+b), synthesized by Yosys, while subsequent rounds use the best conditional adder obtained from the preceding round. To reduce the influence of random variation, we conduct three independent runs for each combination of operand width and partial product scheme. Each run evaluates approximately 10^5 compressor-tree designs, while keeping the baseline search budgets comparable. We report the best ADP achieved among the three runs. The main comparison evaluates end-to-end system performance. Baseline results correspond to the native pipelines reported for the respective methods. For 64-bit experiments, we adjust configurations to accommodate the longer generation sequences and higher memory and runtime requirements. Complete settings are provided in Appendix D.

Baselines and Evaluation Metrics We compare MUL-GPT against seven competitive baselines. **1) Wallace Tree** (Wallace, 1964) is a classical manually designed multiplier architecture. **2) GOMIL** (Xiao et al., 2021), **3) UFO-MAC** (Zuo et al., 2024) formulate arithmetic circuit optimization as an integer programming problem. **4) RL-MUL** (Zuo et al., 2023), **5) MUTE** (Wang et al., 2025), **6) ArithTreeRL** (Lai et al., 2024), and **7) ARITH-DAS** (Xia et al., 2026) are four state-of-the-art machine-learning-based methods. Arithmetic circuit optimization involves competing objectives, particularly circuit area and critical path delay. We evaluate each method using area, delay, and ADP, together with the area-delay Pareto front and its hypervolume. A Pareto front (Deb et al., 2002; Deb, 2011; Song et al., 2022) closer to the lower-left region indicates a better area-delay trade-off, whereas a larger hypervolume indicates better Pareto-front quality and coverage.

Table 1: Numerical comparison of AND and Booth multipliers in area (μm^2), delay (ns), and ADP.

Type	Method	8-bit			16-bit			32-bit			64-bit		
		Area	Delay	ADP	Area	Delay	ADP	Area	Delay	ADP	Area	Delay	ADP
AND	Wallace	372	0.7940	295.3680	1642	1.0921	1793.2282	6682	1.4803	9891.3646	26842	1.9294	51788.9548
	GOMIL	358	0.8410	301.0780	1554	1.1315	1758.3510	6311	1.4505	9154.1055	25326	1.8901	47868.6726
	UFO-MAC	377	0.7609	286.8593	1615	1.1135	1798.3025	6519	1.4251	9290.2269	25803	1.8568	47911.0104
	RL-MUL	371	0.7246	268.8266	1621	1.0498	1701.7258	6543	1.4254	9326.3922	26060	1.7887	46613.5220
	MUTE	**354**	0.7267	257.2518	1545	1.0504	1622.8680	**6310**	1.3781	8695.8110	**25276**	1.7781	44943.2556
	ArithTreeRL	373	0.6475	241.5175	1597	0.9566	1527.6902	6440	1.3032	8392.6080	26066	1.7098	44567.6468
	ARITH-DAS	372	0.6639	246.9708	1569	0.9651	1514.2419	6406	1.2843	8227.2258	25516	1.6527	42170.2932
	MUL-GPT (Ours)	358	**0.6033**	**215.9814**	**1544**	**0.8951**	**1382.0344**	6412	**1.2010**	**7700.8120**	25947	**1.5548**	**40342.3956**
Booth	Wallace	377	0.8712	328.4424	1422	1.2034	1711.2348	5127	1.4794	7584.8838	19343	2.0367	39395.8881
	GOMIL	369	0.8686	320.5134	1337	1.1857	1585.2809	**4908**	1.5026	7374.7608	**18627**	1.9577	36466.0779
	UFO-MAC	381	0.8065	307.2765	1393	1.0749	1497.3357	5074	1.5266	7745.9684	19133	1.8765	35903.0745
	RL-MUL	374	0.7995	299.0130	1378	1.0410	1434.4980	5190	1.3915	7221.8850	19523	1.8260	35648.9980
	MUTE	375	0.7858	294.6750	1340	1.0540	1412.3600	4914	1.4006	6882.5484	18709	1.7654	33028.8686
	ArithTreeRL	378	0.7326	276.9228	1345	1.0259	1379.8355	4993	1.3644	6812.4492	18901	1.7395	32878.2895
	ARITH-DAS	370	0.7327	271.0990	**1325**	1.0223	1354.5475	4911	1.3656	6706.4616	18819	1.6802	31619.6838
	MUL-GPT (Ours)	**367**	**0.6956**	**255.2852**	1344	**0.9393**	**1262.4192**	5018	**1.2335**	**6189.7030**	19338	**1.5398**	**29776.6524**

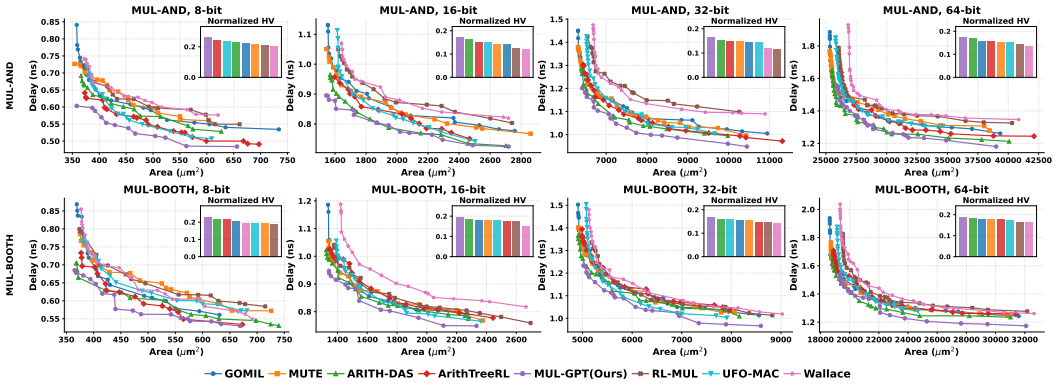


Figure 4: Pareto fronts of MUL-AND and MUL-BOOTH designs.

5.2 Main Results on Multiplier Optimization

We evaluate MUL-GPT on eight multiplier design tasks, covering 8-bit, 16-bit, 32-bit, and 64-bit multipliers with both AND-based and Booth-encoded PPs generators (Booth, 1951), and compare it against competitive baselines. As shown in Table 1, MUL-GPT achieves substantially lower ADP across all tasks and maintains a consistent advantage as the operand width increases. The area-delay Pareto fronts further demonstrate the effectiveness of MUL-GPT. Each Pareto front comprises multiple implementations of the same multiplier task under different timing constraints. As shown in Figure 4, the Pareto fronts obtained by MUL-GPT consistently lie toward the lower left of those produced by the baselines, indicating smaller area under comparable delay and shorter delay under comparable area. Moreover, MUL-GPT achieves larger hypervolume values (Zitzler et al., 2003) across all tasks, demonstrating its ability to discover not only better individual designs but also a larger area-delay trade-off region. Overall, MUL-GPT consistently provides more effective joint optimization of area and delay across different operand widths and multiplier architectures.

5.3 Generalization to Large-Scale AI-accelerator Circuits

To evaluate scalability, we integrate the MAC units (Chen et al., 2016; Markidis et al., 2018) optimized by MUL-GPT and the baseline methods into 8×8 processing element (PE) arrays (Jouppi et al., 2017; Chen et al., 2014), each comprising 64 identical MAC-based PEs. PE arrays are widely used as core computational structures in AI accelerators for matrix multiplication and convolution workloads (Vaswani et al., 2017; Dosovitskiy et al., 2021; He et al., 2016; Devlin et al., 2019). They therefore provide a practical setting for examining whether arithmetic-unit-level improvements can be preserved after integration into larger computational structures. The results in Table 2 and Figure 5 show that MUL-GPT continues to outperform all baseline methods after PE-array integration, achieving better area-delay trade-offs and larger hypervolume. To further assess the generalization capability of the optimized computing units, we also integrate those produced by MUL-GPT and the baseline methods into six representative AI-accelerator circuits from the Koios 2.0 benchmark

Table 2: Numerical comparison of AND and Booth PE arrays in area (μm^2), delay (ns), and ADP.

Method	AND						Booth					
	16-bit			32-bit			16-bit			32-bit		
	Area	Delay	ADP	Area	Delay	ADP	Area	Delay	ADP	Area	Delay	ADP
Wallace	134980	1.3260	178983.4800	481325	1.6763	806845.0975	120237	1.3147	158075.5839	382675	1.6436	628964.6300
GOMIL	127229	1.2250	155855.5250	**458828**	1.5579	714808.1412	**114981**	1.2854	147796.5774	**370188**	1.6425	608033.7900
UFO-MAC	129595	1.2560	162771.3200	470156	1.5247	716846.8532	117793	1.2289	144755.8177	382372	1.5032	574781.5904
RL-MUL	133101	1.2065	160586.3565	475662	1.5692	746408.8104	121829	1.2991	158268.0539	388176	1.5709	609785.6784
MUTE	**126512**	1.2445	157444.1840	457090	1.5248	696970.8320	117179	1.2246	143497.4034	377525	1.5134	571346.3350
ArithTreeRL	129400	1.1247	145536.1800	470993	1.4537	684682.5241	115459	1.2235	141264.0865	375820	1.4744	554109.0080
ARITH-DAS	128274	1.1286	144770.0364	460520	1.4898	686082.6960	117596	1.2211	143596.4756	375799	1.4967	562458.3633
MUL-GPT (Ours)	127914	**1.0424**	**133337.5536**	470431	**1.4049**	**660908.5119**	115295	**1.1666**	**134503.1470**	379499	**1.4532**	**551487.9468**

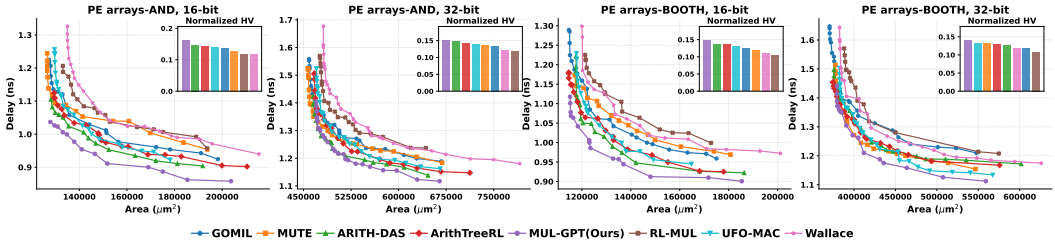


Figure 5: Pareto fronts of PE arrays based on AND and BOOTH designs.

(Arora et al., 2023), with detailed performance analysis reported in Appendix E.4. Improvements observed at the arithmetic-unit level remain measurable after integrating the optimized units into the evaluated PE arrays and Koios compute structures.

5.4 Ablation Study

Table 3: Ablation results on 16-bit and 32-bit multiplier tasks.

Width	Method	Area ↓	Delay ↓	ADP ↓	Imp. ↑
16-bit	Wallace	1642	1.0921	1793.2282	NA
	MUL-GPT	1544	0.8951	**1382.0344**	**22.93%**
	w/o fine-tuning	1648	0.9383	1546.3184	13.77%
	w/o Adder Opt.	1538	0.9358	1439.2604	19.74%
	w/o Arrival Cond.	1545	0.9112	1407.8040	21.49%
	w/o Rewire	1548	0.9199	1424.0052	20.59%
32-bit	Wallace	6682	1.4803	9891.3646	NA
	MUL-GPT	6412	1.2010	**7700.8120**	**22.15%**
	w/o fine-tuning	7132	1.3027	9290.8564	6.07%
	w/o Adder Opt.	6428	1.2725	8179.6300	17.31%
	w/o Arrival Cond.	6412	1.2156	7794.4272	21.20%
	w/o Rewire	6466	1.1985	7749.5010	21.65%

We conducted a systematic ablation study on multiplier design using four variants: **w/o fine-tuning**, which directly uses pre-trained compressor-tree generator; **w/o Adder Optimization**, which removes conditional adder optimization; **w/o Arrival Conditioning**, which retains adder optimization, but removes the arrival-time input; and **w/o Rewire**, which removes critical-path-guided local rewiring. As shown in Table 3, policy fine-tuning provides the largest improvement because the compressor tree is the primary source of structural variation, and synthesis feedback directly guides the generator toward better circuit structures. Adder optimization improves timing, while arrival conditioning further adapts the adder to nonuniform output arrivals. Local rewiring yields a smaller but consistent gain by refining a limited number of critical path connections. Moreover, it is independent of the underlying generation method and can serve as a general post-processing module for different baselines.

6 Conclusion

We present MUL-GPT, an internal-timing-structure-aware generative framework for multiplier optimization. It formulates compressor-tree construction as constrained autoregressive token generation and combines synthesis feedback, critical-path-guided CEM rewiring, and arrival-time-guided adder generation within a unified pipeline. Experiments show that MUL-GPT consistently improves area-delay trade-offs over state-of-the-art baselines. To control sequence complexity and GPU memory overhead, the current representation encodes only compressor positions and types, while signal-level connectivity is optimized separately through CEM rewiring. Incorporating connection decisions directly into the generative model may further improve timing performance.

AI USE STATEMENT

In this work, we used generative AI tools to assist with translation, provide feedback on the presentation of the research methodology, and assist with the implementation of parts of the proposed method. We did not use generative AI tools to generate synthetic datasets, develop theoretical models or conceptual frameworks, formulate or prove mathematical claims, propose or refine hypotheses, clean or reformat datasets, conduct qualitative or thematic data analysis, or interpret experimental results. Other required-disclosure tasks are not applicable to this work. Additionally, we used generative AI tools for brainstorming, information sourcing and searching, manuscript editing to improve readability and clarity, and refinement of the wording and organization of parts of the paper. All AI-assisted text, code, and suggestions were manually reviewed and revised by the authors. AI-assisted code was manually inspected and tested for correctness, and its behavior was verified through our implementation and experimental results. Technical descriptions, methodological claims, experimental results, and conclusions were independently verified against our implementation and experimental records. We take full responsibility for the final content of this work, including any text, claims, code, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility and support a thorough evaluation, we provide an anonymous implementation at `https://anonymous.4open.science/r/MUL_GPT-640C/`. The repository contains the source code, experiment scripts, configuration files, and instructions required to reproduce the main results.

REFERENCES

Tutu Ajayi and David Blaauw. Openroad: Toward a self-driving, open-source digital layout implementation tool chain. In *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019.

Tutu Ajayi, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. Toward an open-source digital flow: First learnings from the openroad project. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–4, 2019.

Arcas. Himaformal: Formal verification platform. `https://www.arcas-da.com`. Accessed: 2026-09-16.

Aman Arora, Andrew Boutros, Seyed Alireza Damghani, Karan Mathur, Vedant Mohanty, Tanmay Anand, Mohamed A Elgammal, Kenneth B Kent, Vaughn Betz, and Lizy K John. Koios 2.0: Open-source deep learning benchmarks for fpga architecture and cad research. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):3895–3909, 2023.

Parhami Behrooz. Computer arithmetic: Algorithms and hardware designs. *Oxford University Press*, 19:512583–512585, 2000.

Jayaram Bhasker and Rakesh Chadha. *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.

KAndrea C Bickerstaff, Michael Schulte, and Earl E Swartzlander. Reduced area multipliers. In *Proceedings of International Conference on Application Specific Array Processors (ASAP'93)*, pp. 478–489. IEEE, 1993.

Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pp. 24–40. Springer, 2010.

Brent and Kung. A regular layout for parallel adders. *IEEE transactions on Computers*, 100(3): 260–264, 1982.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Luigi Dadda. Some schemes for fast serial input multipliers. In *1983 IEEE 6th Symposium on Computer Arithmetic (ARITH)*, pp. 52–59. IEEE, 1983.

Kalyanmoy Deb. Multi-objective optimisation using evolutionary algorithms: an introduction. In *Multi-objective evolutionary optimisation for product design and manufacturing*, pp. 3–34. Springer, 2011.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.

Ruogu Ding, Xin Ning, Ulf Schlichtmann, and Weikang Qian. Prefixgpt: Prefix adder optimization by a generative pre-trained transformer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 20808–20815, 2026.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=YicbFdNTTy`.

Noelia Ferruz, Steffen Schmidt, and Birte Höcker. Protgpt2 is a deep unsupervised language model for protein design. *Nature communications*, 13(1):4348, 2022.

John P Fishburn. A depth-decreasing heuristic for combinational logic: or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 361–364, 1991.

Miguel A González, Riccardo Rasconi, and Angelo Oddi. Metaheuristics for multiobjective optimization in energy-efficient job shops. *Engineering Applications of Artificial Intelligence*, 115: 105263, 2022.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Yiran Guo, Lijie Xu, Jie Liu, Dan Ye, and Shuang Qiu. Segment policy optimization: Effective segment-level credit assignment in rl for large language models. *Advances in Neural Information Processing Systems*, 38:114399–114431, 2026.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. The MIT press, 1992.

Hisao Ishibuchi, Ryo Imada, Yu Setoguchi, and Yusuke Nojima. How to specify a reference point in hypervolume calculation for fair performance comparison. *Evolutionary computation*, 26(3): 411–440, 2018.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*, volume 312. Springer Netherlands, 2011.

Nikita Kazeev, Wei Nong, Ignat Romanov, Ruiming Zhu, Andrey Ustyuzhanin, Shuya Yamazaki, and Kedar Hippalgaonkar. Wyckoff transformer: Generation of symmetric crystals. *arXiv preprint arXiv:2503.02407*, 2025.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793, 1973.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

Yao Lai, Jinxin Liu, David Z Pan, and Ping Luo. Scalable and effective arithmetic tree generation for adder and multiplier designs. *Advances in Neural Information Processing Systems*, 37:139151–139178, 2024.

Jinsu Lee and Hoi-Jun Yoo. An overview of energy-efficient hardware accelerators for on-device deep-neural-network training. *IEEE Open Journal of the Solid-State Circuits Society*, 1:115–128, 2021.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Yuzhe Ma, Subhendu Roy, Jin Miao, Jiamin Chen, and Bei 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*, 38(12):2298–2311, 2018.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=S1jE5L5gl.

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=r1gs9JgRZ`.

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nova, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PmLR, 2016.

Takayuki Moto and Mineo Kaneko. Prefix sequence: Optimization of parallel prefix adders using simulated annealing. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5. IEEE, 2018.

Nangate Inc. Open cell library v2008_10_sp1. Nangate 45nm Open Cell Library, 2008.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

B Ramkumar and Harish M Kittur. Optimal final carry propagate adder design for parallel multipliers. *arXiv e-prints*, pp. arXiv–1110, 2011.

Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 853–858. IEEE, 2021.

Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z Pan. Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures. In *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–8, 2013.

Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

Mohammad Mahdi Salmani-Zarchi, Zahra Rahimi, Heshaam Faili, and Mohammad Javad Dousti. Mdp-grpo: Stabilized group relative policy optimization for multi-constraint instruction following. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 42784–42797, 2026.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pp. 1889–1897. Pmlr, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic computers*, (2): 226–231, 1960.

Jialin Song, Rajarshi Roy, Jonathan Raiman, Robert Kirby, Neel Kant, Saad Godil, and Bryan Catanzaro. Multi-objective reinforcement learning with adaptive pareto reset for prefix adder design. In *Workshop on ML for Systems at NeurIPS*, 2022.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in neural information processing systems*, 33:3008–3021, 2020.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Christopher S Wallace. A suggestion for a fast multiplier. *IEEE Transactions on electronic Computers*, (1):14–17, 1964.

Zhihai Wang, Jie Wang, Dongsheng Zuo, Ji Yunjie, Xilin Xia, Yuzhe Ma, Jianye Hao, Mingxuan Yuan, Yongdong Zhang, and Feng Wu. A hierarchical adaptive multi-task reinforcement learning framework for multiplier circuit design. In *Forty-first international conference on machine learning*, 2024.

Zhihai Wang, Jie Wang, Xilin Xia, Dongsheng Zuo, Lei Chen, Yuzhe Ma, Jianye HAO, Mingxuan Yuan, and Feng Wu. Computing circuits optimization via model-based circuit genetic evolution. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=KWH4UIoQKS`.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pp. 1995–2003. PMLR, 2016.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Stephen Williams and Michael Baxter. Icarus verilog: open-source verilog more than a year later. *Linux Journal*, 2002(99):3, 2002.

Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, pp. 1–6, 2013.

Xilin Xia, Jie Wang, Wanbo Zhang, Zhihai Wang, Mingxuan Yuan, Jianye Hao, and Feng Wu. High-performance arithmetic circuit optimization via differentiable architecture search. *Advances in Neural Information Processing Systems*, 38:22208–22241, 2026.

Weihua Xiao, Weikang Qian, and Weiqiang Liu. Gomil: Global optimization of multiplier by integer linear programming. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 374–379. IEEE, 2021.

Chenhao Xue, Kezhi Li, Jiaxing Zhang, Yi Ren, Zhengyuan Shi, Chen Zhang, Yibo Lin, Lining Zhang, Qiang Xu, and Guangyu Sun. Ac-refiner: Efficient arithmetic circuit optimization using conditional diffusion models. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 289–296. IEEE, 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Jixiao Zhang and Chunsheng Zuo. Grpo-lead: A difficulty-aware reinforcement learning approach for concise mathematical reasoning in language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 5642–5665, 2025.

Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M Fonseca, and Viviane Grunert Da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on evolutionary computation*, 7(2):117–132, 2003.

Dongsheng Zuo, Yikang Ouyang, and Yuzhe Ma. Rl-mul: Multiplier design optimization with deep reinforcement learning. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2023.

Dongsheng Zuo, Jiadong Zhu, Chenglin Li, and Yuzhe Ma. Ufo-mac: A unified framework for optimization of high-performance multipliers and multiply-accumulators. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024.

Dongsheng Zuo, Jiadong Zhu, Yikang Ouyang, and Yuzhe Ma. Rl-mul 2.0: Multiplier design optimization with parallel deep reinforcement learning and space reduction. *ACM Transactions on Design Automation of Electronic Systems*, 31(4):1–20, 2026.

A Related Work

A.1 Computing Circuits Optimization

Arithmetic circuits, such as adders and multipliers, are fundamental components in modern computing systems. To meet the increasing demand for high-performance hardware, extensive algorithmic efforts have been devoted to the design and optimization of prefix-tree and compressor-tree structures. Existing approaches can be broadly categorized into three stages. In the early stage, circuit structures were mainly designed manually. For example, Wallace (Wallace, 1964), Dadda (Dadda, 1983), and RA multipliers (Bickerstaff et al., 1993) rely on predefined compression rules to efficiently reduce partial products, while Sklansky (Sklansky, 1960), Kogge-Stone (Kogge & Stone, 1973) and Brent-Kung adders (Brent & Kung, 1982) exploit regular prefix networks to enable parallel carry computation. Subsequently, heuristic optimization algorithms were introduced to automatically explore the large design space of circuit structures, including greedy recursive methods (Fishburn, 1991), pruning algorithms (Roy et al., 2013), simulated annealing (Moto & Kaneko, 2018), and integer programming (Xiao et al., 2021). These methods typically optimize simplified structural metrics, such as size and logic depth. However, since they overlook the complex interactions introduced by practical design flows, including logic synthesis and physical implementation, their optimization objectives may not accurately reflect the final hardware performance. More recently, with the rapid development of machine learning and reinforcement learning techniques (Zuo et al., 2023; 2026; Xue et al., 2026; Wang et al., 2024; 2025; Song et al., 2022; Roy et al., 2021; Ding et al., 2026; Lai et al., 2024; Xia et al., 2026; Ma et al., 2018), data-driven approaches have been increasingly explored for circuit design. By directly leveraging performance feedback from synthesis tools, these methods guide the search and optimization of circuit structures based on actual hardware metrics, opening new opportunities for automated generation of high-performance arithmetic circuits.

A.2 Reinforcement Learning

Reinforcement Learning (RL) is a general framework for sequential decision-making, where an agent learns optimal policies through environment interactions and reward feedback. Existing RL

algorithms can be classified into value-based and policy-based approaches. Value-based methods learn action values to derive policies, including Q-learning (Watkins & Dayan, 1992), Deep Q-Networks (DQN) (Mnih et al., 2015), and its variants (Van Hasselt et al., 2016; Wang et al., 2016; Hessel et al., 2018). In contrast, policy-based methods directly optimize policies, including policy gradient methods (Williams, 1992), actor-critic algorithms (Konda & Tsitsiklis, 1999; Mnih et al., 2016), Trust Region Policy Optimization (TRPO) (Schulman et al., 2015), and Proximal Policy Optimization (PPO) (Schulman et al., 2017). Recently, RL has been extended to optimize large-scale generative models, where policy optimization methods such as Reinforcement Learning from Human Feedback (RLHF) (Christiano et al., 2017) and Group Relative Policy Optimization (GRPO) (Shao et al., 2024; Guo et al., 2025) with its subsequent extensions (Yu et al., 2026; Salmani-Zarchi et al., 2026; Liu et al., 2025; Zhang & Zuo, 2025; Guo et al., 2026) have been developed to improve generation quality and alignment. In this work, we adopt a DAPO-based (Yu et al., 2026) optimization framework to refine the circuit generation process using synthesis-driven rewards.

B Implementation Details

B.1 Synthesis Tool Setup

Under the widely adopted NanGate 45 nm open cell library (Nangate Inc., 2008) in the semiconductor industry, we perform logic synthesis using Yosys (Wolf et al., 2013) and conduct timing analysis using OpenROAD (Ajayi & Blaauw, 2019) to obtain the area and delay of the multiplier designs. Listing 1 and Listing 2 provide the SDC constraints and synthesis scripts, respectively.

Listing 1: SDC Constraints

```
set_driving_cell BUF_X1
set_load 10.0 [all_outputs]
```

Listing 2: Synthesis Script

```
read -sv {verilog_path}
synth -top {top_name}
flatten
opt
abc -fast -D {target_delay} -constr {constr_path} -liberty {liberty_path}
write_verilog {netlist_path}
```

B.2 Legality Guarantee of Constrained Generation

We first analyze the complexity of the multiplier design space explored by MUL-GPT. A multiplier design is represented as an autoregressive token sequence, where each compressor token is defined as $z_i = (c_i, s_i, t_i)$. For an N-bit multiplier, assuming that the compressor tree contains S_c possible stages and T_c available compressor types, the candidate space of each compressor token is

$$|\Omega_z| = 2N \times S_c \times T_c. \tag{10}$$

Let M denote the number of compressors in the compressor tree. Since the number of compressors increases quadratically with the multiplier bit-width, i.e., $M = O(N^2)$ (Zuo et al., 2023), the search space of the complete compressor tree is

$$|\Omega_C| = (2NS_cT_c)^M \leq (2NS_cT_c)^{O(N^2)}. \tag{11}$$

The design space grows exponentially with the multiplier bit-width, making exhaustive enumeration impractical and motivating an efficient learning-based generation framework. The exponentially large design space also necessitates effective legality constraints to eliminate structurally invalid designs during the generation process. The implementation of the dynamic legality constraints is described in Section 4.2. Here, we establish the structural validity of the compressor trees produced by the

constrained generation process. Let H_t denote the complete generation state at step t, including the stage-wise column heights and the associated levelized signal dependencies. We distinguish between valid intermediate states and valid terminal states. Specifically, let $\mathcal{S}_{\text{safe}}$ denote the set of structurally valid intermediate states satisfying non-negativity, valid column indices, and levelized dependency constraints. The initial partial-product matrix may contain columns with more than two signals and is therefore not required to satisfy the final two-row reduction condition. For a state H, let $h_H(c)$ denote the number of live signals in column c at the output frontier. The set of valid terminal states is defined as

$$\mathcal{S}_{\text{term}} = \{H \in \mathcal{S}_{\text{safe}} \mid h_H(c) \leq 2,\ \forall c\}\,. \tag{12}$$

Lemma 1 (Structural validity of accepted sequences)**.** *Suppose that the initial partial-product state satisfies $H_0 \in \mathcal{S}_{\text{safe}}$, each generated action satisfies $a_t \in \mathcal{A}(H_t)$, and the decoder accepts a sequence $Z = (z_1, \ldots, z_T)$ only when its terminal state satisfies $H_T \in \mathcal{S}_{\text{term}}$. Then every intermediate state of an accepted sequence belongs to $\mathcal{S}_{\text{safe}}$, and the accepted sequence represents a structurally valid two-row compressor tree.*

Proof. We first show that the dynamic legality constraints preserve the validity of all intermediate states. Consider a legal compressor action applied at stage s and column c. For an FA and an HA, the required numbers of input signals are $k = 3$ and $k = 2$, respectively. The dynamic mask permits the action only when

$$H_t(s, c) \geq k. \tag{13}$$

Therefore, the state update

$$\begin{aligned} H_{t+1}(s, c) &= H_t(s, c) - k, \\ H_{t+1}(s+1, c) &= H_t(s+1, c) + 1, \\ H_{t+1}(s+1, c+1) &= H_t(s+1, c+1) + 1 \end{aligned} \tag{14}$$

preserves the non-negativity of all column heights. The dynamic mask also excludes actions with invalid column indices or invalid stage transitions. Moreover, the sum and carry outputs are placed only at stage $s + 1$, thereby preserving the levelized and acyclic signal dependencies. Hence,

$$H_t \in \mathcal{S}_{\text{safe}}, \quad a_t \in \mathcal{A}(H_t) \quad \Longrightarrow \quad H_{t+1} \in \mathcal{S}_{\text{safe}}. \tag{15}$$

Since $H_0 \in \mathcal{S}_{\text{safe}}$, induction over the generated actions gives

$$H_t \in \mathcal{S}_{\text{safe}}, \qquad \forall t \leq T. \tag{16}$$

Finally, the decoder accepts the sequence only if $H_T \in \mathcal{S}_{\text{term}}$, which additionally requires

$$h_{H_T}(c) \leq 2, \qquad \forall c. \tag{17}$$

Therefore, every accepted sequence satisfies all intermediate structural constraints as well as the final two-row reduction requirement, and thus represents a structurally valid two-row compressor tree. □

Sequences that do not satisfy the two-row terminal condition within the maximum decoding length are rejected and excluded from subsequent evaluation. Accordingly, Lemma 1 establishes the structural validity of accepted designs rather than guaranteeing successful termination of every sampling trajectory.

B.3 Reward Optimization Analysis of RL Fine-tuning

Let π_θ denote the multiplier generation policy, and define the expected synthesis reward as

$$J(\theta) = \mathbb{E}_{Z \sim \pi_\theta}[R(Z)], \tag{18}$$

where $R(Z)$ is the synthesis-based reward defined in Section 4.3. During RL fine-tuning, we adopt a DAPO-style (Yu et al., 2026) clipped policy optimization objective with KL regularization, as described in equation 7. For on-policy samples, π_{old} denotes the policy used to generate the current batch. For replayed samples, we store the behavior policy π_b that generated each sample and use the corresponding behavior-policy ratio to reduce the distribution mismatch between the current and behavior policies:

$$r_t(\theta) = \frac{\pi_\theta(z_t \mid z_{<t})}{\pi_b(z_t \mid z_{<t})}. \tag{19}$$

For replayed samples, π_{old} is equivalent to π_b. Because replayed trajectories are selected from an elite set, this correction does not generally provide an exactly unbiased estimator of the original on-policy policy gradient. Instead, elite replay is treated as a high-reward sample-reuse mechanism. To analyze the policy-gradient component in isolation, we omit the clipping operation, KL regularization, and elite-selection effects.

Lemma 2. *Consider the idealized expected-reward objective*

$$J(\theta) = \mathbb{E}_{Z\sim\pi_\theta}[R(Z)]. \tag{20}$$

Assume that $J(\theta)$ is twice continuously differentiable in a neighborhood of θ_k. For the idealized on-policy exact gradient-ascent update

$$\theta_{k+1} = \theta_k + \eta\nabla_\theta J(\theta_k), \tag{21}$$

there exists a sufficiently small learning rate $\eta > 0$ such that

$$J(\theta_{k+1}) \geq J(\theta_k). \tag{22}$$

Proof. Using the log-derivative identity, the gradient of the expected synthesis reward can be written as

$$\nabla_\theta J(\theta) = \mathbb{E}_{Z\sim\pi_\theta}\left[R(Z)\nabla_\theta \log \pi_\theta(Z)\right]. \tag{23}$$

Let

$$\Delta\theta = \eta\nabla_\theta J(\theta_k). \tag{24}$$

Since $J(\theta)$ is twice continuously differentiable in a neighborhood of θ_k, its Taylor expansion around θ_k gives

$$\begin{aligned} J(\theta_k + \Delta\theta) &= J(\theta_k) + \nabla_\theta J(\theta_k)^T\Delta\theta + O(\|\Delta\theta\|^2) \\ &= J(\theta_k) + \eta\left\|\nabla_\theta J(\theta_k)\right\|^2 + O(\eta^2). \end{aligned} \tag{25}$$

If $\nabla_\theta J(\theta_k) \neq 0$, the first-order term

$$\eta\left\|\nabla_\theta J(\theta_k)\right\|^2 \tag{26}$$

is strictly positive and is of order $O(\eta)$, whereas the remainder is of order $O(\eta^2)$. Hence, for sufficiently small $\eta > 0$, the positive first-order term dominates the higher-order remainder, yielding

$$J(\theta_{k+1}) > J(\theta_k). \tag{27}$$

If $\nabla_\theta J(\theta_k) = 0$, then

$$\theta_{k+1} = \theta_k, \tag{28}$$

and therefore

$$J(\theta_{k+1}) = J(\theta_k). \tag{29}$$

Combining the two cases gives

$$J(\theta_{k+1}) \geq J(\theta_k), \tag{30}$$

which establishes the local ascent property of the idealized on-policy policy-gradient update. □

This lemma provides an idealized interpretation of the policy-gradient component in the RL fine-tuning stage of MUL-GPT. Because the reward is directly derived from post-synthesis circuit metrics, ascending the expected-reward objective encourages the generator to assign higher probability to multiplier structures with improved area-delay characteristics. In practice, the recorded behavior-policy ratio reduces the distribution mismatch for replayed samples, while elite replay focuses optimization on high-reward designs. Clipping and KL regularization further constrain policy updates and discourage substantial deviation from the pretrained legality-aware generation policy. These mechanisms guide optimization and improve sample utilization, but they do not establish a monotonic-improvement guarantee for the true expected synthesis reward under every practical update.

B.4 CEM-BASED REWIRING ANALYSIS

CEM formulation and implementation. Let $\mathcal{X}_{\text{crit}}$ denote the finite set of feasible inter-compressor rewiring assignments within the current critical slices. Let $p_{\phi_k}(x)$ denote the sampling distribution over $x \in \mathcal{X}_{\text{crit}}$ at iteration k, where lower values of $J(x)$ indicate better post-synthesis performance.

At iteration k, an elite threshold γ_k is selected such that

$$\mathcal{E}_k = \{x \in \mathcal{X}_{\text{crit}} : J(x) \leq \gamma_k\} \tag{31}$$

contains the best-performing candidates. The corresponding idealized elite $q_k(x)$ distribution is

$$q_k(x) = \frac{p_{\phi_k}(x)\mathbb{I}(J(x) \leq \gamma_k)}{\sum_{x' \in \mathcal{X}_{\text{crit}}} p_{\phi_k}(x')\mathbb{I}(J(x') \leq \gamma_k)}. \tag{32}$$

The population-level CEM update fits the sampling distribution to the elite distribution by maximizing the expected elite log-likelihood:

$$\phi^*_{k+1} = \arg\max_{\phi} \mathbb{E}_{x \sim q_k} \left[\log p_\phi(x)\right]. \tag{33}$$

Since q_k is fixed with respect to ϕ, this update is equivalent to minimizing the KL divergence from the elite distribution to the sampling distribution:

$$\phi^*_{k+1} = \arg\min_{\phi} D_{\text{KL}}\left(q_k \,\|\, p_\phi\right). \tag{34}$$

Thus, the idealized CEM update moves the sampling distribution toward high-performing rewiring assignments. Before CEM sampling, the RTL compressor tree is parsed to construct a signal-provenance graph. Each intermediate signal is associated with the partial products that contribute to it, and each compressor is recorded together with its input signals and sum/carry outputs. After synthesis, STA provides a gate-level timing path consisting of the startpoint, endpoint, and a sequence of mapped standard-cell instances. Rather than directly matching these gate instances to RTL compressors, we trace the input nets of the path cells through the mapped-netlist connectivity and recover their contributing partial products. These partial products are then propagated through the RTL signal-provenance graph to identify the corresponding compressor instances and reconstruct the RTL compressor path associated with the STA-reported timing path. The recovered compressors are expanded to slices sharing the same column and stage. CEM then optimizes a one-to-one mapping from slice source signals to compressor input ports or bypass slots, while preserving the compressor types, positions, and output signals. The initial assignment follows a delay-based heuristic, whereas the CEM connection-score matrix is initialized uniformly.

In practice, we implement the CEM update using the Gumbel–Hungarian procedure described in Section 4.4. Candidate rewiring assignments are sampled under the one-to-one matching constraint, evaluated using post-synthesis metrics, and ranked according to their ADP values. The top ρ_e fraction is selected as the elite set, and the connection-score matrix is updated according to the empirical frequencies of source–port assignments in these elite samples. This update provides a tractable, moment-based approximation to the population-level CEM fitting objective. The CEM formulation gives the objective-level interpretation, whereas the Gumbel–Hungarian procedure provides its efficient approximate implementation.

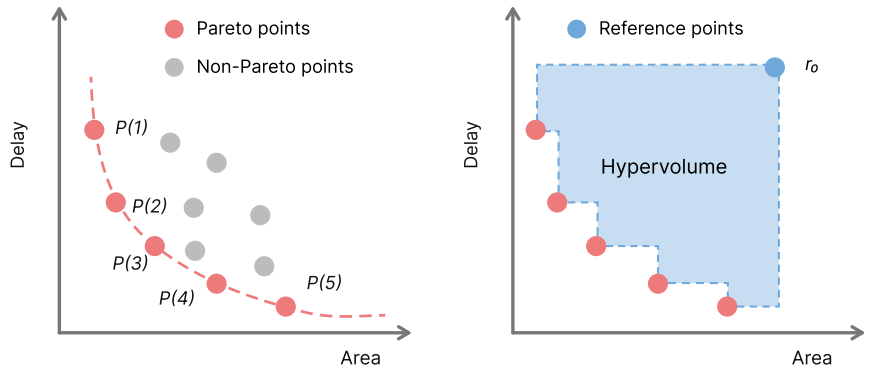
B.5 EVALUATION METRICS

For each generated circuit, we synthesize the Verilog design under the same technology library and timing constraints, and report the post-synthesis area and critical-path delay (Ajayi et al., 2019; Mirhoseini et al., 2021; Kahng et al., 2011). A candidate implementation x is represented by a two-dimensional objective vector

$$\mathbf{f}(x) = (A(x), D(x)), \tag{35}$$

where $A(x)$ denotes the synthesized area and $D(x)$ denotes the critical-path delay. Both objectives are minimized. Since area and delay are typically conflicting objectives, we use the area-delay product (ADP) as a scalar optimization objective during the search stage:

$$\text{ADP}(x) = A(x) \cdot D(x). \tag{36}$$



(a) Pareto Points **(b) Hypervolume**

Figure 6: Illustration of Pareto optimality and hypervolume. (a) Dominated and non-dominated solutions in a two-objective minimization problem. (b) Hypervolume dominated by the Pareto front with respect to the reference point $\mathbf{r}_0$.

ADP provides a simple and efficient way to balance the trade-off between hardware cost and timing performance when ranking individual candidate circuits. This is particularly useful during iterative optimization, where evaluating and comparing a large number of generated circuits is required. Constructing a complete Pareto front for every intermediate candidate would require repeated synthesis under multiple timing targets, which is computationally expensive. Therefore, we first use ADP to identify promising candidates and then perform a more detailed Pareto analysis on the selected high-quality designs. For final evaluation, we further characterize the area-delay trade-off by constructing Pareto fronts. Compared with a single scalar metric such as ADP, the Pareto front preserves the complete trade-off structure between area and delay and does not impose a fixed preference weight between the two objectives. Specifically, for each selected design or design family, we synthesize it under multiple delay targets and collect the resulting candidate implementations. These points characterize the circuit's area–delay trade-offs under different timing constraints, and the non-dominated subset forms the Pareto front. Figure 6(a) illustrates the distinction between dominated and non-dominated candidate implementations. For two candidate implementations $x_i, x_j \in \mathcal{X}$, we say that x_i dominates x_j, denoted by $x_i \prec x_j$, if

$$A(x_i) \le A(x_j), \qquad D(x_i) \le D(x_j),$$

with at least one strict inequality. The Pareto front is formed by the non-dominated candidate implementations among all synthesized results under different delay targets. To quantitatively summarize the quality of a Pareto front, we use the hypervolume indicator. Figure 6(b) illustrates the hypervolume dominated by a Pareto front with respect to a selected reference point. In our experiments, hypervolume is used to evaluate the performance of the same selected design or design family across different delay targets. For a set of non-dominated candidate implementations $\mathcal{P}$, the hypervolume measures the region dominated by $\mathcal{P}$ with respect to a reference point $\mathbf{r} = (A_{\text{ref}}, D_{\text{ref}})$:

$$\text{HV}(\mathcal{P}) = \lambda \left(\bigcup_{x \in \mathcal{P}} [A(x), A_{\text{ref}}] \times [D(x), D_{\text{ref}}] \right), \tag{37}$$

where $\lambda(\cdot)$ denotes the two-dimensional area measure. Since both area and delay are minimized, the reference point is chosen to be worse than all evaluated points (Ishibuchi et al., 2018). To make the hypervolume values comparable within the same task, we use a shared reference point for all methods(González et al., 2022):

$$A_{\text{ref}} = 1.05 \max_x A(x), \qquad D_{\text{ref}} = 1.05 \max_x D(x), \tag{38}$$

where the maximum is taken over all valid evaluated points from all methods for that task. The normalized hypervolume is then computed as

$$\widetilde{\text{HV}} = \frac{\text{HV}}{A_{\text{ref}} D_{\text{ref}}}. \tag{39}$$

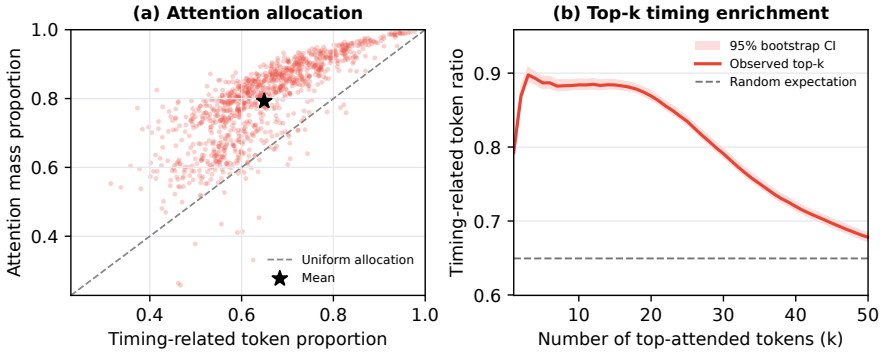


Figure 7: Attention alignment with timing-related compressor tokens. (a) Timing-related compressor tokens receive a larger fraction of attention mass than their token proportion. (b) Top-ranked attention tokens are enriched with timing-related compressor tokens compared with random expectation.

The same reference point is used for all methods within the same task. A larger hypervolume indicates that the design achieves better area-delay trade-offs over the tested delay targets. Thus, ADP and hypervolume serve complementary roles in our evaluation pipeline. ADP is used as an efficient scalar objective for candidate search and preliminary selection, while Pareto fronts and hypervolume are used for the final evaluation of area-delay trade-offs under multiple timing targets.

C Additional Empirical Observations

C.1 Timing Awareness of Compressor Trees Enabled by the Transformer Architecture

A multiplier compressor tree is encoded as an autoregressive sequence of tokens specifying each compressor's column, stage, and type. Causal self-attention (Vaswani et al., 2017) conditions each prediction on preceding operations, enabling the model to capture structural dependencies across columns and compression stages.

To examine whether the learned attention patterns correlate with timing-relevant circuit structures, we compare attention over historical tokens with timing information obtained from post-synthesis STA. For each target token, we label as timing-related all preceding compressors whose outputs can affect its arrival time through the compressor-tree dependency graph, rather than considering only the single longest critical path. For this analysis, we sample 1,000 16-bit multipliers and consistently select the last compressor token on the critical path as the target token for each multiplier. We then compute the attention scores assigned by this target token to all previously generated compressor tokens. Let H denote the set of all valid historical compressor tokens, and let $S_t \subseteq H$ denote the subset of timing-related historical tokens. For token-level attention weights a_i, we measure the fraction of timing-related tokens and the fraction of attention mass assigned to them as

$$\rho_{\text{tok}} = \frac{|S_t|}{|H|}, \qquad \rho_{\text{attn}} = \frac{\sum_{i \in S_t} a_i}{\sum_{j \in H} a_j}. \tag{40}$$

If attention were allocated uniformly with respect to timing relevance, these two quantities would be expected to be similar. Therefore, $\rho_{\text{attn}} > \rho_{\text{tok}}$ indicates that timing-related tokens receive a disproportionally larger amount of attention. We further evaluate whether timing-related tokens are enriched among the highest-attended tokens. Let A_k denote the set of top-k historical compressor tokens ranked by attention score. We define the top-k timing enrichment ratio as

$$\eta_k = \frac{|A_k \cap S_t|}{k}, \tag{41}$$

This metric measures the proportion of timing-related tokens among the most attended historical tokens. We compare it with a random-ranking baseline given by the average timing-related token frequency. As shown in Figure 7, timing-related compressor tokens receive more attention than would

be expected from their token frequency alone. In Figure 7(a), most samples lie above the uniform-allocation line, indicating that the attention mass assigned to timing-related tokens is larger than their proportion among all historical tokens. In Figure 7(b), the top-ranked attention tokens also exhibit clear timing enrichment: the observed η_k remains above the random expectation over a wide range of k. These results suggest that the Transformer does not attend uniformly to all previously generated compression operations. Instead, its attention distribution captures structural dependencies that are correlated with timing-relevant regions of the compressor tree. This provides empirical support for using the Transformer architecture to model multiplier compressor trees, since the model assigns greater attention to previously generated compressors associated with timing-relevant structural dependencies when predicting subsequent tokens.

We emphasize that this attention alignment should not be interpreted as causal evidence that the model explicitly optimizes timing through attention alone. Rather, the analysis serves as an interpretable diagnostic showing that the learned sequence representation encodes structural information correlated with STA-derived timing behavior. Such timing-aware representation capability is consistent with the objective of generating compressor-tree structures with improved area-delay characteristics.

C.2 Effectiveness of Conditional Adder Generation

The final adder of a multiplier receives the two-row partial sums produced by the compressor tree. Unlike a standalone adder, the input bits of this final adder do not arrive simultaneously. Instead, their arrival times are determined by the preceding compressor tree and can vary significantly across bit positions. Therefore, a fixed adder topology may fail to fully exploit the timing characteristics of non-uniform input arrivals, leading to suboptimal area-delay trade-offs.

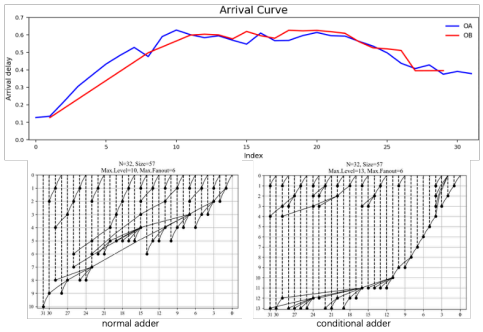


Figure 8: Effect of arrival-time conditioning on final adder generation. The normal adder is generated without arrival-time conditioning, whereas the conditional adder takes the arrival-time curve shown above as input and adapts its structure accordingly.

To address this issue, our conditional adder generation explicitly conditions the adder structure on the bit-wise arrival-time profile produced by the compressor tree. This enables the model to adapt the degree of carry parallelism according to the timing characteristics of different bit regions. Figure 8 presents an illustrative example showing that the normal adder is timing-oblivious, whereas the conditional adder adapts to the non-uniform arrival-time profile. In the lower bit region (0–12 bits), the arrival time increases rapidly across adjacent bits. In this case, later input bits are not available until much later, and aggressively parallel carry computation across this region provides limited timing benefit because the delay is dominated by input availability. A more serial adder structure is therefore preferable, as it can reduce hardware cost while exploiting the staggered input arrivals to overlap part of the carry computation with input arrival. In contrast, in the higher bit region (13–31 bits) where the arrival-time curve becomes relatively flat, multiple adjacent input bits arrive at similar times. In this regime, carry propagation delay becomes a more dominant component of the final adder delay. The conditional adder consequently favors a more parallel structure, which reduces the logic depth of carry computation and improves timing performance. This behavior differs from a normal adder generator, which typically determines the adder topology without explicitly observing the arrival-time profile from the compressor tree. As a result, a normal adder may apply a similar structural preference across bit positions, or optimize only for generic depth and size characteristics, without distinguishing whether a given bit region is limited by input arrival time or by carry propagation.

This observation explains why conditioning on arrival times is beneficial. The conditional adder does not simply minimize the depth of the adder uniformly across all bits. Instead, it allocates parallelism where it is most effective and preserves more serial structures where additional parallelism would mainly increase area without significantly improving delay. As a result, the generated adder better matches the timing profile produced by the compressor tree and achieves a more favorable area-delay trade-off.

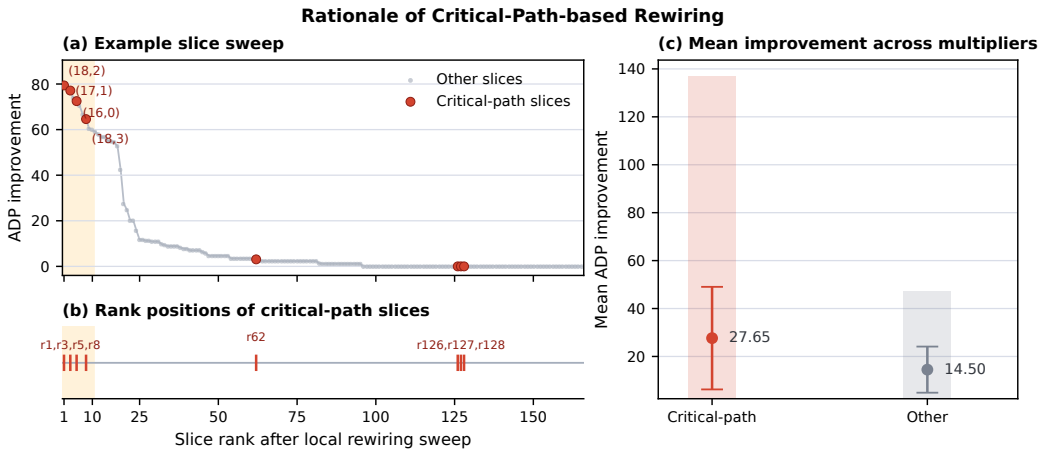


Figure 9: Rationale of critical-path-based rewiring. (a) Representative slice-level rewiring sweep, where red points denote critical-path slices. (b) Rank positions of critical-path slices in the same example. (c) Mean ADP improvement comparison across generated multipliers; markers, error bars, and shaded regions denote the mean, standard deviation, and minimum-to-maximum range, respectively.

C.3 Rationale of Critical-Path-based Rewiring

Our rewiring procedure focuses on compressor slices associated with the current critical path, rather than uniformly exploring all slices in the compressor tree. This design choice is motivated by both timing analysis and empirical observations. From a timing perspective, the post-synthesis delay of a combinational multiplier is determined by its longest input-to-output path. Therefore, local modifications outside the logic cone (Micheli, 1994) of the current critical path are less likely to reduce the overall circuit delay. In contrast, rewiring compressor slices on or near the critical path can directly affect local signal assignment, arrival-time propagation, and the structure of the timing bottleneck. Critical-path-related slices therefore provide a more promising local search space for interconnection optimization.

To empirically validate this intuition, we perform a slice-level rewiring sweep on generated 16-bit multipliers. For each multiplier, each candidate compressor slice is individually selected for local rewiring while the rest of the compressor tree is kept unchanged. The modified circuit is then synthesized and evaluated using the same STA flow, and the improvement is measured by the ADP reduction relative to the original design. Figures 9(a) and 9(b) show a representative example. Candidate slices are ranked by the ADP improvement obtained after local rewiring, where red points denote critical-path-related slices reported by STA and the shaded region denotes the top-10 candidates. In this example, four out of eight critical-path slices fall within the top-10 candidates, suggesting that slices near the critical path are more likely to yield effective improvements. Beyond this example, we repeat the sweep over 100 generated multipliers and compare the average improvement of critical-path slices with that of the remaining slices. As shown in Figure 9(c), critical-path slices achieve larger mean ADP improvement than non-critical slices, nearly doubling the improvement. This consistent gap indicates that the critical path provides an informative prior for selecting local rewiring locations.

Based on this observation, our method restricts the CEM-based local rewiring search to critical-path-related compressor slices. This restriction reduces the combinatorial search space and allocates the optimization budget to regions that are more likely to affect the current timing bottleneck.

D Hyperparameters

In the preceding sections, we have described the implementation details and the major hyperparameters used in our framework. Here, we further summarize the common hyperparameter settings used for comparative evaluation in Table 4.

It is worth noting that the 64-bit setting introduces substantially higher memory overhead. In this case, the generated compressor-tree sequence can contain more than 4,000 tokens. If full self-attention is applied to such long sequences, the GPU memory consumption grows quadratically with the sequence length, making training and sampling significantly more expensive. To reduce memory usage, we adopt sliding-window attention for the 64-bit experiments, with a window size of 512. In addition, the number of samples per context is reduced from $G = 512$ to $G = 256$. For computational efficiency, we also simplify the iterative optimization schedule in the 64-bit setting. Specifically, we run one optimization round, consisting of 200 iterations for compressor-tree generation followed by 200 iterations for conditional-adder generation. This setting provides a practical trade-off between optimization quality and computational cost under the larger design space of 64-bit multipliers.

Table 4: Hyperparameter settings of MUL-GPT.

Component	Setting
Transformer layers	4
Embedding dimension	128
$\lambda^{\text{col}}_{\text{legal}}$ / $\lambda^{\text{stage}}_{\text{legal}}$ / $\lambda^{\text{type}}_{\text{legal}}$	0.02 / 0.02 / 0.005
$\lambda^{\text{col}}_{\text{margin}}$ / $\lambda^{\text{stage}}_{\text{margin}}$ / $\lambda^{\text{type}}_{\text{margin}}$	0.01 / 0.01 / 0.005
δ	1.0
RL group size / iterations	512 / 70
Learning rate / KL coefficient	1.0×10^{-4} / 0.001
CEM samples / rounds / Elite fraction/ α	32 / 30 / 0.2 / 0.5
Tree–Adder loop rounds	3
Elite replay mixing ratio	10%
Temperature	1

E More Experiment Results

We implement MUL-GPT in PyTorch and conduct all experiments on a Linux server equipped with NVIDIA RTX 6000 Ada Generation GPUs and an AMD EPYC 9754 128-core CPU. The software environment uses PyTorch 2.3.1 with CUDA 11.8.

E.1 Functional Simulation and Formal Equivalence Checking

Functional correctness is validated through two complementary procedures. First, RTL simulation is performed using Icarus Verilog (Williams & Baxter, 2002) with directed and randomly generated test vectors. Second, formal equivalence checking is conducted using HimaFormal (Arcas) to verify that each generated RTL design is functionally equivalent to its bit-accurate reference implementation. All generated designs included in the evaluation passed both verification stages.

E.2 Training Dynamics and Run-to-Run Variability

We independently repeat the optimization three times for each multiplier task. Table 5 reports the mean ADP and sample standard deviation across the three runs for the AND-based tasks. The reported variation provides an indication of the run-to-run stability of the optimization procedure under the specified experimental configuration.

Table 5: MUL-GPT ADP on AND-based multipliers over three independent runs (mean ± sample standard deviation).

Bit width	ADP ($\mu\text{m}^2 \cdot \text{ns}$)
8-bit	217.1798 ± 1.0574
16-bit	1387.5684 ± 7.4096
32-bit	7785.7406 ± 79.9516
64-bit	40414.9245 ± 120.4669

To further analyze the optimization behavior of MUL-GPT, we report the training curve of a 32-bit multiplier during the first optimization round. The curve records the evolution of the average design quality during the search process and provides insight into how the policy explores and improves over the design space. It should be noted that MUL-GPT adopts group-relative advantage

Table 6: Cross-generator evaluation of the rewiring module.

Method	16-bit Multiplier			32-bit Multiplier		
	Before	After	Reduction	Before	After	Reduction
Wallace	1793.2282	1722.4240	3.95%	9891.3646	9556.2770	3.39%
GOMIL	1758.3510	1597.7625	9.13%	9154.1055	8986.8443	1.83%
UFO-MAC	1798.3025	1717.6482	4.49%	9290.2269	9275.4480	0.16%
RL-MUL	1701.7258	1630.1408	4.21%	9326.3922	9223.4990	1.10%
MUTE	1622.8680	1620.8595	0.12%	8695.8110	8665.8144	0.34%
ArithTreeRL	1527.6902	1520.0350	0.50%	8392.6080	8220.7392	2.05%
ARITH-DAS	1514.2419	1493.3072	1.38%	8227.2258	8200.9612	0.32%

estimation for policy optimization, and therefore does not directly optimize a conventional supervised loss during this stage. Accordingly, we use the average ADP-based reward of each candidate group as the training metric. This metric reflects the ability of the current policy to generate high-quality arithmetic structures under synthesis-based feedback.

The training curve provides complementary evidence to the final comparison results. While the main results report the final optimized designs, the curve shows how the search evolves over optimization iterations. In particular, it indicates whether the model can continuously discover arithmetic structures with better area-delay characteristics and whether the optimization process gradually stabilizes after sufficient exploration. As shown in Figure 10, MUL-GPT steadily improves the quality of generated designs during training. As more candidate structures are explored, the average ADP-based reward improves, indicating that the policy gradually learns effective search directions in the design space and increases the probability of generating high-performance structures. After an initial stage of rapid improvement, the curve gradually becomes stable. These results show that the proposed training procedure forms a stable and effective optimization trajectory, rather than relying only on a few accidentally discovered high-quality samples. The continuous improvement trend further supports the effectiveness of the policy update mechanism and the synthesis-feedback-driven design optimization used in MUL-GPT.

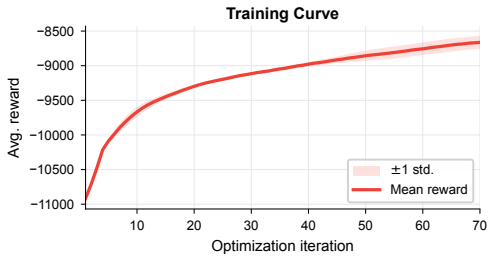


Figure 10: Training curve of MUL-GPT on a 32-bit multiplier in round 1. The average group reward improves as optimization proceeds.

E.3 Cross-Generator Generality of the Rewiring Module

To assess whether the proposed rewiring procedure is specific to the MUL-GPT generator, we apply it as a post-processing step to multiplier designs produced by seven representative multiplier design methods. For all methods, we use the same rewiring configuration, including the CEM population size, number of iterations, elite fraction, and synthesis budget. Table 6 reports the post-synthesis ADP before and after rewiring, together with the relative ADP reduction.

The rewiring procedure reduces ADP for all evaluated methods and operand widths, although the magnitude of the improvement varies across generators. These results indicate that the module is not intrinsically tied to the MUL-GPT generation policy and can be applied to a broader class of compressor-tree-based multiplier designs.

E.4 More Results of Generalization Evaluation

The area and delay of Koios 2.0 benchmark (Arora et al., 2023) designs integrated with our MUL-GPT and baselines are provided in Table 12. It reports the total synthesized area of each complete circuit and the local delay of the replaced arithmetic unit. For the pipelined Koios 2.0 designs evaluated in this work, the clock period is determined by the slowest pipeline stage rather than by

Table 7: Numerical comparison of AND and Booth MACs in area (μm^2), delay (ns), and ADP.

Method	AND						Booth					
	16-bit			32-bit			16-bit			32-bit		
	Area	Delay	ADP	Area	Delay	ADP	Area	Delay	ADP	Area	Delay	ADP
Wallace	1865	1.1810	2202.5650	7039	1.5690	11044.1910	1615	1.1959	1931.3785	5485	1.5516	8510.5260
GOMIL	1741	1.1120	1935.9920	6689	1.4506	9703.0634	**1533**	1.2222	1873.6326	**5269**	1.5673	8258.1037
UFO-MAC	1780	1.1095	1974.9100	6870	1.4138	9712.8060	1592	1.1526	1834.9392	5490	1.4205	7798.5450
RL-MUL	1839	1.0804	1986.8556	6965	1.4288	9951.5920	1640	1.1461	1879.6040	5589	1.4695	8213.0355
MUTE	**1732**	1.0586	1833.4952	**6666**	1.3657	9103.7562	1586	1.1161	1770.1346	5412	1.3618	7370.0616
ArithTreeRL	1779	0.9966	1772.9514	6889	1.3138	9050.7682	1535	1.0463	1606.0705	5363	1.3562	7273.3006
ARITH-DAS	1756	1.0068	1767.9408	6717	1.3224	8882.5608	1586	1.0481	1662.2866	5386	1.3385	7209.1610
MUL-GPT (Ours)	1749	**0.9032**	**1579.6968**	6865	**1.2252**	**8410.9980**	1534	**0.9658**	**1481.5372**	5439	**1.2645**	**6877.6155**

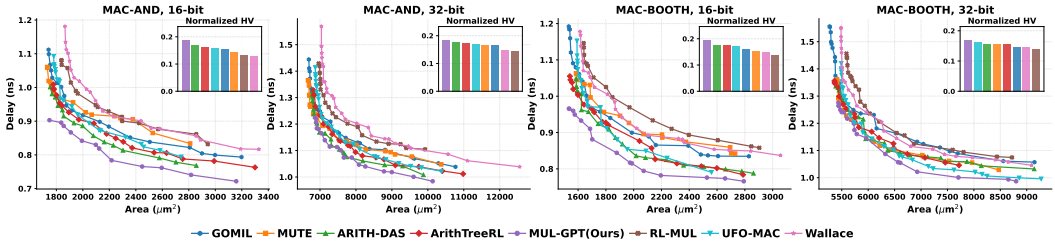


Figure 11: Pareto fronts of MAC-AND and MAC-BOOTH designs.

Table 8: Hypervolume comparison of different multiplier optimization methods.

Method	8-bit Hypervolume		16-bit Hypervolume		32-bit Hypervolume		64-bit Hypervolume	
	AND	Booth	AND	Booth	AND	Booth	AND	Booth
Wallace	156.56	152.52	434.35	533.39	2194.23	2185.98	12663.38	12310.72
GOMIL	172.59	159.91	503.47	651.15	2672.19	2359.18	14974.71	13457.73
UFO-MAC	180.48	151.45	536.96	645.25	2667.93	2412.24	14309.11	13044.74
RL-MUL	162.59	144.61	445.15	622.24	2144.65	2240.66	13441.81	12380.10
ArithTreeRL	185.23	167.82	540.92	632.50	2777.57	2261.73	14960.68	13245.72
MUTE	167.79	148.74	513.03	647.43	2713.34	2332.95	14529.65	13402.80
ARITH-DAS	175.85	169.34	587.86	668.48	2825.71	2415.83	16190.31	13921.71
MUL-GPT (Ours)	**200.38**	**178.72**	**608.00**	**701.78**	**3016.67**	**2573.58**	**16285.39**	**14197.67**

any individual arithmetic unit. We therefore report the local combinational delay of the replaced arithmetic unit to isolate and evaluate its timing contribution.

E.5 More Results of MAC Evaluation

In this section, we present our evaluation of MAC units. The optimal MAC (Chen et al., 2016; Han et al., 2016) implementations achieved by different methods are summarized in Table 7. The visualization of the Pareto front is shown in Figure 11.

E.6 More Results of Hypervolume

In this section, we present the hypervolume evaluation of the Pareto fronts obtained by different methods. The hypervolume results of multiplier optimization are summarized in Table 8, while the results of MAC units and PE arrays are presented in Table 9.

E.7 Runtime Comparison

Because the compared methods explore different numbers of candidate circuits per run, Table 11 reports both the end-to-end wall-clock time per run, covering the complete optimization pipeline, and the runtime normalized to 1,000 explored compressor-tree candidates. Several post-synthesis-driven baselines (Xia et al., 2026; Wang et al., 2025; Lai et al., 2024) use an update loop in which the next design decision depends on the synthesis and timing results of previous candidates. For methods that generate only a small number of candidates per update, this dependency limits within-run parallelism. In contrast, MUL-GPT adopts a batched update strategy. At each update, it generates $G = 512$

Table 9: Hypervolume comparison of different MAC and PE arrays optimization methods.

Method	16-bit AND Hypervolume		16-bit Booth Hypervolume		32-bit AND Hypervolume		32-bit Booth Hypervolume	
	MAC	PE array	MAC	PE array	MAC	PE array	MAC	PE array
Wallace	565.84	37934.33	607.94	32397.71	3195.58	176304.06	2386.75	134863.20
GOMIL	670.06	44562.55	659.76	36073.89	3586.05	196316.68	2419.17	133195.23
UFO-MAC	682.91	45418.70	709.16	37784.24	3658.34	201693.63	2652.30	150416.70
RL-MUL	576.75	38776.70	560.51	30328.85	3163.94	173545.80	2264.17	122625.23
ArithTreeRL	707.57	46280.90	720.13	39514.73	3729.61	206337.75	2566.13	145865.40
MUTE	626.54	41697.44	626.83	34415.40	3615.06	200100.14	2578.87	148877.68
ARITH-DAS	739.31	48142.89	723.75	39757.85	3852.24	214423.81	2579.85	144227.40
MUL-GPT (Ours)	**804.95**	**52531.82**	**793.06**	**42912.46**	**3952.14**	**218974.98**	**2751.33**	**158441.41**

Table 10: Native and unified-post-processing comparison on 16-bit and 32-bit multipliers.

Method	16-bit		32-bit	
	Native ADP	+ CPA + Rewire	Native ADP	+ CPA + Rewire
MUTE	1622.8680	1534.3871	8695.8110	8362.5108
ArithTreeRL	1527.6902	1457.8146	8392.6080	8143.9478
ARITH-DAS	1514.2419	1415.3508	8227.2258	7908.9014

legal compressor-tree candidates and dispatches them to multiple independent EDA workers for parallel synthesis and performance evaluation. The model is updated after the results of the entire batch have been collected. This transforms candidate evaluation from a largely serial process into a batch-parallel process, thereby reducing elapsed time. The primary practical overhead introduced by this strategy is the increased GPU memory consumption during batched candidate sampling and model evaluation. To mitigate this overhead, we employ FP16 mixed-precision training (Micikevicius et al., 2018; Markidis et al., 2018) and gradient accumulation (Goyal et al., 2017), reducing memory consumption while preserving the effective batch size required for optimization. For the 32-bit and 64-bit multiplier tasks, we further employ two-GPU distributed training to accelerate the generation and evaluation of large-scale candidate structures.

The batched optimization strategy evaluates approximately $512 \times 210 = 107{,}520$ compressor-tree structures in a complete MUL-GPT run. Experiments are independently repeated three times, resulting in approximately $3 \times 107{,}520 = 322{,}560$ candidate evaluations for MUL-GPT in total. To improve comparability, we adjust the number of runs and search budgets of the baseline methods so that their total numbers of evaluated candidates are comparable to that of MUL-GPT. Therefore, all methods in Table 1 and Table 2 are compared under similar aggregate compressor-tree exploration budgets. For 64-bit tasks, we uniformly halve the search budgets of all methods due to the higher computational cost of evaluating each candidate. Because MUL-GPT additionally includes conditional final-adder optimization and critical-path-guided rewiring, the comparison should be interpreted as an end-to-end quality comparison under a matched compressor-tree search budget. The computational costs introduced by these two stages are treated as additional method-specific overhead and are therefore not included in the matched budget.

We further perform a unified-post-processing comparison to examine whether the performance advantage of MUL-GPT is solely attributable to its additional downstream optimization modules. Specifically, for the strongest baseline methods, we select the best-performing compressor tree obtained under the corresponding search budget and apply the same conditional final-adder optimization and critical-path-guided rewiring configurations used by MUL-GPT. As shown in Table 10, applying the unified conditional final-adder optimization and critical-path-guided rewiring further reduces the ADP of all evaluated baseline methods, demonstrating that the unified post-processing procedure effectively improves the area-delay trade-off of multiplier designs. Nevertheless, the resulting ADPs of all evaluated baseline methods remain higher than that of MUL-GPT for both operand widths. These results indicate that the performance advantage of MUL-GPT arises from the synergistic optimization of compressor-tree generation, conditional adder optimization, and local interconnection rewiring, rather than from any single component in isolation.

Table 11: Runtime comparison of different multiplier optimization methods.

Method	8-bit Mult.	16-bit Mult.	32-bit Mult.	64-bit Mult.
End-to-end runtime per optimization run (hours)				
ArithTreeRL	1.27	2.27	6.29	26.60
MUTE	4.39	13.40	21.20	81.84
ARITH-DAS	3.06	7.50	38.62	76.43
MUL-GPT (Ours)	**3.56**	**8.20**	**27.22 (2 GPUs)**	**45.98 (2 GPUs)**
Runtime for Exploring 1,000 Candidate Structures (min)				
ArithTreeRL	25.40	45.40	125.73	532.23
MUTE	26.34	80.40	127.20	491.04
ARITH-DAS	18.38	41.65	231.70	458.58
MUL-GPT (Ours)	**1.99**	**4.58**	**15.19 (2 GPUs)**	**53.88 (2 GPUs)**

F LICENSES

The licenses of the tools and baselines used in our experiments are listed below.

1. OpenROAD (Ajayi & Blaauw, 2019): BSD 3-Clause License
2. Yosys (Wolf et al., 2013): ISC License
3. ArithTreeRL (Lai et al., 2024): No License
4. GOMIL (Xiao et al., 2021): No License
5. Icarus Verilog (Williams & Baxter, 2002): GPL-2.0 License
6. MUTE (Wang et al., 2025): No License
7. ARITH-DAS (Xia et al., 2026): No License
8. HimaFormal (Arcas): Educational License

Table 12: Area and Delay of Koios 2.0 benchmark integrated with our MUL-GPT and baselines.

Design	Method	Area(μm^2)↓	Area Impr.(%)↑	Delay(ns)↓	Delay Impr.(%)↑
attention_layer	Wallace	1.0602E+06	NA	1.1396	NA
	GOMIL	1.0563E+06	0.37	1.2205	-7.10
	UFO-MAC	1.0547E+06	0.52	1.1273	1.08
	RL-MUL	1.0619E+06	-0.16	1.1719	-2.83
	MUTE	1.0565E+06	0.35	1.1985	-5.17
	ArithTreeRL	1.0581E+06	0.20	1.1393	0.03
	ARITH-DAS	1.0553E+06	0.46	1.1301	0.83
	MUL-GPT (Ours)	**1.0538E+06**	**0.60**	**1.0891**	**4.43**
conv_layer	Wallace	1.0375E+07	NA	1.0483	NA
	GOMIL	1.0371E+07	0.04	1.1923	-13.74
	UFO-MAC	1.0371E+07	0.04	1.0235	2.37
	RL-MUL	1.0376E+07	-0.01	1.0753	-2.58
	MUTE	1.0370E+07	0.05	1.0624	-1.35
	ArithTreeRL	1.0374E+07	0.01	0.9914	5.43
	ARITH-DAS	1.0370E+07	0.05	1.0247	2.25
	MUL-GPT (Ours)	**1.0369E+07**	**0.06**	**0.9502**	**9.36**
eltwise_layer	Wallace	1.2282E+07	NA	1.0235	NA
	GOMIL	**1.2281E+07**	**0.01**	0.8830	13.73
	UFO-MAC	**1.2281E+07**	**0.01**	0.9090	11.19
	RL-MUL	1.2282E+07	0.00	0.9898	3.29
	MUTE	**1.2281E+07**	**0.01**	0.8672	15.27
	ArithTreeRL	1.2282E+07	0.00	0.9232	9.80
	ARITH-DAS	**1.2281E+07**	**0.01**	0.8285	19.05
	MUL-GPT (Ours)	**1.2281E+07**	**0.01**	**0.8046**	**21.39**
gemm_layer	Wallace	1.7841E+06	NA	0.9626	NA
	GOMIL	1.7625E+06	1.21	0.8824	8.33
	UFO-MAC	1.7626E+06	1.21	0.909	5.57
	RL-MUL	1.7843E+06	-0.01	0.9842	-2.24
	MUTE	1.7613E+06	1.28	0.9165	4.79
	ArithTreeRL	1.7785E+06	0.31	0.9232	4.09
	ARITH-DAS	1.7660E+06	1.01	0.8221	14.60
	MUL-GPT (Ours)	**1.7590E+06**	**1.41**	**0.8127**	**15.57**
lstm	Wallace	1.4570E+06	NA	1.1995	NA
	GOMIL	1.4380E+06	1.30	1.2205	-1.75
	UFO-MAC	1.4301E+06	1.85	1.1273	6.02
	RL-MUL	1.4636E+06	-0.45	1.1997	-0.02
	MUTE	1.4357E+06	1.46	1.1787	1.73
	ArithTreeRL	1.4475E+06	0.65	1.1393	5.02
	ARITH-DAS	1.4319E+06	1.72	1.1365	5.25
	MUL-GPT (Ours)	**1.4254E+06**	**2.17**	**1.0891**	**9.20**
softmax	Wallace	1.1421E+06	NA	1.1922	NA
	GOMIL	1.1368E+06	0.46	1.0658	10.60
	UFO-MAC	1.1311E+06	0.96	1.0982	7.88
	RL-MUL	1.1361E+06	0.53	1.1291	5.29
	MUTE	1.1366E+06	0.48	1.0325	13.40
	ArithTreeRL	1.1344E+06	0.67	1.0620	10.92
	ARITH-DAS	1.1328E+06	0.81	0.9730	18.39
	MUL-GPT (Ours)	**1.1303E+06**	**1.03**	**0.9532**	**20.05**