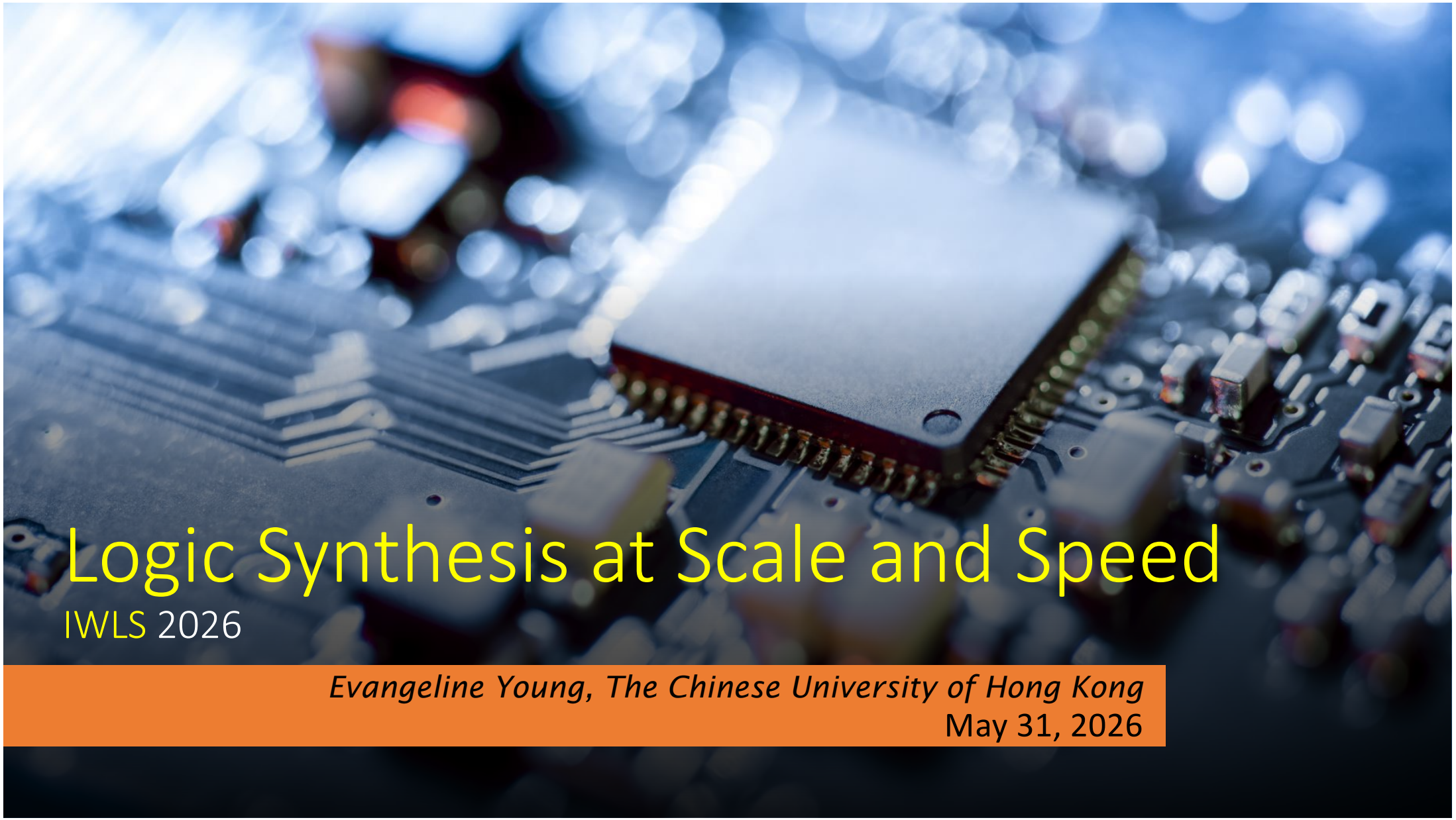


# Logic Synthesis at Scale and Speed

IWLS 2026

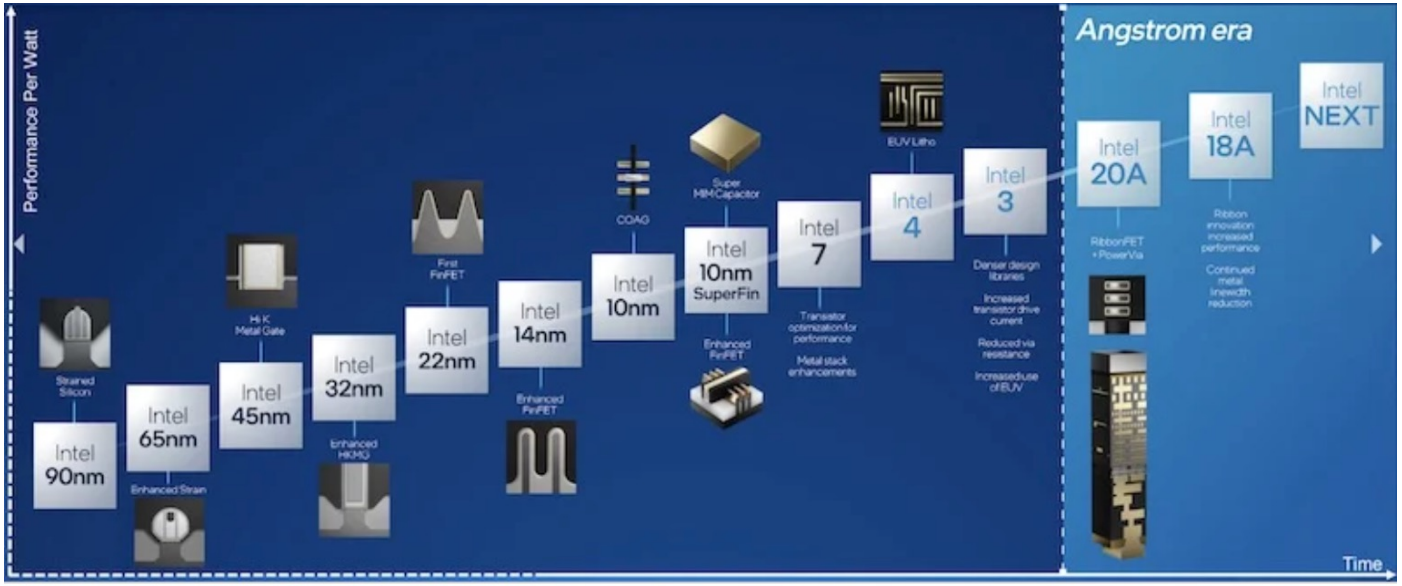
*Evangeline Young, The Chinese University of Hong Kong*

May 31, 2026

# Agenda

- GPU Acceleration on:
  - Logic Optimization
  - Mapping
  - Circuit Equivalence Check
- Accelerated Logic Synthesis Flow
  - Demo

# Motivations



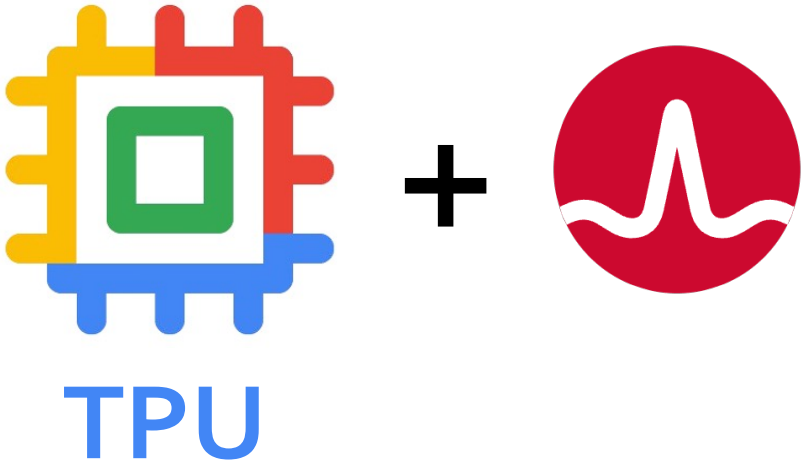

Trillion-transistor processor by 2030

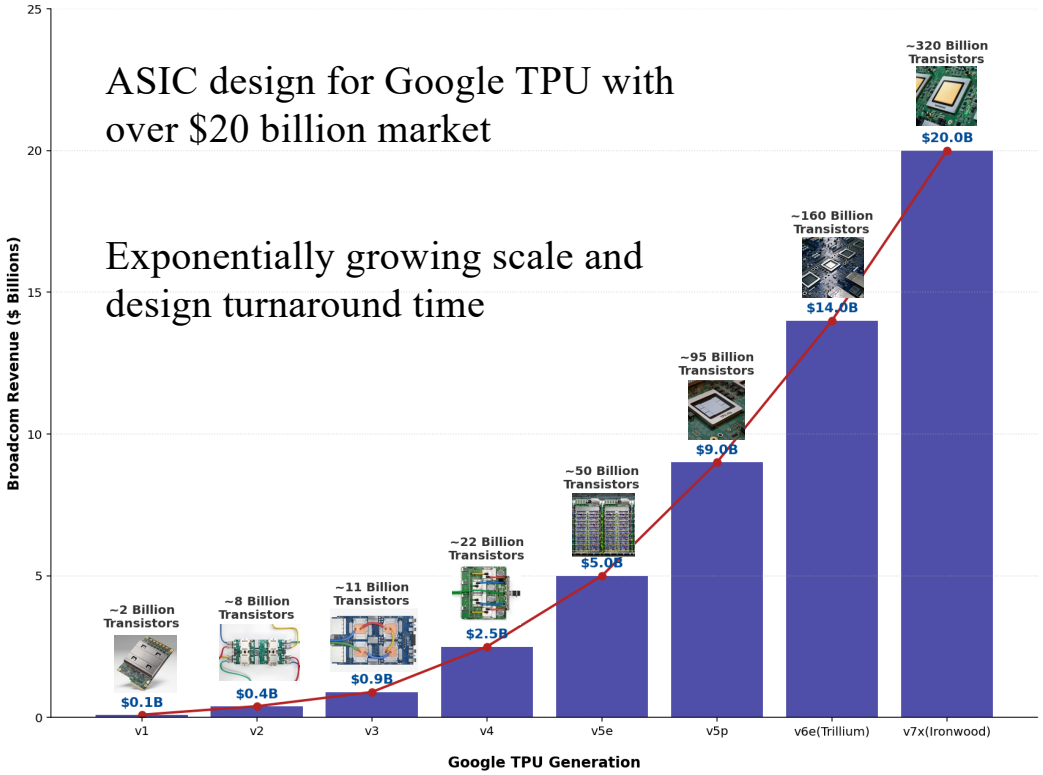
Large AI models today with 200 billions parameters

*copyright@intel*

- The lengthy chip design cycle, coupled with the slow performance of traditional EDA tools, hampers the pace of tape-out
- Emerging technologies such as AI-empowerment and **GPGPU** continue to evolve, presenting significant opportunities to revolutionize traditional EDA tools
- A modern EDA tool should meet the criteria of being **"fast, accurate, and effective"**

# AI Accelerators

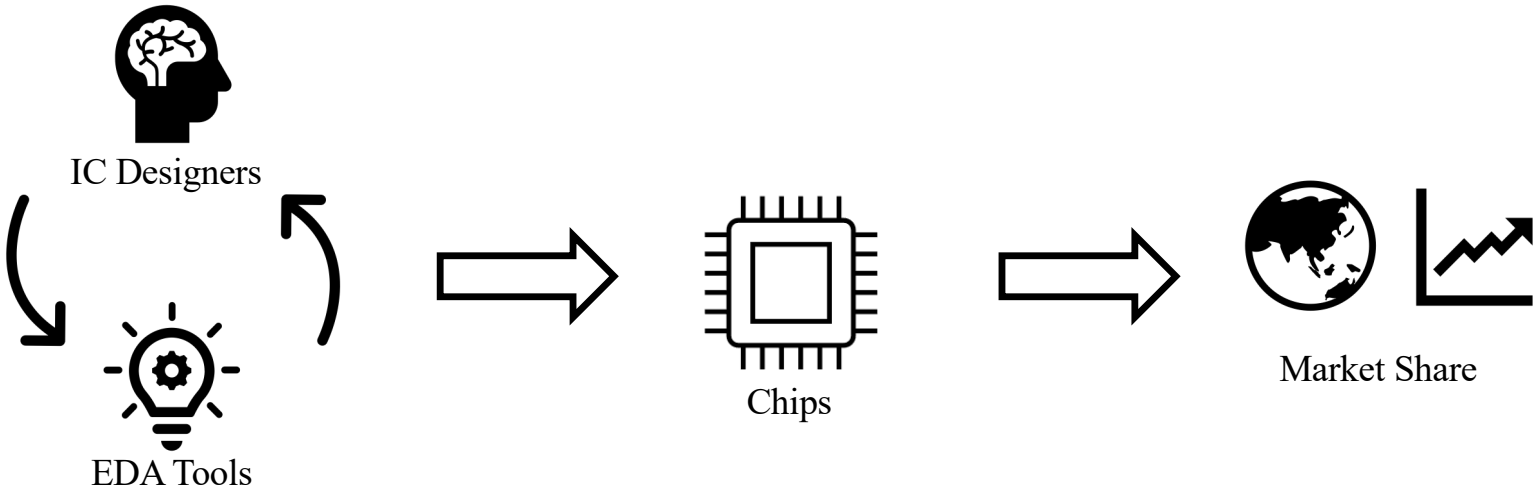



- The ASIC design for AI accelerators has explosively growing market.
- RTL to tape-out takes more than 6 months. VLSI synthesis takes massive portion of the design cycle.

# Our Solution: Heterogeneous CPU-GPU Accelerated EDA Tools

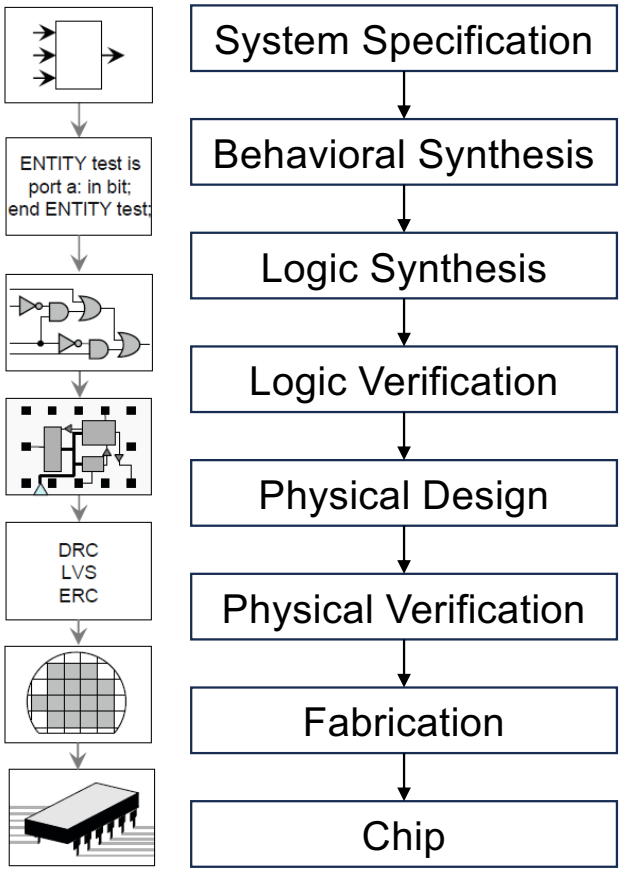
- With **30-50x or more** acceleration compared to conventional EDA tools
- Unlock the full potential of CPU-GPU acceleration
- **Agile Chip Development:**
  - **Faster design closure -> better PPA -> larger market share**




**More complex designs**
**More powerful GPUs**
**More open IC ecosystem**

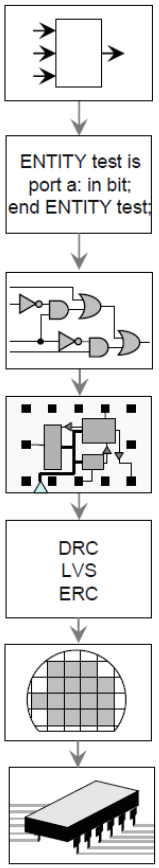
**An opportune moment to develop GPU-accelerated logic synthesis**

# Flow of VLSI Design Automation




Adapted from: Kahng et al., "VLSI Physical Design: From Graph Partitioning to Timing Closure"

# Flow of VLSI Design Automation




System Specification

Behavioral Synthesis

Logic Synthesis

Logic Verification

Physical Design

Physical Verification

Fabrication

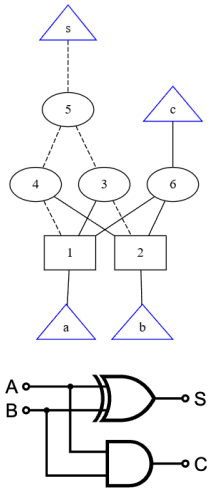
Chip

RT-level Synthesis

Logic Optimization

Technology Mapping

```
module add(input a,
           input b,
           output c,
           output s);
   assign {c, s} = a + b;
endmodule
```

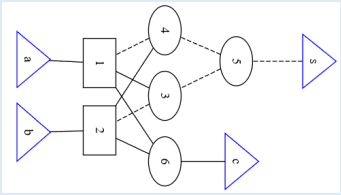



Adapted from: Kahng et al., “VLSI Physical Design: From Graph Partitioning to Timing Closure”

# Logic Synthesis

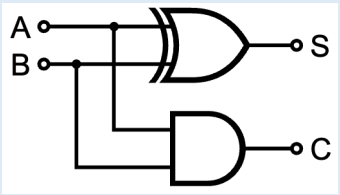
```
module add(input a,
           input b,
           output c,
           output s);
    assign {c, s} = a + b;
endmodule
```

**HDL/RTL Code**



**Tech-independent Logic Network**

- AIG, XMG, …




**Mapped Netlist**

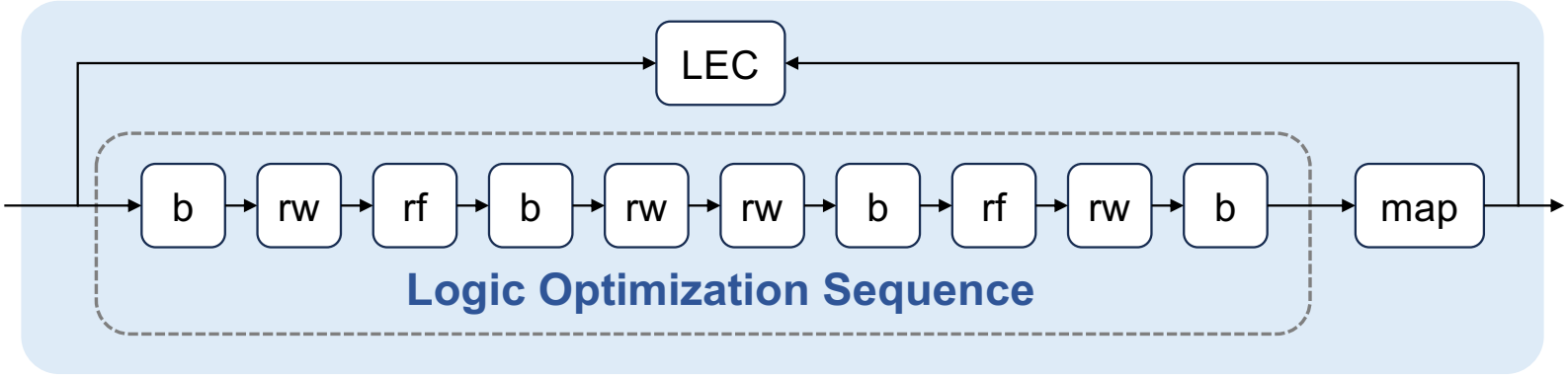
- LUTs
- Standard cells

RT-level Synthesis → Logic Optimization → Technology Mapping

**Gate-level Synthesis**

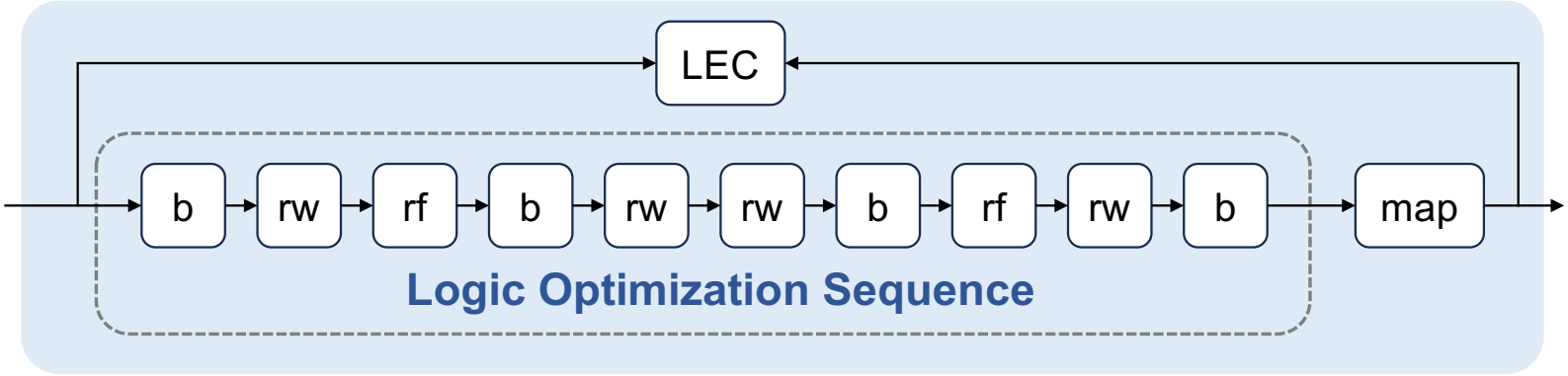
# Gate-level Synthesis and Verification

- Logic optimization
  - Different algorithms: balance (b), rewrite (rw), refactor (rf), resubstitution (rs), …
- Technology mapping after logic optimization
- Functional / Logic verification
  - Equivalence checking (EC) after synthesis



# Accelerating Synthesis and Verification

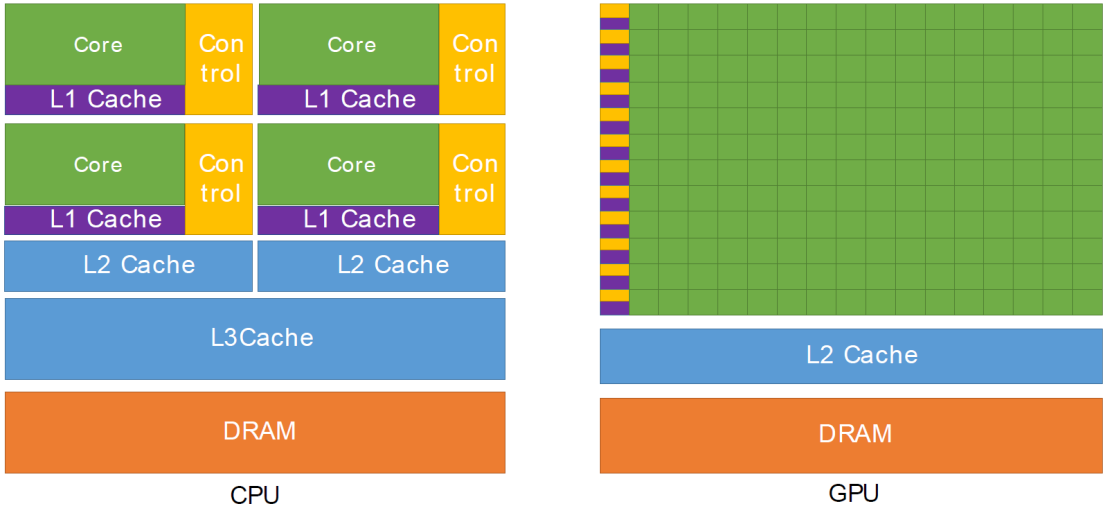
- Faster means better quality
  - More high-effort optimization
  - Better exploration of design space
- Handle increasing design scale and complexity
  - +30M gate AIG,  needs ~**2h** (synthesis) + >**4 months** (combinational EC)




**Speedup 10x → Repeat 10x with same runtime budget**

# Graphics Processing Unit (GPGPU)

- Much higher degree of parallelism vs. multi-core CPU
- Applications
  - Scientific computing, physical design in EDA
  - **Less explored in logic synthesis and verification**




Source: NVIDIA

# Speedup

| Logic Optimization | Technology Mapping | Logic Verification |
| --- | --- | --- |
| ▪ [DAC'23] GPU refactor & balance<br>▪ [TODAES UR] Parallel e-graph-based logic opt | ▪ [ASP-DAC'24] GPU LUT mapping<br>▪ [TCAD'24] GPU map-based logic opt | ▪ [DAC'25] Simulation-based CEC<br>▪ [TCAD'26] Multi-GPU simulation-based CEC |
| × 26.4 | × 34.6 | × 12 |

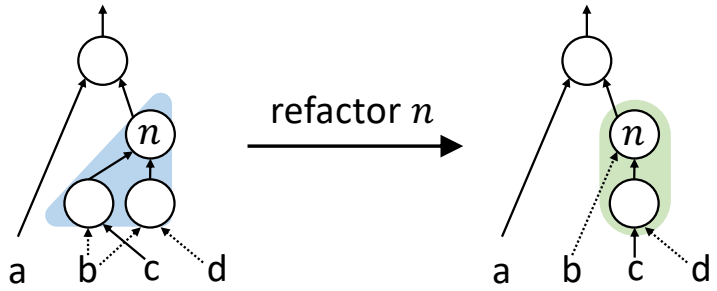
UR = Under Review

# Agenda

- GPU Acceleration on:
  - Logic Optimization
  - Mapping
  - Circuit Equivalence Check
- Logic Synthesis Flow
  - Demo

# AIG Refactoring

- Resynthesize logic cones using factored forms

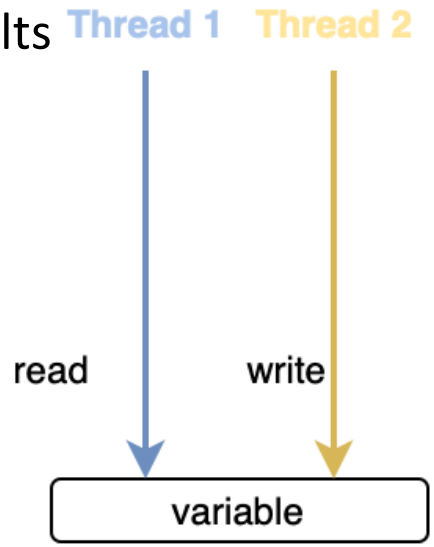



Repeat the transformation
for each node

A. Mishchenko, S. Chatterjee, and R. Brayton,
"DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", DAC'06.

# Challenges for Parallelization

- Need to prevent **data races**
  - Data race leads to non-determinism or incorrect results

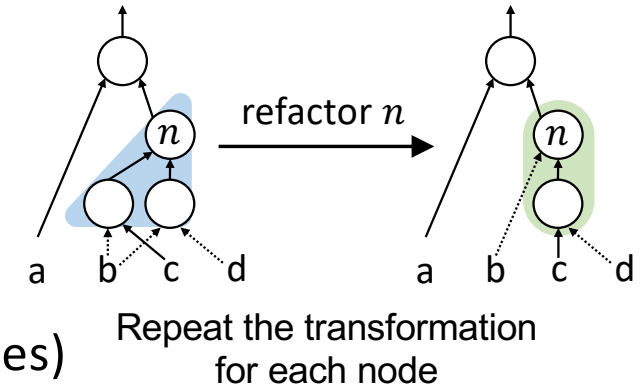


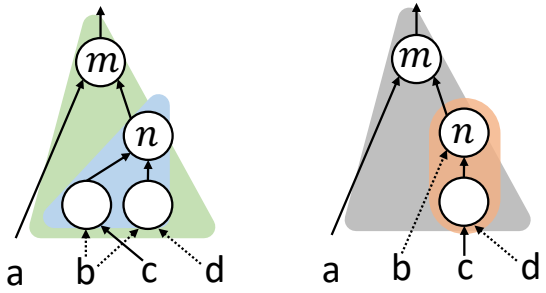

- Parallel logic synthesis is non-trivial
  - (✗) Direct parallelization → data races
  - (✓) New, inherently data-race-free algorithms

Source: https://stackoverflow.com/questions/11276259

# Challenges for Parallel Refactoring

- Concurrent resynthesis for each node?
  - **Data race can happen!**
- In the case of refactoring
  1. Cones to be deleted can overlap
  2. Cones to be inserted can overlap (resynthesized cones)



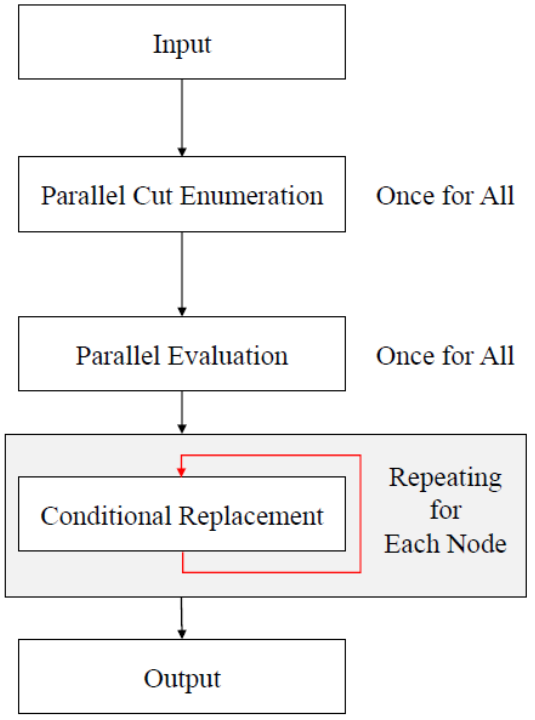
# Related Work - Parallel AIG Rewriting

- Idea: parallel resynthesis, sequential update
  - Resynthesis is read-only, thus no data race
- Refactoring is a variant of rewriting
- Use this idea for parallel refactoring?
  - Too slow!

**60% longer than parallel rewrite**

| Algorithm | GPU `rw` [9] | `rf` w/ seq. replace | `rf` (proposed) |
|---|---|---|---|
| Norm. seq. time | 1.0 | 1.6 | 0.6 |

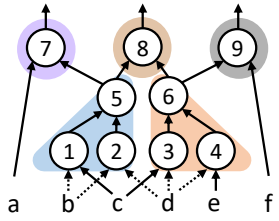
Normalized Sequential Part Runtime



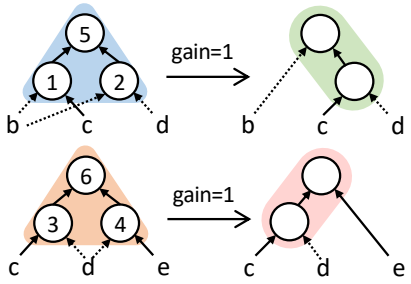

S. Lin, J. Liu, T. Liu, M. D. F. Wong, and E. F. Y. Young, "NovelRewrite: Node-Level parallel AIG rewriting", DAC'22.

# Parallel Refactoring - Overall Flow

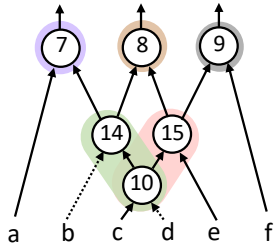
- Stage 1: identify and collect logic cones (subgraphs) in parallel that are
  - (1) disjoint from each other, and (2) cover all the logic in the AIG
- Stage 2: resynthesize all local functions in parallel
- Stage 3: replace the original cones by the new cones in parallel



Identification



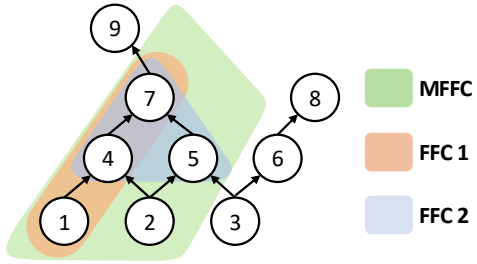
Resynthesis



Replacement

# Parallel Refactoring - Stage 1

- Maximum fanout-free cone (MFFC) of a node
  - Containing all the logic dedicated to driving the node
- Fanout-free cone (FFC) of a node
  - A logic cone that is a subset of the node's MFFC



(M)FFCs of node 7

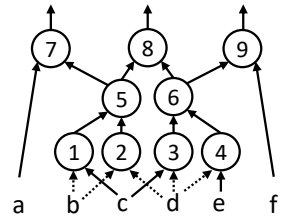
❖Property

❖MFFCs of different nodes are either subsets of each other, or disjoint from each other.

# Parallel Refactoring - Stage 1

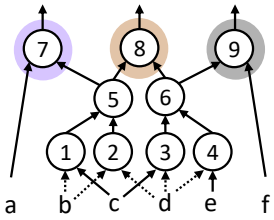
- These MFFCs can be identified in parallel.
- **Level-wise parallel** identification of MFFCs:
  - Maintain a frontier array, initialized by all POs
  - As long as there exists non-PI node in the frontier, repeat:
    - Identify MFFCs rooted at each node in the frontier, one thread per node
    - Gather all the inputs to the MFFCs to be the new frontier

frontier: 7 8 9



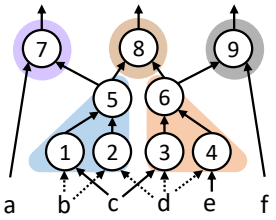

before level 0

frontier: a 5 6 f




before level 1

frontier: b c d e




before level 2

# Parallel Refactoring - Stage 1

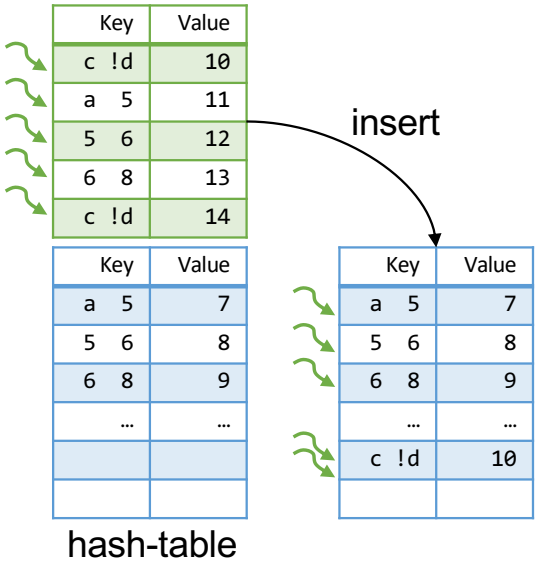
- A practical issue
  - Need to bound #inputs of a cone by $k$ to restrict time complexity
  - What if an MFFC has more than $k$ inputs?
- Solution
  - Stop identification of an MFFC when reaching $k$ inputs (i.e., obtaining a FFC)

❖Theorem

❖The FFCs identified using our level-wise parallel approach are all disjoint.

# Parallel Refactoring - Stages 2 & 3

- Stage 2
  - Resynthesize all local functions in parallel, using one thread per function
- Stage 3
  - When updating the circuit, insert new nodes into a **dedicated GPU-parallel hashtable**
  - Avoid duplicate nodes and enable logic sharing



# Parallel Refactoring - Results

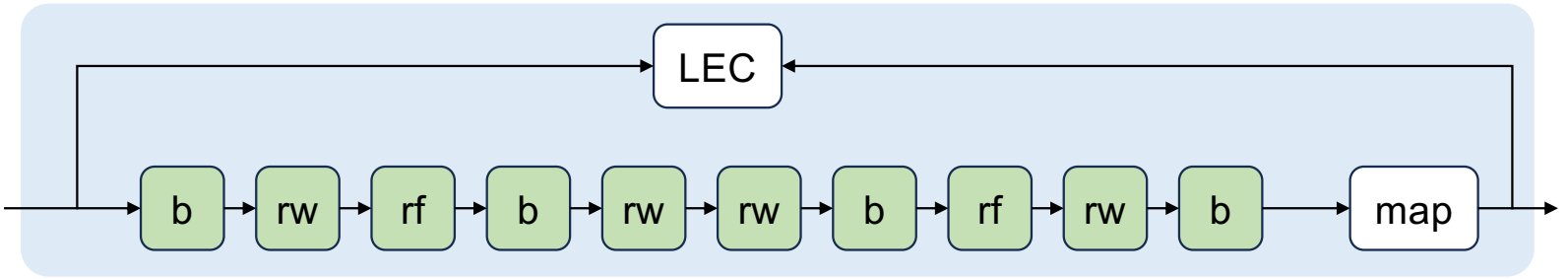
- Setup
  - RTX3090 GPU (mid-range consumer-level)
  - Benchmarks: large-scale AIGs (4~60M nodes)
  - Compare with ABC
- Results

| Algorithm / Sequence | Speed-up vs. ABC | Size vs. ABC | Level vs. ABC |
|---|---|---|---|
| balance | **6.0x** | **-0.1%** | **-0.0%** |
| refactor | **20.4x** | **-1.7%** | **-2.0%** |
| resyn2 | **26.4x** | +0.3% | **-1.8%** |

resyn2 = b, rw, rf, b, rw, rw, b, rf, rw, b

# Parallel Refactoring - Summary

- Novel parallel algorithms for AIG refactoring (rf) and balancing (b)
- Fully GPU-accelerated `resyn2`, a commonly used AIG optimization script




**GPU-based Logic Optimization**

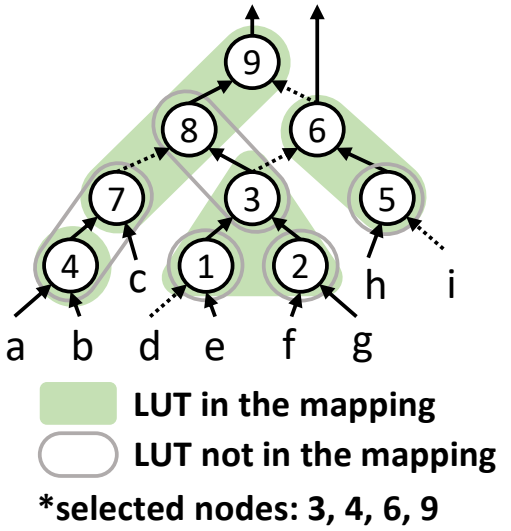
https://github.com/cuhk-eda/CULS

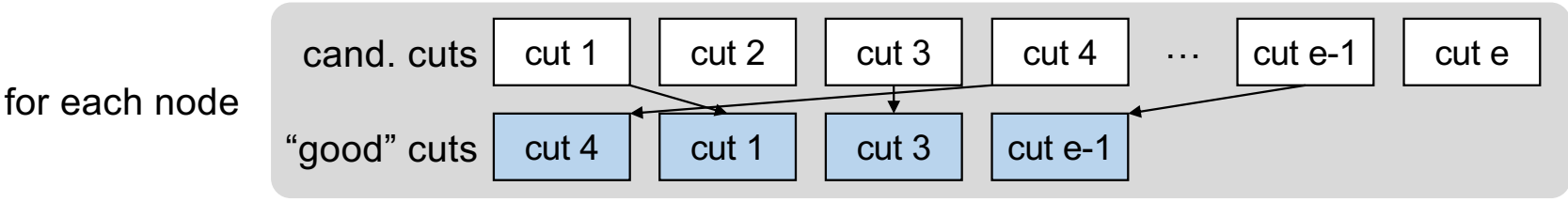
# Agenda

- GPU Acceleration on:
  - Logic Optimization
  - Mapping
  - Circuit Equivalence Check
- Logic Synthesis Flow
  - Demo

# LUT Mapping

- Objective
  - Subject graph (e.g., AIG) → LUT network
  - Minimize LUT count (area) & level (delay)
- Cut-based technology mapping
  - Generate multiple candidate cuts for each node
  - Rank the cuts and select a set of “good” ones
    - Metrics: arrival time, area-flow, exact-area, …



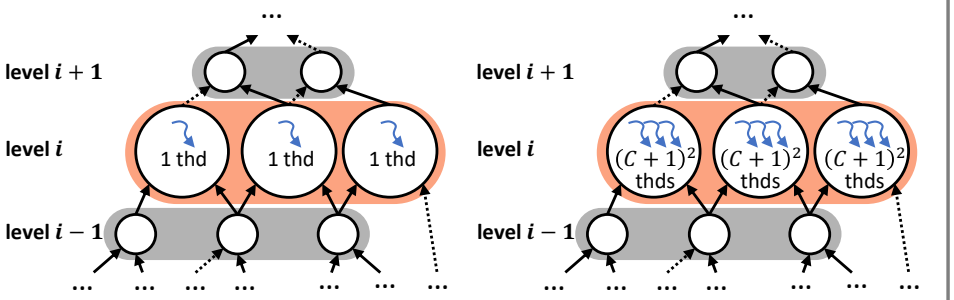




A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton,
"Combinational and sequential mapping with priority cuts", ICCAD’07.

# GPU LUT Mapping

- Four key improvements

**Impr. 1: Fine-grained Parallelism**




Previous work Ours

**Impr. 3: Dedicated Memory Management**


GPU-based Memory Pool for Cuts

**Impr. 2: New Cut Evaluation Metric**

- Data race in computation of the old metric
- Data-race-free and effective new metric provably equivalent to previous one

**Impr. 4: Fully-parallel Mapping Engine**

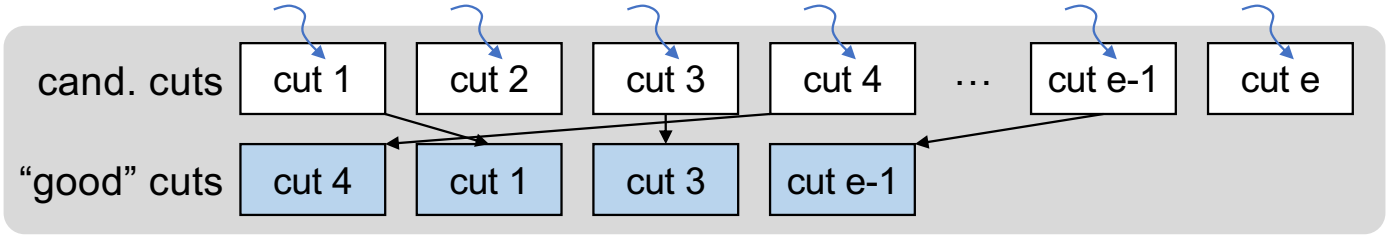
- All subroutines in LUT mapping are parallelized

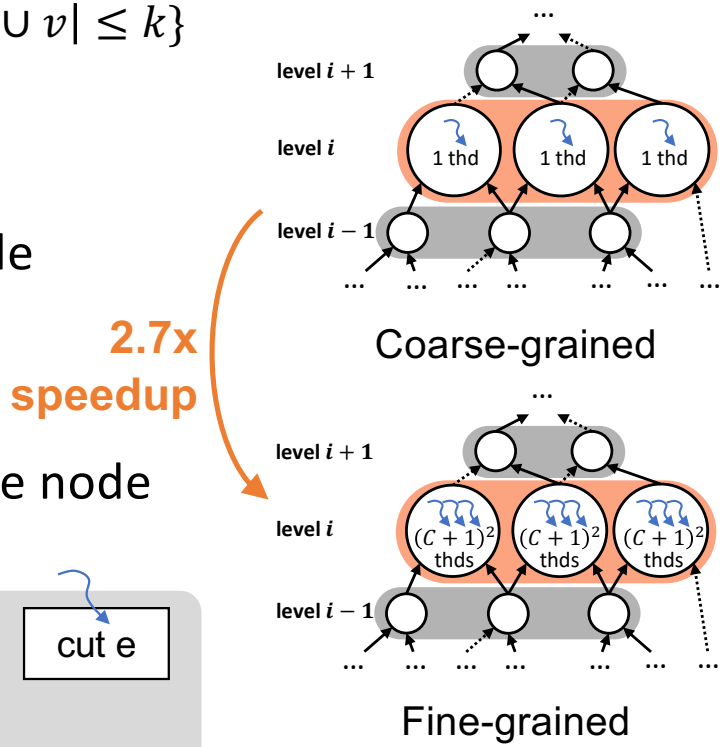
# GPU LUT Mapping

- Candidate cut generation via **cut enumeration**

$$E(n) = \{u \cup v \ : \ u \in P\big(in_0(n)\big), v \in P\big(in_1(n)\big), |u \cup v| \leq k\}$$

- Previous work on GPU LUT mapping [FPL'10]
  - Coarse-grained level-wise parallel: one thread per node
- Our method: fine-grained parallel
  - One thread per candidate cut
  - Cut ranking and selection by all threads assigned to the node



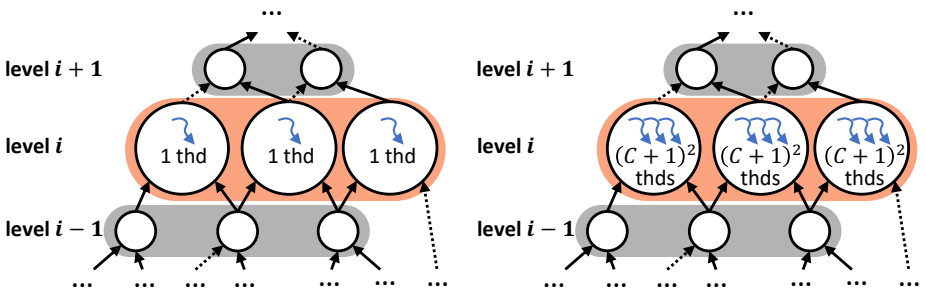




D. Chen and D. Singh, "Parallelizing FPGA technology mapping using Graphics Processing Units (GPUs)", FPL'10.

# GPU LUT Mapping

- Four key improvements

**Impr. 1: Fine-grained Parallelism**




Previous work Ours

**Impr. 3: Dedicated Memory Management**


GPU-based Memory Pool for Cuts

**Impr. 2: New Cut Evaluation Metric**

- Data race in computation of the old metric
- Data-race-free and effective new metric provably equivalent to previous one

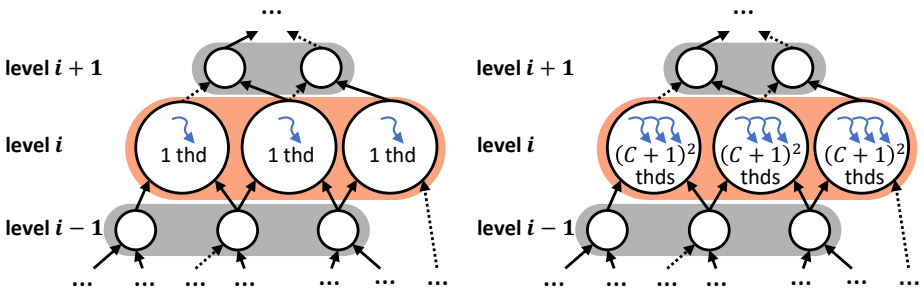
**Impr. 4: Fully-parallel Mapping Engine**

- All subroutines in LUT mapping are parallelized

# GPU LUT Mapping

- Four key improvements

## Impr. 1: Fine-grained Parallelism




Previous work Ours

## Impr. 2: New Cut Evaluation Metric

- Data race in computation of the old metric
- Data-race-free and effective new metric provably equivalent to previous one

## Impr. 3: Dedicated Memory Management


GPU-based Memory Pool for Cuts

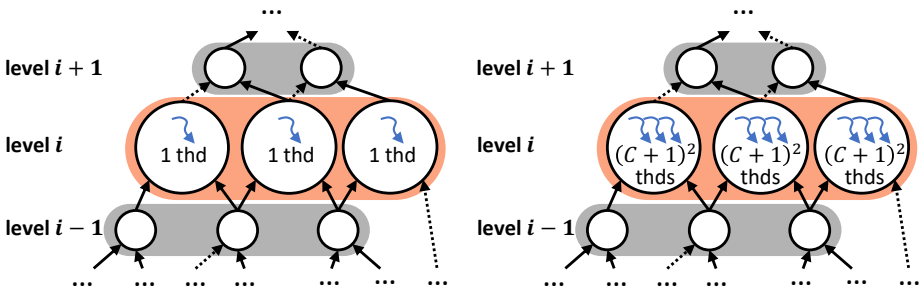
## Impr. 4: Fully-parallel Mapping Engine

- All subroutines in LUT mapping are parallelized

# GPU LUT Mapping

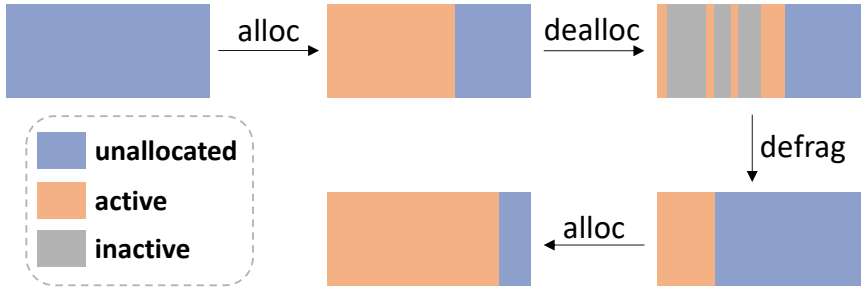
- Four key improvements

## Impr. 1: Fine-grained Parallelism




Previous work Ours

## Impr. 3: Dedicated Memory Management




GPU-based Memory Pool for Cuts

## Impr. 2: New Cut Evaluation Metric

- Data race in computation of the old metric
- Data-race-free and effective new metric provably equivalent to previous one

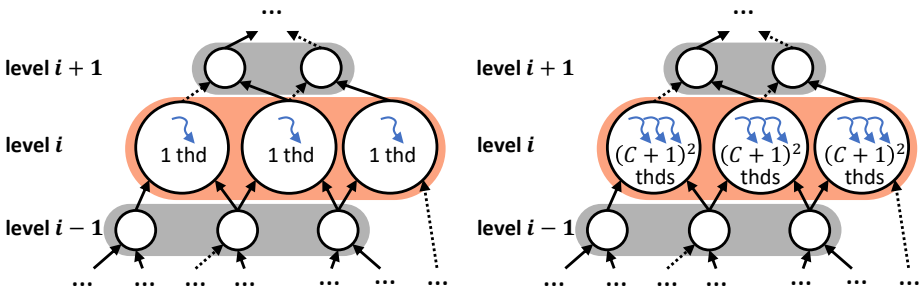
## Impr. 4: Fully-parallel Mapping Engine

- All subroutines in LUT mapping are parallelized

# GPU LUT Mapping

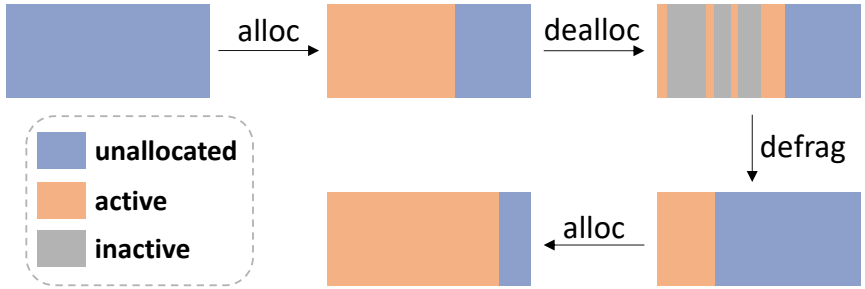
- Four key improvements

## Impr. 1: Fine-grained Parallelism




Previous work Ours

## Impr. 3: Dedicated Memory Management




GPU-based Memory Pool for Cuts

## Impr. 2: New Cut Evaluation Metric

- Data race in computation of the old metric
- Data-race-free and effective new metric provably equivalent to previous one

## Impr. 4: Fully-parallel Mapping Engine

- All subroutines in LUT mapping are parallelized

# GPU LUT Mapping - Results

- Setup
  - RTX A6000 GPU (mid-range consumer-level)
  - Benchmarks: large-scale AIGs (4~60M nodes)
- Compare with ABC LUT mapper (if)

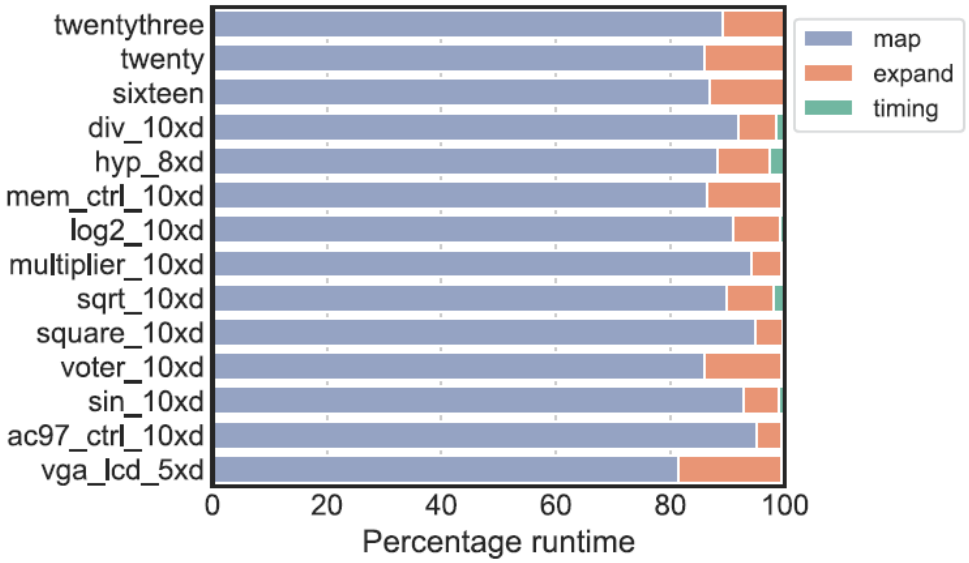
| Speed-up | #LUTs | Level |
|---|---|---|
| **34.6x** | **-0.2%** | **-0.0%** |

- Compare with FPL'10 GPU LUT mapper

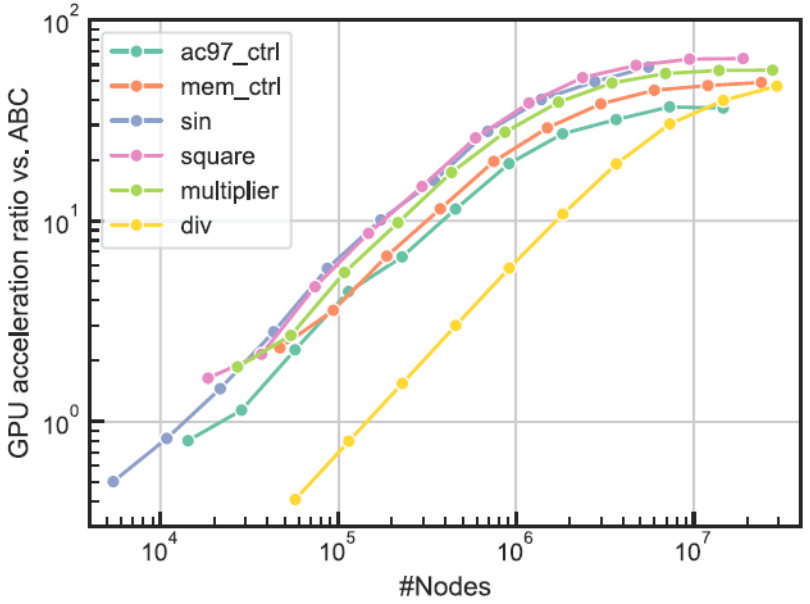
| Type | Speed-up |
|---|---|
| cut gen&sel only | **2.7x** |
| full flow | **4.7x** |

# GPU LUT Mapping - Results

- Runtime breakdown

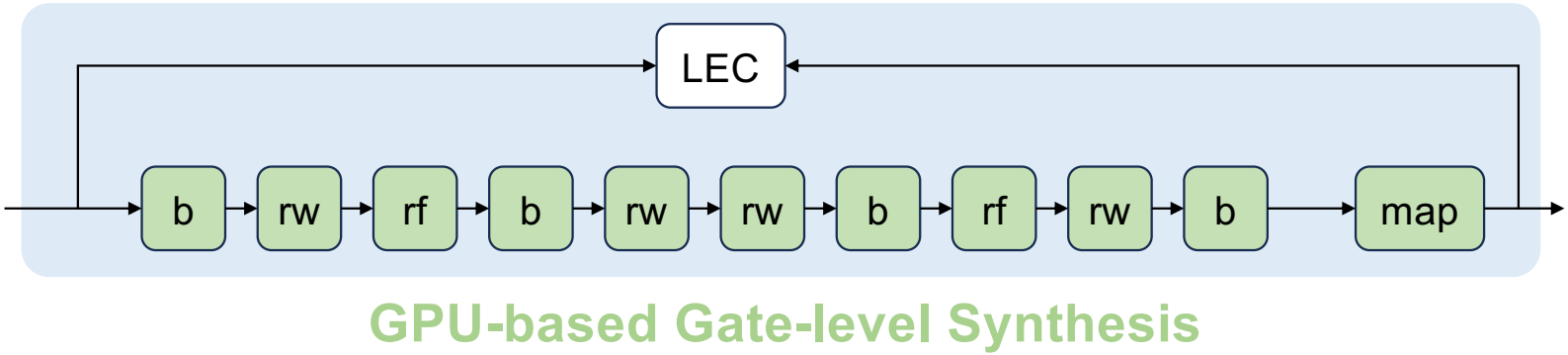



- Scaling



# GPU LUT Mapping - Summary

- An ultra-fast GPU LUT mapping engine with four key improvements
  - Fine-grained parallel cut enumeration and selection
  - A new metric evaluation effective in parallel scenario
  - Dynamic memory management
  - Fully parallel mapping flow



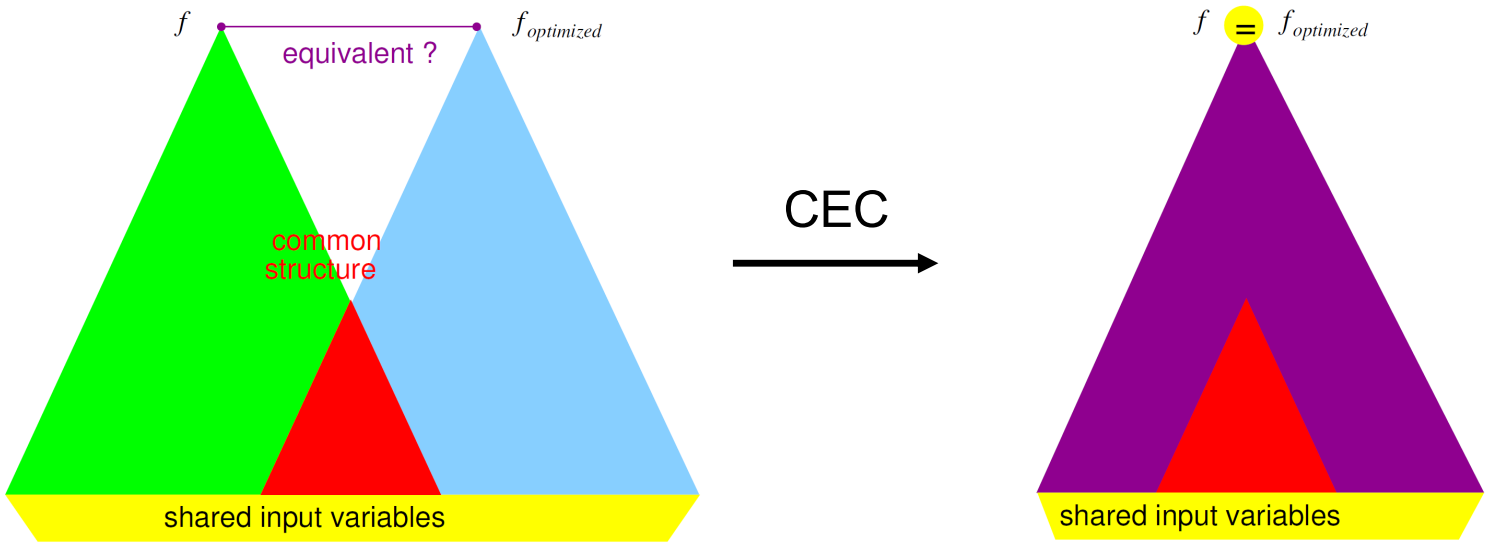

**GPU-based Gate-level Synthesis**

# Agenda

- GPU Acceleration on:
  - Logic Optimization
  - Mapping
  - Circuit Equivalence Check
- Logic Synthesis Flow
  - Demo

# Combinational Circuit Equivalence Checking (CEC)

- Prove the equivalence of two netlists implementing the same circuit
- CEC is co-NP-complete: no universally efficient algorithm
- Applications: verification, logic synthesis, functional ECO



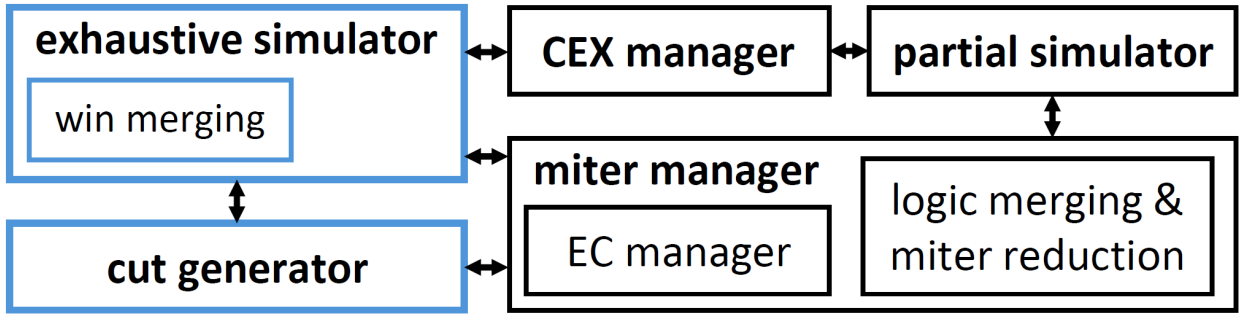

Source: Armin Biere, “SAT in Formal Hardware Verification”, SAT’05 (invited talk)

# CEC by Sweeping

- Miter: combine POs with XORs, sharing PIs of the two circuits,
  - Two circuits are equivalent <=> miter is constant 0
- CEC by sweeping
  - Gradually reduce the miter by proving and merging internal equivalent nodes, which finally makes proving the POs easier.
  - Use a formal method (e.g., BDD, SAT) to perform the proof
- Assumption: there are **internal equivalent nodes** in the miter
  - Usually this is the case if one circuit is the optimized version of another, e.g., during synthesis

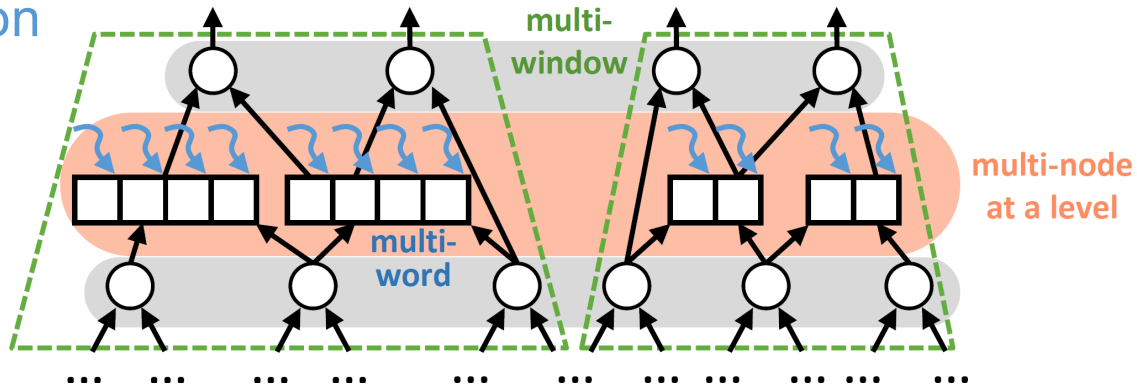
# Sim-based CEC - Overview

- General Idea: use **exhaustive simulation** to prove equivalences
- Massive computational power: parallel computation with **GPUs**
- Have a **local function checking** scheme for reducing time complexity



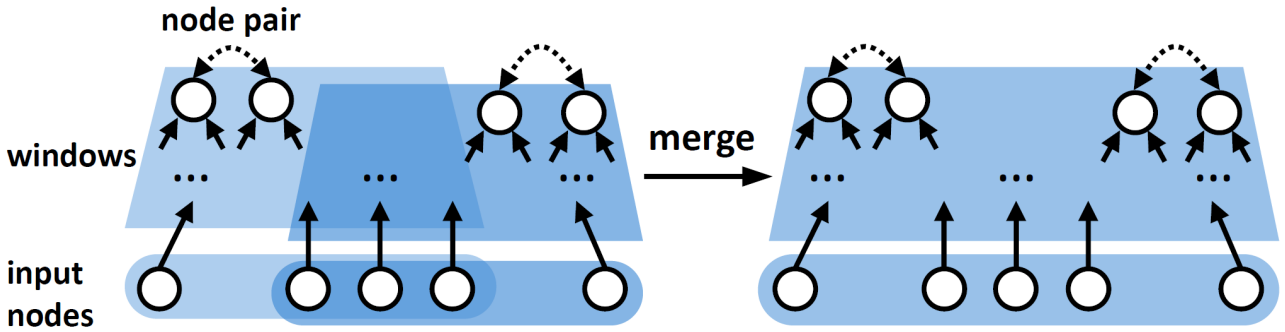
# Sim-based CEC - Exhaustive Simulation

- Exhaustive Simulator:
  - Check equivalence of candidate node pairs
  - For each pair, find a window, enumerate all input patterns at the window input and compute response at the two nodes
- Three dimensions of parallelism
  1. Parallel simulation of different pairs (**windows**)
  2. Level-wise parallel node simulation
  3. Parallel multi-word simulation



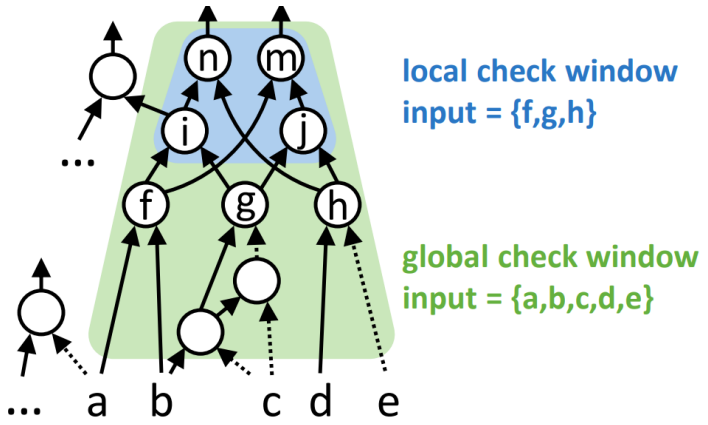
# Sim-based CEC – Global Function Checking

- Windows with PI as input.
- Merge windows with high overlaps:
  - One window for simulating multiple pairs
  - Reduce simulation effort and window number



# Sim-based CEC - Local Function Checking

- Idea:
  - Check the equivalence of two nodes' functions in terms of a common cut
  - Restrict the common cut size
- Properties:
  - EQ local function ⇒ EQ pair
  - NEQ local function ⇏ NEQ pair
  - The patterns leading to NEQ can be **satisfiability don't cares** (SDCs)
- Avoid SDCs at the cut nodes:
  - Check multiple cuts for a node pair
  - Improve the "quality" of the cuts



# Sim-based CEC - Cut Quality

- How to select best cuts?

| Pass | Main Metric | Tie-breaker Metric 1 | Tie-breaker Metric 2 |
|---|---|---|---|
| 1 | large avg. **fanout** | small **cut size** | small avg. **level** |
| 2 | small avg. level | small cut size | large avg. fanout |
| 3 | large avg. level | small cut size | large avg. fanout |

- Multiple passes with different selection metrics
  - Ensuring high diversity of cuts
- Finding similar cuts for two nodes in a pair

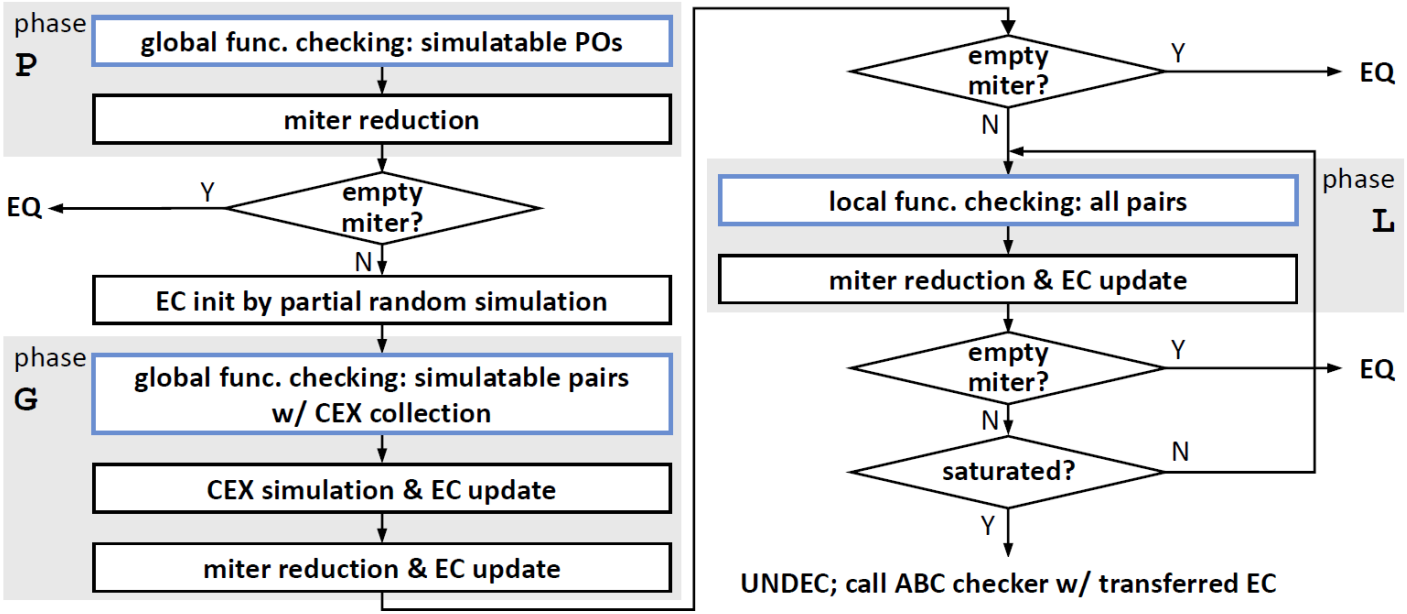
# Sim-based CEC – Simulation and Cut Generation Flow

On-the-fly exhaustive simulation along with the cut generation process:

- A constant-sized common cut buffer
- Collect common cuts into the buffer by level-wise parallel cut enumeration
- Once the buffer is full, launch a batch of exhaustive simulation
- Clear the buffer and continue the cut enumeration process

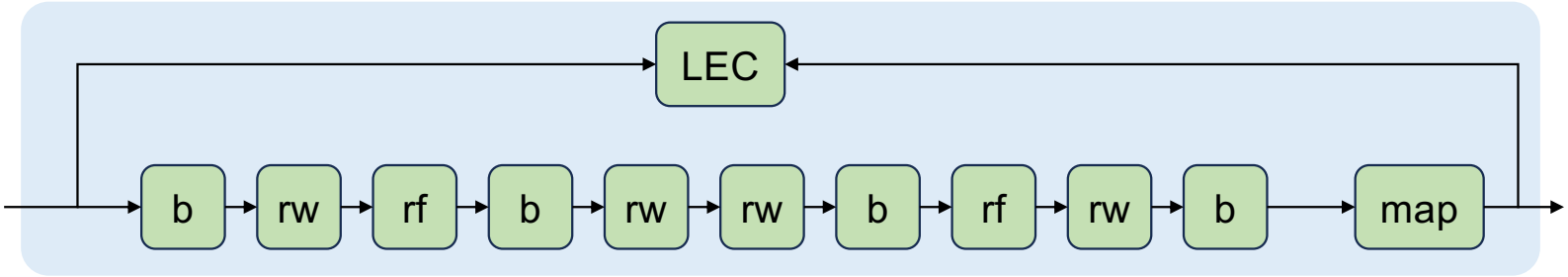
# Sim-based CEC - Overall Flow

- Three phases:
  - Phase **P**: check the **global** functions of simulatable (#support <= threshold) PO pairs
  - Phase **G**: check the **global** functions of simulatable internal node pairs
  - Phase **L**: check the **local** functions of all internal node pairs (until saturation)



# Sim-based CEC - Summary

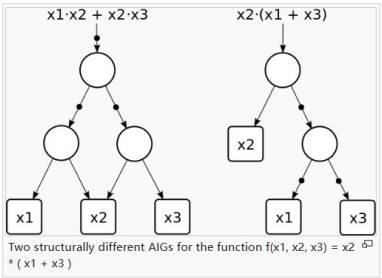
- A new perspective of checking equivalences using parallel exhaustive simulation
- Local function checking scheme for reducing checking effort
- Efficient parallel algorithms for exhaustive simulation and local function checking



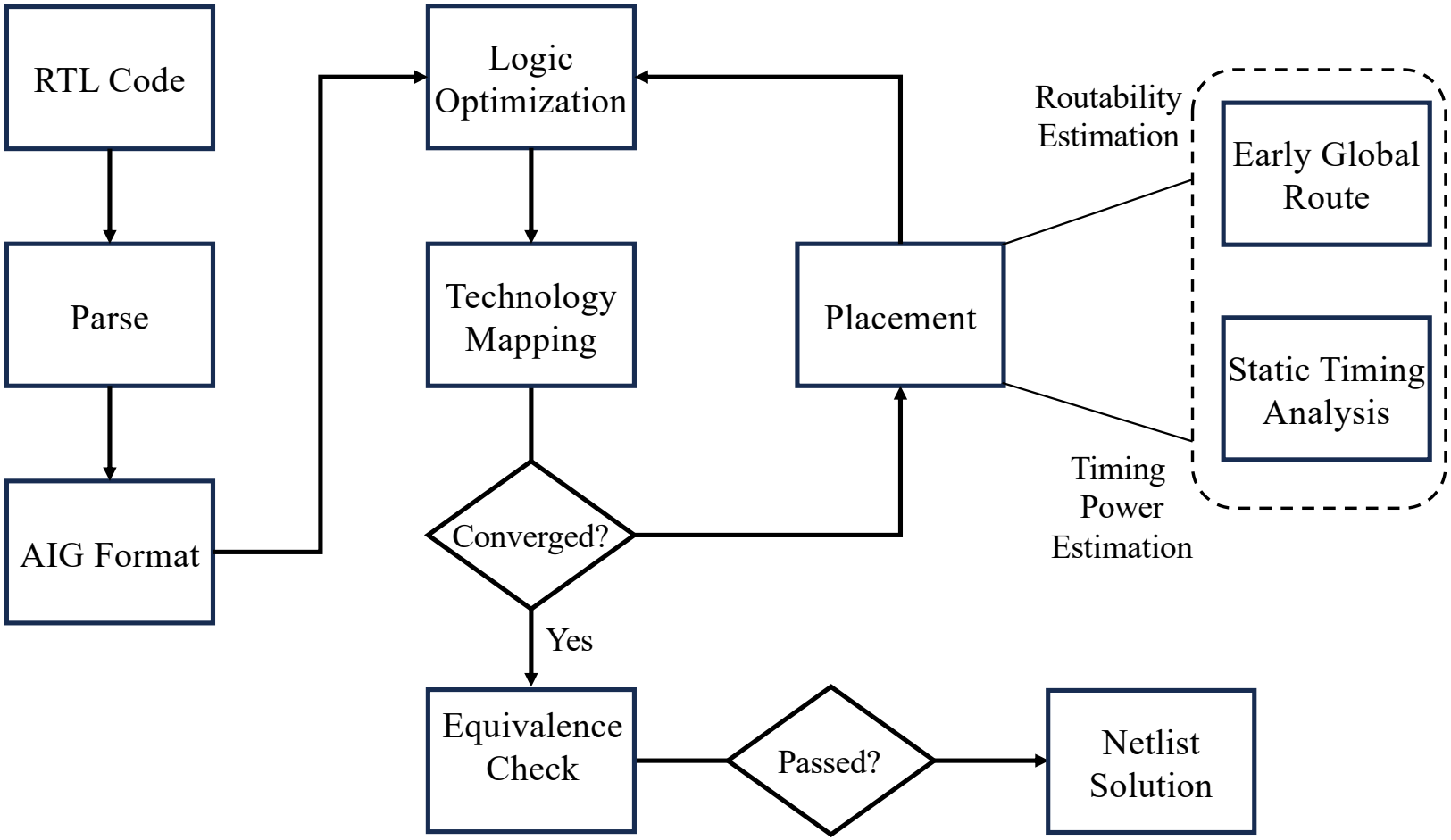
# Agenda

- GPU Acceleration on:
  - Logic Optimization
  - Mapping
  - Circuit Equivalence Check
- Fast Logic Synthesis Flow
  - Demo

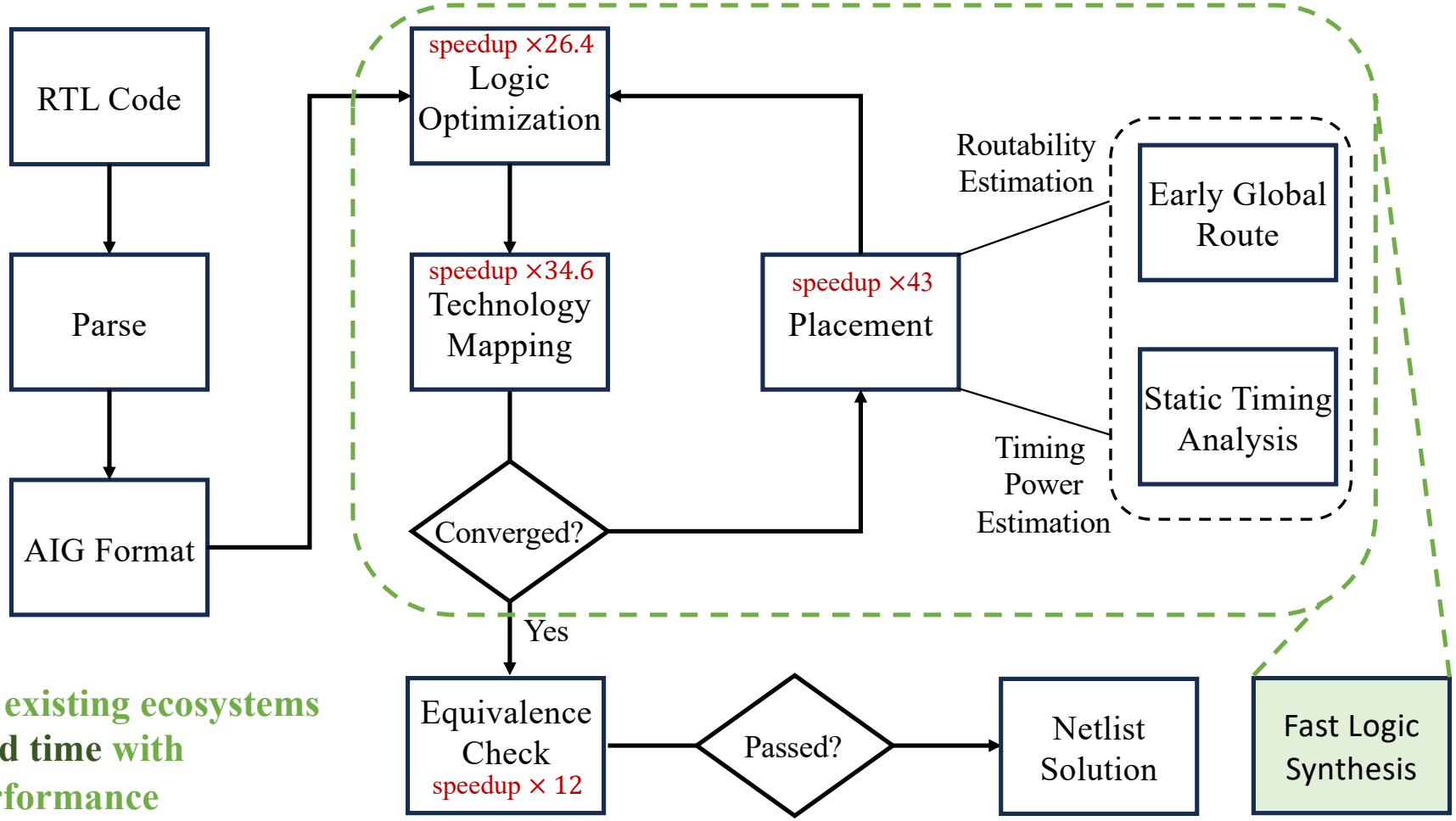
# Traditional Flow: CPU-based Logic Synthesis




A simple AIG example



# Our Solutions: GPU-accelerated Logic Synthesis




**Goal:**

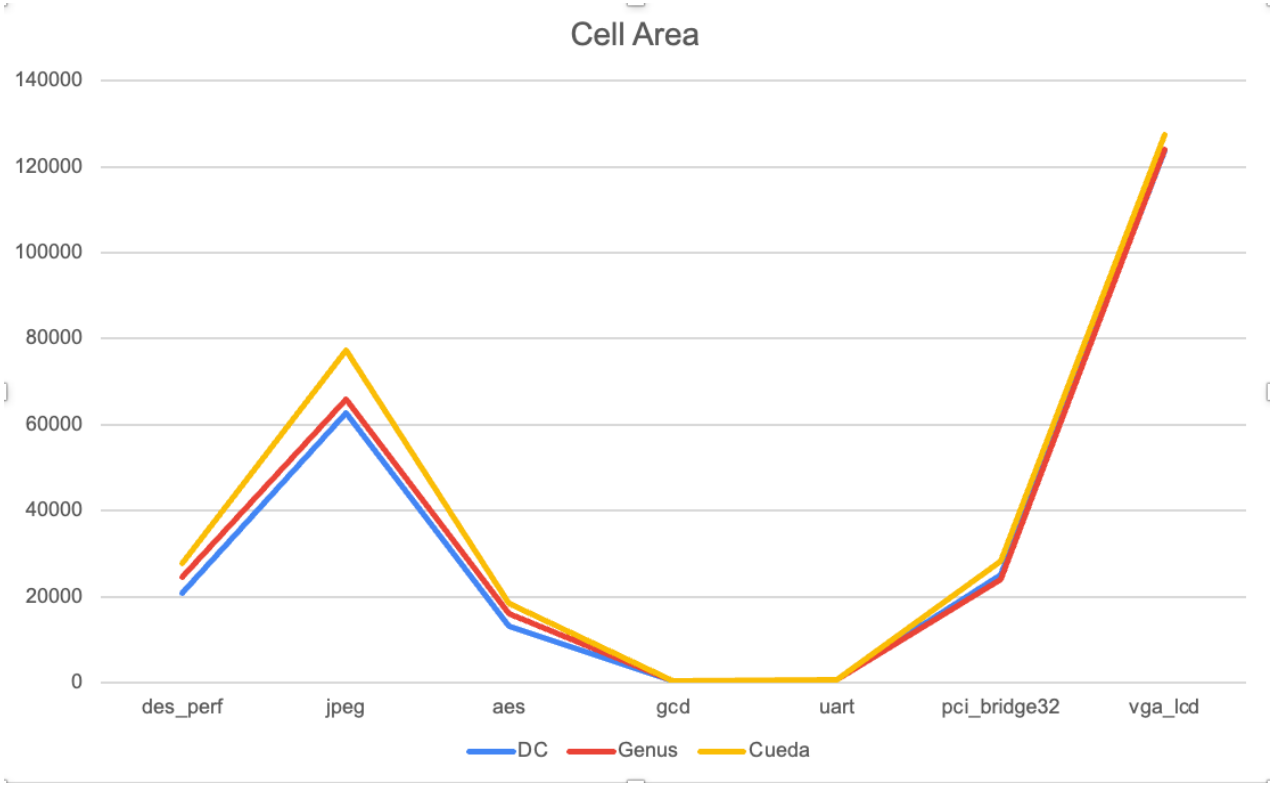
- **Integrated into existing ecosystems**
- **Fast turnaround time with competitive performance**

# Recent Result on Combinational Circuits

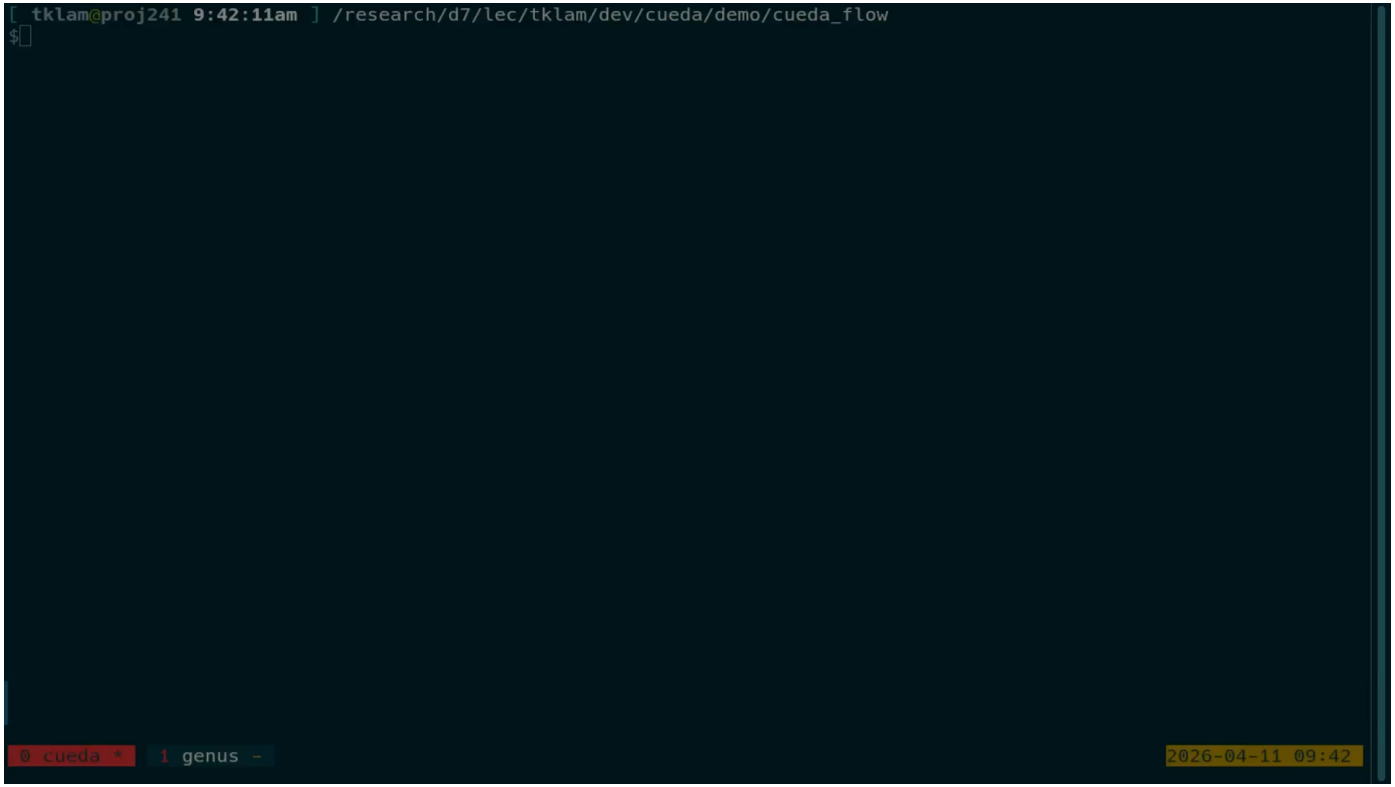
EPFL Benchmarks (Combinational)

| design | ABC &Deepsyn + (Ours) ExactMap + GPUBufSizer | | | | | |
|---|---|---|---|---|---|---|
| | WNS | TNS | Power | Area | Runtime(s) | #Cells |
| ac97_ctrl | -0.58 | -728.72 | 0.044844 | 6186.63 | 1 | 5360 |
| adder | -5.89 | -387.55 | 0.008949 | 761.83 | 1.012 | 637 |
| arbiter | -1.2 | -121.6 | 0.006705 | 1170.93 | 0.94 | 1263 |
| bar | -0.6 | -74.45 | 0.020045 | 1722.34 | 0.933 | 1476 |
| cavlc | -0.72 | -5.94 | 0.002428 | 288.08 | 0.902 | 285 |
| ctrl | -0.3 | -5.31 | 0.000459 | 64.37 | 0.896 | 69 |
| dec | -0.34 | -71.14 | 0.001496 | 318.67 | 0.912 | 324 |
| div | -187.34 | -16107.5 | 0.13724 | 15405.98 | 21.541 | 13922 |
| hyp | -687.19 | -32425.4 | 4.12703 | 152885.6 | 103.099 | 137932 |
| i2c | -0.55 | -47.05 | 0.002691 | 479.86 | 0.929 | 477 |
| int2float | -0.44 | -2.56 | 0.000808 | 104.01 | 0.88 | 105 |
| log2 | -13.21 | -394.96 | 0.343688 | 18121.68 | 5.881 | 14900 |
| max | -7.79 | -1002.22 | 0.022945 | 1860.92 | 2.485 | 1781 |
| mem_ctrl | -1.55 | -745.89 | 0.027906 | 4643.58 | 2.487 | 4137 |
| multiplier | -8.19 | -600.97 | 0.371136 | 15127.96 | 3.878 | 11563 |
| netcard | -2.04 | -147527 | 1.391033 | 310424.8 | 14.546 | 233028 |
| router | -0.32 | -0.94 | 0.000283 | 70.22 | 0.862 | 67 |
| sin | -6.32 | -139.94 | 0.064292 | 3403.47 | 2.456 | 3011 |
| sqrt | -163.95 | -3703.79 | 0.480022 | 13141.5 | 21.474 | 12428 |
| square | -6.46 | -423.31 | 0.218288 | 12045.19 | 2.355 | 10650 |
| vga_lcd | -1.43 | -22207.6 | 0.288212 | 52179.12 | 3.072 | 39224 |
| voter | -2.39 | -2.39 | 0.115648 | 6180.28 | 1.516 | 5103 |
| MEAN | -49.9455 | -10305.7 | 0.348916 | 28026.69 | 8.8207273 | 22624.64 |
| RATIO | 0.768936 | 0.798216 | 1.04573 | 1.003901 | 0.0122718 | 0.907706 |

# Correlations with Commercial Tools

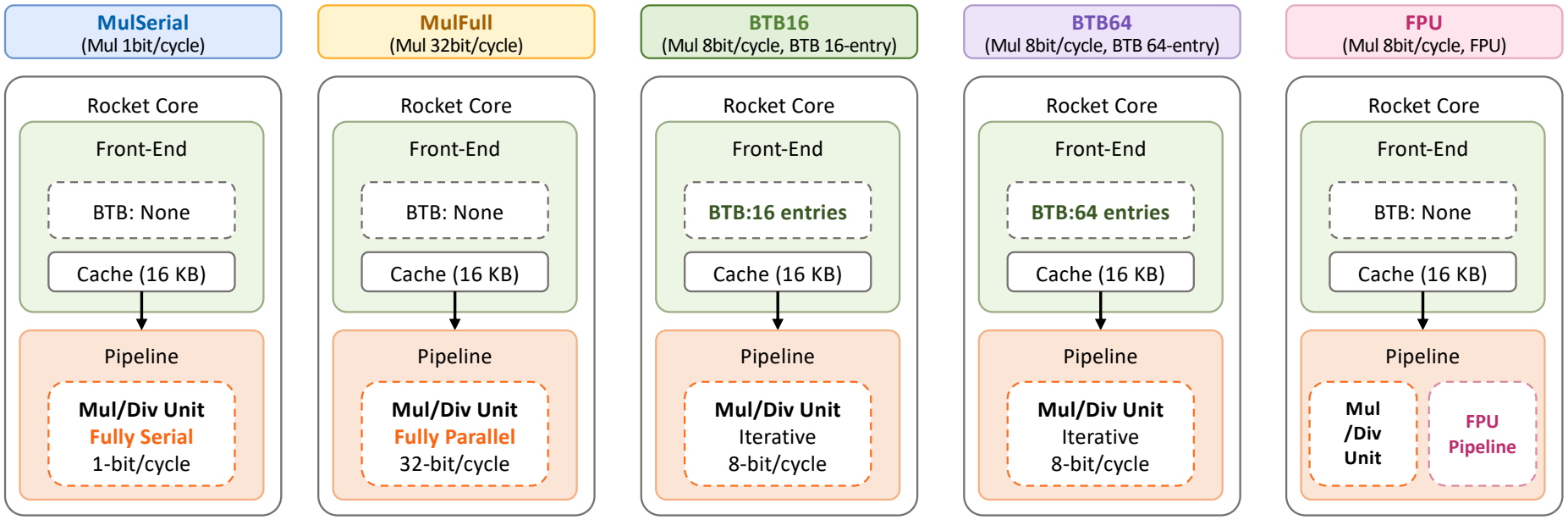


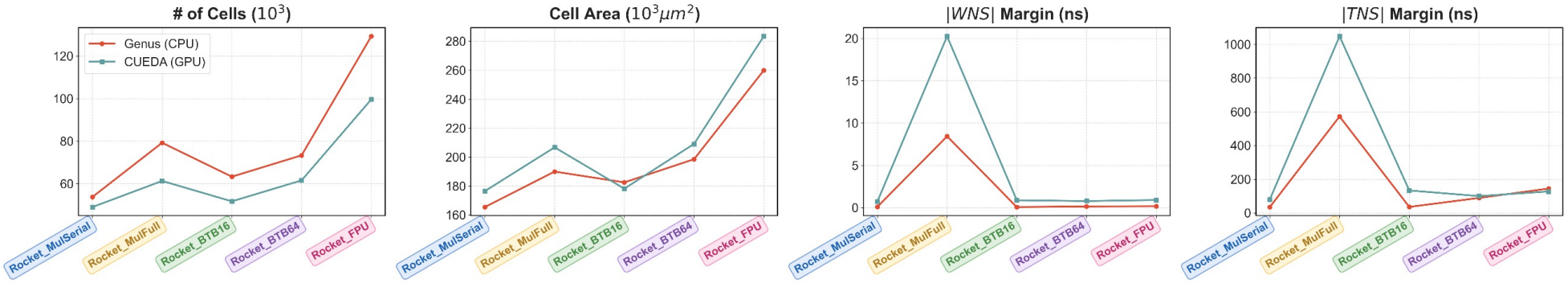
# Demo


[ tklam@proj241 9:42:11am ] /research/d7/lec/tklam/dev/cueda/demo/cueda_flow
$
0 cueda *
1 genus -
2026-04-11 09:42

# Design Space Exploration

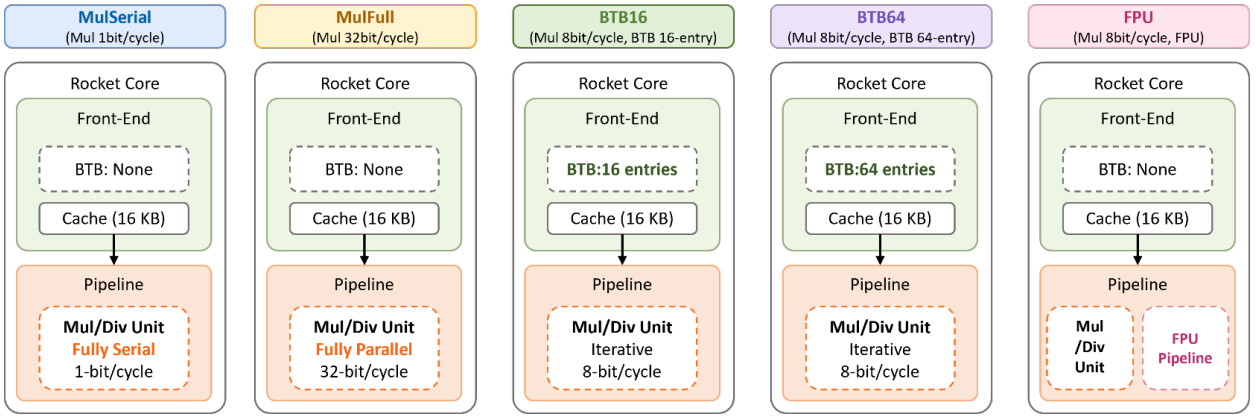
## How to customize the design of Rocket Core?

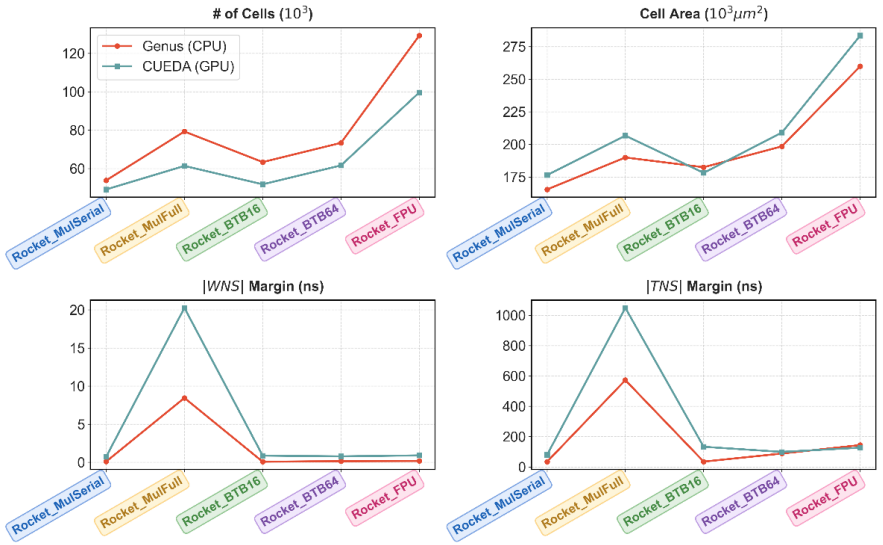


# Design Space Exploration

## How to customize the design of Rocket Core?




How long it takes to do the Design Space Searching?

>1 Hour with CPU commercial tool per design

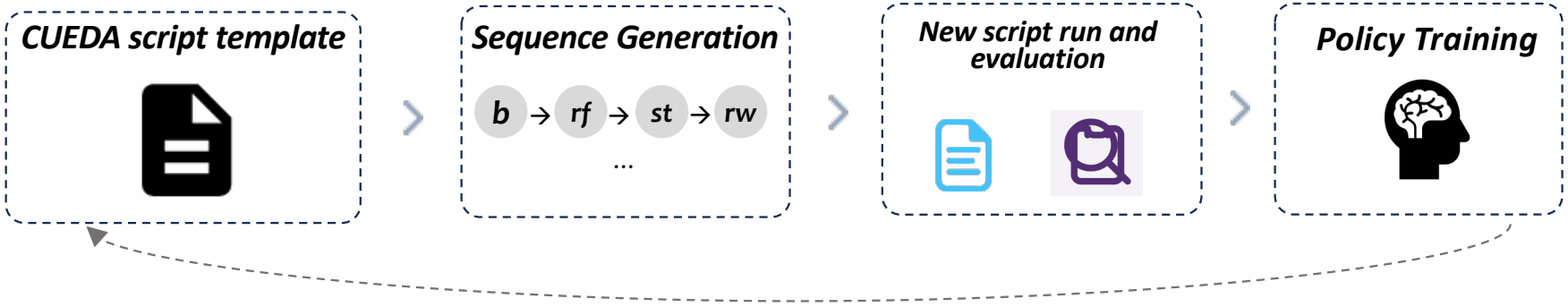
<1 **Min** with CUEDA per design

# Customize Design Flow by Reinforcement Learning

Berkeley ABC **resyn2** synthesis flow consists of several logic operators

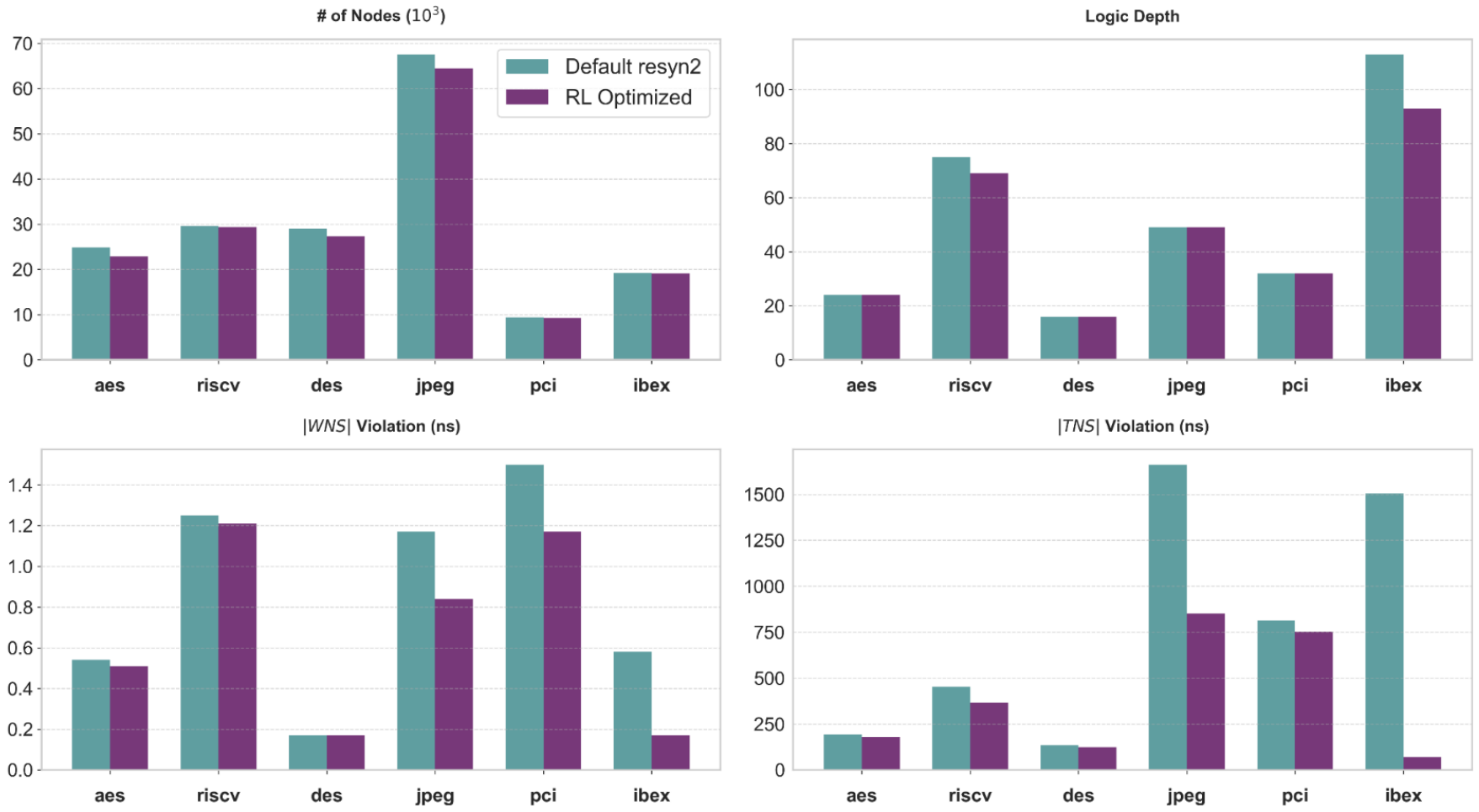
*balance, strash, rewrite, refactor*

b → rw → rf → st → b → rw → rw → rw → b → rf → st → rw → rw → b




6 hours training with **3000** optimization sequence

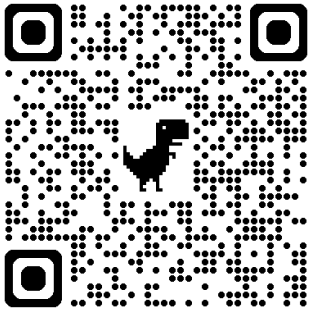
# Customize Design Flow by Reinforcement Learning



# Conclusion

GPU-accelerated Logic Synthesis:

- An exciting endeavor **for the chip design and microelectronics industry**.
- It can be used for front end designer to synthesize their designs quickly to estimate performance.
- It can also be used as an LLM agent to generate a design automatically.



https://cueda.cse.cuhk.edu.hk/

cuedahk@gmail.com

# Thank YOU