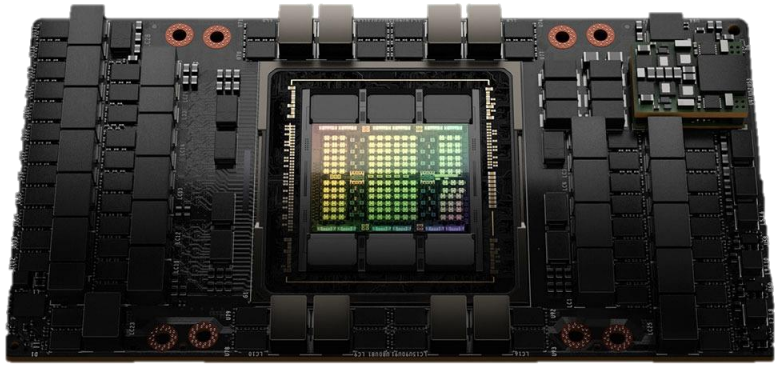
Graph-to-Gates: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning

Presented by **Cory Brynds, Francisco Soriano, Michael Castiglia, and John Gierlach**

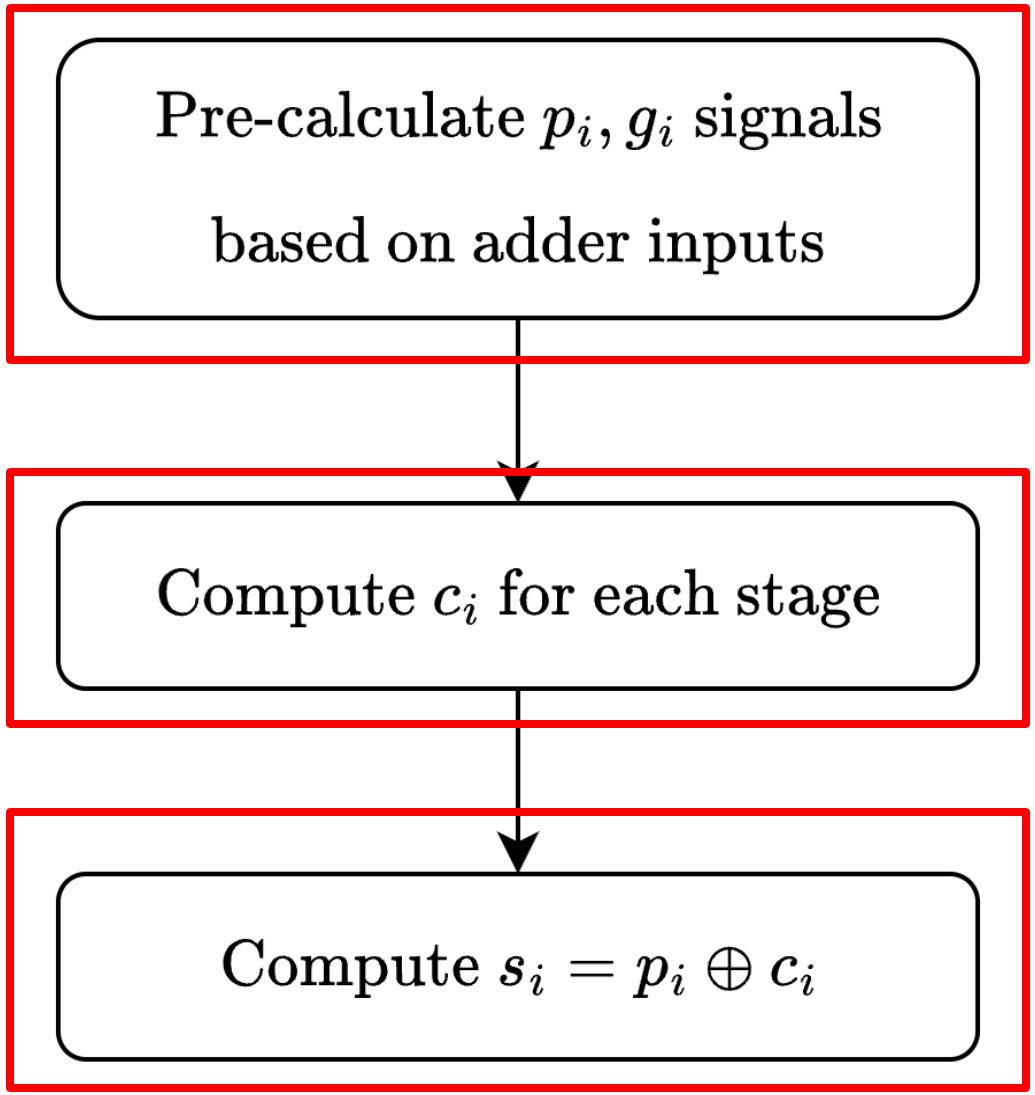
Motivation

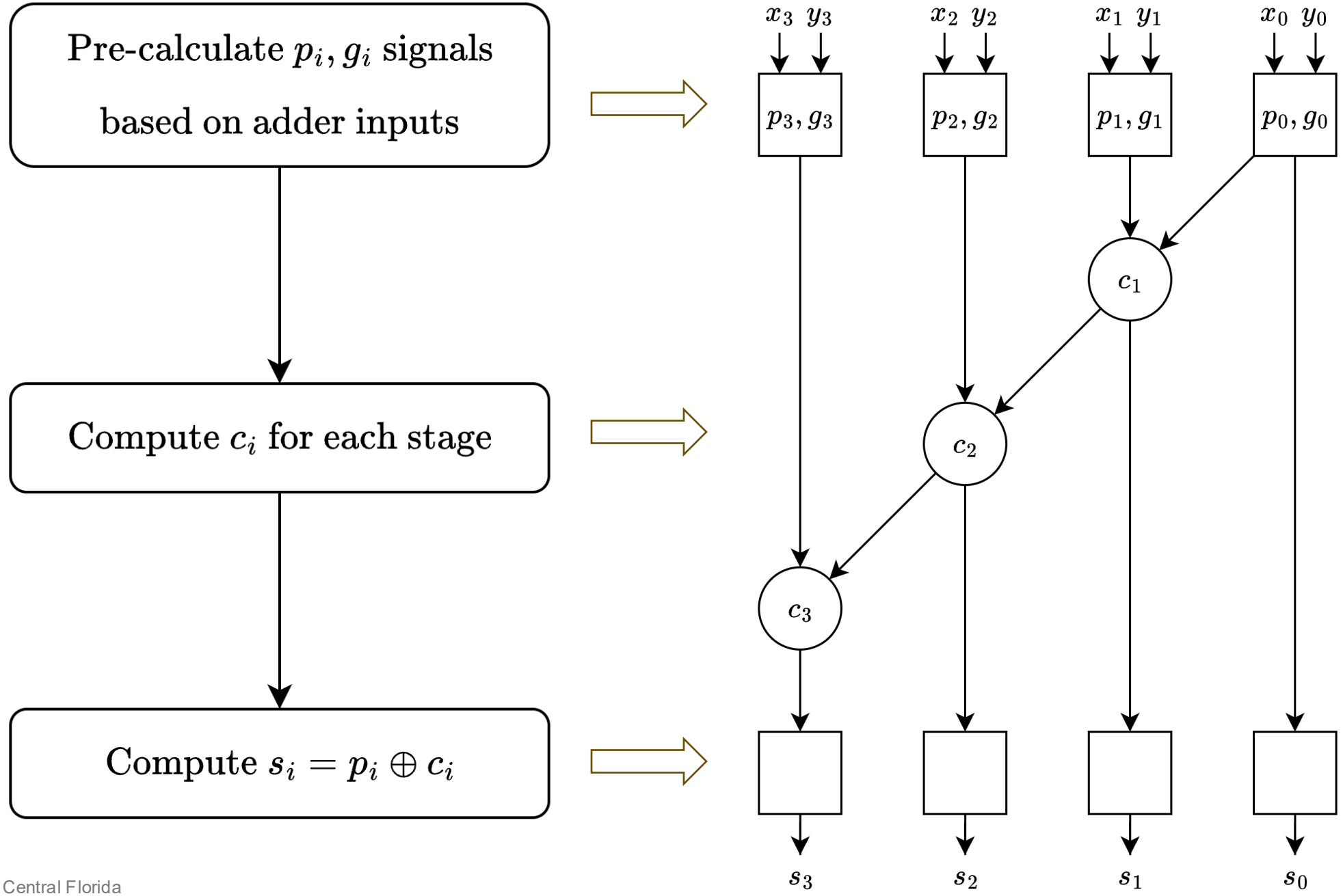
- Vast amounts of arithmetic circuits are used in CPUs and GPUs to achieve unprecedented acceleration for AI, high-performance computing, and computer graphics
- Improving the design of these arithmetic circuits has a massive impact on the performance and efficiency of these processors
- Binary adders are fundamental & extensively used throughout processors (e.g. ALUs, counters)
- Example: **NVIDIA H100**
  - 144 streaming multiprocessors (SMs) per GPU
  - 128 FP32 CUDA cores and 4 tensor cores per SM
  - **18,432 FP32s CUDA cores** and **576 tensor cores** per H100



The importance of design optimizations scales with core count

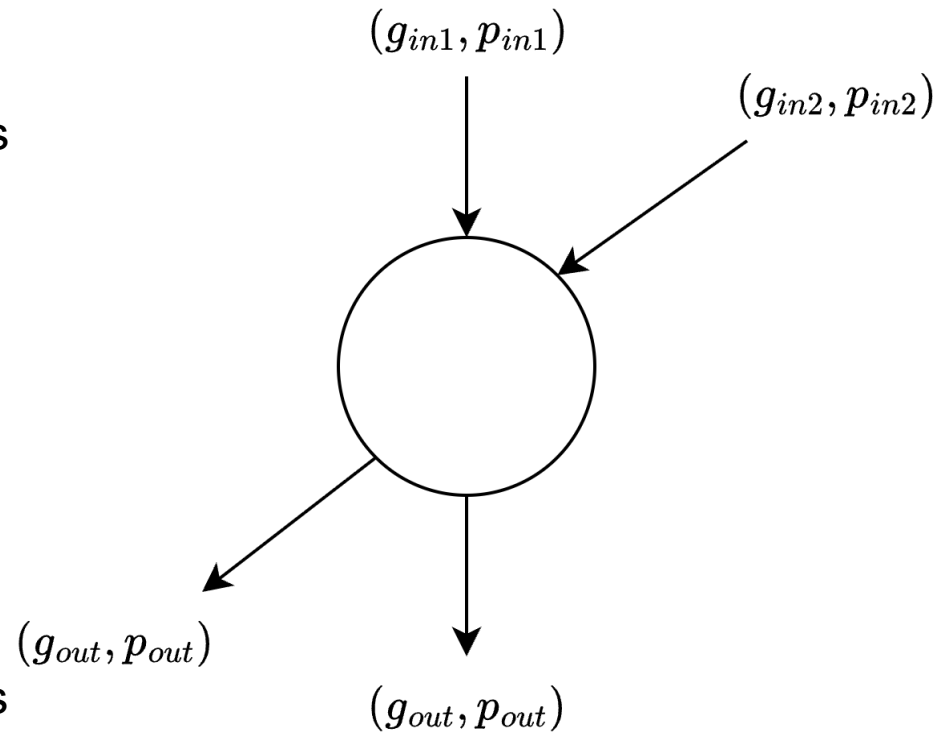
Review: Parallel Prefix Addition



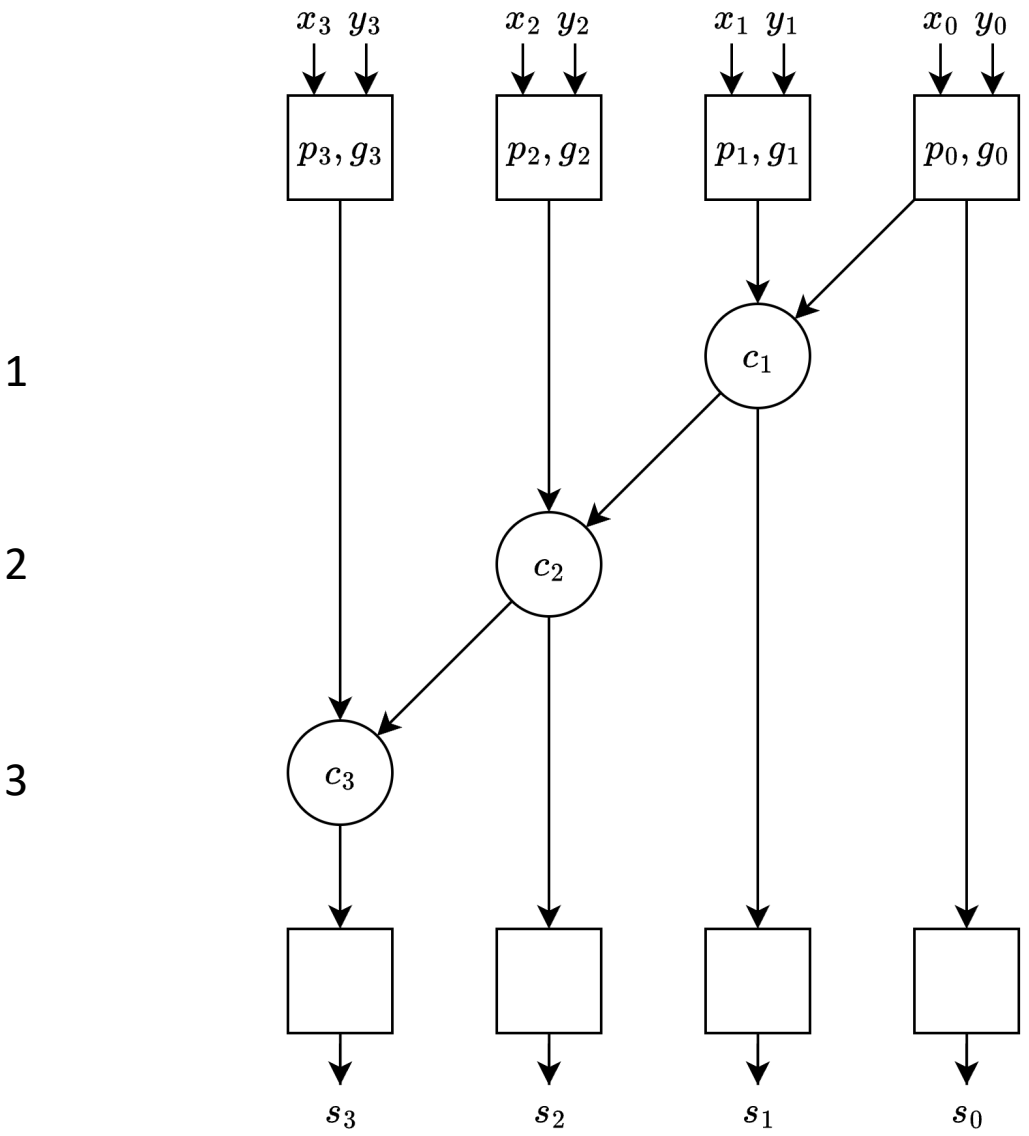

Pre-calculate p_i, g_i signals based on adder inputs
Compute c_i for each stage
Compute $s_i = p_i \oplus c_i$
x_3 y_3
x_2 y_2
x_1 y_1
x_0 y_0
p_3, g_3
p_2, g_2
p_1, g_1
p_0, g_0
c_1
c_2
c_3
s_3
s_2
s_1
s_0

Prefix Nodes

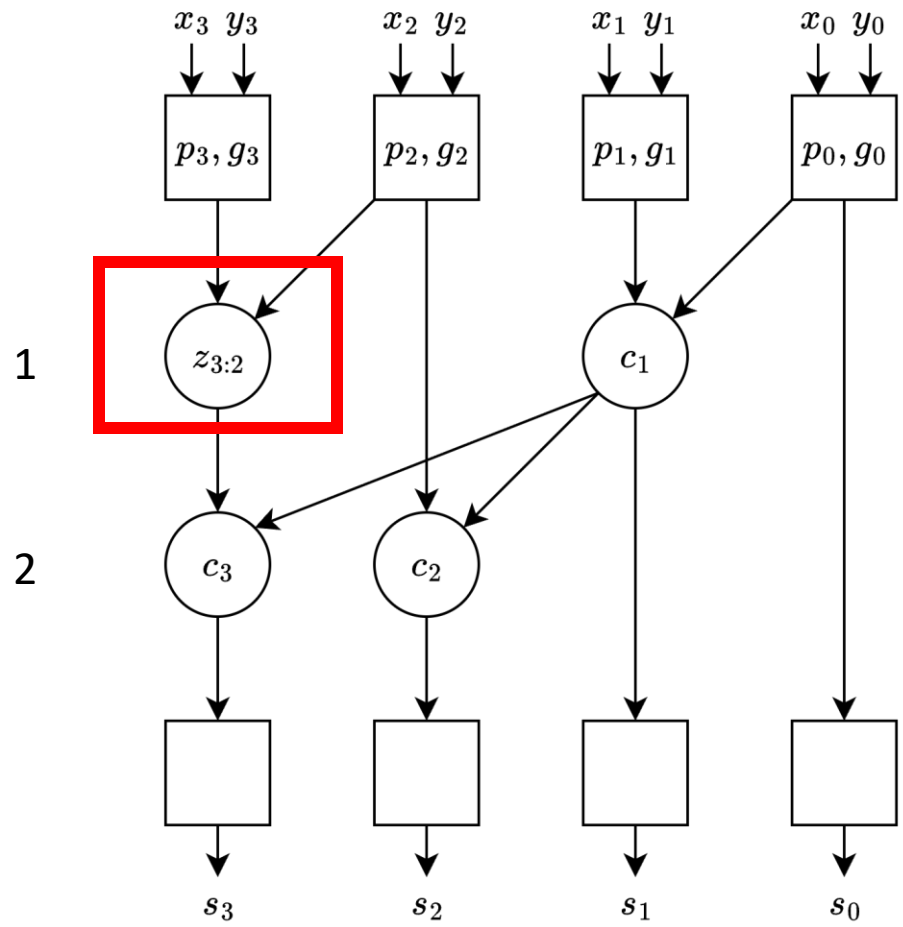
- Here we define an operator where the inputs are two 2-bit vectors (g_i, p_i) used to calculate the following bits
- $\boldsymbol{Generate_{out}}$
  - $g_{out} = g_{in1} \vee (g_{in2} \wedge p_{in1})$
- $\boldsymbol{Propagate_{out}}$
  - $p_{out} = p_{in1} \wedge p_{in2}$
- The output is a 2-bit vector $(g_{\text{out}}, p_{\text{out}})$ that represents the combined generate and propagate signals from both input signals



Designing a Parallel Prefix Adder

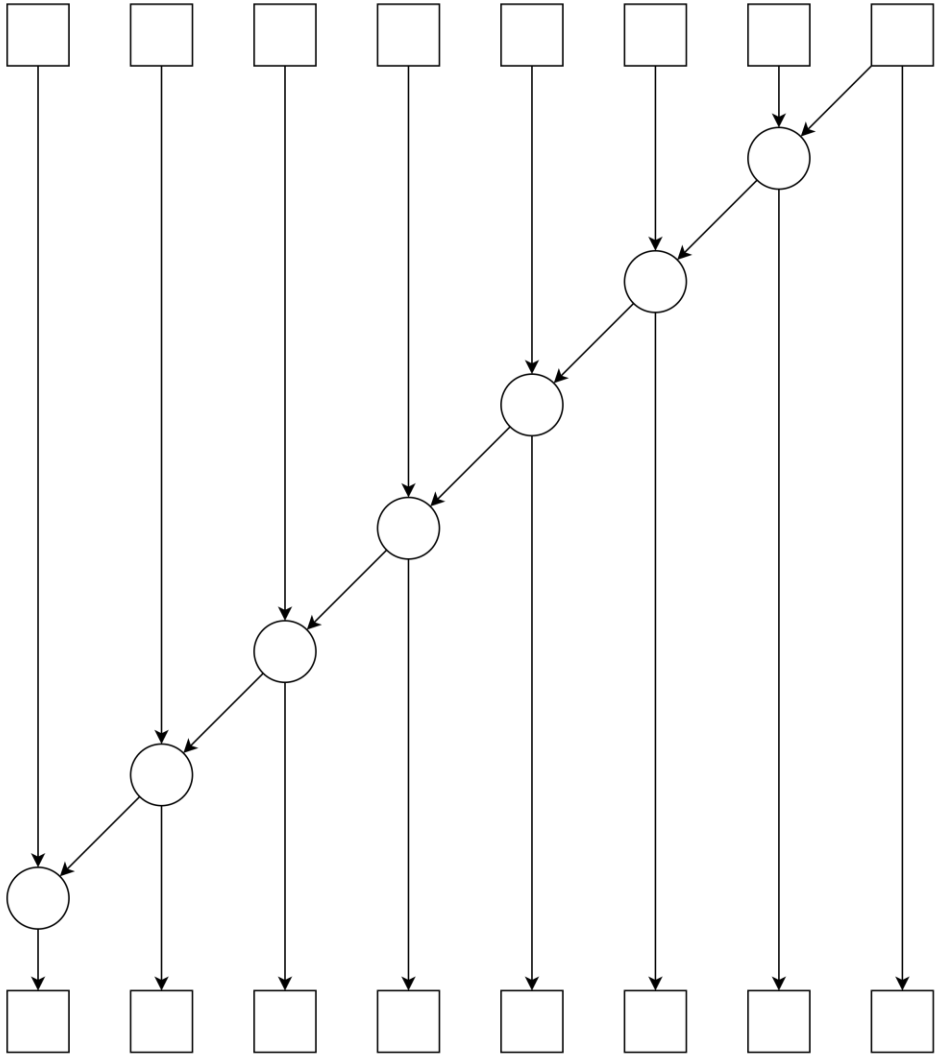


Sequential Carry (RCA)

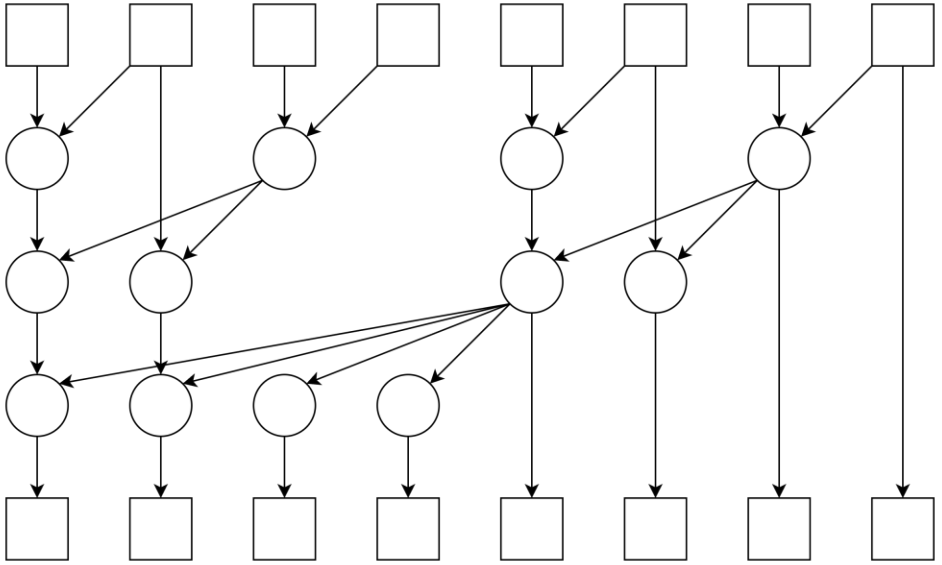


Parallel Carry

8-Bit Parallel Prefix Adder



Serial (RCA)



Parallel (Sklansky)

Adder Type	Delay (Stages)	Area (Node count)
Serial (RCA)	7	7 (7 OR, 14 AND gates)
Parallel (Sklansky)	$log_2(8) = 3$	12 (12 OR, 24 AND gates)

The Parallel Prefix Problem

- Existing parallel prefix algorithms tend to offer minimum delay **OR** reduced area but do not offer a balanced tradeoff between the two metrics
- There are $\frac{(N-1)(N-2)}{2}$ locations where a prefix node may or may not exist, meaning there are $O(2^{N^2})$ valid N-input prefix graphs!
- Exhaustively searching for the best tradeoff in this enormous design space is impractical
  - The total design space for valid prefix graphs for a 32-bit adder is $O(2^{465})$

Is it possible for a model to explore and optimize graphs within this enormous design space?

Existing Prefix Graph Optimization Solutions

- **Historical Parallel Prefix Algorithms**
  - Serial (RCA): **Worst delay**, best area, minimal fanout
  - Sklansky (1960): Best delay, higher area than serial, **worst fanout**
  - Kogge-Stone (1973): Best delay, **worst area**, minimal fanout
  - Brent-Kung (1982): Higher delay than Sklanksy/KS, lower area than Sklansky, bounded but not minimal fanout

Algorithm	Delay (# of stages)	Area (# of 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	$2log_2 n - 2$	$2n - log_2 n - 2$	$log_2 n$
Kogge-Stone	$log_2 n$	$n log_2 n - n + 1$	2

Existing Prefix Graph Optimization Solutions

- Historical Parallel Prefix Algorithms
  - Serial (RCA): **Worst delay**, best area, minimal fanout
  - Sklansky (1960): Best delay, higher area than serial, **worst fanout**
  - Kogge-Stone (1973): Best delay, **worst area**, minimal fanout
  - Brent-Kung (1982): Higher delay than Sklanksy/KS, lower area than Sklansky, bounded but not minimal fanout
- **Analytic Approaches**
  - Pruning via heuristics and delay/area prediction with ML model
  - Simulated annealing: randomly modify prefix graph towards optimal point – using an analytic model

Existing Prefix Graph Optimization Solutions

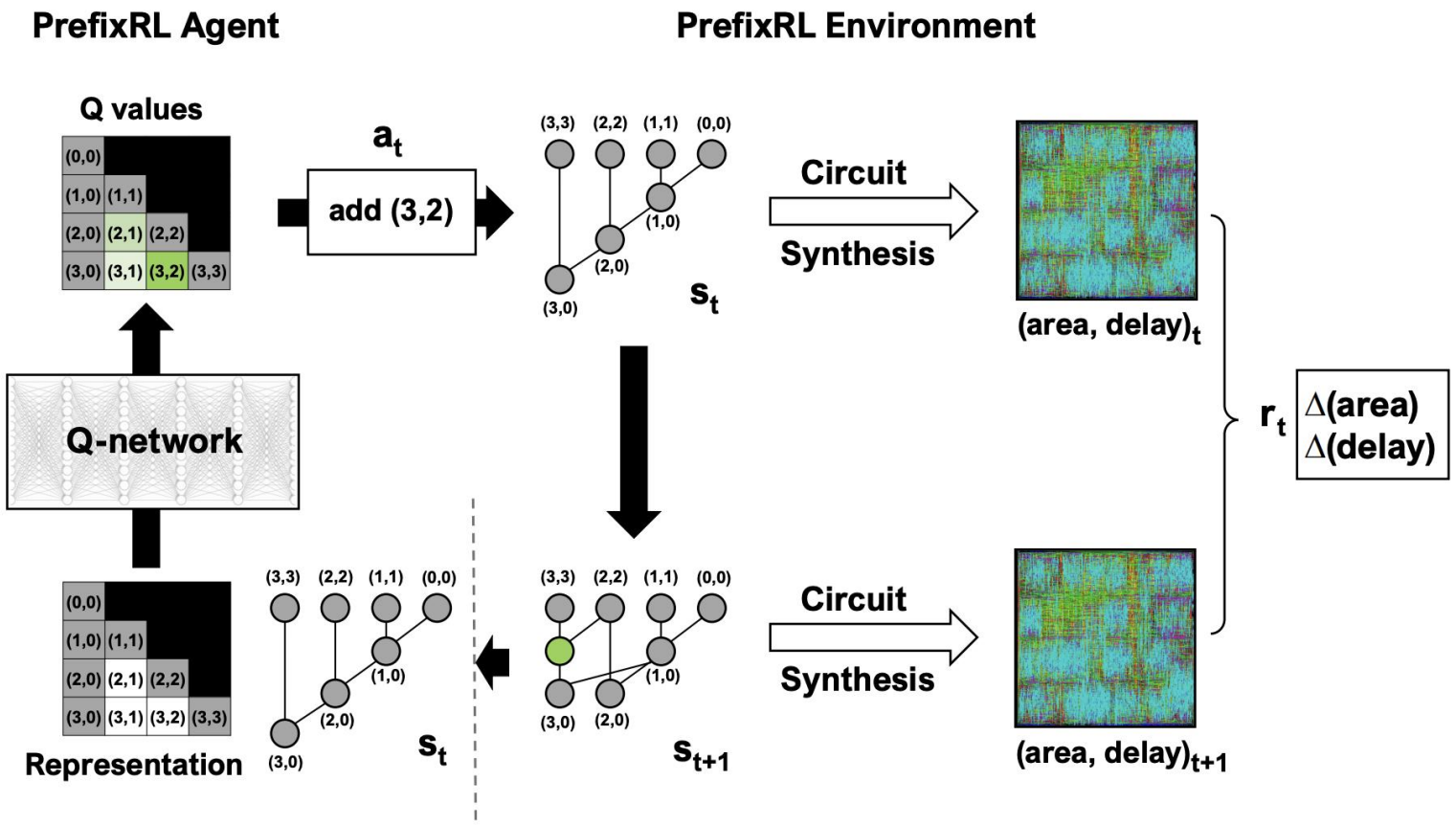
- Historical Parallel Prefix Algorithms
  - Serial (RCA): **Worst delay**, best area, minimal fanout
  - Sklansky (1960): Best delay, higher area than serial, **worst fanout**
  - Kogge-Stone (1973): Best delay, **worst area**, minimal fanout
  - Brent-Kung (1982): Higher delay than Sklanksy/KS, lower area than Sklansky, bounded but not minimal fanout
- Analytic Approaches
  - Pruning via heuristics and delay/area prediction with ML model
  - Simulated annealing: randomly modify prefix graph towards optimal point – using an analytic model
- **PrefixRL**: RL-based approach that explores design space with physical synthesis evaluation metrics

PrefixRL: An RL Model for Optimizing Parallel Prefix Circuits

- Reinforcement learning framework that trains an agent to explore an unrestricted prefix graph design space to **co-optimize** for delay and area
    - Incorporates physical synthesis **inside of the training loop** to extract more accurate metrics than prior analytical models
    - Optimizes prefix circuits for 32-bit and 64-bit adders that **Pareto-dominate** all prior designs – offering better area-delay tradeoffs with at least 1 improved metric and no metrics worse than previous designs

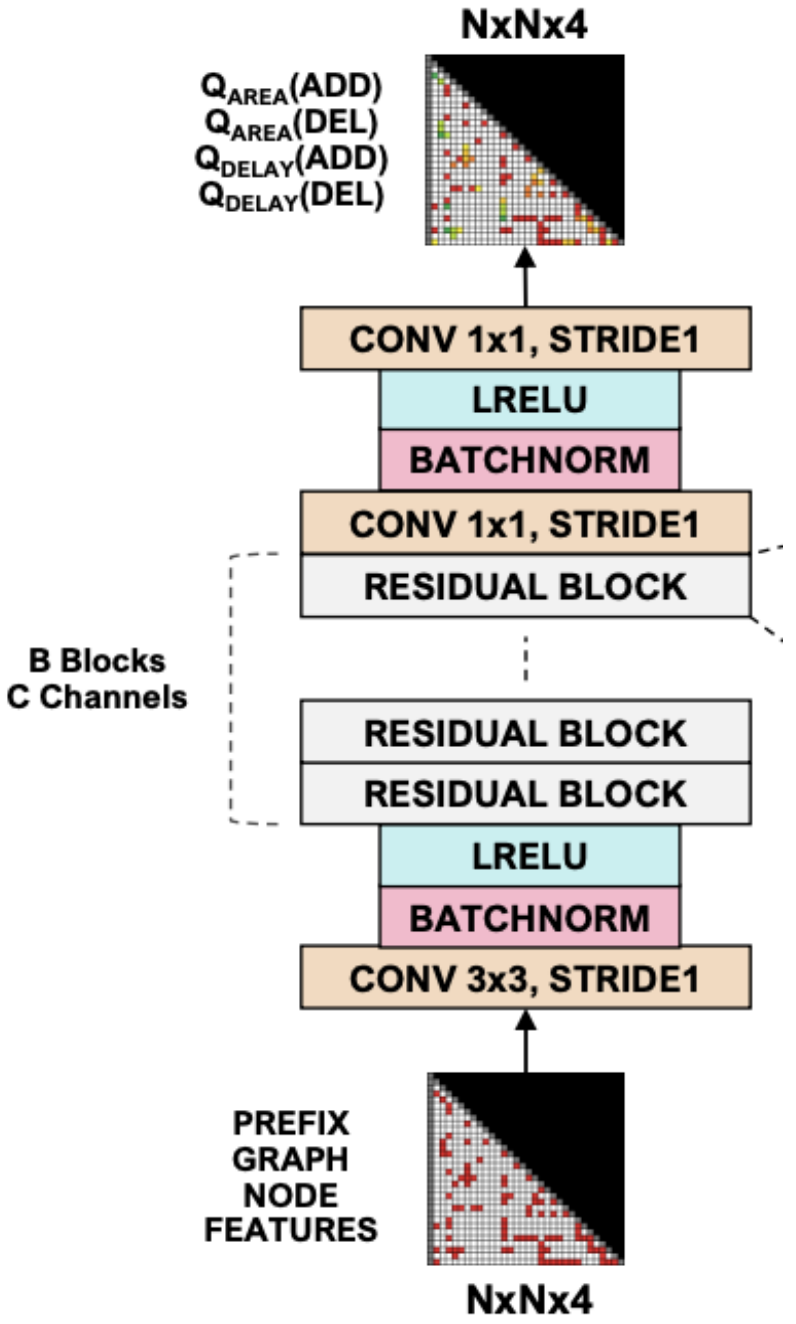
Reinforcement Learning Environment

- **Task** – maximize cumulative reward via finding prefix adder graphs that optimize area & delay tradeoff
- **State space** – all valid NxN prefix graphs
- **Action** – add or delete a prefix node
- **Reward** – returns area-delay tradeoff



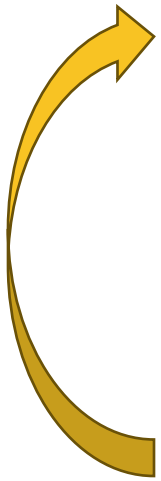
Deep Q-Network (DQN) Architecture

- Our Q-network utilizes a CNN architecture
- The **input** to the DQN is an NxN grid with 4 channels indicating:
    1. Presence of a node (1 if present, else 0)
    2. Deletability of a node (1 if delectable, else 0)
    3. Level of node (Normalized to [0,1])
    4. Fanout of node (Normalized to [0,1])
- The DQN **outputs** an NxN grid with 4 channels, corresponding to the Q values for adding and deleting a node, in terms of area and delay
- The RL agent iteratively constructs the prefix graph by **selecting actions** according to its policy informed by **Q-values**
- We use **B=32, C=256** for 32b and 64b adder training



Deep Q-Learning Algorithm

- To train our reinforcement learning model, we implement the classic Deep Q-Learning Algorithm[1]



1. Take an action and observe (s_i, a_i, s_{i+1}, r_i), adding it to the replay buffer
2. Sample a random batch (s_j, a_j, s_{j+1}, r_j) from the replay buffer
3. Compute the bellman target $y_j = r_j + \gamma max_{a_j'} Q_{\phi'}(s_j', a_j')$ (the sum of the current reward and the expected future reward from the **target network** $Q_{\phi'}$)
4. Update the parameters of the **online network** based on $\text{Loss}(Q_\phi(s_j, a_j), y_j)$
5. Every N steps, update $\phi' \leftarrow \phi$ (synchronize target network's parameters with online network)

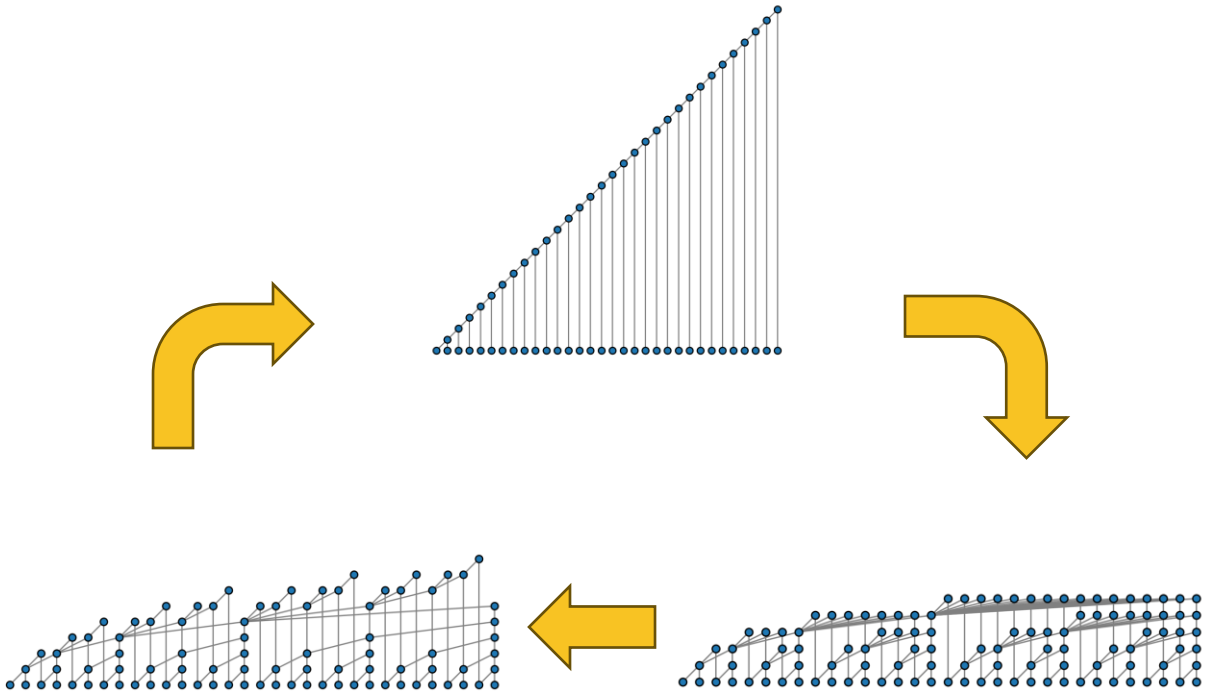
[1]Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." (2013).

Network Initialization

- In the initialization phase, the online and target Q-networks are initialized with random weights
- Separate target and online networks are used to create a separation between network evaluation and network updating to avoid feedback loops and improve error reduction while training.
- The Adam optimizer is initialized with a dynamic learning rate - exponentially decreasing as training goes on
- We use a **replay buffer** with 400000 elements to cache the results of past experiences to improve training stability. During the network update phase, batches will be randomly sampled from the replay buffer

Episodic Reset

- At the beginning of a new episode, the environment is reset to an initial graph state
- The PrefixRL paper resets to a ripple-carry adder (RCA) graph at the beginning of each episode
- Modifying the initial starting state may allow the Q-network to convergence to an optimal policy more quickly[1]
- For our implementation, we cyclically reset to a ripple carry, Sklansky, and Brent-Kung adder on each episode



[1]Song, Roy, et. al. Multi-objective reinforcement learning with adaptive pareto reset for prefix adder design. NeurIPS 2022.

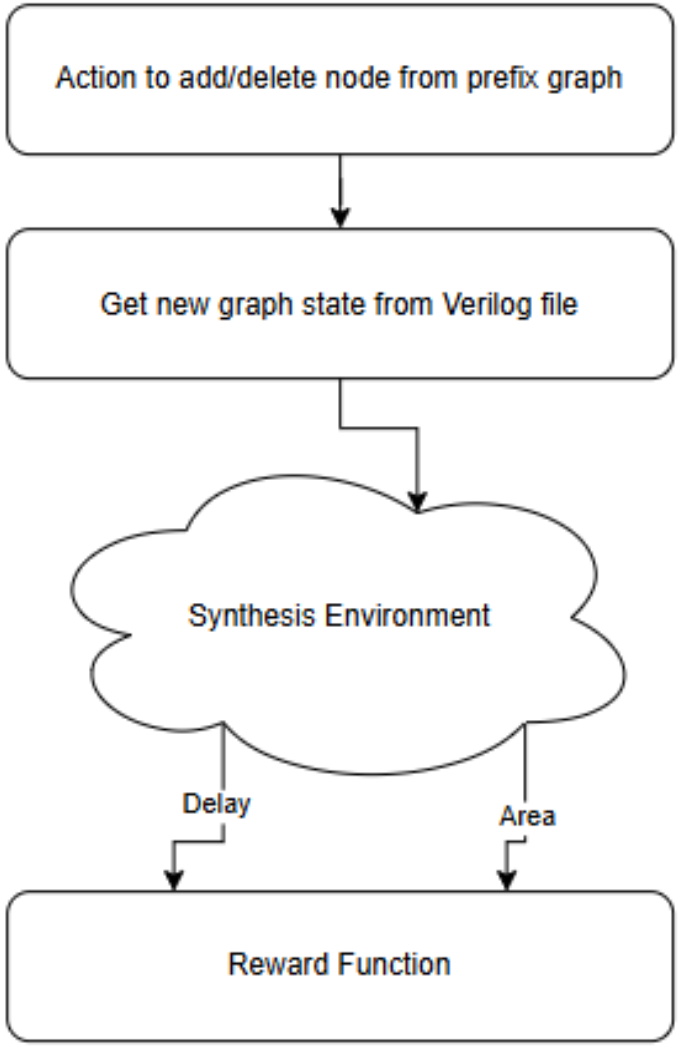
Epsilon Greedy

- The epsilon greedy approach is used to determine if the network will randomly add or delete a node from the current prefix graph
- This is done by take current progress of the episodes that are currently completed and applying an exponential increase to the epsilon value.
- Over the course of the training, the rate at with the nodes are added or deleted will decrease.

$$\epsilon(e) = 0.4^{\left(1+7\frac{i(e)}{1023}\right)}$$

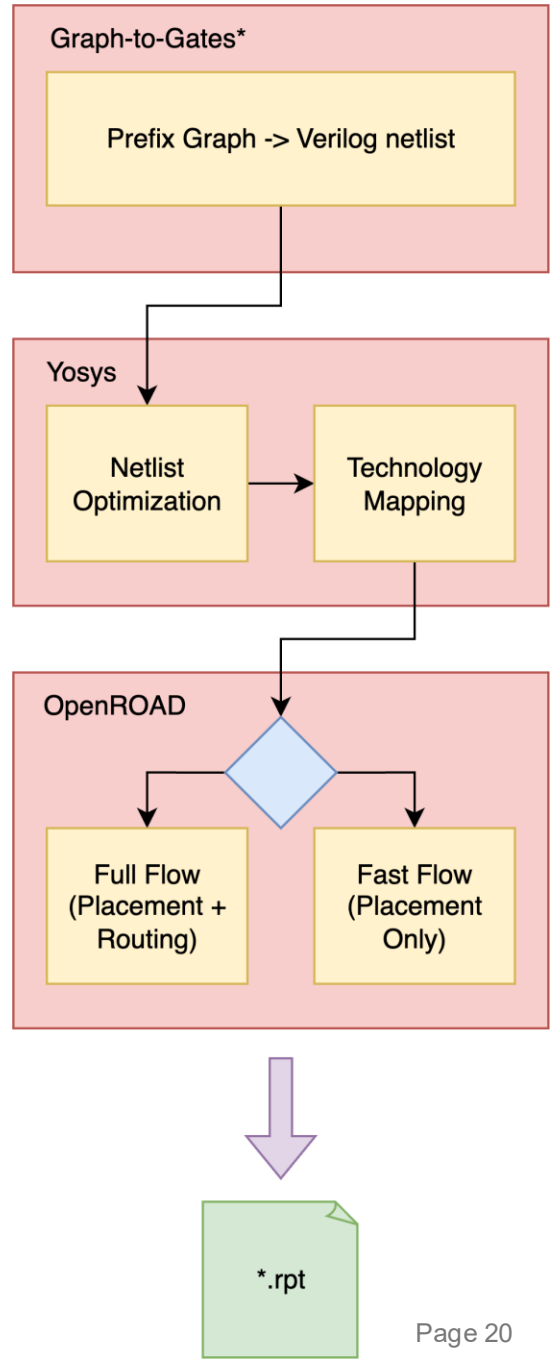
Prefix Graph Action & Reward Function Calculation

- Once an action is selected, a node will be added or deleted from the current graph state
- To evaluate the reward within our environment, the prefix graph will be converted to a Verilog netlist
- From there, the netlist is synthesized to generate the delay and area metrics of the new parallel prefix adder
- Finally, the reward is calculated based on the delay and area reported from the synthesis tool



Evaluation Flow

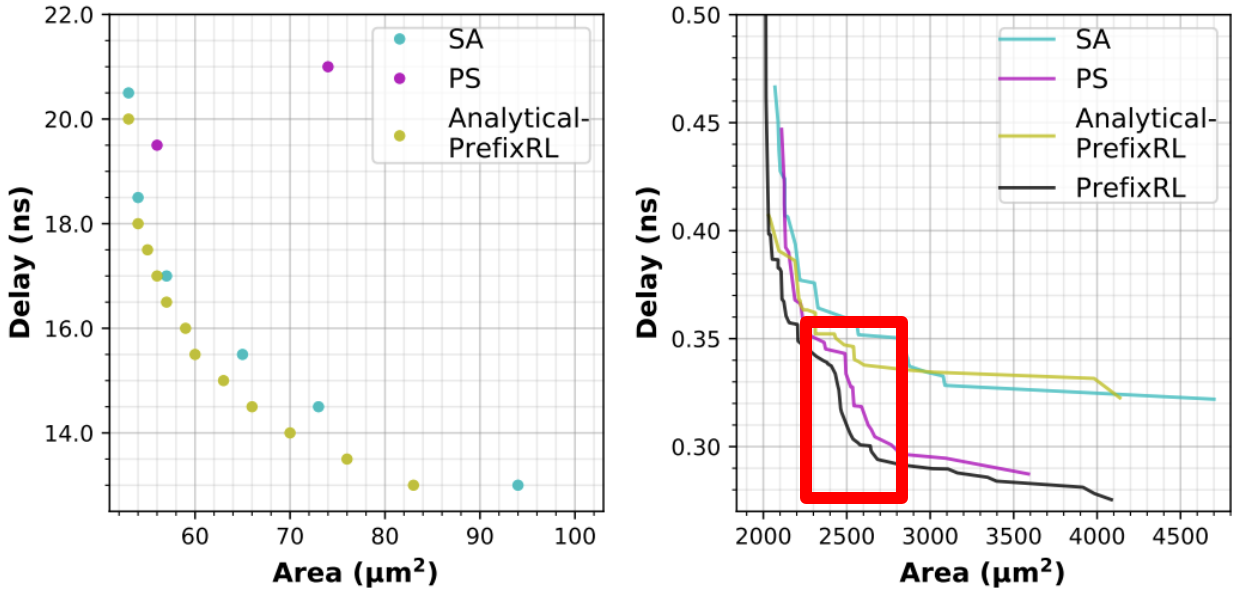
- Our evaluation flow uses a suite of open-source EDA tools
- **Graph-to-netlist conversion** – generates RTL netlist based on graph produced by the RL agent
- **Yosys** – logic synthesis to produce get technology-mapped netlist (Using open-source Nangate45 standard cell library)
- **OpenROAD** – open-source place-and-route flow to evaluate circuit delay & area (Adapted from ArithTreeRL repository)
    - Two separate flows: full flow (full placement and routing optimizations, ~10 seconds for 32b adder) and fast flow (only placement, no routing, ~1 second for 32b adder)



*Adapted from ArithTreeRL repository

Analytic Model of Area and Delay

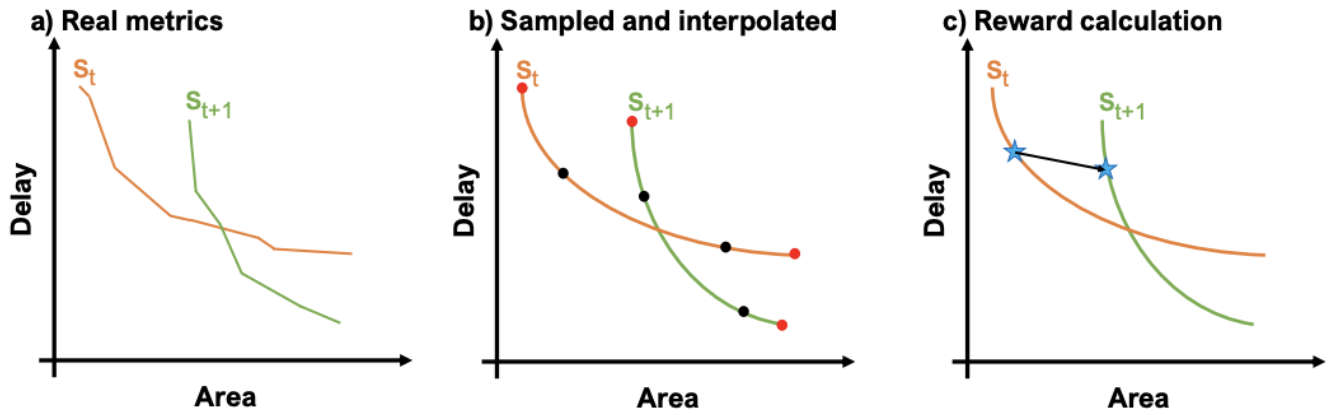
- Performing circuit synthesis each environment step can be quite time-consuming, especially without parallelization
- Training can be sped up significantly by using analytic metrics for area and delay, at the cost of accuracy
- $area = \text{prefix}\ node\ count$
- $delay = \text{delay}_{\text{internal}} + \text{delay}_{\text{driving}} + \text{delay}_{\text{sum}}$
  - $\text{delay}_{\text{internal}} = \text{stage count} + (\text{D}_\text{I} * \text{delay}_{\text{node}})$
  - $\text{delay}_{\text{driving}} = \text{D}_\text{O} * (\text{critical path fanout})$
  - $\text{delay}_{\text{sum}} = 1$ (delay of the final $p_i \oplus c_i$ stage)
  - $\text{D}_\text{O} = 0.5, \text{D}_\text{I} = 1.0$[1]



[1]T. Moto and M. Kaneko, "Prefix Sequence: Optimization of Parallel Prefix Adders using Simulated Annealing," *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*

Scalarized Multi-Objective Reward

- The Q-network learns the Q-values for our two objectives, area and delay, separately
- To select the optimal action (add or delete a node), these separate Q values must be scalarized according to a weight scalar **w**
- $\mathrm{r_t} = w(\mathrm{delay_t} - \mathrm{delay_{t+1}}) + (1 - w)(\mathrm{area_t} - \mathrm{area_{t+1}})$
- **w** is a value in [0,1] that allows for different tradeoffs in area and delay
  - $\mathrm{w} = 1$ optimizes only for delay and $\mathrm{w} = 0$ optimizes only for area
- The Q-network's goal is to find a policy to maximize $\mathrm{r_t}$ for a given **w**

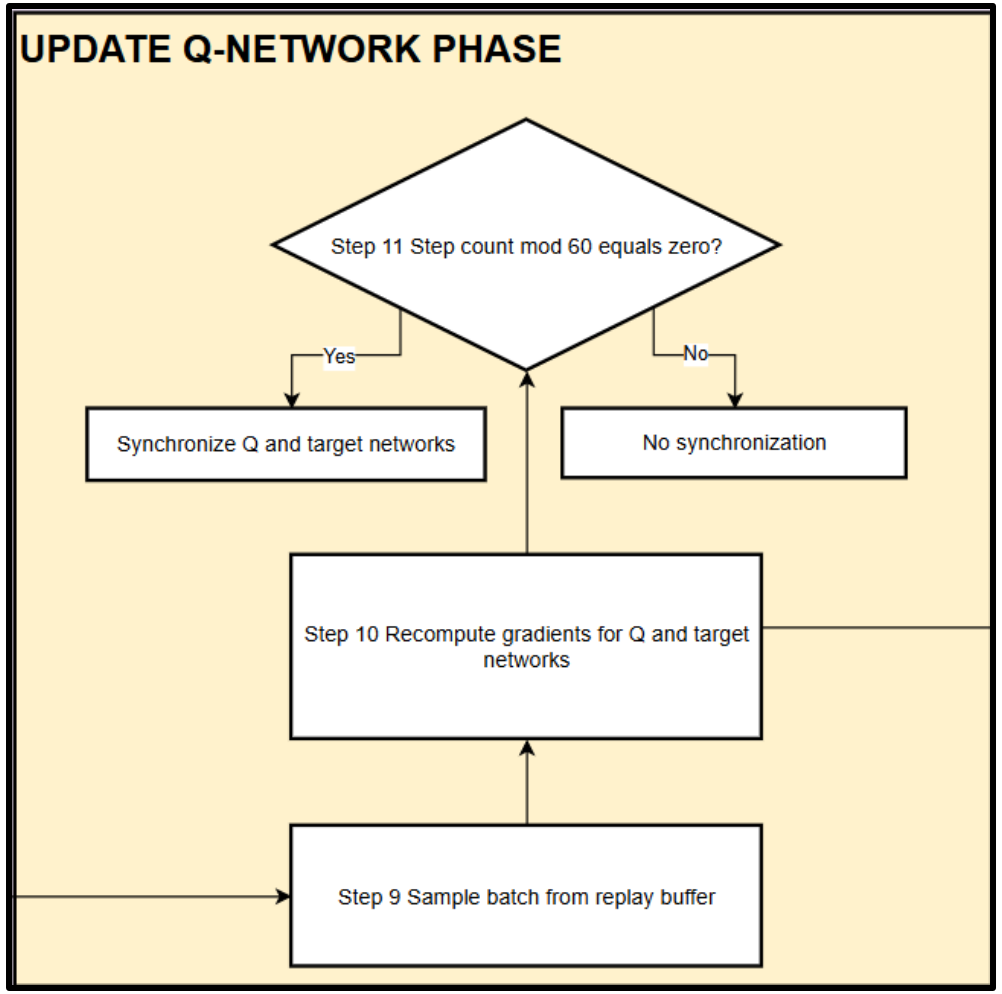


Q-Network Update

- To update the parameters of the online network based on gather experience, a batch of (s_j, a_j, s_{j+1}, r_j) is sampled from the replay buffer
- The target network is evaluated to compute the bellman target y_j for the sampled experience
- The online network's parameters are updated according to

$$\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi(s_j,a_j)}{d\phi}\left(Q_\phi(s_j,a_j) - y_j\right)$$

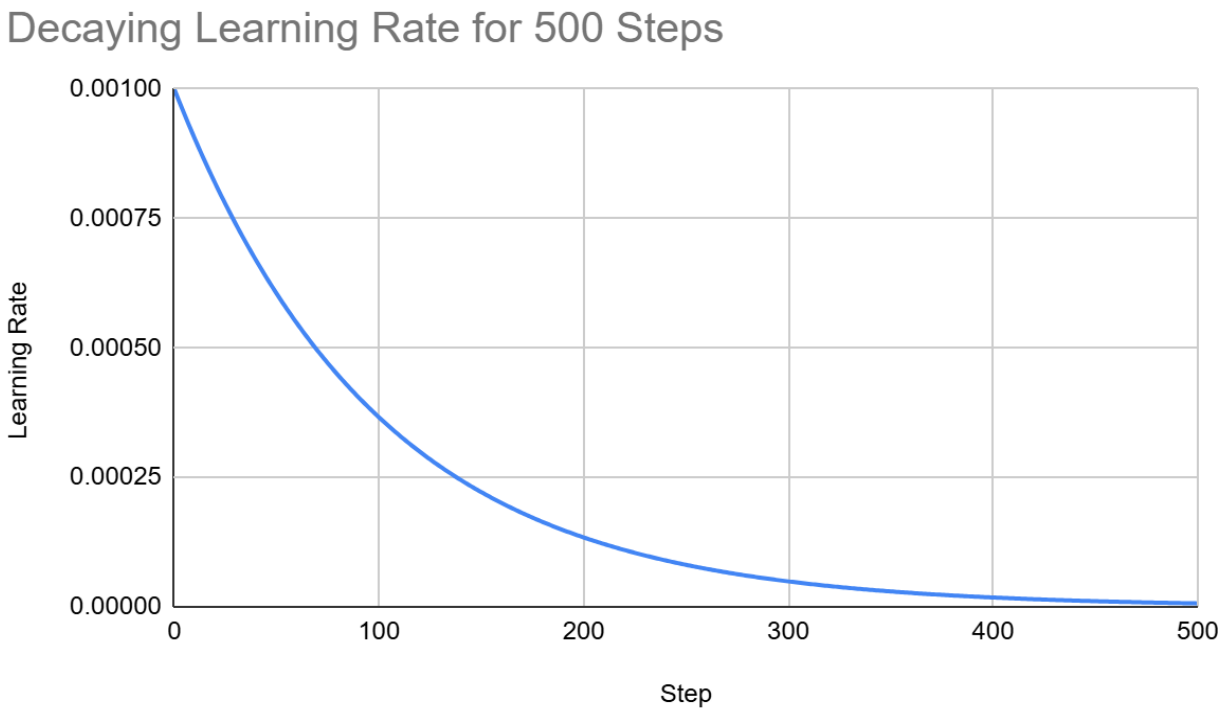
- Every 60 steps of the training loop of a given episode, the target network and online network are synchronized to ensure the target network has the updated weights from the online network



Decaying Learning Rate

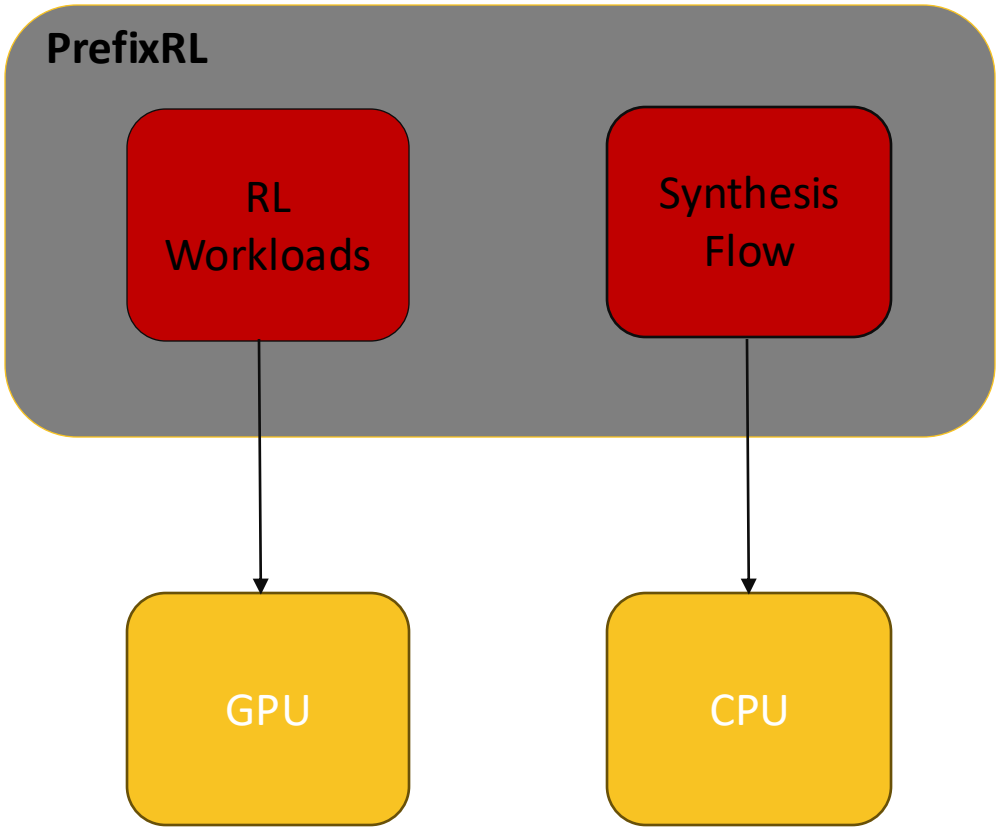
$$\mathrm{lr}(t) = \mathrm{lr}_0 \gamma^t$$

- The learning rate of the Q-network determines how much the parameters of a model are updated each environment step
- In general, the parameters should be updated in smaller increments as the Q-network converges to an optimal policy
- A decaying learning rate will slow down the rate at which the Q-network's parameters are updated with each episode to avoid any drastic design changes near the end of training
- This is done by using a scheduler to decrease the learning rate over each episode and step



Workload Parallelization

- To ensure timely results, we parallelized the RL computations using AMD & NVIDIA GPUs on personal machines and clusters (e.g. ARCC, VLSI)
- Additionally, there were parallel processes that called Yosys and OpenROAD

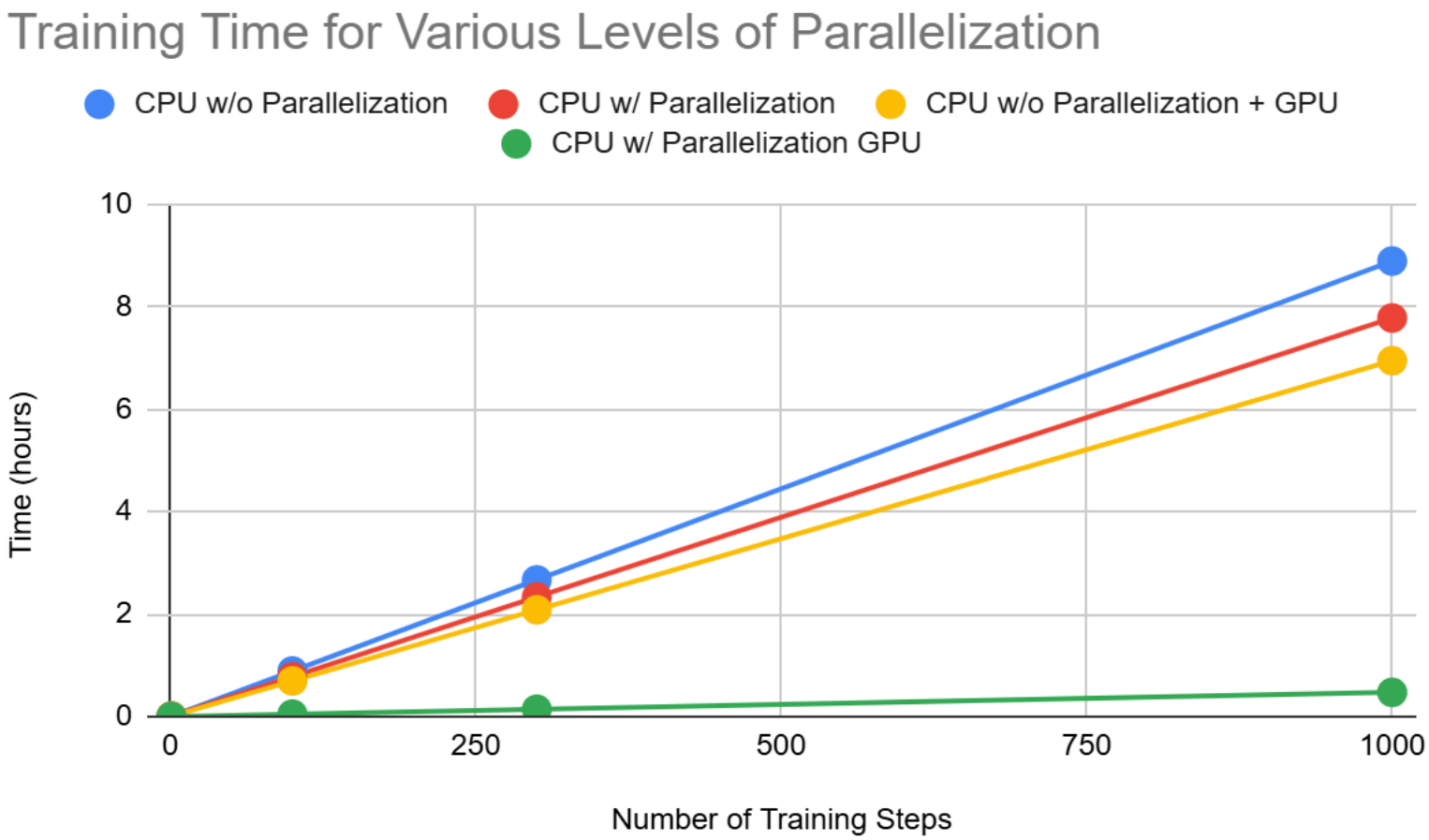


Note on GPU/CPU Utilization

```
Every 0.1s: rocm-smi                                                                                                draco-eng-A01: Sat Nov 15 13:53:09 2025



========================================= ROCm System Management Interface =========================================
================================================= Concise Info =====================================================
Device  Node  IDs              Temp    Power    Partitions          SCLK    MCLK     Fan    Perf  PwrCap  VRAM%  GPU%
^[3m              (DID,     GUID)  (Edge)  (Avg)    (Mem, Compute, ID)                                                ^[0m
====================================================================================================================
0       1     0x744c,   4672   54.0°C  51.0W    N/A, N/A, 0         42Mhz   772Mhz   14.9%  auto  339.0W  71%    0%
1       2     0x164e,   18139  43.0°C  22.131W  N/A, N/A, 0         N/A     2400Mhz  0%     auto  N/A     3%     0%
====================================================================================================================
============================================= End of ROCm SMI Log ==================================================
```

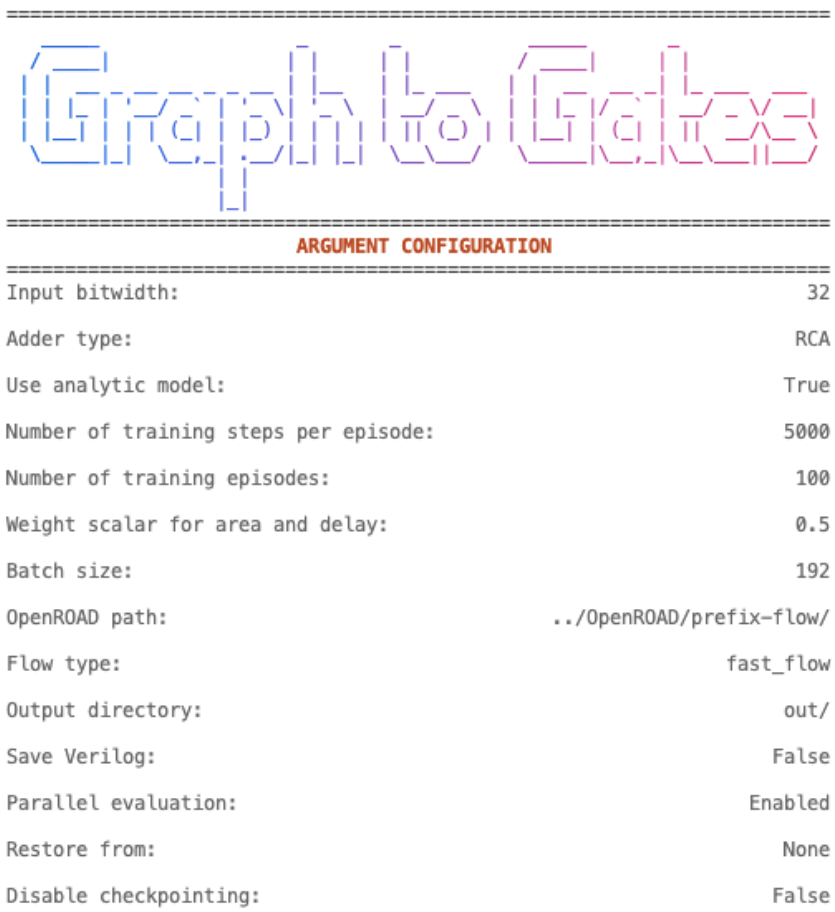
```
  0[          0.0%]   4[          0.0%]   8[          0.0%]  12[          0.0%]   16[          0.0%]  20[          0.0%]  24[          0.0%]  28[          0.0%]
  1[||||||||66.7%]   5[|||||     37.5%]   9[          0.0%]  13[          0.0%]   17[|||       20.0%]  21[          0.0%]  25[          0.0%]  29[          0.0%]
  2[||        12.5%]   6[          0.0%]  10[          0.0%]  14[          0.0%]   18[          0.0%]  22[          0.0%]  26[          0.0%]  30[          0.0%]
  3[          0.0%]   7[|         6.2%]  11[          0.0%]  15[          0.0%]   19[          0.0%]  23[|         6.7%]  27[          0.0%]  31[          0.0%]
Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||33.1G/93.4G]   Tasks: 352, 3297 thr; 2 running
Swp[|                                                                   1.25M/2.00G]   Load average: 11.36 8.11 5.36
                                                                                       Uptime: 24 days, 02:39:59

    PID USER      PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
2810535 root       20   0 31.0G 4209M 2329M R 83.0  4.4  6:43.52 python3 graph_to_gates.py -n 32 --adder_type 0 -b 32 --w_scalar 0.7 --output_dir adder_32b_batch16_weight0p7_st
2863089 mcastigli  20   0 26456  9984  3840 R 38.3  0.0  0:32.43 htop -d 1
 485070 mcastigli  20   0 28.2G  761M  135M S  6.4  0.8  2:37.56 /snap/firefox/7084/usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0:44657 -prefMapHandle 1:276
 622705 mcastigli  20   0 2755M  317M  108M S  6.4  0.3  0:21.64 /snap/firefox/7084/usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0:44658 -prefMapHandle 1:276
1329803 mcastigli  20   0 8458M  474M  185M S  6.4  0.5 21:10.91 /usr/bin/gnome-shell
2740348 mcastigli  20   0 28.2G  761M  135M S  6.4  0.8  0:01.36 /snap/firefox/7084/usr/lib/firefox/firefox -contentproc -isForBrowser -prefsHandle 0:44657 -prefMapHandle 1:276
      1 root       20   0  164M 12544  7680 S  0.0  0.0  2:11.07 /sbin/init splash
```

Parallelization Optimization Results

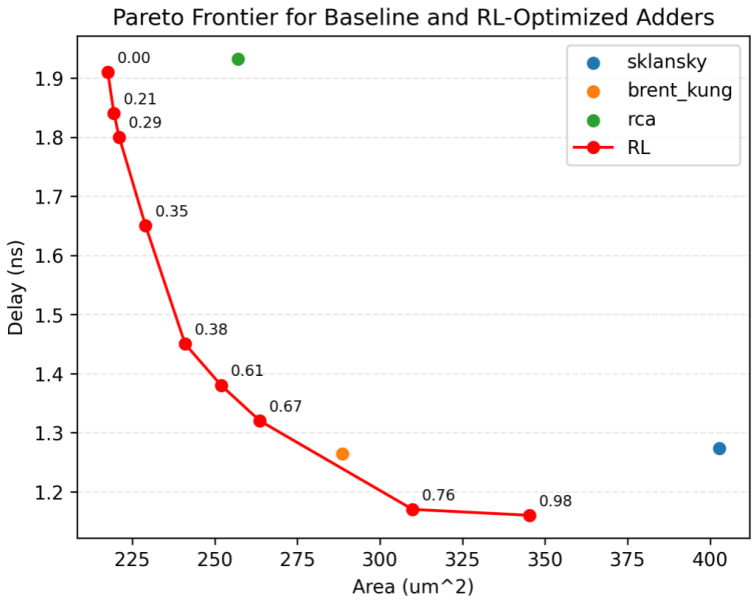


Graph to Gates User Interface

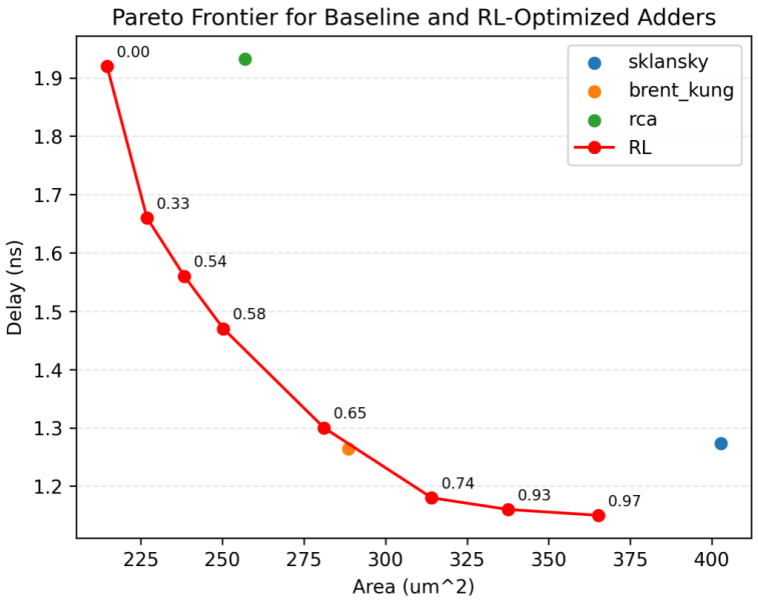
- Input arguments printed to screen:
  - Input adder bit-width/type
  - Use of analytic model or physical-aware model
  - Weight scalar (to incentivize area or delay)
  - Training hyperparameters



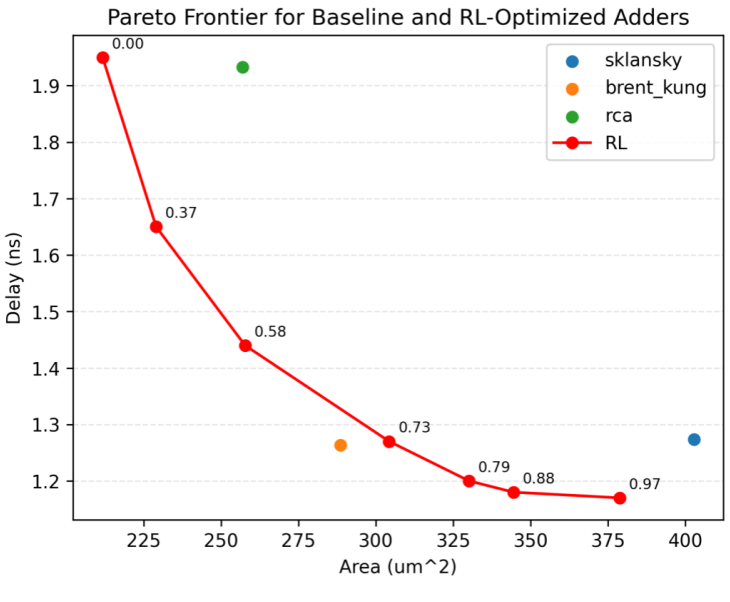
32b Adder Training Results for Different Weight Scalars



w = 0.3



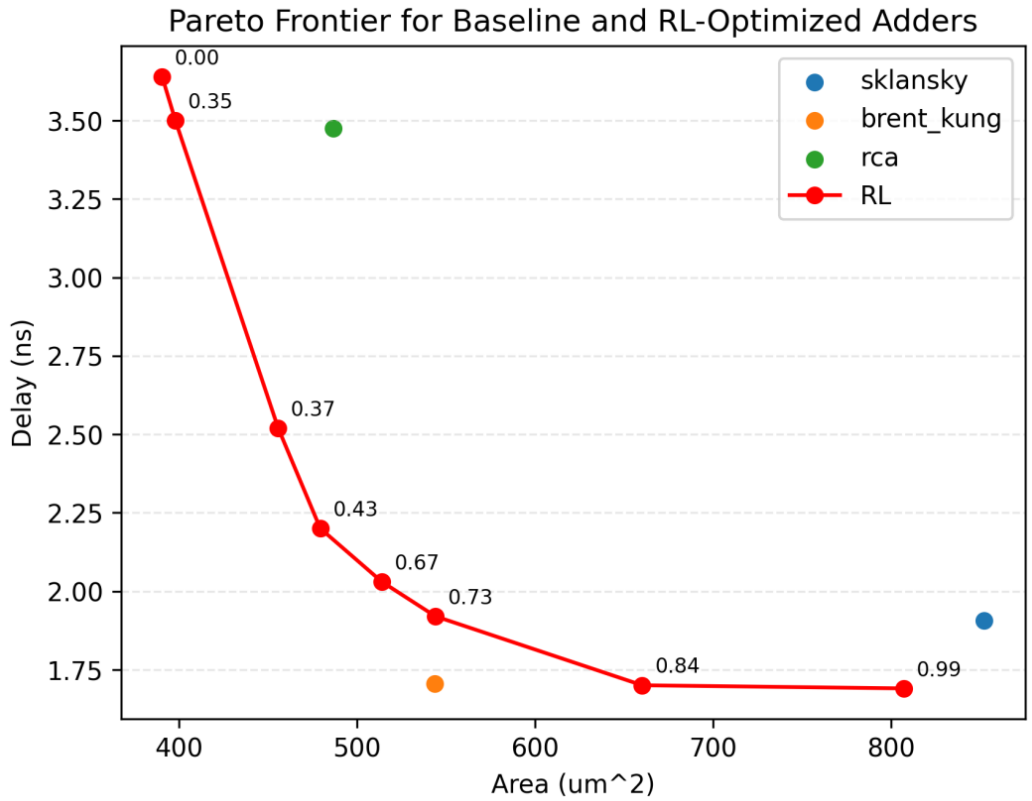
w = 0.5



w = 0.7

32b Adder Results (Trained with w=0.3)

64b Adder Training Results



$$w = 0.7$$

64b Adder Results (Trained with w=0.7)

32b Adder Results: Analytic Model

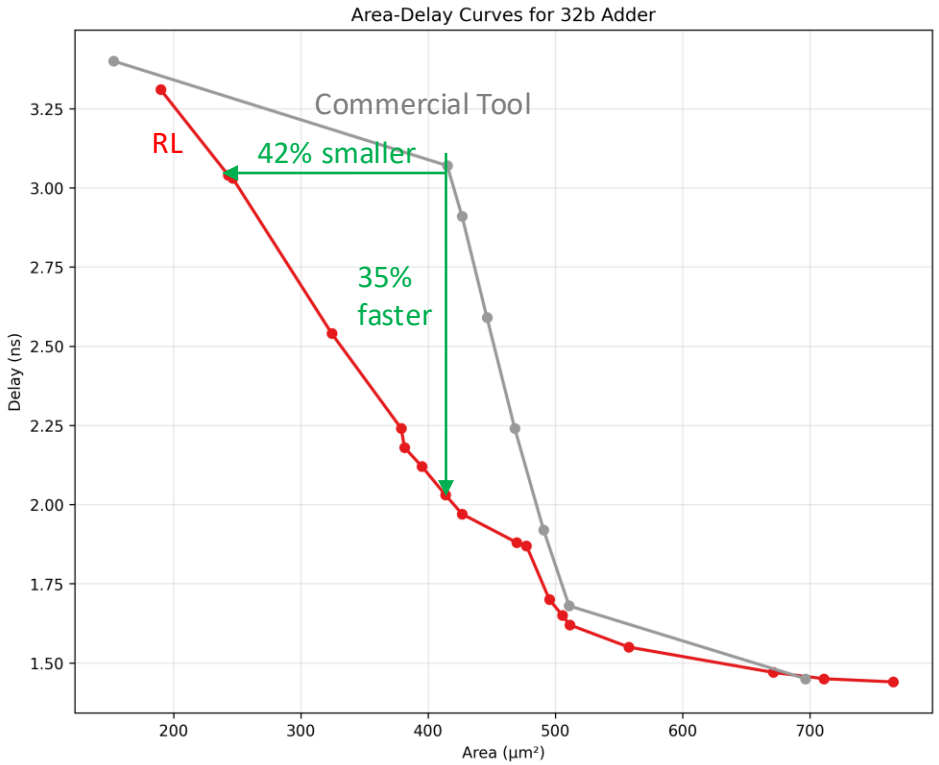
- Since the analytic model for delay computes only the critical path, it does not have context for the fanout of most intermediate nodes
- This leads to non-ideal graph modifications

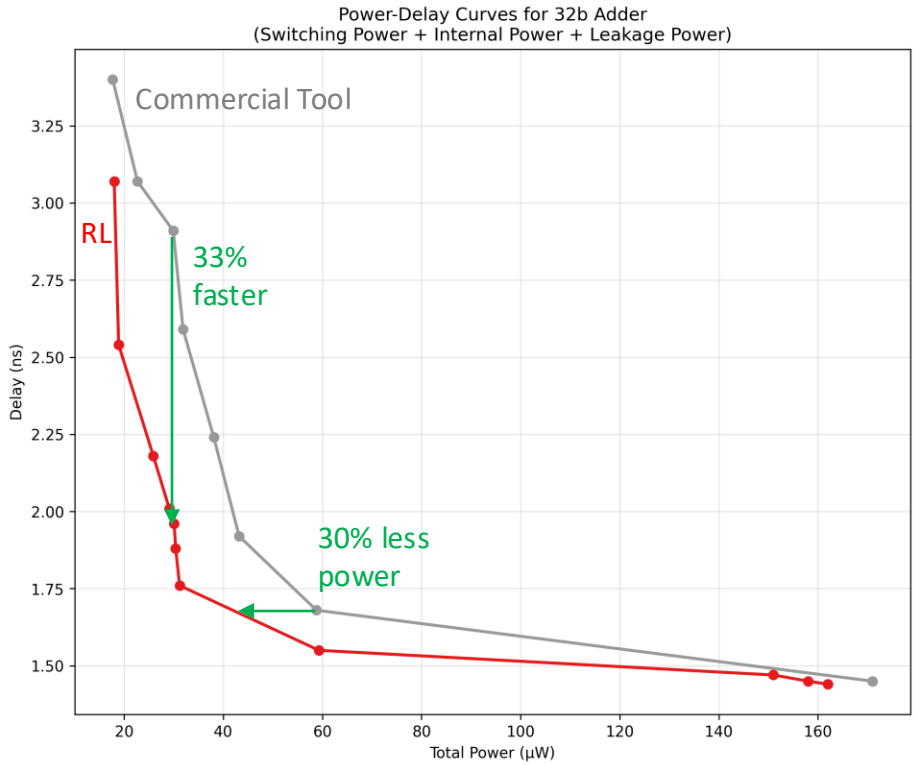
Evaluation Against Commercial Tools

- Our RL-optimized adders were evaluated using open-source EDA tools and standard cell libraries
- To see how our results generalize to commercial EDA tools and cell libraries, we evaluate results with Synopsys RTL Architect using a 32nm process technology
- In addition, we measure the power consumption using PrimePower
- For the following results, we compare our RL-generated adders against Synopsys DesignWare adders

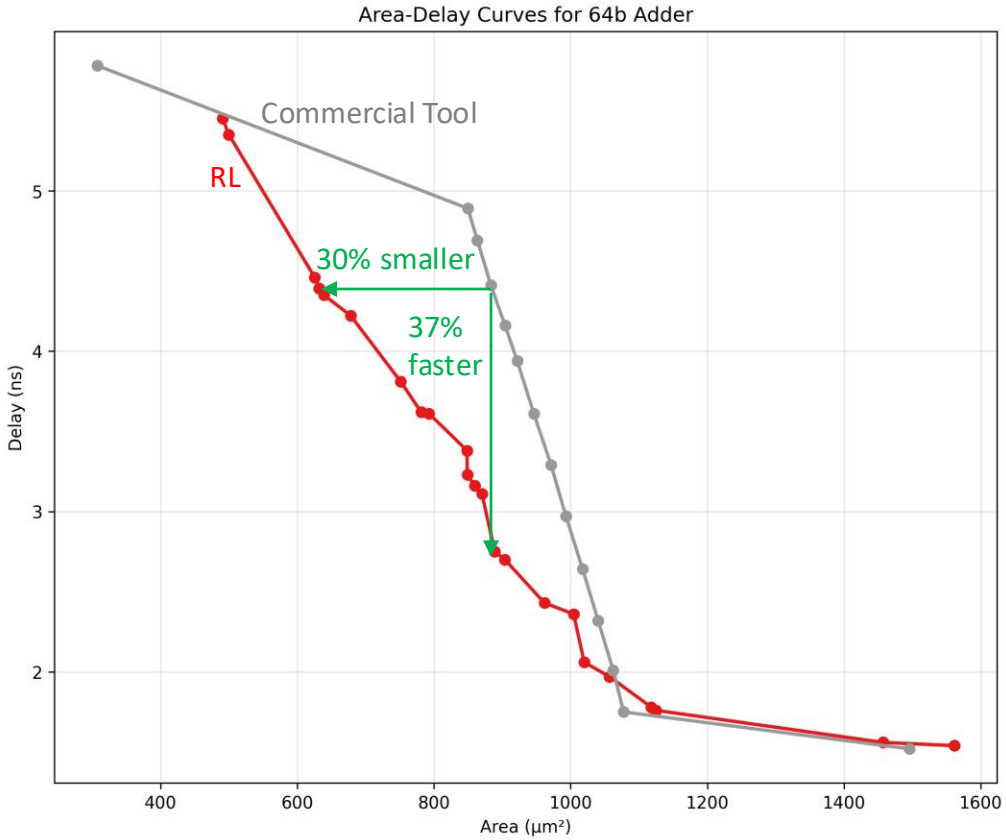
```
module adder_top #(parameter N = 32)(
    input [N-1:0] a,
    input [N-1:0] b,
    output [N-1:0] s,
    output cout
);

assign {cout, s} = a + b;

endmodule
```

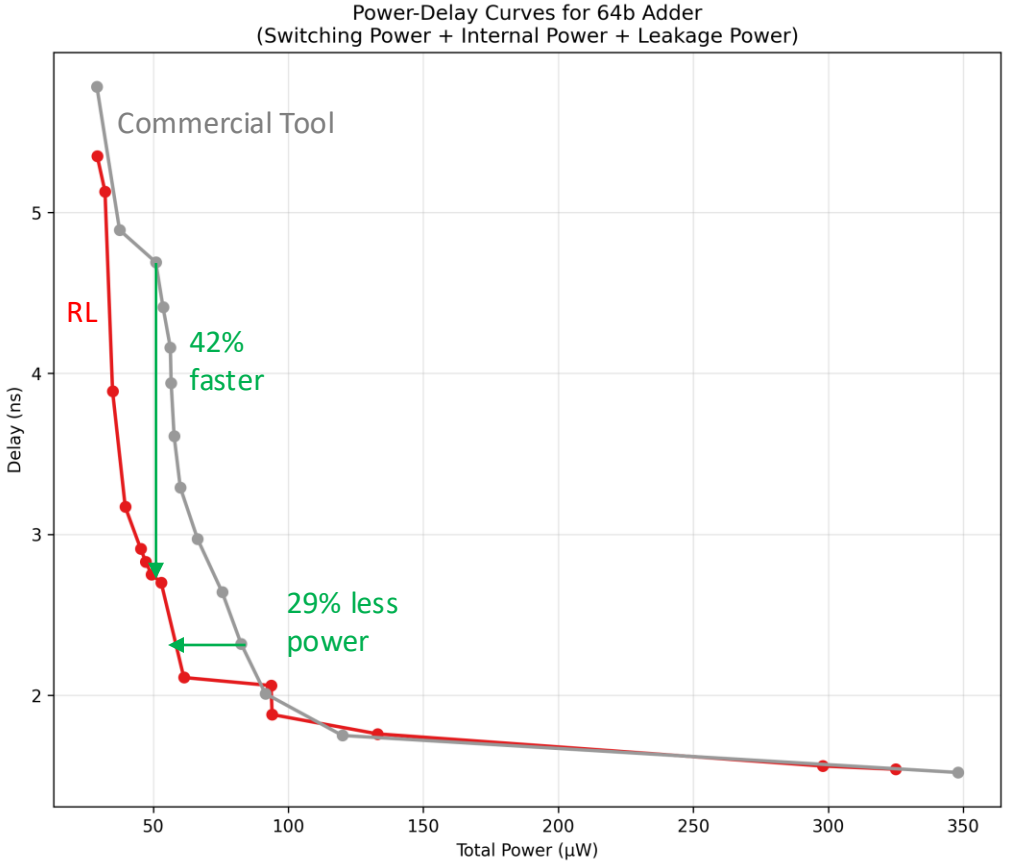
32b Evaluation Against Commercial Tools





64b Evaluation Against Commercial Tools





Project Limitations

Compute, Compute, Compute

- Using the exact parameters from PrefixRL for 32b adder training (5000 steps/episode, 100 episodes, batch size of 192), full training would take 150 CPU days sequentially
- To obtain our results, we used a much-reduced parameter set (500 steps/episode, 10 episodes, batch size of 16/32)

Power Optimization

- The training process does **not** optimize for power – minimizing delay and/or area *may* increase power
- Without multi-objective optimization including power, the RL agent may produce designs that have balanced delay and area but not power-efficiency

Signal Arrival Times

- We assume that all input signals arrive to the graph with a uniform latency
- In reality, input bits may arrive with non-uniform latencies, requiring different optimizations

Thank You

Questions?