Graph-to-Gates: Optimization of Parallel Prefix Adders using Reinforcement Learning

Cory Brynds, Michael Castiglia, John Gierlach, Francisco Soriano
Department of Electrical and Computer Engineering
University of Central Florida, Orlando, FL
{co224015, michaelc, jo105085, fr015568}@ucf.edu

***Abstract*—Binary adders are fundamental components within ASIC implementations of arithmetic circuits. Often, EDA tools will instantiate an adder circuit from a standardized set of existing adders, which provide less-than-ideal tradeoffs in power, performance, and area. This is due to the massive $O(2^{N^2})$ design space of adder structures, where the optimal design cannot be feasibly enumerated for large-bitwidth adders. We propose Graph-to-Gates (G2G), a DNN-based reinforcement learning agent that incorporates synthesis inside of the training loop to co-optimize parallel prefix circuits within binary adders for delay and area. Using open-source tooling, G2G is able to achieve an average of 12% less delay and 21.7% less area over commercially-generated 32b adders. The source code for G2G can be found at this github repository.**

***Index Terms*—Parallel Prefix, RL, CNN, Q-Learning**

I. Introduction

Binary adders are used throughout all ASIC and FPGA-based circuits. Countless small adders are used for counters and incrementers. Larger adders ($\geq$32 bits) are used for address computation and arithmetic instructions. Cryptographic hardware (e.g. AES) routinely uses adders as large as 128 bits.

In many use cases, instantiating an adder is as simple as a '+' HDL operator, whereby synthesis will infer the adder's implementation based on the context of the greater circuit. Often, it will choose the most area-efficient implementation (the ripple-carry adder) in situations that are not timing path critical. However, when speed is essential, faster but more expensive adders can be designed by parallelizing the computation of intermediate signals.

The primary bottleneck in an adder circuit is the expensive carry computation, which creates a dependency on the carry out of each subsequent full adder. This carry computation can be formulated as a parallel prefix problem - a class of problems involving the placement and ordering of associative operators to trade off depth (delay) and number of operators (area). The placement and ordering of nodes in a parallel prefix graph have an $O(2^{N^2})$ design space, and prior state-of-the-art parallel prefix adders instantiated by commercial tools can provide suboptimal trade-offs in area and delay, especially for larger adder designs.

II. Related Work

Adders are the fundamental building blocks of virtually all other arithmetic operators, and as such, adder optimization has been extensively studied. Various parallel prefix adder algorithms have been proposed by Sklansky [10], Brent-Kung [1], Kogge-Stone [3], and others, that provide different tradeoffs in area and delay. Table I summarizes the delay,area, and fanout of these approaches as a function of bitwidth.

Algorithm	Delay (# stages)	Area (# nodes)	Max fanout
Serial	$n-1$	$n-1$	2
Sklansky	$\log_2 n$	$\frac{1}{2} n \log_2 n$	$\frac{1}{2} n$
Brent–Kung	$2\log_2 n - 2$	$2n - \log_2 n - 2$	$\log_2 n$
Kogge–Stone	$\log_2 n$	$n \log_2 n - n + 1$	2

TABLE I: Comparison of prefix adder algorithms

Many analytical approaches have been proposed to search and prune the massive $O(2^{N^2})$ design space, including the use of heuristics [9] and simulated annealing [7]. All of these methods are limited to an analytic approach and neglect to account for the impact of logic synthesis and physical implementation.

Recently, machine learning and synthesis-aware optimization approaches have been applied to the parallel prefix circuit problem. PrefixRL [8] proposed the use of a CNN-based reinforcement learning approach with synthesis inside of the training loop for a more accurate reward metric. PrefixAgent [12] utilized LLM agents to explore the design space and E-graphs to provide interpretable reasoning traces. However, these learning-based approaches optimized the adder circuits in a vacuum, neglecting other physical effects such as power consumption and non-uniform signal arrival times. These omissions limit the usefulness of produced designs in actual hardware, where the input stimulus is entirely context-dependent.

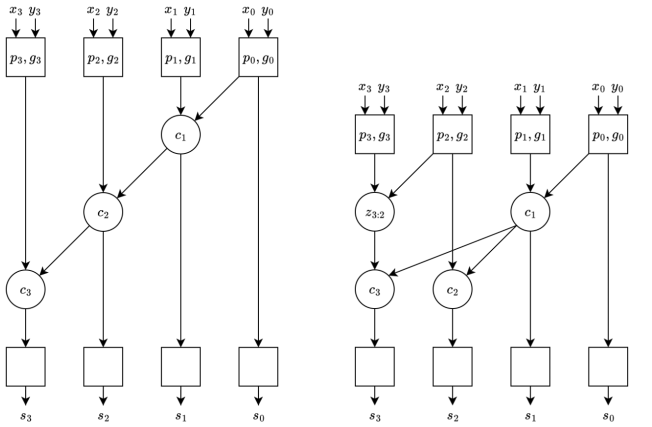
III. Background

A. Prefix Graphs

The carry propagation problem for an N-bit adder circuit can be formulated as a prefix sum computation of the form $y_i = x_i \circ x_{i-1} \circ ... \circ x_0$ for $0 \leq i < N$, where $\circ$ is an associative binary operator on two-bit values representing the generate and propagate bits, shown in Equation 1.

$$(G_1, P_1) \circ (G_2, P_2) = (G_1 + P_1 G_2, P_1 P_2) \tag{1}$$

Constructing a graph of these operators enables control of the tradeoff between the area and delay of carry computation.



(a) Serial carry prefix adder (b) Parallel carry prefix adder

Fig. 1: 4-bit prefix adder structures

For instance, one can create a serial chain of prefix operators similar to a ripple-carry adder, which leads to minimal area but a linear delay. The architecture of a 4-bit serial prefix adder is shown in Figure 1a. In the first stage, the propagate and generate signals for each of the input bit pairs are computed. The second stage, the prefix graph being optimized in this project, computes the composition of its two parents' propagate and generate signals, forwarding the signal to one or more child nodes. Finally, the third stage computes the sum bits by XORing the original propagate signal with each of the computed carry signals (the G_i bit in the two-bit vector). The prefix operations for this four-input sum are

$$c_0 = x_0, c_1 = x_1 \circ c_0, c_2 = x_2 \circ c_1, c_3 = x_3 \circ c_2$$

Alternatively, c_3 and c_2 can be computed in parallel by introducing a term $z_{3:2}$

$$c_0 = x_0, c_1 = x_1 \circ c_0, c_2 = x_2 \circ c_1, z_{3:2} = x_3 \circ x_2, c_3 = z_{3:2} \circ c_1$$

Shown in Figure 1b, $z_{3:2}$ breaks the dependency of c_3 on c_2. The order of parallel prefix computations can be represented as a directed acyclic graph where each node $z_{i:j}$ represents a single operation on a pair of inputs: $z_{i:j} = z_{i:k} \circ z_{k-1:j}$, where $i \geq k > j$. Throughout this paper, we adopt the notation from [8].

B. Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning where an agent learns to make decisions based on how it responds to the environment it is in. The goal of RL is to try different solutions or actions and receive feedback in the form of rewards. Specifically, there is a reward function that will lean towards or against the desired goal of the model. Over time, an RL agent will attempt to maximize its cumulative reward by discovering which steps lead to the best possible outcome.

This is formalized using a Markov Decision Process (MDP), which can be defined as a tuple (S, A, P, R, γ). S is the set of states the agent can be in, A is a set of actions, $P(s'|s, a)$ is the transition probability of moving between states based on the result of the action, $R(s, a)$ is the reward function for a given state with a certain action taken, and $\gamma \in [0, 1]$ is a discount factor to balance short and long term rewards. Overall, the agent will use a policy $\pi(a|s)$, which maps states to actions to maximize the cumulative reward that strikes the best long-term tradeoff. Equation 2 shows the overall RL functionality where G_t is the return of the model at a given time t.

$$G_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \tag{2}$$

C. Deep Q-Network

A DQN is a reinforcement learning method that uses a neural network to approximate the Q-function $Q(s, a)$, which estimates the expected discounted return for taking an action a in a given state s. Instead of relying on a lookup table as in traditional Q-learning, a DQN uses a parameterized function $Q(s, a \mid \theta)$ that can handle high-dimensional state spaces. During training, the network updates its parameters by minimizing the discrepancy between its predicted Q-values and target values derived from the Bellman equation. For each transition (s, a, r, s'), the target value is shown from Equation 3.

$$y = r + \gamma \max_{a'} Q(s'_j, a'_j \mid \theta^-) \tag{3}$$

A target network can be used to stabilize learning by providing consistent target values during training. Instead of using the same network to both select and evaluate actions, DQN maintains a separate network with parameters θ^-, called the target network, which is updated more slowly than the online network.

IV. Implementation

A. Reinforcement Learning Framework

Algorithm 1: Q-learning Algorithm

for *episode* **do**
  reset to base state s_0;
  for *step* **do**
    take action a_i;
    observe (s_i, a_i, s'_i, r_i); add it to $\mathcal{D}$;
    sample (s_j, a_j, s'_j, r_j) from $\mathcal{D}$;
    compute $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using the *target* network $Q_{\phi'}$;
    $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi(s_j, a_j)}{d\phi}\big(Q_\phi(s_j, a_j) - y_j\big)$;
    every N steps, update $\phi' \leftarrow \phi$

For our reinforcement learning framework, we implement the Q-learning algorithm from [6], which defines the RL training process in terms of a sequence of *episodes* and *environment steps*.

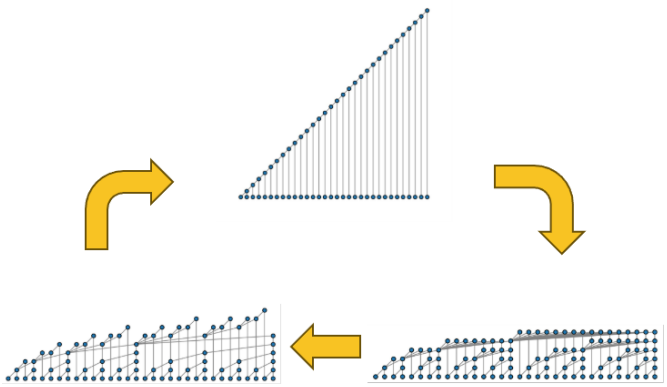


Fig. 2: Episodic Reset Structure

At the beginning of each episode, the environment is initialized to a default state. In the case of PrefixRL, this state was a ripple-carry adder. For our implementation, we opted for a cyclic reset, which resets the initial graph between ripple-carry, Sklansky, and Brent-Kung adders. By altering the initial state of the graph, the Q-network can theoretically converge toward an optimal policy at a faster rate. Figure 2 shows each of the three initial graphs in the cycle.

For each environment step, the Q-network predicts the optimal action that will maximize the cumulative reward for the current state. To encourage exploration, each step has a small probability of selecting a random action, which is determined by an epsilon greedy function defined as:

$$\epsilon(e) = 0.4^{(1+7\,\frac{i(e)}{1023})} \tag{4}$$

Over the course of training, the ϵ-greedy value will exponentially decay such that by the end of training, only the optimal actions are selected. This functionality removes the possibility of over-correcting the graph by adding or deleting more times than necessary.

After taking action a_i, the reward r_i is calculated as the delta between s_i and s'_i. These four values are added to the replay buffer, which acts as a cache for gathered experience. During the gradient update step, a random batch of experience (s_j, a_j, s'_j, r_j) is sampled from the replay buffer. This ensures that the network parameters can be updated not just from current experience but also from experience gathered in past environment states, which improves the quality of learning between iterations.

From the sampled experience, the Bellman target y_j is computed based on the sum of the current reward and the expected future reward (discounted by a factor γ) for the optimal action, evaluated using the *target network*, an outdated copy of the online network currently being trained. Then the parameters of the *online network* are updated based on the loss between y_j and the expected reward from the online network for the sampled action a_j in state s_j. A target network is used to decouple network evaluation and network training. At specified intervals of N, the target network's weights are synchronized with the online network.

In a similar vein to the ϵ-greedy function, a decaying learning rate is applied to reduce dramatic changes to the parameters of the network throughout the course of training, which also assists with over-correction issues. Equation 5 shows the function used for decaying learning rate, where α is a given learning rate over time. Figure 3 shows the DQN update flow based on a given prefix graph per episode/step.

$$\alpha_t = \alpha_0\,\gamma^t \tag{5}$$

B. Graph Encoding

Each environment step, the RL agent has the option of adding or deleting a node from the current graph state. Not all nodes are valid, so a boolean *node list* keeps track of which nodes are present in a graph, and a boolean *min list* keeps track of which nodes can be deleted. Input nodes, output nodes, and nodes that are lower parents of other nodes cannot be deleted. In addition, the *level list* keeps track of the depth at which each node exists, and the *fanout list* tracks how many children each node has. For a given graph state, these four lists allow for the construction of an RTL netlist and form the NxNx4 tensor that is input to the deep neural network approximating the Q-values for each action.

During the RL process, there is a structural legalization check on every prefix graph to ensure that each one adheres to the prefix-adder rules. The validation begins by verifying valid bit ordering, ensuring that each node only combines information from lower index bits. Generate and propagate signals should only be collected in the forward direction from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). Furthermore, this legalization step ensures that no cycles can exist in a prefix graph as a cycle would imply recursive dependency, which would violate the logic flow required for parallel prefix computation.

Additionally, there must be complete carry-dependency paths ensuring the root of all bit computations can be traced back to bit index 0 to ensure the carry-in is fully defined. This would ensure that complete carry-dependency paths exist for every bit. Specifically, the cumulative prefix computation at each prefix node must be traceable back to bit index 0, ensuring that the carry-in at the LSB is well-defined and influences all subsequent bit computations. If any bit's prefix logic cannot be traced back to the LSB, the prefix graph would be deemed incomplete and therefore illegal.

C. Q-Network Architecture

The architecture of our Q-value approximator is adapted directly from PrefixRL, which uses a convolution deep neural network architecture in a residual network configuration (See Figure 4). The input tensors described in Section IV-B serve as the four input channels to the DNN, and the output is an NxNx4 tensor with the estimated Q values Q^{add}_{area}, Q^{del}_{area}, Q^{add}_{delay}, and Q^{del}_{delay}. To select the optimal action for the current

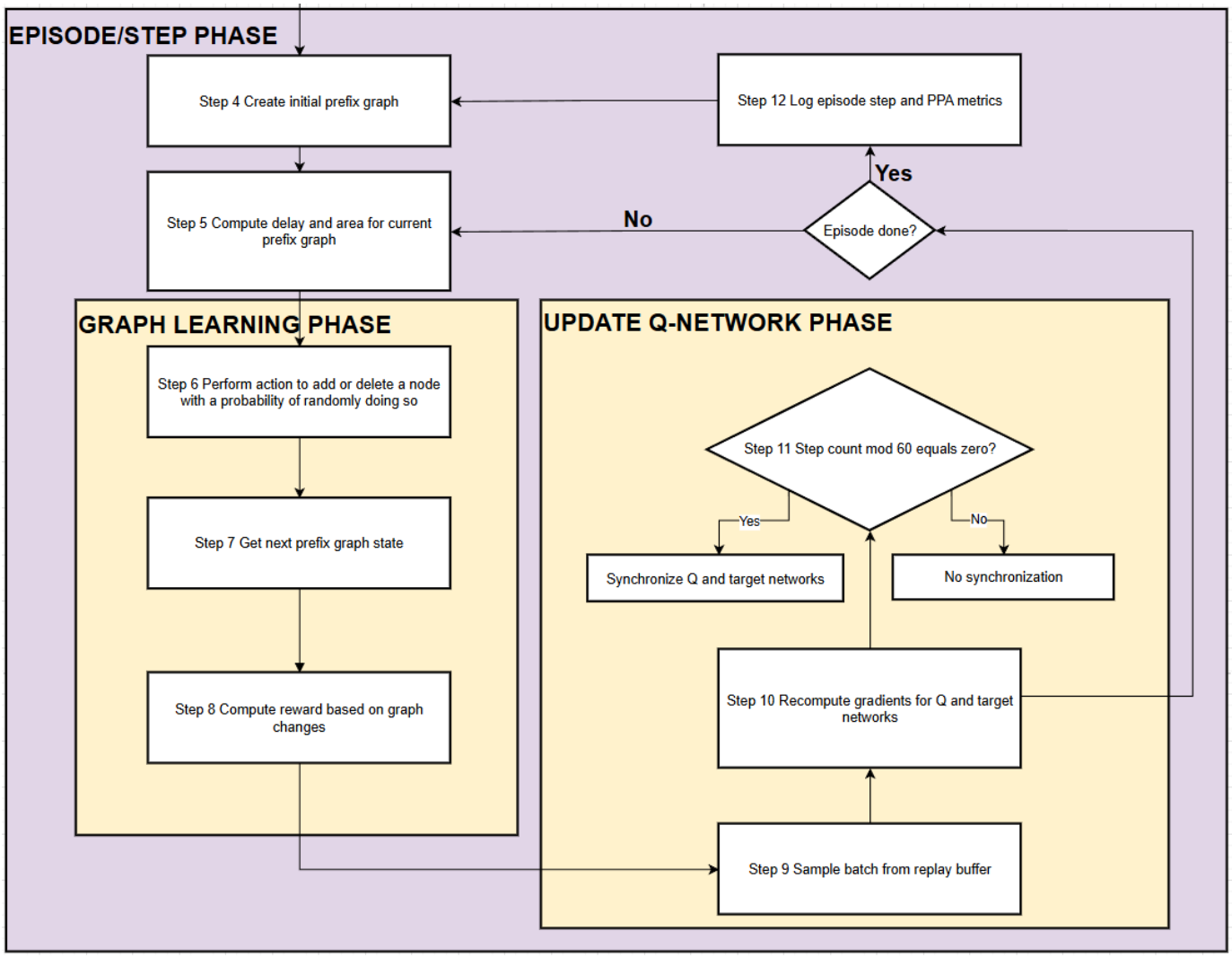


Fig. 3: Flowchart for Q-learning algorithm, highlighting graph creation, environment evaluation, and network update phases.

state, invalid actions (such as locations where a node legally cannot exist) must be masked out by setting those Q values to negative infinity. Afterwards, the four channels are reduced to two channels, which represent the Q-values for adding and deleting a node from the graph. From there, the optimal action can be selected by finding the location of the maximum value in either channel. Scalarized area-delay values are expanded on in Section IV-F.

D. Synthesis-in-the-Loop

After choosing to add or delete a node, the environment must evaluate the next state to compute the reward. Our agent's reward is calculated based on area and delay, which must be obtained from circuit synthesis. Typically, the entire synthesis process is lengthy and includes several rounds of optimizing a design to meet specified constraints. We will leverage a lightweight three-stage evaluation approach that uses a python script for prefix graph to RTL netlist conversion [4], Yosys [11] for logical synthesis, and OpenROAD [2] for placement and routing.

Depicted in Figure 5, the generated gate-level netlist is input to Yosys, which uses ABC to optimize and map the netlist to the Nangate45 standard cell library. Then, OpenROAD performs placement and (optionally) routing on this optimized netlist, which provides the most accurate PPA metrics relative to a fabricated IC.

[4] leverages a two-level retrieval strategy, where PnR is divided into two different flows. The *full flow* performs full placement and routing, with timing-driven optimizations, and the *fast flow* stops after performing placement, neglecting to perform the routing step. While the full flow provides more accurate PPA results, it takes nearly 10x the amount of time as the fast flow to evaluate a single design. A hybrid approach could provide the ideal tradeoff between evaluation speed and accuracy, but we elected to perform RL training using only the fast flow in favor of evaluation speed.

From the generated PnR results, we extract area and delay

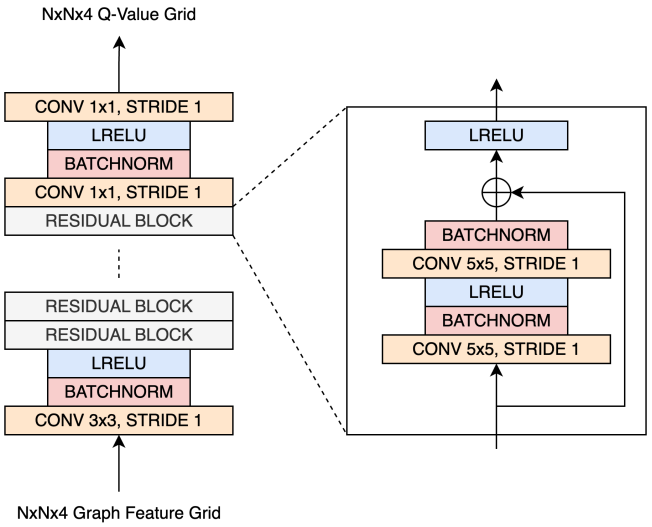


Fig. 4: Q-network architecture

metrics to compute the reward for the RL agent.

E. Analytic Model

In addition to training an RL agent with physical feedback, we developed an analytic model of area and delay. The analytic model is used as a baseline to quickly verify the training results of the synthesis-aware RL agent, which takes much longer to train. The area for a given prefix graph can be approximated by taking the total number of nodes within a given graph. The delay can be approximated by summing the internal node delay, the cumulative fanout delay, and the output gate delay, taken from [5]. Equation 6 shows the simple equation for calculating analytical delay, where $delay_i$, $delay_f$, and $delay_s$ are the internal, fanout, and out bit sum delay values, respectively.

$$delay = delay_i + delay_f + delay_s \tag{6}$$

F. Multi-Objective Scalarized Reward

After taking a given action, the agent must receive feedback in the form of a reward to determine whether the action was advantageous. Ultimately, this reward must be a singular scalar value where a positive reward is correlated with a "good" action and a negative reward is correlated with a "bad" action. However, our circuit optimization problem has multiple objectives, area and delay, so we must scalarize this reward when providing feedback to the agent.

$$r_t = w \times c_{area} \times area_d + (1-w) \times c_{delay} \times delay_d \tag{7}$$

$$area_d = area_t - area_{t+1} \tag{8}$$

$$delay_d = delay_t - delay_{t+1} \tag{9}$$

Equation 7 represents the computation of the scalarized multi-objective reward. c_{area} and c_{delay} are constants used to adjust for the differences in unit scale between area (which is measured in μm^2) and delay (which is measured in ns). For our implementation, we use $c_{area} = 10^{-2}$ and $c_{delay} = 1$. $area_d$ and $delay_d$ represent the difference between the area and delay values of the current and previous states.

A scalar **w** controls the tradeoff between area and delay. **w** is a value in [0,1] that controls how much the network should incentivize area vs delay. For example, $w = 0$ will cancel out the area term and reward the network for pursuing delay-optimized designs, and $w = 1$ will do the same for area. For each training run, this scalar can be varied to expose different tradeoffs in area and delay, which is demonstrated later on in Figure 9.

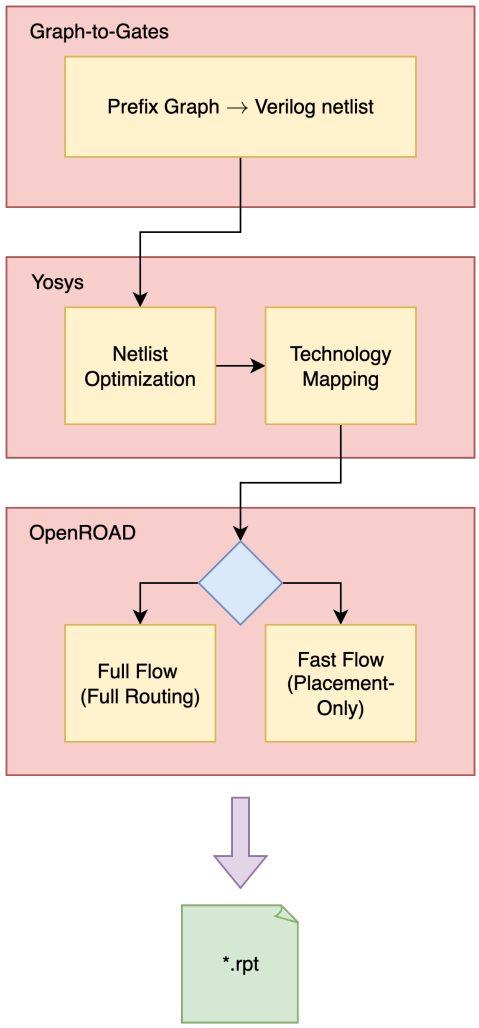


Fig. 5: Synthesis + place-and-route flow used within the RL training loop

G. Training System

We found that parallelizing each part of the workload generally had quite a significant effect on the time required to compute each step. The types of parallelization can be broadly broken down into the following:

Parameter	Value/Configuration
Residual blocks (B)	32
Residual channels (C)	256
Discount factor (γ)	0.75
Experience buffer size	4×10^5
Network synchronization interval	60 gradient steps
Optimizer	Adam
Starting learning rate	4×10^{-5}
c_{area}	10^0
c_{delay}	10^{-2}

TABLE II: DQN Parameter Configuration

Statistic	32b	64b
Number of states	2^{465}	2^{1953}
Number of steps/episode	100	100
Number of episodes	500	100
Batch Size	16	8
w scalar	0.3	0.3
Synthesis and Placement time (sec, avg)	0.75	0.80
Training step time (sec, avg)	2.16	2.25

TABLE III: RL training configuration, using AMD Radeon RX 7900XTX

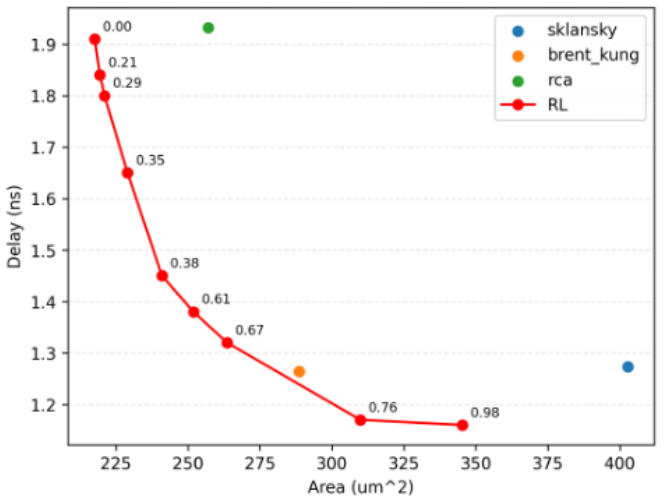


Fig. 6: RL training results for 32b adder against baselines

GPU Parallelizations: By parallelizing the GPU Tensor operations of generating the next step in the training process, through use of the ROCm and CUDA natively in PyTorch, we were able to find a significant reduction in process time for the generation step, where the initialization stage now takes the majority of the time required in these computations. As there is limited VRAM on the GPUs, we found that with the use of a AMD Radeon RX 7900XTX (24GB), the model could support 32 bit at 32 batch size or 64 bit at 8 batch size. Similarly, a NVIDIA RTX 3060 (8GB) could support a 32 bit adder at a batch size of 8.

CPU Parallelizations: By parallelizing the generation of the products used in the rewards function, power and area, we found a great reduction in the time used to compute results with the synthesis-based placement flow. In particular, we find that this workload is approximately 90% parallizable using Amdahl's Law, as the Yosys Synthesis and OpenROAD placement each run primarily on a single core and therefore are heavily bottlenecked by a serialized workflow. For the same reasons, there is little slowdown in this portion of the flow for a batch size up to the number of CPU cores used. The unparallizable part of this flow is the migration of Tensors from the GPU to the CPU, when GPU Parallelization is used. This overhead causes an overall slowdown when parallelization is used with the analytical model.

V. Results

A. RL Training

We evaluated the efficiency of the G2G framework by generating 32-bit and 64-bit adders and comparing them against standard parallel prefix architectures (ripple carry, Sklansky, and Brent-Kung) synthesized with Yosys and implemented using the OpenROAD flow. Table II shows the hyperparameter configuration for the RL network, and Table III shows the training configuration used to gather our results for 32b and 64b prefix graphs. It is important to note that the number of episodes and environment steps is a fraction of those used by PrefixRL; this was due to resource and time constraints. With longer training iterations, we could have likely achieved even better quality of results.

Figure 6 depicts the area-delay metrics of nine different prefix adders evaluated by the RL environment for a *training* weight scalar of $w = 0.3$. These reflect designs that minimized a scalarized area-delay score for an *evaluation* weight scalar w_e. For instance, the design with an area of $\approx 250\mu m^2$ and a delay of $\approx 1.38ns$, corresponding to the "0.61" point in the plot, minimized the scalar score for $w_e = 0.61$:

$$score = w_e \times c_{area} \times area + (1 - w_e) \times c_{delay} \times delay$$

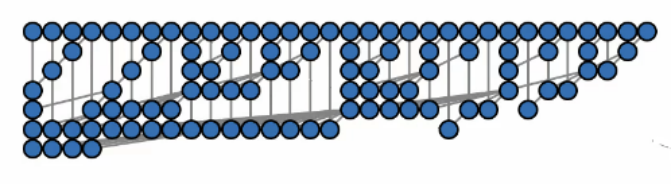
$$(0.61 \times 10^{-2} \times 250) + (0.39 \times 1 \times 1.38) = 2.036$$

Legacy graphs such as Sklansky, Brent–Kung, and RCA are shown as fixed reference points. Out of the reference designs, RCA achieves minimal area but suffers from high delay, while Sklansky minimizes delay but at the expense of significantly larger area. Brent–Kung offers marginally lower delay than Sklansky with 39% less area.

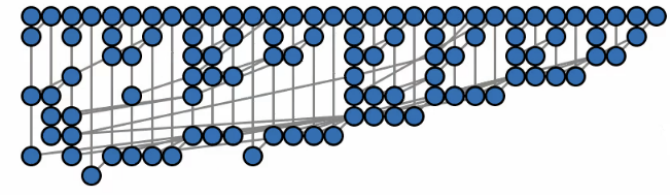
The red curve traces designs produced by the RL agent at different values of w_e. For $w_e = 0$, the adder structure resembles an RCA in terms of delay, and for larger values of w_e, RL-optimized adders are able to outperform Brent-Kung and Sklansky in terms of delay with lower area. As learning progressed, the RL agent identified prefix graph structures with combinations that reduced carry propagation depth, enabling substantial delay reductions with only incremental area increases. This illustrates the RL agent's ability to intelligently explore the enormous design space and uncover optimized architectural tradeoffs.

B. Importance of Synthesis in the Loop

It was imperative for PrefixRL to include logic synthesis within the training loop [8] as it captured real physical data



(a) Analytic model



(b) Synthesis-in-the-loop

Fig. 7: Fastest 32b adder circuits produced by using analytic model vs synthesis-in-the-loop when training

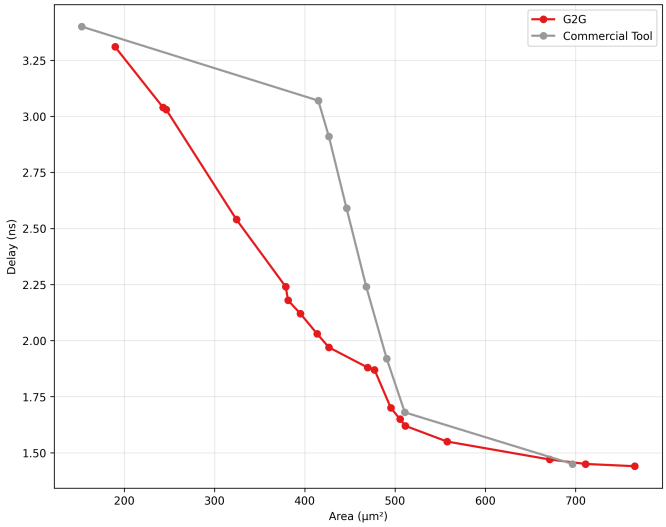
that the RL agent could leverage. This feedback allowed the agent to not rely on analytical delay and area scalars but instead receive rewards from post-synthesis delay and area results. Pure analytical delay and area scalars often fail to capture activity in real hardware. For example, many of the prefix circuits declared "optimal" by the analytical model possessed large fanouts on internal nodes and clustering of many nodes in the same local area, which could lead to less-than-optimal delay due to fanout and routing congestion. This is demonstrated by Figure 7, which shows 32b prefix circuits declared "fastest" for training using only the analytic model and only synthesis-in-the-loop.

This methodology ensured that every modification to a prefix graph is evaluated from a physical design/implementation perspective. As a result, the RL agent is able to go beyond just proxy metrics optimizations and generate designs that can generalize across standard cell libraries.

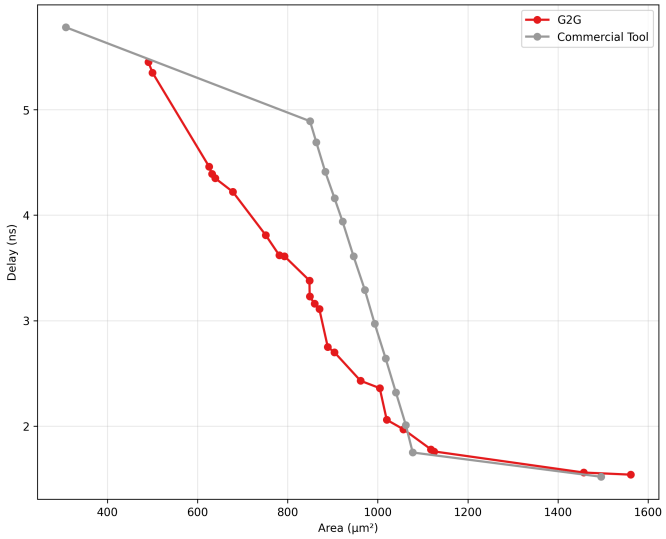
C. Commercial Tool Evaluation

To evaluate the optimal prefix circuits designed by G2G, we compare the designs against those produced by commercial EDA tools. By comparing G2G's adder circuits against industry-standard adder macros, we demonstrate the quality of G2G's results and its ability to generalize to commercial process technologies. All designs were synthesized using Synopsys RTL Architect on a 32nm process. In Figure 9, "G2G" results represents the synthesis results of 10 different adder circuits obtained during RL training with **w**-optimal scalar values ranging from 0.0-1.0. "Commercial Tool" represents the macro cell components inferred by Synopsys' synthesis engine from a behavioral instantiation of **a+b** in RTL.

Figures 8a and 8b show the area-delay Pareto curves for 32b and 64b adders, respectively. G2G is able to produce adders that Pareto-dominate commercial tools for all but the lowest delay targets. Results for 32b adders demonstrate an average of 12.01% reduced delay and 21.7% reduced area, with maximum



(a) 32b adder synthesis

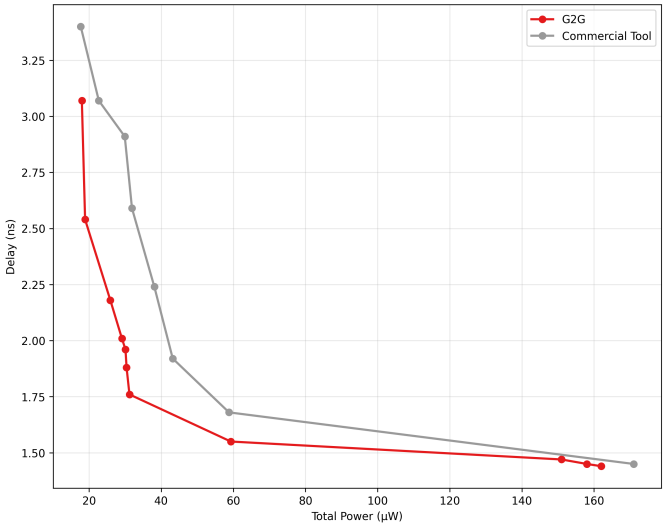


(b) 64b adder synthesis

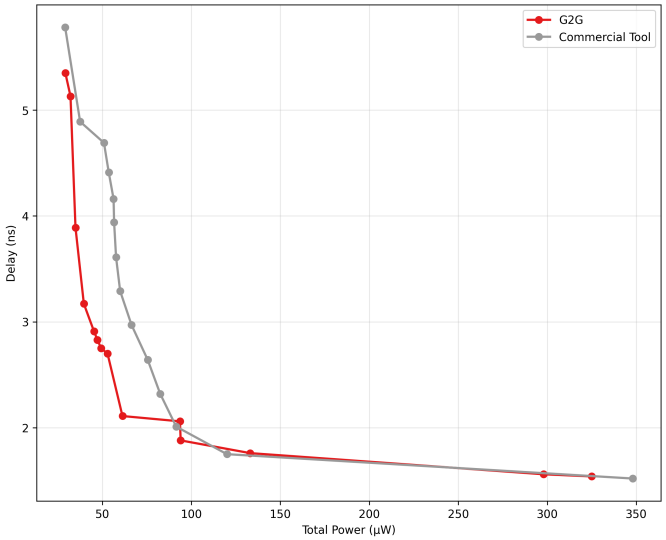
Fig. 8: Area-delay Pareto curves for 32b and 64b adders (Commercial tool vs RL-optimized adders)

savings of **33.9% for delay** and **42.8% for area**. Comparable savings are seen for 64b results, with average savings of 10.8% for delay and 14.9% for area. The maximum reduction in delay in area is 36.6% and 33.4%, respectively.

In addition, although we did not explicitly optimize for power efficiency, the design decisions made by the RL agent produced impressive power reductions over standard Synopsys adders. Results were obtained using Synopsys PrimePower and are expressed as the sum of switching, internal, and leakage power. Figures 9a and 9b show the power-delay Pareto curves for 32b and 64b adder designs. G2G Pareto dominates commercial tooling for 32b results, and either dominates or matches the results of commercial tool for the 64b designs. For 32b results, G2G produced adders with average power



(a) 32b adder synthesis



(b) 64b adder synthesis

Fig. 9: Power-delay Pareto curves for 32b and 64b adders (Commercial tool vs RL-optimized adders)

savings of 35.7% and a **max power saving of 51.0%**. For 64b results, the average power reduction was 26.0%, and the max power reduction was 39.1%.

These results exceeded our expectations of the potential savings from RL-driven optimization, especially due to the limited training we were able to run with respect to PrefixRL's implementation. With increased training time, G2G can likely produce adders with further-improved PPA metrics over the results presented in this report.

VI. Conclusion

In this work, we presented G2G, a reinforcement learning framework leveraging deep Q-networks to optimize parallel prefix circuits. By integrating physical synthesis directly into the training loop, G2G is able to produce circuits closer to the global optimum when compared to analytical proxy metrics. Our results demonstrate that the proposed agent can effectively navigate the $O(2^{N^2})$ design space, generating adder topologies that Pareto-dominate standard architectural baselines such as Sklansky and Brent-Kung, as well as commercial standard designs from Synopsys.

Furthermore, the introduction of episodic resets and physical-aware rewards allowed the agent to converge significantly faster than random search or traditional heuristic methods. While this study focused on 32-bit and 64-bit adders using the Nangate45 technology library, the underlying framework is technology-agnostic. Future work will explore the application of G2G to non-uniform signal arrival times and the co-optimization of datapath components beyond simple arithmetic units, pushing the boundaries of ML-driven circuit design.

References

[1] Brent and Kung. "A Regular Layout for Parallel Adders". In: *IEEE Transactions on Computers* C-31.3 (1982), pp. 260–264. DOI: 10.1109/TC.1982.1675982.

[2] A.B. Kahng and T Spyrou. "The OpenROAD Project: Unleashing Hardware Innovation". In: *Proc. Government Microcircuit Applications and Critical Technology Conference* 2021 (). URL: https://vlsicad.ucsd.edu/Publications/Conferences/383/c383.pdf.

[3] Peter M. Kogge and Harold S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *IEEE Transactions on Computers* C-22.8 (1973), pp. 786–793. DOI: 10.1109/TC.1973.5009159.

[4] Yao Lai et al. "Scalable and Effective Arithmetic Tree Generation for Adder and Multiplier Designs". In: *Advances in Neural Information Processing Systems*. Ed. by A. Globerson et al. Vol. 37. Curran Associates, Inc., 2024, pp. 139151–139178. DOI: https://doi.org/10.48550/arXiv.2405.06758. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/fb23cf87a9e04d7677b73c47acd060ef-Paper-Conference.pdf.

[5] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[6] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[7] 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)*. 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351414.

[8] Rajarshi Roy et al. "PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE,

Dec. 2021, pp. 853–858. ISBN: 9781665432740. DOI: 10.1109/DAC18074.2021.9586094. URL: https://ieeexplore.ieee.org/document/9586094/ (visited on 09/25/2025).

[9] S. Roy et al. "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.10 (2014), pp. 1517–1530. DOI: 10.1109/TCAD.2014.2334313.

[10] J. Sklansky. "Conditional-Sum Addition Logic". In: *IRE Transactions on Electronic Computers* EC-9.2 (1960), pp. 226–231. DOI: 10.1109/TEC.1960.5219822.

[11] Clifford Wolf, Johann Glaser, and Johannes Kepler. "Yosys-A Free Verilog Synthesis Suite". In: 2013. URL: https://api.semanticscholar.org/CorpusID:202611483.

[12] Dongsheng Zuo et al. *PrefixAgent: An LLM-Powered Design Framework for Efficient Prefix Adder Optimization*. 2025. arXiv: 2507.06127 [cs.AR]. URL: https://arxiv.org/abs/2507.06127.