HeteroTime: Accelerating Static Timing Analysis using GPUs

Dr. Tsung-Wei Huang

Department of Electrical and Computer Engineering

University of Utah, UT

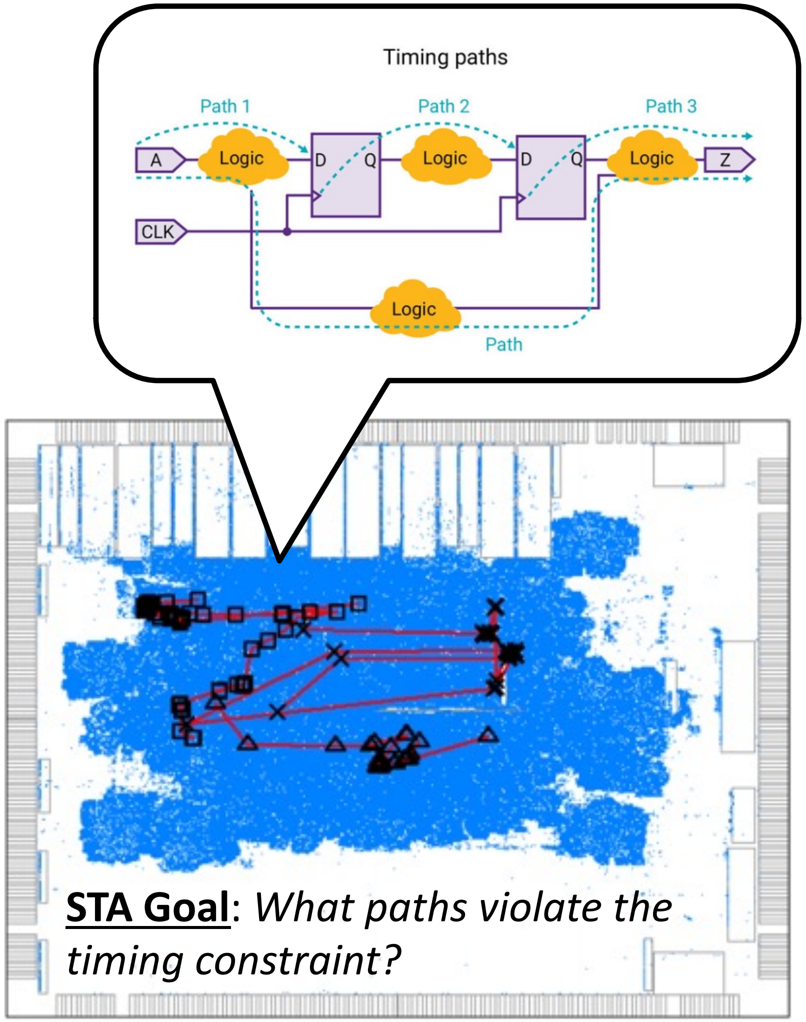
https://tsung-wei-huang.github.io/

Agenda

- ❑ **Introduce the scalability problem of STA**
  - ❑ What is STA and its challenges?
  - ❑ Why do we need new parallel paradigm for STA?
- ❑ **Accelerate graph-based analysis using GPU**
- ❑ **Accelerate path-based analysis using GPU**

Static Timing Analysis

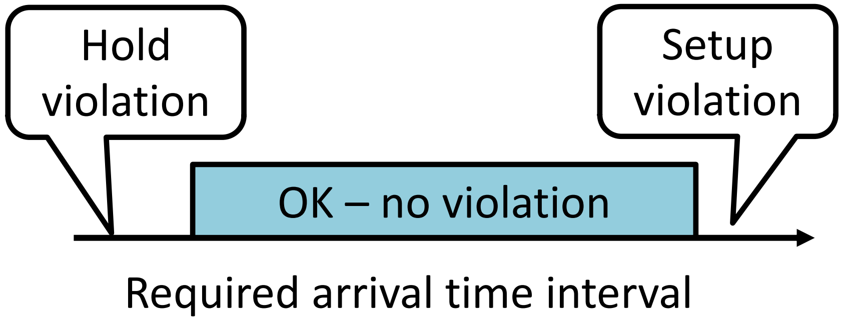
- ❑ **Static timing analysis (STA)**
  - ❑ Key step in the VLSI design
  - ❑ Verify the circuit timing
- ❑ **Analyze worst-case timing**
  - ❑ Minimum timing values
  - ❑ Maximum timing values
- ❑ **Challenges**
  - ❑ Compute giant graphs
  - ❑ Analyze millions of paths
  - ❑ Balance the loads
  - ❑ …



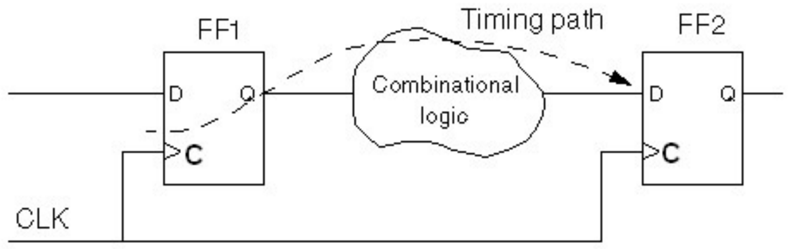
STA Goal: *What paths violate the timing constraint?*

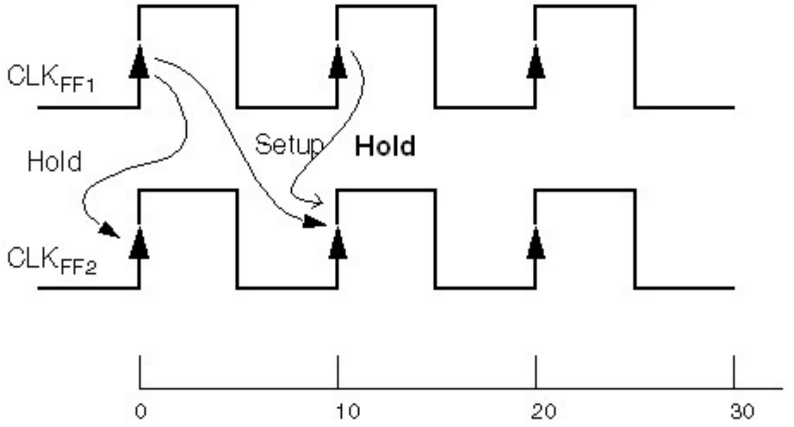
Timing Checks (Required Arrival Time)

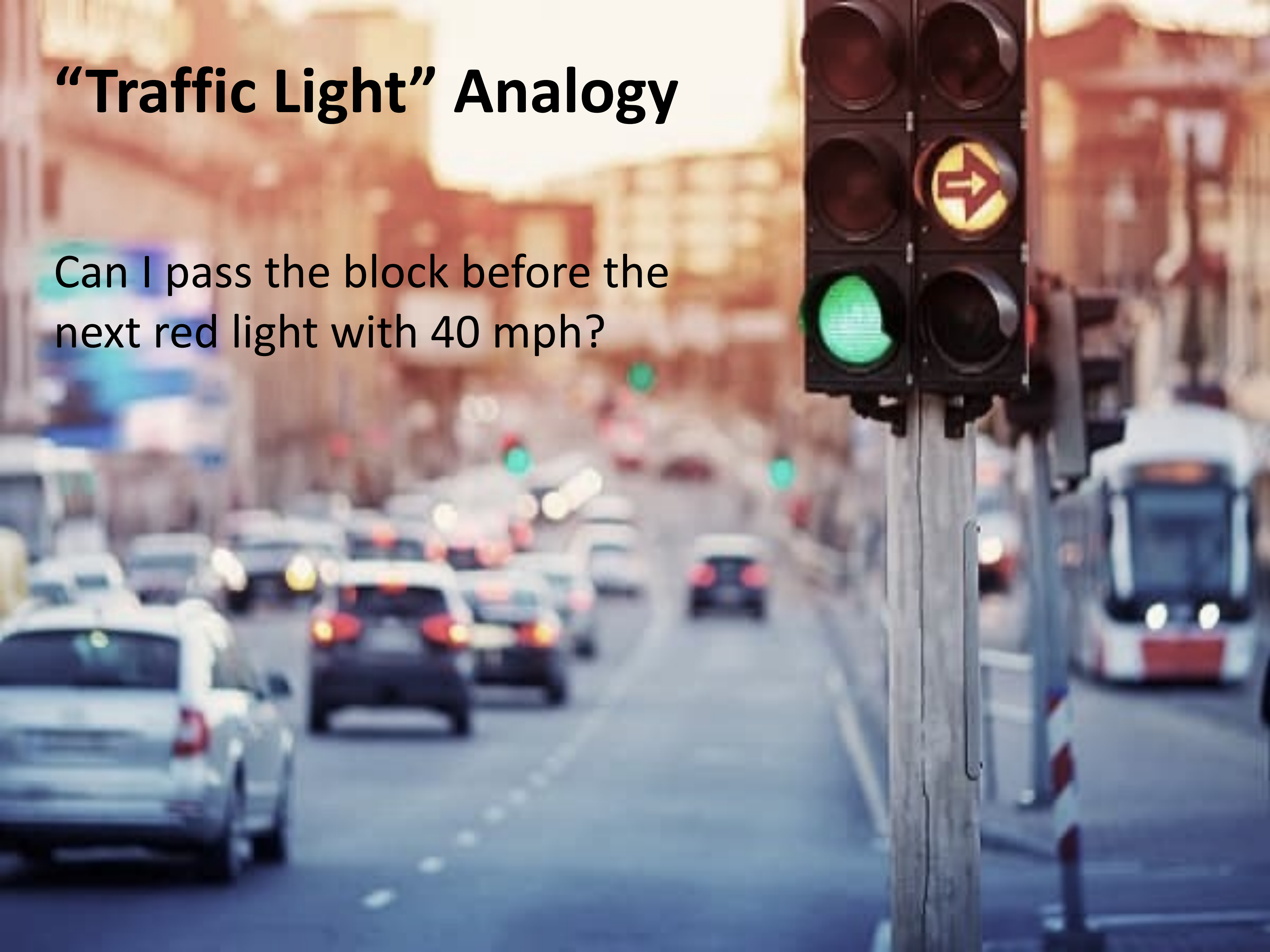
- ❑ **Modern circuits are sequential**
  - ❑ Drive data signal via clocks
  - ❑ Capture data via flip-flops (FF)s
- ❑ **Timing constraints**
  - ❑ Min required arrival time
    - After clock: hold
  - ❑ Max required arrival time
    - Before clock: setup



Required arrival time interval







“Traffic Light” Analogy

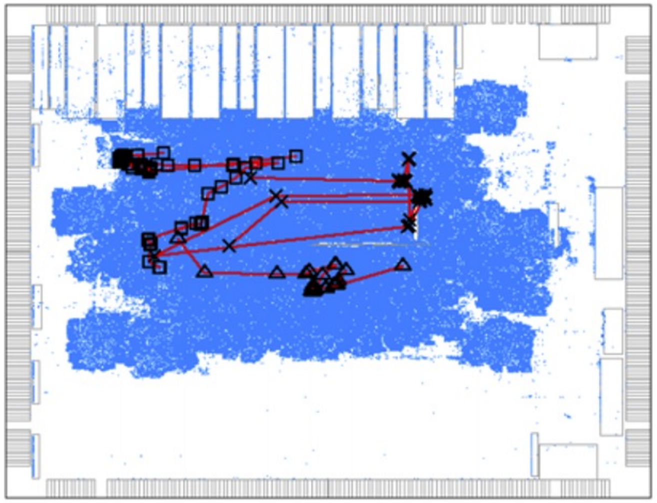
Can I pass the block before the next red light with 40 mph?

Building a Good Traffic System is Hard

Trillions of sections and traffic lights to analyze …

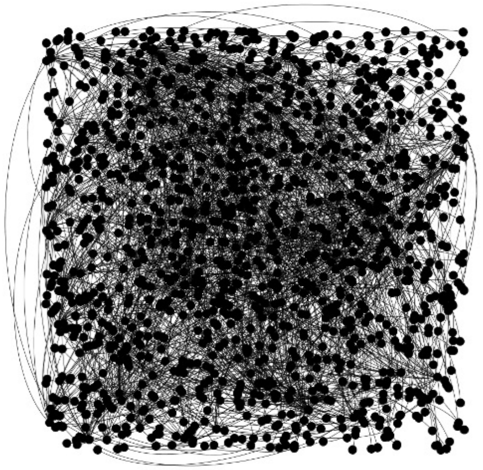
STA is Computationally Challenging!

- **STA graphs is extremely large and irregular**
  - Millions to billions of nodes and edges
  - Propagate timing information along giant graphs

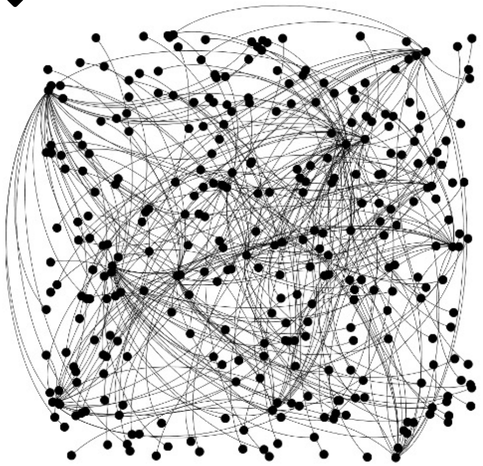

ISPD circuit design (10M gates)

Complete analysis can take **8 hours** and **800 GB RAM**

STA graphs



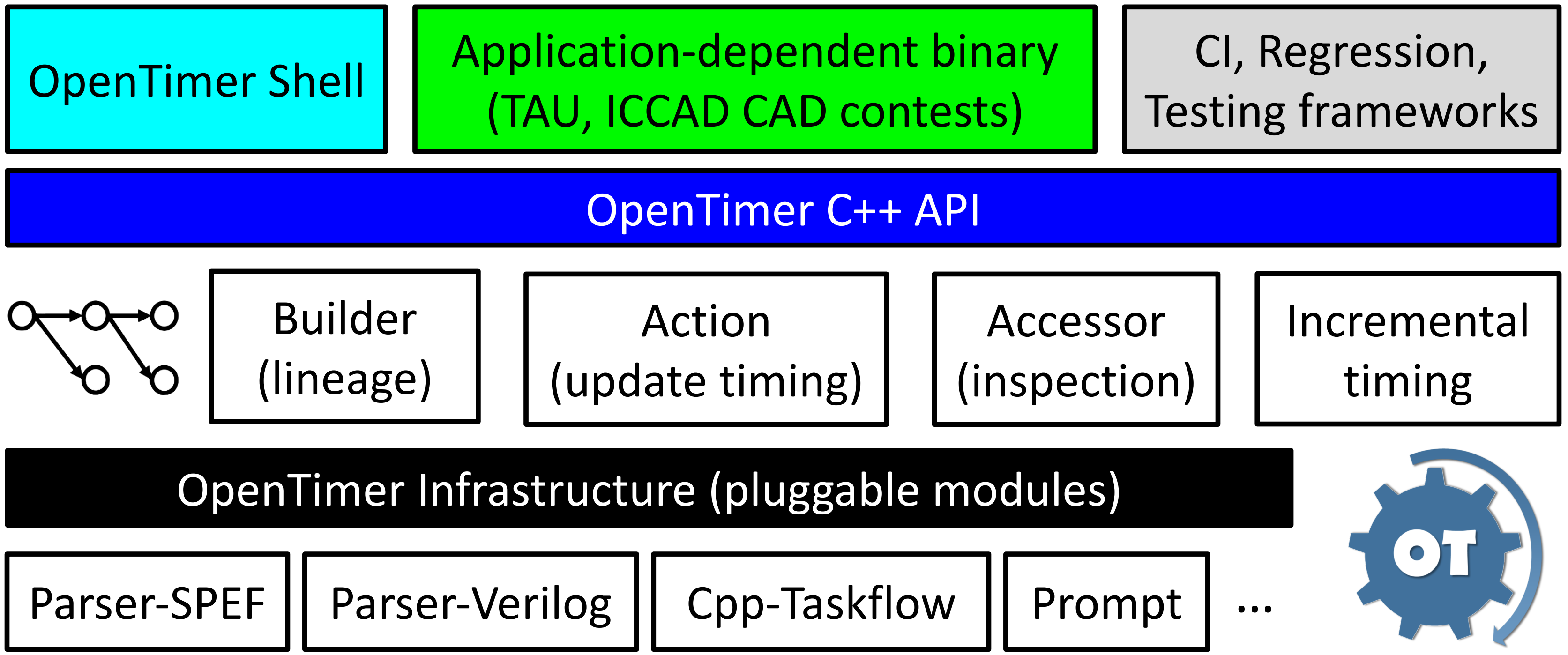
A datapath



STA graphs are extremely large and irregular

Our STA Solution: OpenTimer

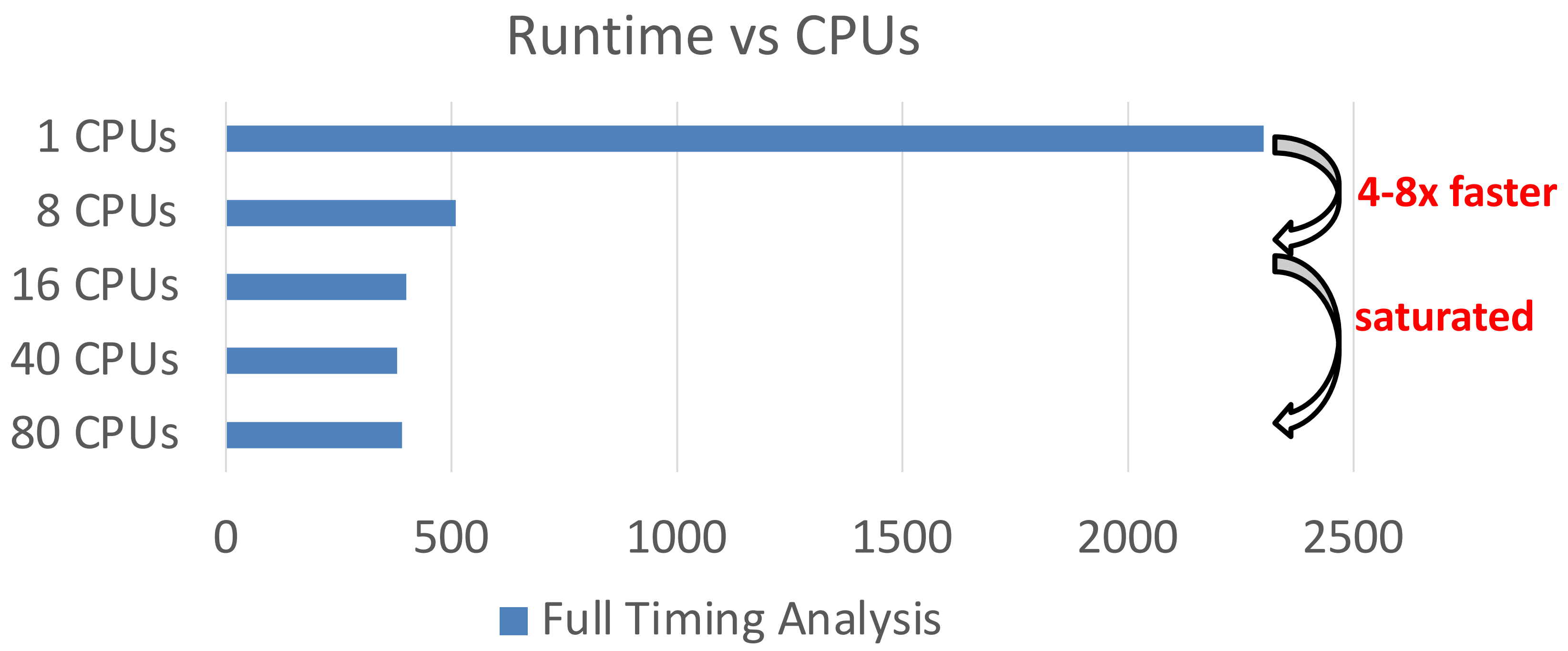
- Open-source incremental timing analysis software
  - https://github.com/OpenTimer/OpenTimer



T.-W. Huang et al., "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," IEEE TCAD21

How Can We Make STA Run Faster?

- ❑ **Leverage many-core CPUs to speed up the runtime**
  - ❑ Dramatic speed-up using 8 cores
  - ❑ Yet, scalability ***saturates*** at about 10—16 cores



Observed Scalability Bottleneck

- ❑ **CPU-only parallelism stagnates at about 10 cores**
  - ❑ “Amdahl’s Law” limits the strong scalability
    - The maximum speed-up you can achieve from a problem is proportional to the part that can be parallelized
  - ❑ Circuit graph structures limits the maximum parallelism
    - If the graph has only 10 parallel nodes at a level, we won’t achieve 40x speed-up
  - ❑ Irregular computations limits the memory bandwidth
    - STA is graph-oriented, not cache-friendly
- ❑ **Need to incorporate new parallel paradigms**
  - ❑ GPU opens opportunities for new scalability milestones
    - e.g., 20—80x speed-up reported in placement

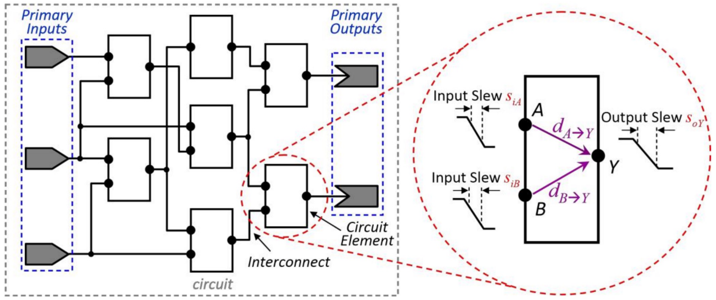
How can we leverage new GPU parallel paradigms to accelerate static timing analysis algorithms and achieve transformational performance milestones?

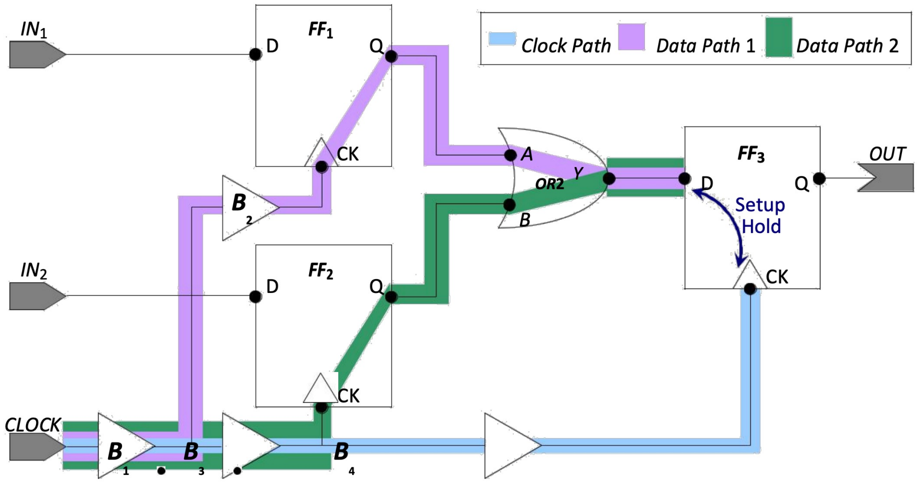
Leverage GPU to Accelerate STA

- **We target two important STA steps:**
  - Graph-based analysis (GBA) – *shortest path finding*
  - Path-based analysis (PBA) – *k-shortest path finding*
- **We design new GPU-accelerated STA algorithms:**
  - CPU-GPU task decomposition
  - GPU kernels for timing update

<u>PBA</u> analyzes critical paths one by one on a updated graph

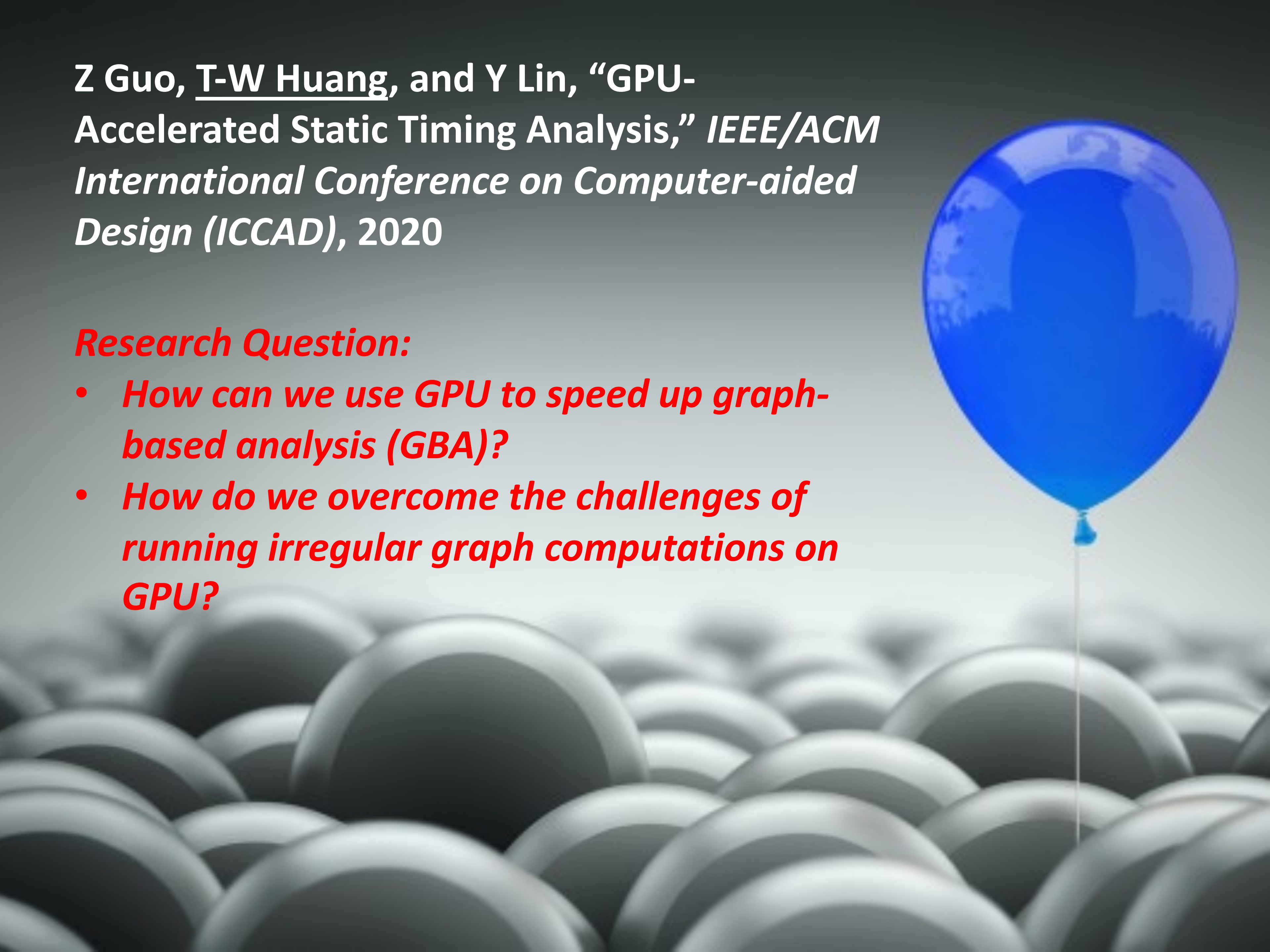
<u>GBA</u> computes the delay, slew, arrival time at each node and edge





Agenda

- ❑ **Introduce the scalability problem of STA**
    - ❑ What is STA and its challenges?
    - ❑ Why do we need new parallel paradigm for STA?
- ❑ **Accelerate graph-based analysis using GPU**
- ❑ **Accelerate path-based analysis using GPU**



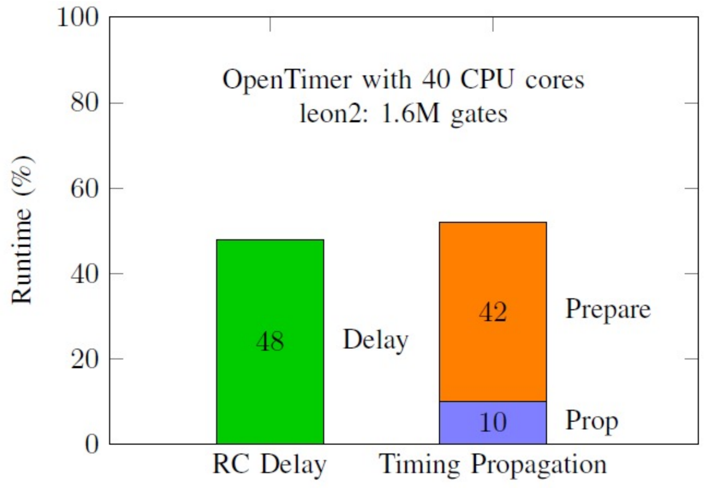
Z Guo, T-W Huang, and Y Lin, "GPU-Accelerated Static Timing Analysis," *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, 2020

Research Question:

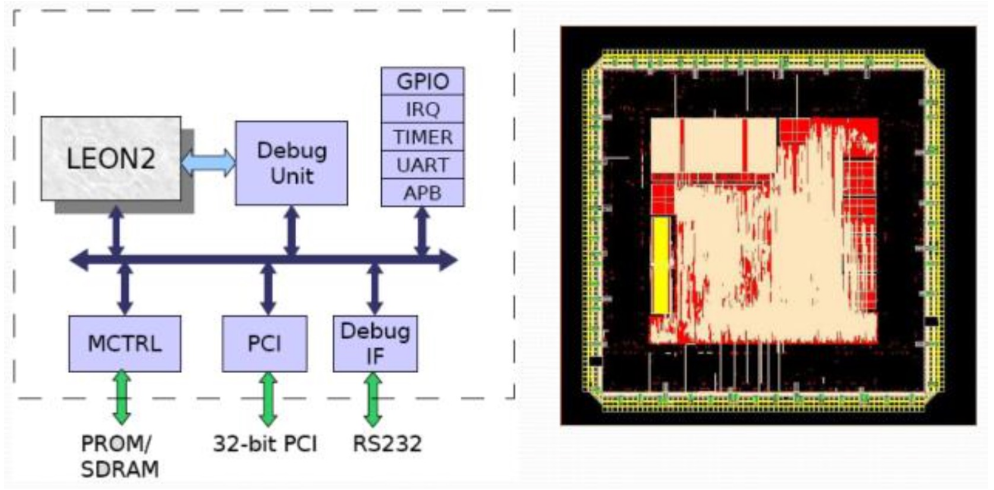
- *How can we use GPU to speed up graph-based analysis (GBA)?*
- *How do we overcome the challenges of running irregular graph computations on GPU?*

Runtime Breakdown of GBA

- ❑ **GBA has three time-consuming steps**
  1. Prepare tasks through levelization → 42% runtime
     - All pins at the same level can run in parallel => data parallelism
  2. Compute RC delay → 48% runtime
     - RC network (e.g., SPEF) can take GBs of data to compute
  3. Propagate timing → 10% runtime

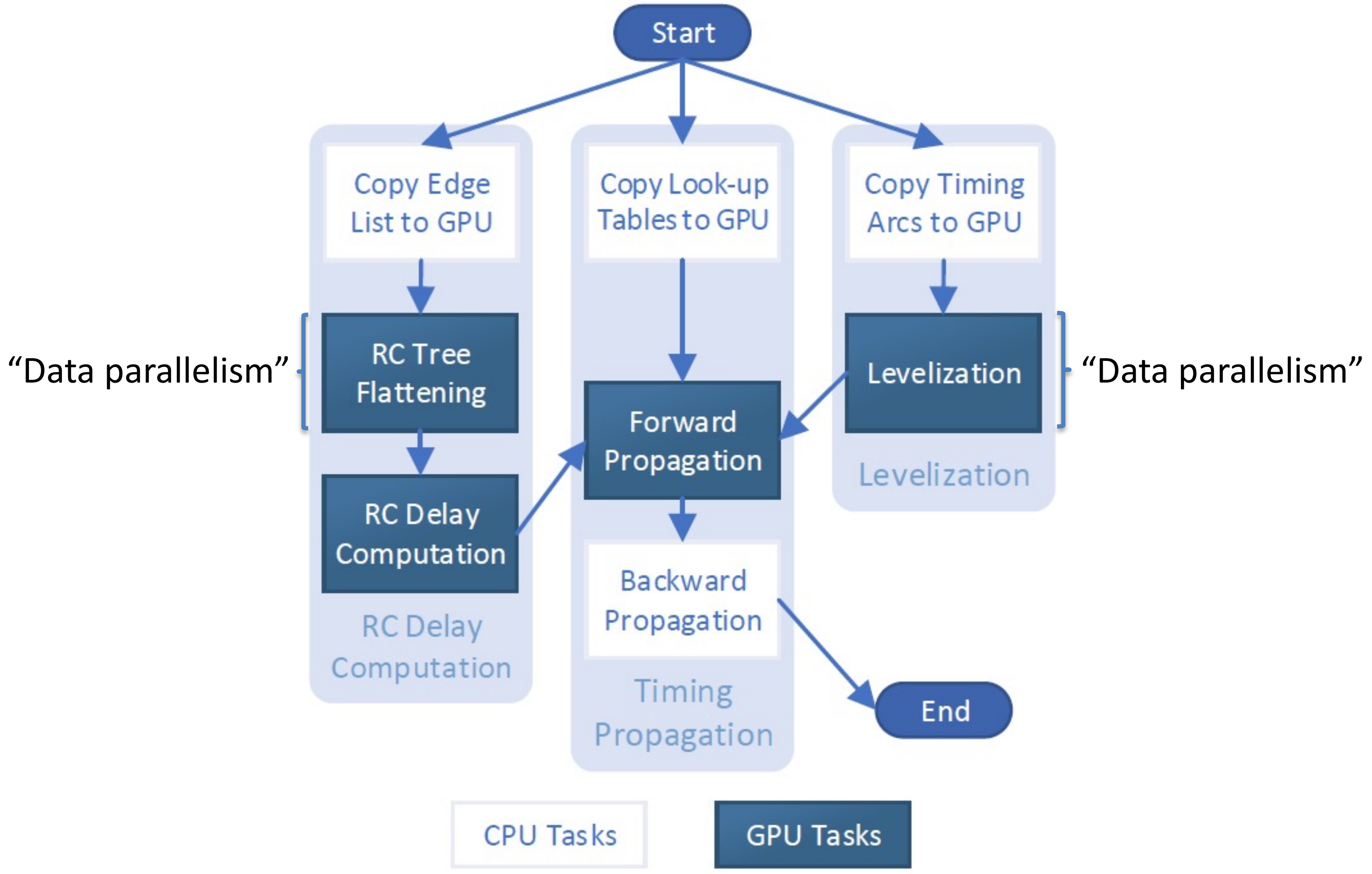


Leon2 GBA runtime breakdown



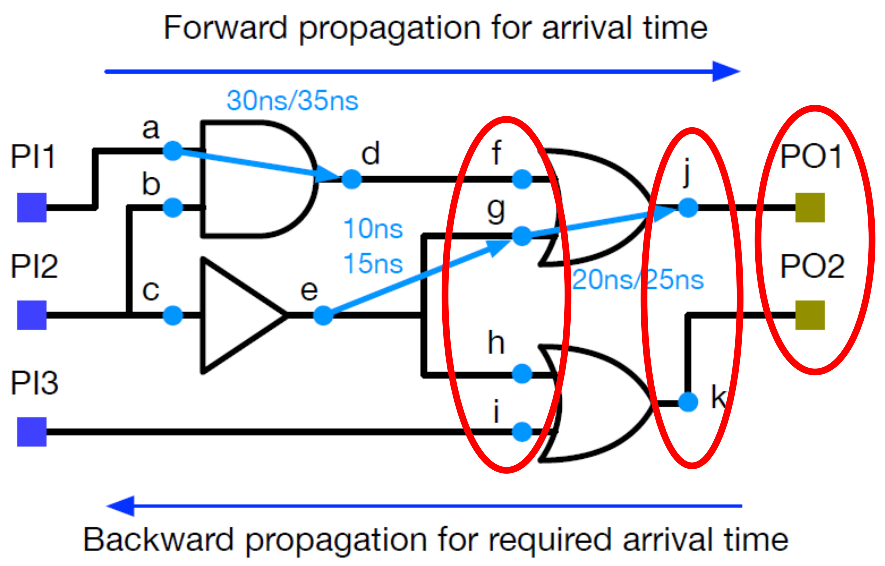
Leon2 circuit diagram and layout

GPU-Accelerated GBA Algorithm Flow

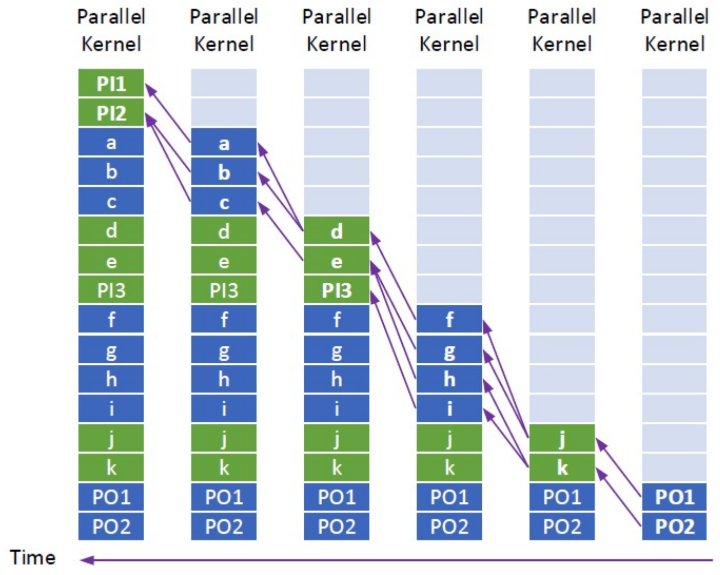


Step #1: Levelization

- ❑ **Levelize the circuit graph to a 2D levellist**
  - ❑ Nodes at the same level can run in parallel (red circle)
  - ❑ Nodes at the same level can be modeled as a batch



Forward propagation for arrival time
30ns/35ns
PI1
PI2
PI3
a
b
c
d
e
f
g
h
i
j
k
10ns
15ns
20ns/25ns
PO1
PO2
Backward propagation for required arrival time



- ❑ **GPU-accelerated levelization using parallel frontiers**

Step #1: Levelization (cont'd)

- Levelization kernels

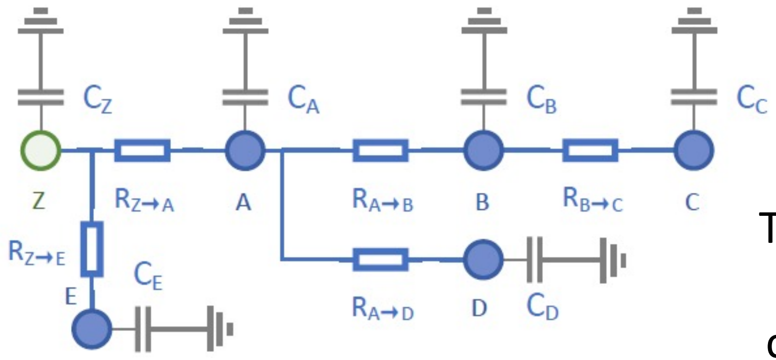
```
Input: the set of nodes $nodes$
Data: the adjacency list $out$, the current in-degree $in$
Output: a level list of nodes
1 $F \leftarrow \{f \in nodes : in_f = 0\}$;
2 while $F$ is not empty do
3     output $F$;
4     $F' \leftarrow \{\}$;
5     Call advanceFrontier on $F$ and get $F'$;
6     $F \leftarrow F'$;
7 end
```

Each GPU thread to expand pin level from *l* to *l*+1

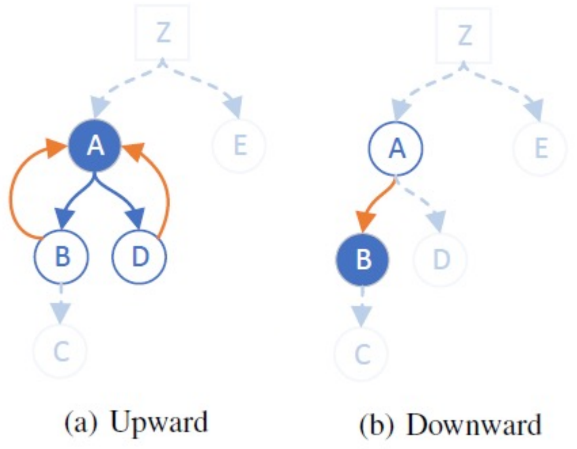
```
Input: the old frontier $F$
Data: the adjacency list $out$, in-degree array $in$
Output: the new frontier $F'$
1 $nodeID \leftarrow$ blockIdx.x $\times$ blockDim.x + threadIdx.x;
2 if $nodeID \geq$ size($F$) then return;
3 for $v$ in $out[nodeID]$ do
4     $oldvalue \leftarrow$ atomicAdd($in[v]$, -1);
5     if $oldvalue = 1$ then
6         Add $v$ to $F'$;
7     end
8 end
9 return $G$;
```

Step #2: RC Update

- The Elmore delay model
- Phase 1: $load_u = \sum_{v \text{ is child of } u} cap_v$
  - For example, $load_A = cap_A + cap_B + cap_C + cap_D = cap_A + load_B + load_D$
- Phase 2: $delay_u = \sum_{v \text{ is any node}} cap_v \times R_{Z \to LCA(u,v)}$
  - For example, $delay_B = cap_A R_{Z\to A} + cap_D R_{Z\to A} + cap_B R_{Z\to B} + cap_C R_{Z\to B} = delay_A + R_{A\to B} load_B$



Two-phase tree traversal to compute delay

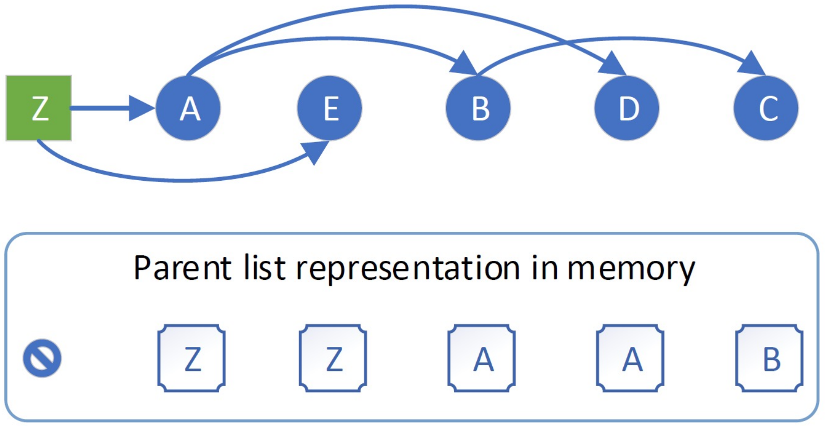


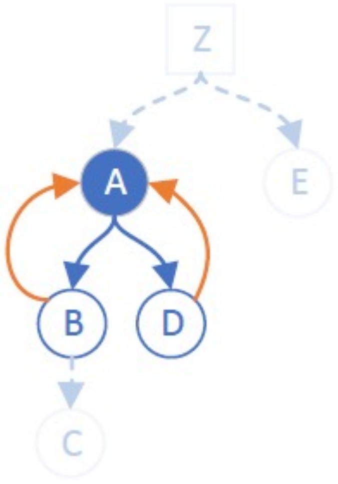
Step #2: RC Update Upward Phase

- Store the parent index of each node on GPU
- Perform dynamic programming on trees

```
DFS_load(u):
    load[u] = cap[u]
    For child v of u:
        DFS_load(v)
        load[u] += load[v]
```

```
GPU_load:
    For u in [C, D, B, E, A]:
        load[u] += cap[u]
        load[u.parent] += load[u]
```





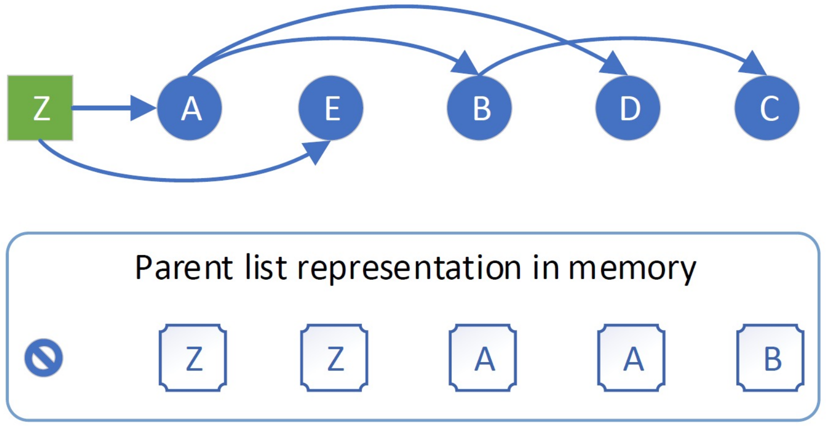
(a) Upward

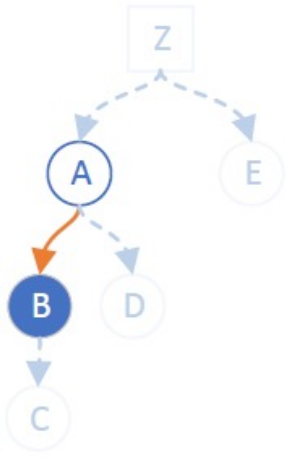
Step #2: RC Update Downward Phase

- Store the parent index of each node on GPU
- Perform dynamic programming on trees

```
DFS_delay(u):
    For child v of u:
        temp := R[u,v]*load[v]
        delay[v] = delay[u] + temp
        DFS_delay(v)
```

```
GPU_delay:
    For u in [A, E, B, D, C]:
        temp := R[u.parent,u]*load[u]
        delay[u]=delay[u.parent] + temp
```

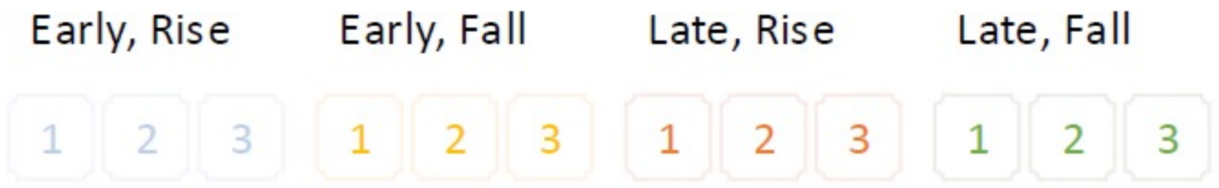




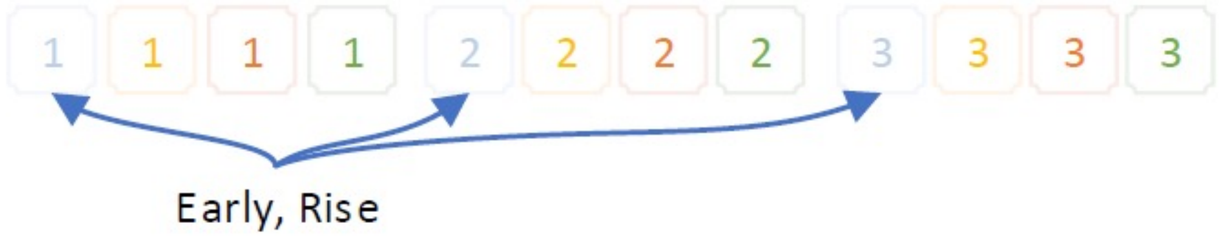
(b) Downward

Step #2: RC Update Memory Coalesce

- ❑ Consecutive threads access consecutive memory
- ❑ RC update has four cases: {Rise, Fall} x {Early, Late}



(a)



(b)

Step #2: RC Update Kernels (flatten + update)

```
Input: $N$ as #nets, $(M, E)$ as (#nodes, #edges) in all nets
Input: $roots[0..N-1]$, the index of root of each net
Input: $edges[0..E-1]$, the undirected edges $\{(a, b)\}$
Input: $nodestart[0..N]$, the offsets of each net in arrays of
        nodes, with $nodestart[N] = M$
Input: $edgestart[0..N]$, the offsets of each net in arrays of
        edges, with $edgestart[N] = E$
Input: $distances[0..M] = \infty$, $counts[0..M] = 0$
Output: $order[0..M-1]$, nodes in BFS order for each net
/* Process one net w/ blockDim.x threads                    */
$netID$ = blockIdx.x;                           ▷ gridDim.x = #nets
$threadID$ = threadIdx.x;                       ▷ blockDim.x = 64
$nst = nodestart[netID]$;                       ▷ node offset start
$nend = nodestart[netID + 1]$;                  ▷ node offset end
$est = edgestart[netID]$;                       ▷ edge offset start
$eend = edgestart[netID + 1]$;                  ▷ edge offset end
$distances[nst + roots[netID]] = 0$;
for $d = 0, 1, 2, ..., (nend - nst)$ do
    for $i = est + threadID$ to $eend$ step blockDim.x do
        $(a, b) = edgelist[i]$;
        if $distances[a] == d$ and $distances[b] > d + 1$ then
            $distances[b] = d + 1$;
            atomicAdd($counts[d]$, 1);
        end
        else if $distances[b] == d$ and $distances[a] > d + 1$ then
            $distances[a] = d + 1$;
            atomicAdd($counts[d]$, 1);
        end
    end
    __syncthreads();                      ▷ Sync threads within a block
    break when $counts[d] == 0$;
end
countingSort($distances, counts, order, threadID$);
```

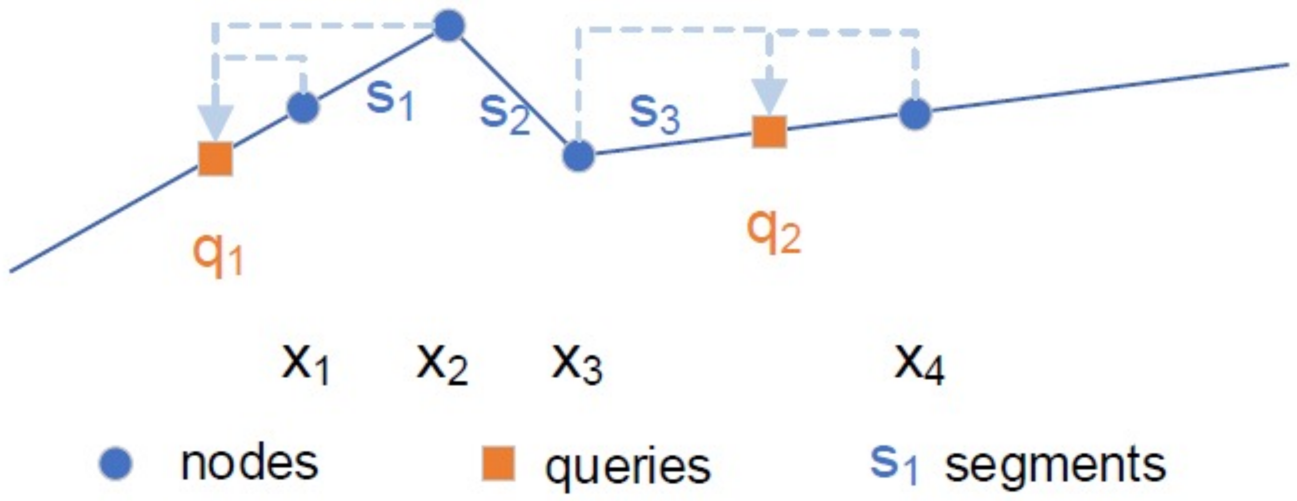
Flatten the data to array for "**data parallelism**"

```
Input: $N$ as #nets, $M$ as #nodes in all nets
Input: $start[0..N]$, the offsets of each net in arrays of nodes
Input: $parent[0..M-1]$, the index of parent of every nodes
Input: $pres[0..M-1]$, the resistance between nodes and their
        parent
Input: $cap[0..4M-1]$, the capacitance of nodes, each in 4
        different combinations
Output: $load[0..4M-1]$, $delay[0..4M-1]$, $impulse[0..4M-1]$:
         arrays of results of load, delay and impulse,
         respectively
$netID$ = blockIdx.x × blockDim.x + threadIdx.x;
$condID$ = threadIdx.y;
if $netID \geq N$ then return;
$offsetL = start[netID]$;                       ▷ node offset start
$offsetR = start[netID + 1]$;                   ▷ node offset end
Initialize $load$, $delay$, $ldelay$ to zero;
Initialize $\beta = 0$ as an auxiliary array;
for $i = offsetR - 1$ down to $offsetL$ do
    $load[4i + condID]$ += $cap[4i + condID]$;
    $load[4parent[i] + condID]$ += $load[4i + condID]$;
end
for $i = offsetL + 1$ to $offsetR - 1$ do
    $t = load[4i + condID] \times pres[i]$;
    $delay[4i + condID] = delay[4parent[i] + condID] + t$;
end
for $i = offsetR - 1$ down to $offsetL$ do
    $ldelay[4i + condID]$ +=
      $cap[4i + condID] \times delay[4i + condID]$;
    $ldelay[4parent[i] + condID]$ += $load[4i + condID]$;
end
for $i = offsetL + 1$ to $offsetR - 1$ do
    $t' = ldelay[4i + condID] \times pres[i]$;
    $\beta[4i + condID] = \beta[4parent[i] + condID] + t'$;
    $impulse[4i + condID] =$
      $2\beta[4i + condID] - delay[4i + condID]^2$;
end
```

Each thread computes one RC tree

Step #3: Cell Delay Update

- **Perform linear inter- and extra-polation in batches**
  - x-axis (input slew) and then y-axis (output load)



Step #3: Cell Delay Update Kernels

```
/* Input: line $(x_1, y_1)$--$(x_2, y_2)$            */
/* Input: the $x$ value queried                      */
Function interpolate($x_1, x_2, y_1, y_2, x$):
    if $x_1 = x_2$ then return $y_1$;
    else return $d_1 + (d_2 - d_1)\frac{x - x_1}{x_2 - x_1}$;
end
/* Input: $n \times m$ look-up table                 */
/* Input: the point queried $(x, y)$                 */
Function lut_lookup($n, m, X, Y, mat, x, y$):
    $i' \leftarrow 0$;
    $i \leftarrow \min(1, n - 1)$;
    while $i + 1 < n$ and $X[i] \leq x$ do
        $i' \leftarrow i$;
        $i \leftarrow i + 1$;
    end
    $j' \leftarrow 0$;
    $j \leftarrow \min(1, m - 1)$;
    while $j + 1 < m$ and $Y[j] \leq y$ do
        $j' \leftarrow j$;
        $j \leftarrow j + 1$;
    end
    $r_{i'} \leftarrow$ interpolate($Y[j'], Y[j], mat[i', j'], mat[i', j]$);
    $r_i \leftarrow$ interpolate($Y[j'], Y[j], mat[i, j'], mat[i, j]$);
    $r \leftarrow$ interpolate($X[i'], X[i], r_{i'}, r_i$);
    return $r$;
end
```

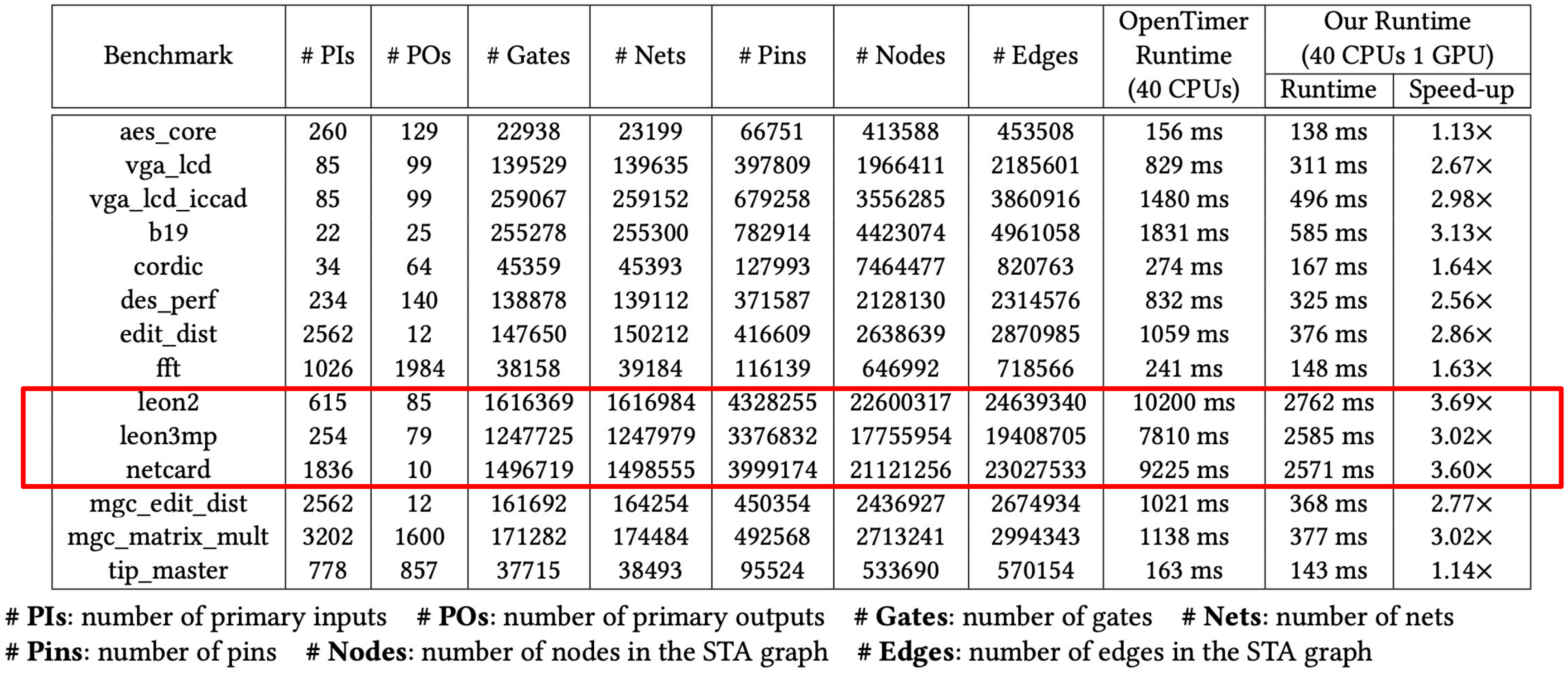
Each thread performs binary search on x-axis and y-axis to find the slew and capacitance

Experimental Setting

- ❑ **Machine configuration**
  - ❑ Nvidia CUDA, RTX 2080
  - ❑ 40 Intel Xeon Gold 6138 CPU cores
- ❑ **Execution parameters for GPU kernels**
  - ❑ RC Tree Flattening
    - 64 threads per block with one block for each net
  - ❑ Levelization
    - 128 threads per block
  - ❑ RC delay computation
    - 4 threads for each net (one for each Early/Late and Rise/Fall condition) with a block of 64 nets
  - ❑ Cell delay computation
    - 4 threads for each arc, with a block of 32 arcs

Overall Performance

- ❑ **Comparison with OpenTimer of 40 CPUs**
  - ❑ Run on large TAU15 Benchmarks (>20K gates)



Benchmark	# PIs	# POs	# Gates	# Nets	# Pins	# Nodes	# Edges	OpenTimer Runtime (40 CPUs)	Our Runtime (40 CPUs 1 GPU)	
									Runtime	Speed-up
aes_core	260	129	22938	23199	66751	413588	453508	156 ms	138 ms	1.13×
vga_lcd	85	99	139529	139635	397809	1966411	2185601	829 ms	311 ms	2.67×
vga_lcd_iccad	85	99	259067	259152	679258	3556285	3860916	1480 ms	496 ms	2.98×
b19	22	25	255278	255300	782914	4423074	4961058	1831 ms	585 ms	3.13×
cordic	34	64	45359	45393	127993	7464477	820763	274 ms	167 ms	1.64×
des_perf	234	140	138878	139112	371587	2128130	2314576	832 ms	325 ms	2.56×
edit_dist	2562	12	147650	150212	416609	2638639	2870985	1059 ms	376 ms	2.86×
fft	1026	1984	38158	39184	116139	646992	718566	241 ms	148 ms	1.63×
leon2	615	85	1616369	1616984	4328255	22600317	24639340	10200 ms	2762 ms	3.69×
leon3mp	254	79	1247725	1247979	3376832	17755954	19408705	7810 ms	2585 ms	3.02×
netcard	1836	10	1496719	1498555	3999174	21121256	23027533	9225 ms	2571 ms	3.60×
mgc_edit_dist	2562	12	161692	164254	450354	2436927	2674934	1021 ms	368 ms	2.77×
mgc_matrix_mult	3202	1600	171282	174484	492568	2713241	2994343	1138 ms	377 ms	3.02×
tip_master	778	857	37715	38493	95524	533690	570154	163 ms	143 ms	1.14×

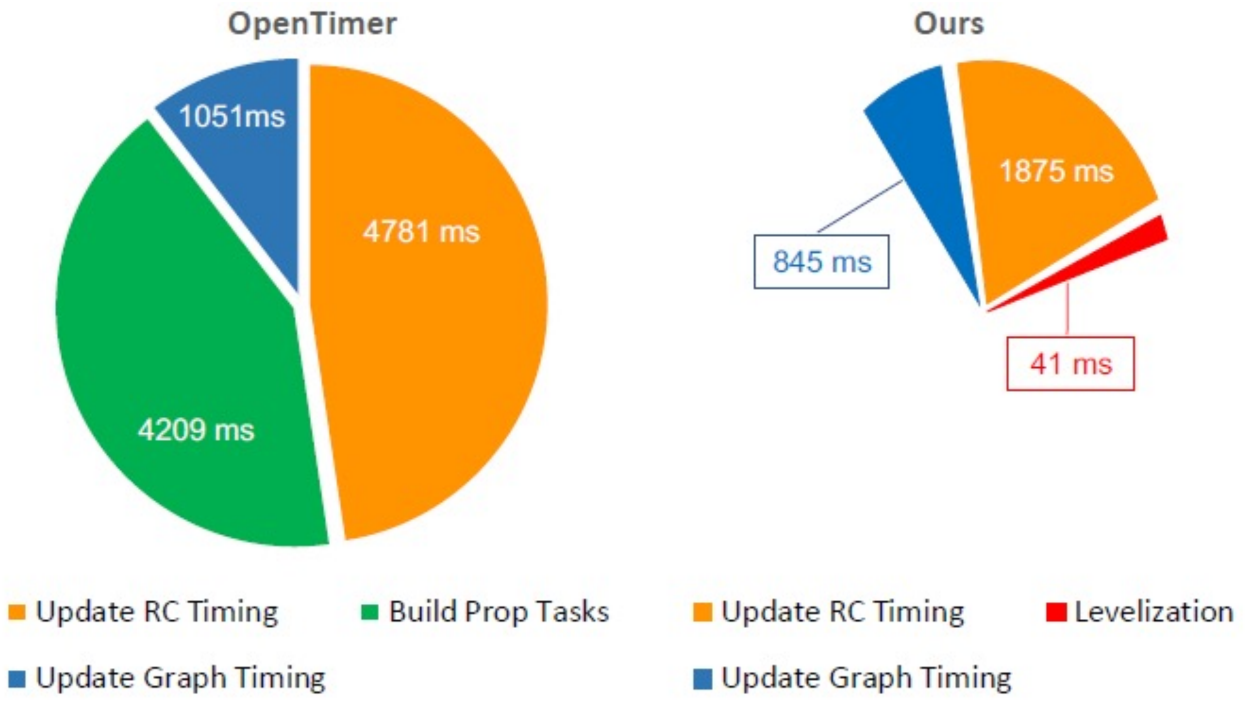
# PIs: number of primary inputs **# POs**: number of primary outputs **# Gates**: number of gates **# Nets**: number of nets
# Pins: number of pins **# Nodes**: number of nodes in the STA graph **# Edges**: number of edges in the STA graph

TAU15 Contest on Incremental Timing Analysis: https://sites.google.com/site/taucontest2015

T.-W. Huang et al., "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," IEEE TCAD21

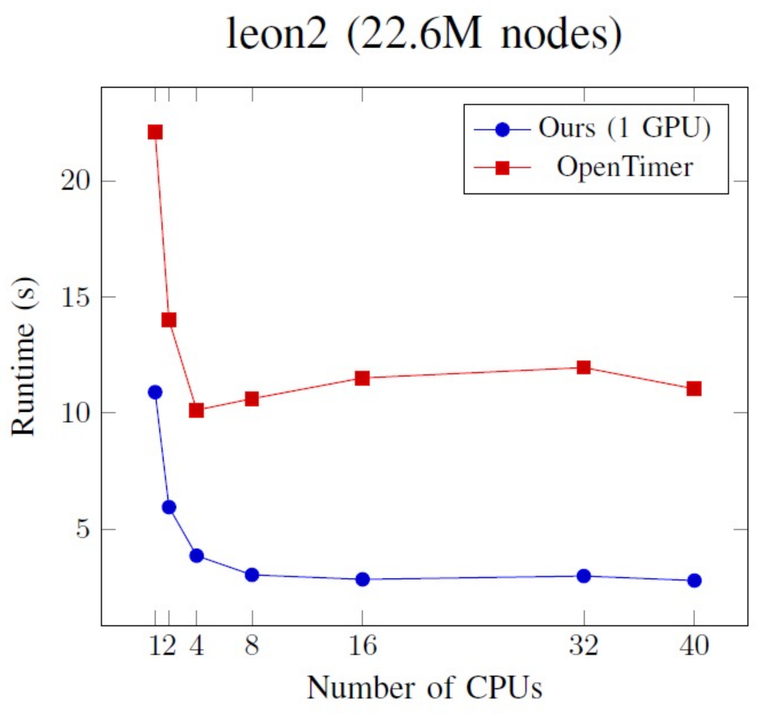
Runtime Breakdown

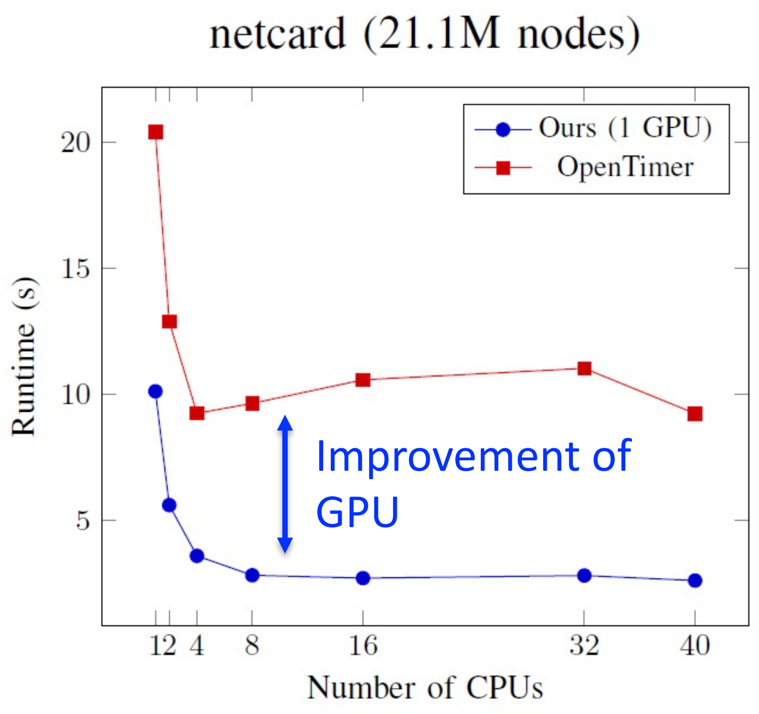
- Circuit leon2 (21 M nodes)



Runtime vs CPUs

❑ **Significant performance gap between CPU and GPU**

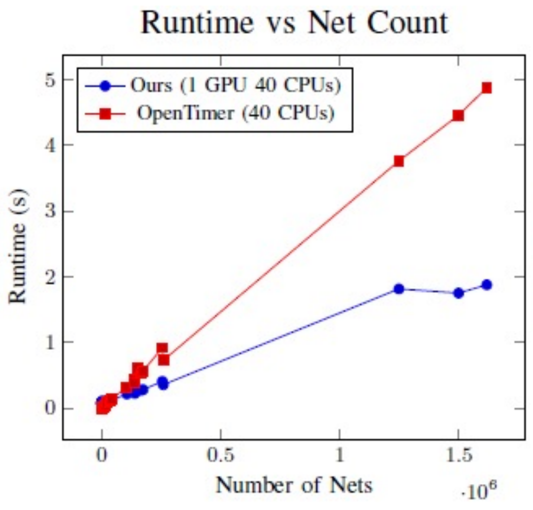


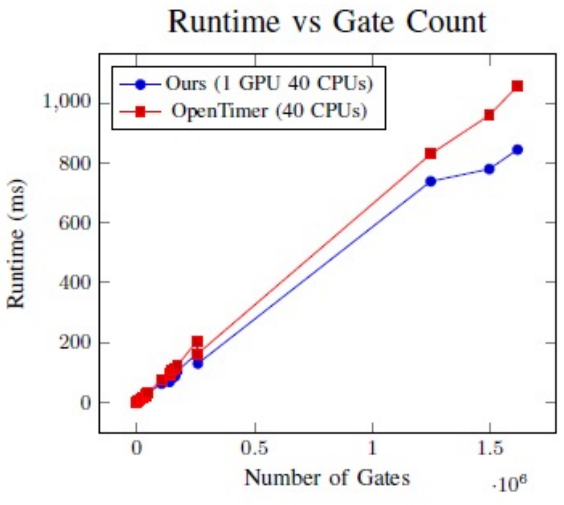


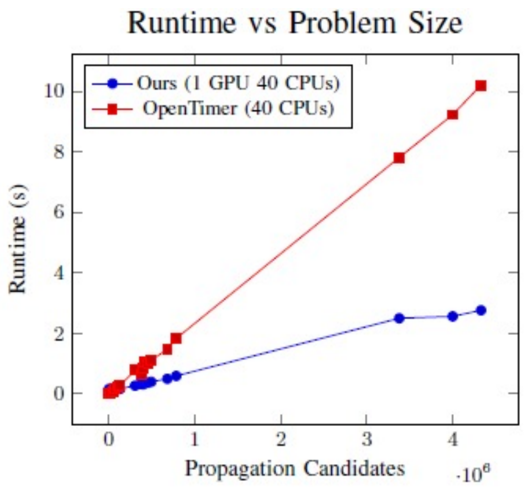
Our runtime of 1 CPU and 1 GPU is very close to OpenTimer of 40 CPUs

Runtime vs Problem Sizes

- ❑ **Problem size matters for GPU acceleration**
- ❑ **When to enable GPU acceleration?**
  - ❑ Net count > 20K
  - ❑ Gate count > 50K
  - ❑ Propagation candidate count > 15K







Agenda

- ❑ **Introduce the scalability problem of STA**
  - ❑ What is STA and its challenges?
  - ❑ Why do we need new parallel paradigm for STA?
- ❑ **Accelerate graph-based analysis using GPU**
- ❑ **Accelerate path-based analysis using GPU**

G Guo, T-W Huang, Y Lin, and M Wong, "GPU-Accelerated Path-based Timing Analysis," *IEEE/ACM Design Automation Conference (DAC)*, 2021

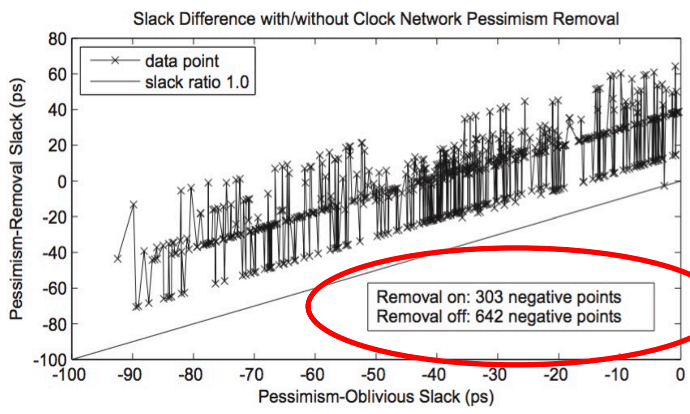
Research Question:

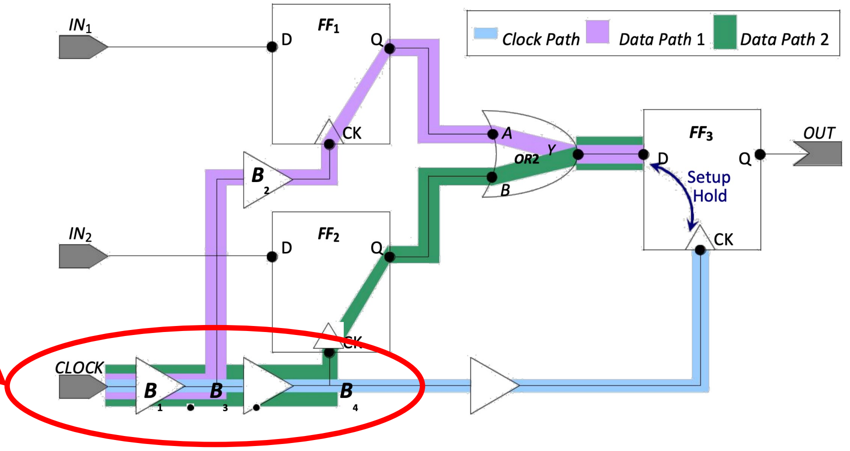
- ***How can we use GPU to speed up path-based analysis (PBA)?***
- ***How do we overcome the challenges of generating large numbers of critical paths to analyze with GPU?***

Path-based Analysis (PBA)

- ❑ **Identify a set of critical paths from a updated graph**
  - ❑ Exponential number of paths in the circuit graph
- ❑ **Re-analyze each path with path-specific update**
  - ❑ Re-propagate the slew and remove pessimism
  - ❑ Advanced on-chip variation (AOCV)
  - ❑ Common path pessimism removal (CPPR)
  - ❑ …

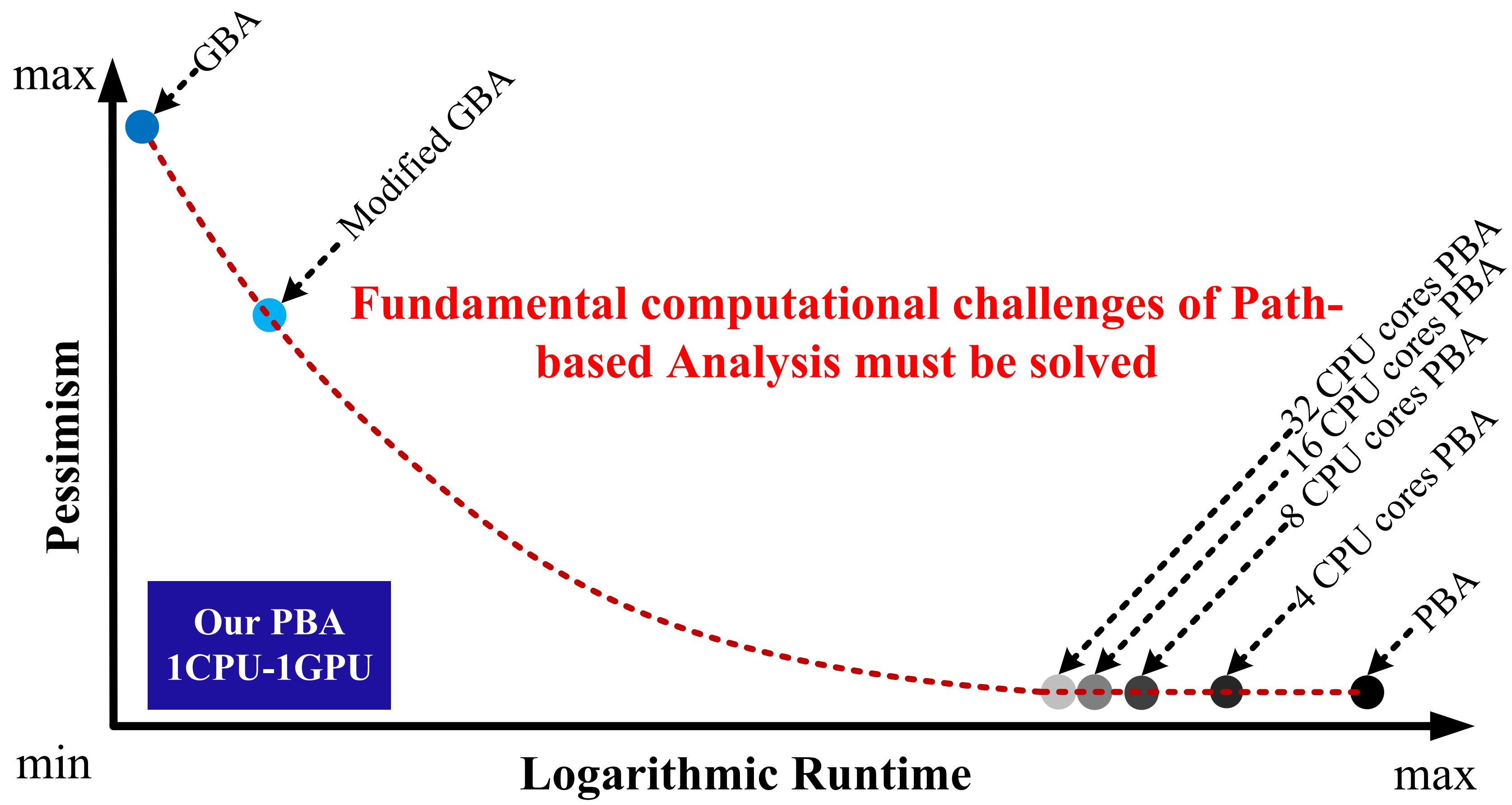
Paths marked failing at GBA may become passing after PBA!





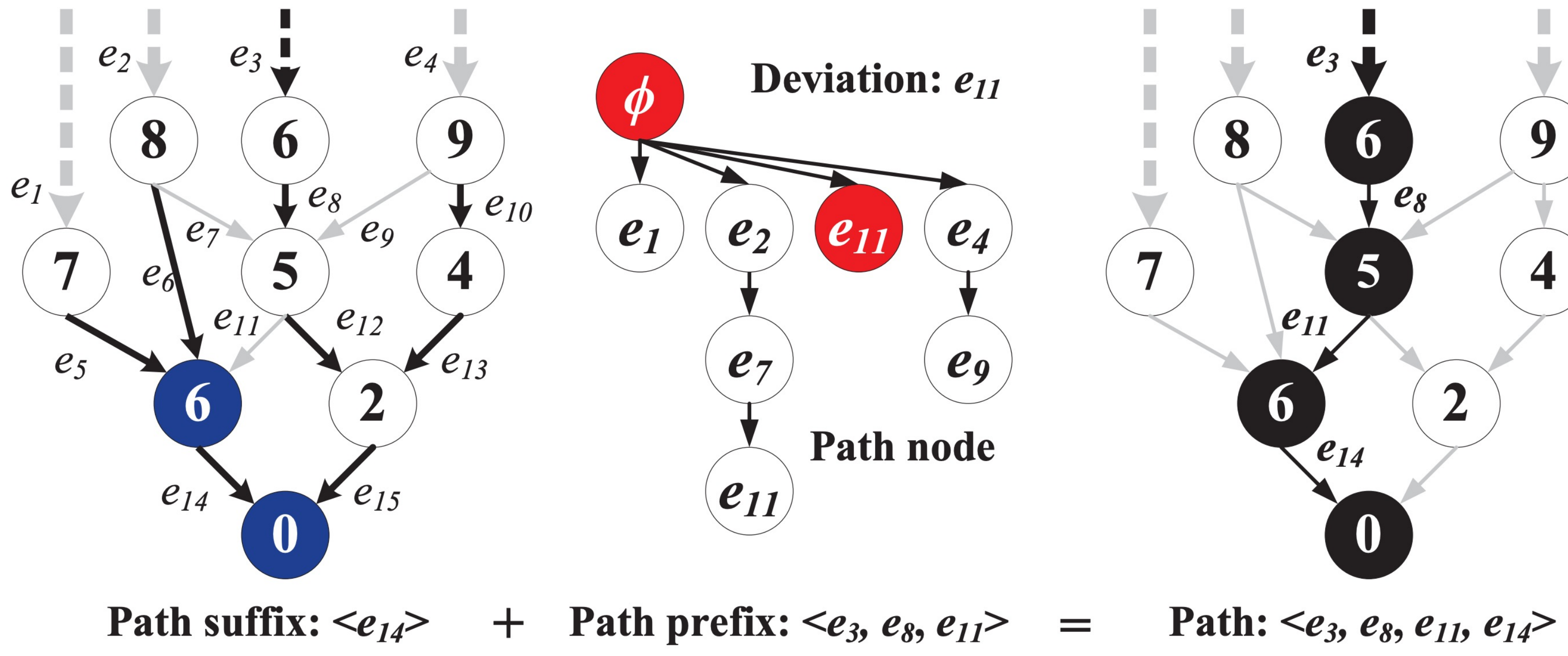
PBA is Extremely Time-Consuming

- Speed vs Accuracy (pessimism removal) tradeoff



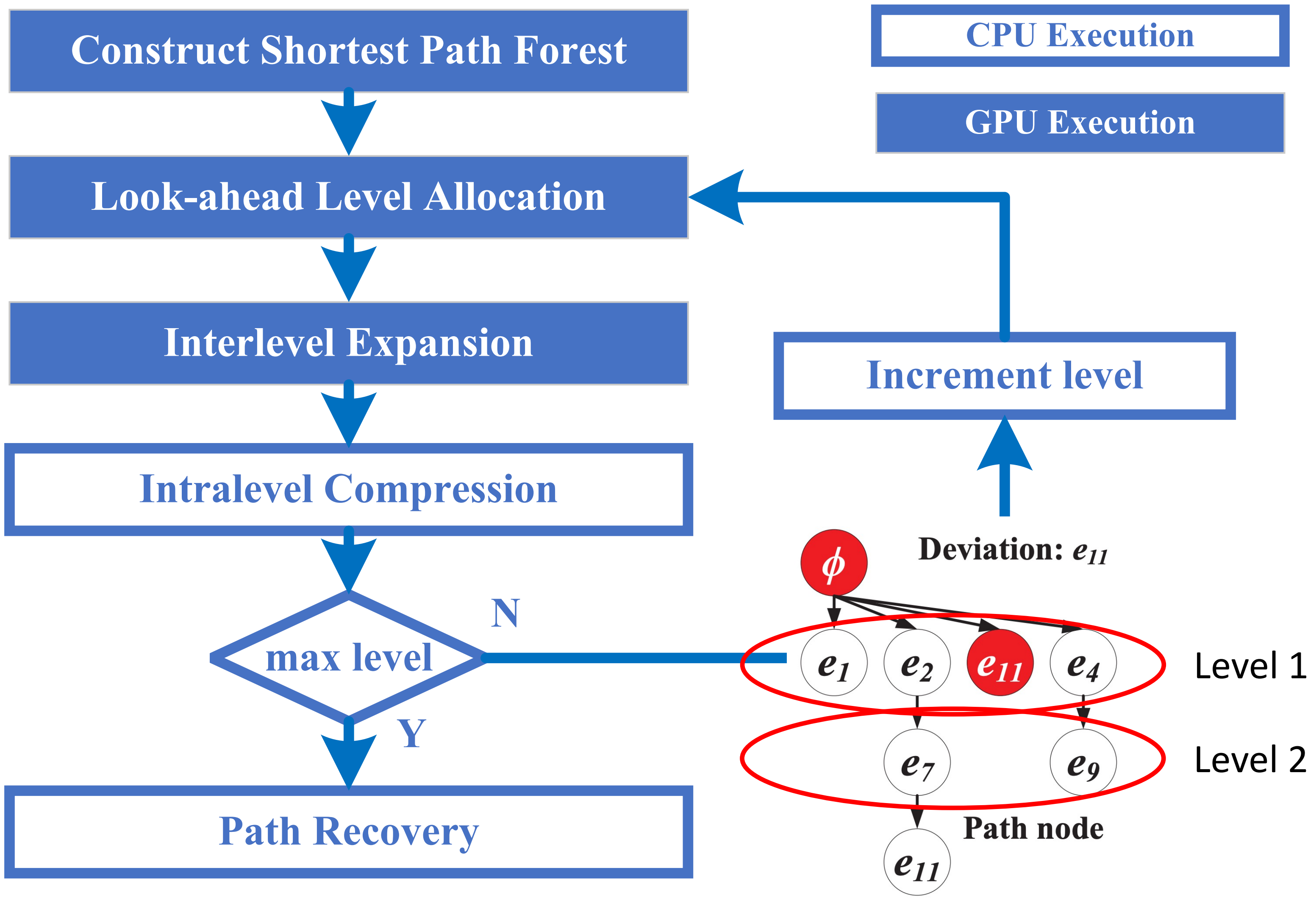
A Key Step: Generate Critical Paths

- ❑ **We introduce implicit path representation**
  - ❑ Each path is represented using <u>*O(1)* space</u> and time
  - ❑ Each path is ranked through a *prefix* tree & a *suffix* tree

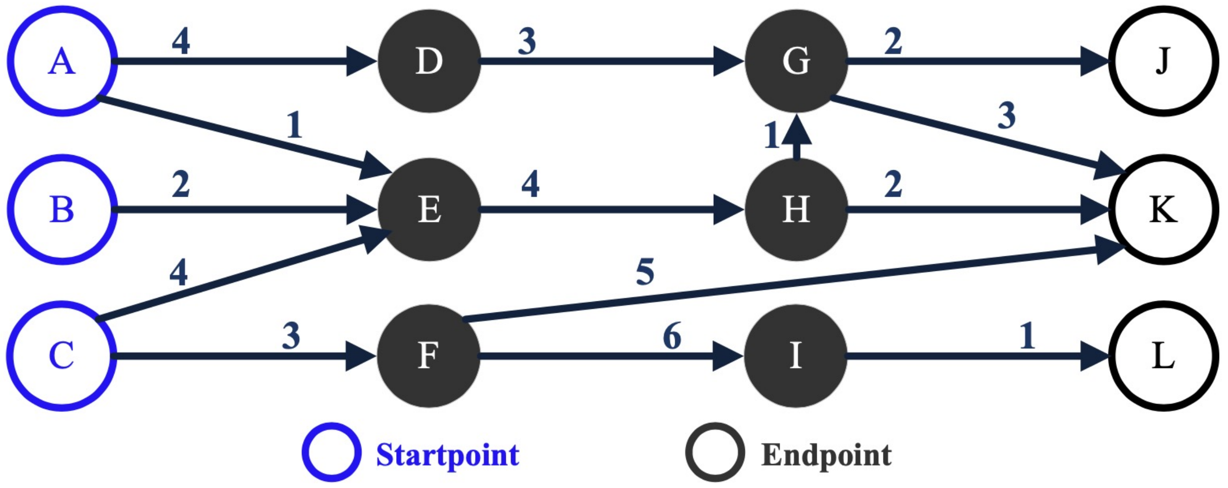


Deviation: e_{11}
ϕ
Path node
Path suffix: $<e_{14}>$ + Path prefix: $<e_3, e_8, e_{11}>$ = Path: $<e_3, e_8, e_{11}, e_{14}>$

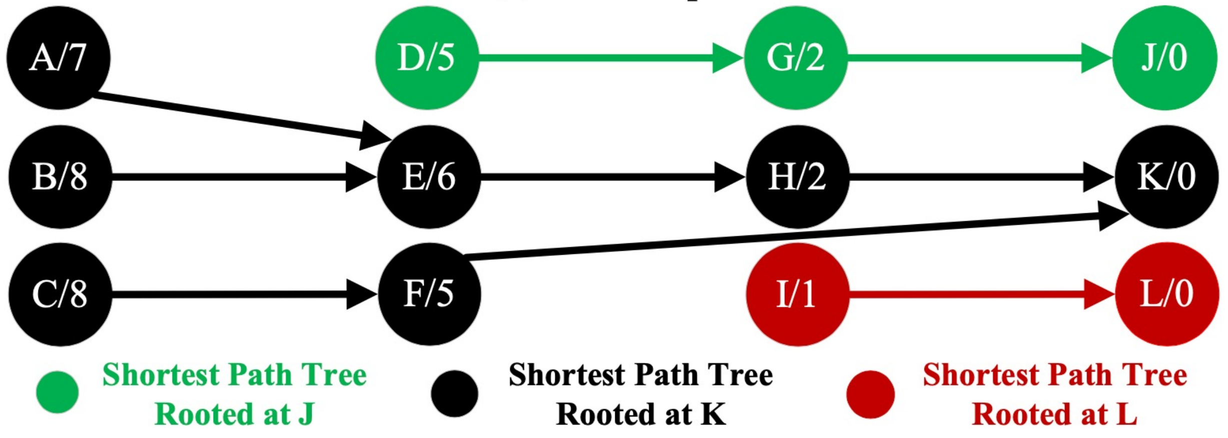
GPU-Accelerated PBA Algorithm Flow



Step #1: Generate Suffix Tree on GPU

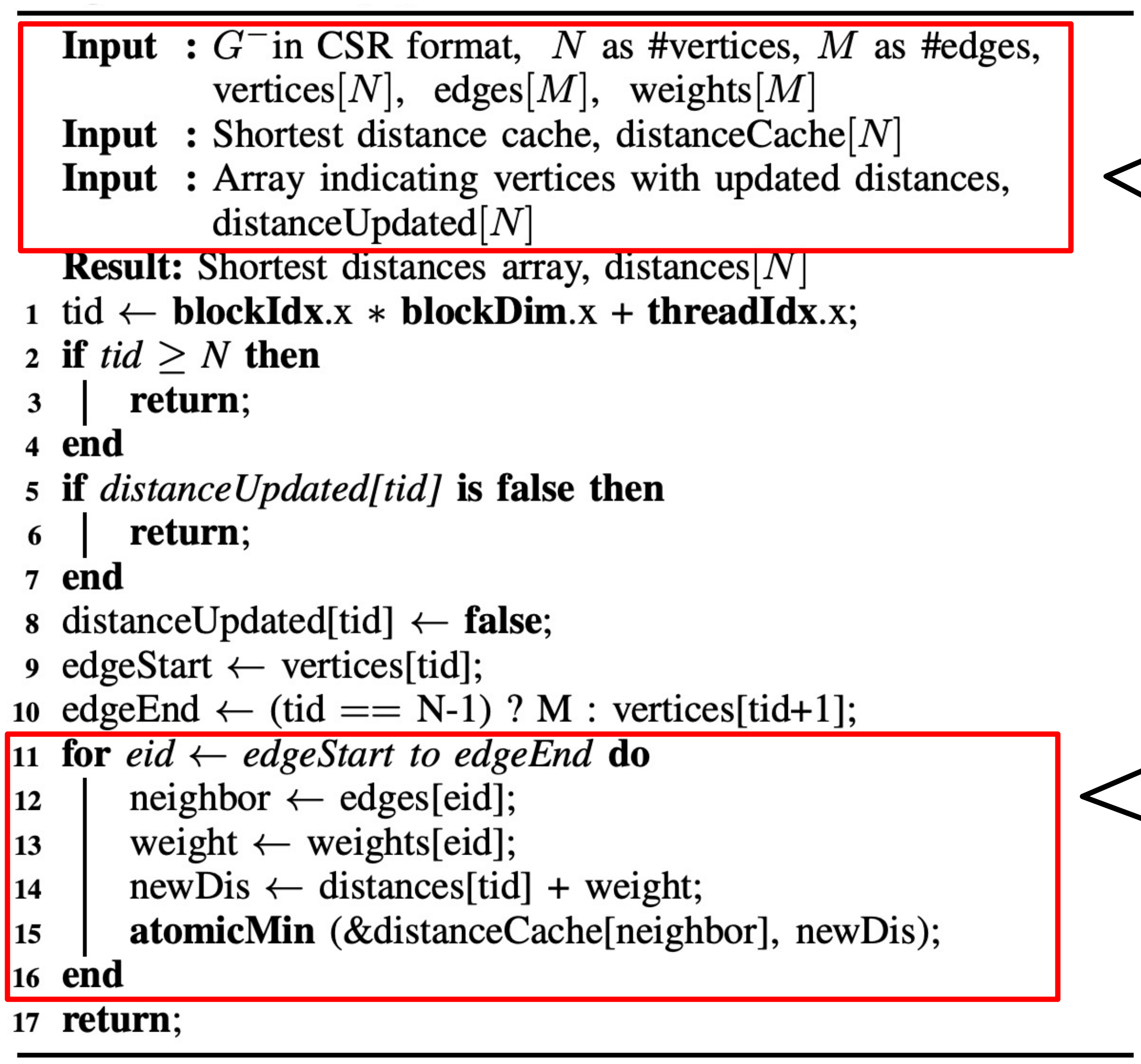


(a) STA Graph.


A/7
B/8
C/8
D/5
E/6
F/5
G/2
H/2
I/1
J/0
K/0
L/0
Shortest Path Tree Rooted at J
Shortest Path Tree Rooted at K
Shortest Path Tree Rooted at L

(b) Shortest path forest.

Step #1: Generate Suffix Tree Kernel

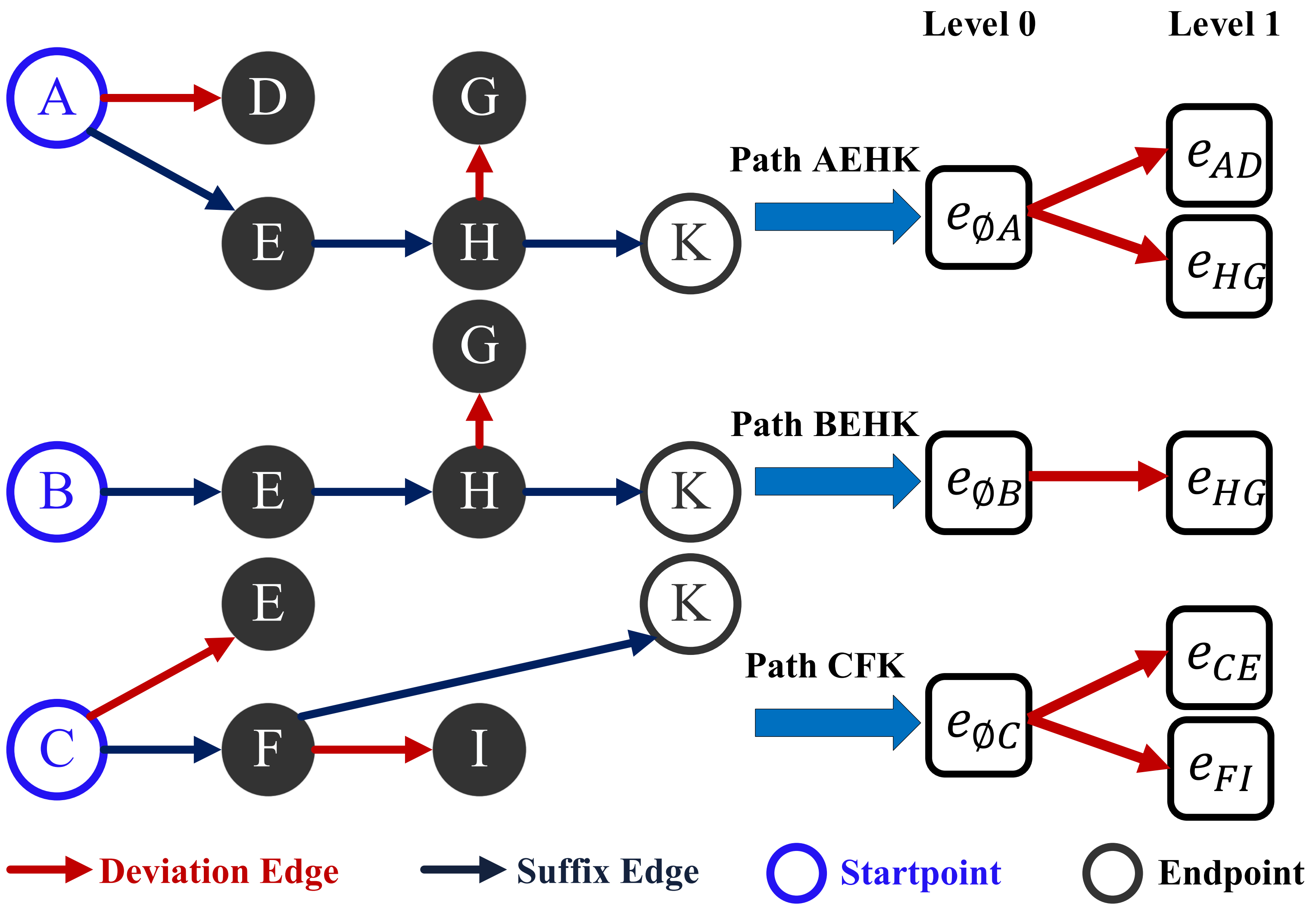


```
Input  : $G^-$ in CSR format, $N$ as #vertices, $M$ as #edges,
         vertices[$N$], edges[$M$], weights[$M$]
Input  : Shortest distance cache, distanceCache[$N$]
Input  : Array indicating vertices with updated distances,
         distanceUpdated[$N$]
Result: Shortest distances array, distances[$N$]
tid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
if tid ≥ N then
    return;
end
if distanceUpdated[tid] is false then
    return;
end
distanceUpdated[tid] ← false;
edgeStart ← vertices[tid];
edgeEnd ← (tid == N-1) ? M : vertices[tid+1];
for eid ← edgeStart to edgeEnd do
    neighbor ← edges[eid];
    weight ← weights[eid];
    newDis ← distances[tid] + weight;
    atomicMin (&distanceCache[neighbor], newDis);
end
return;
```

Flattened data structure to benefit data parallelism

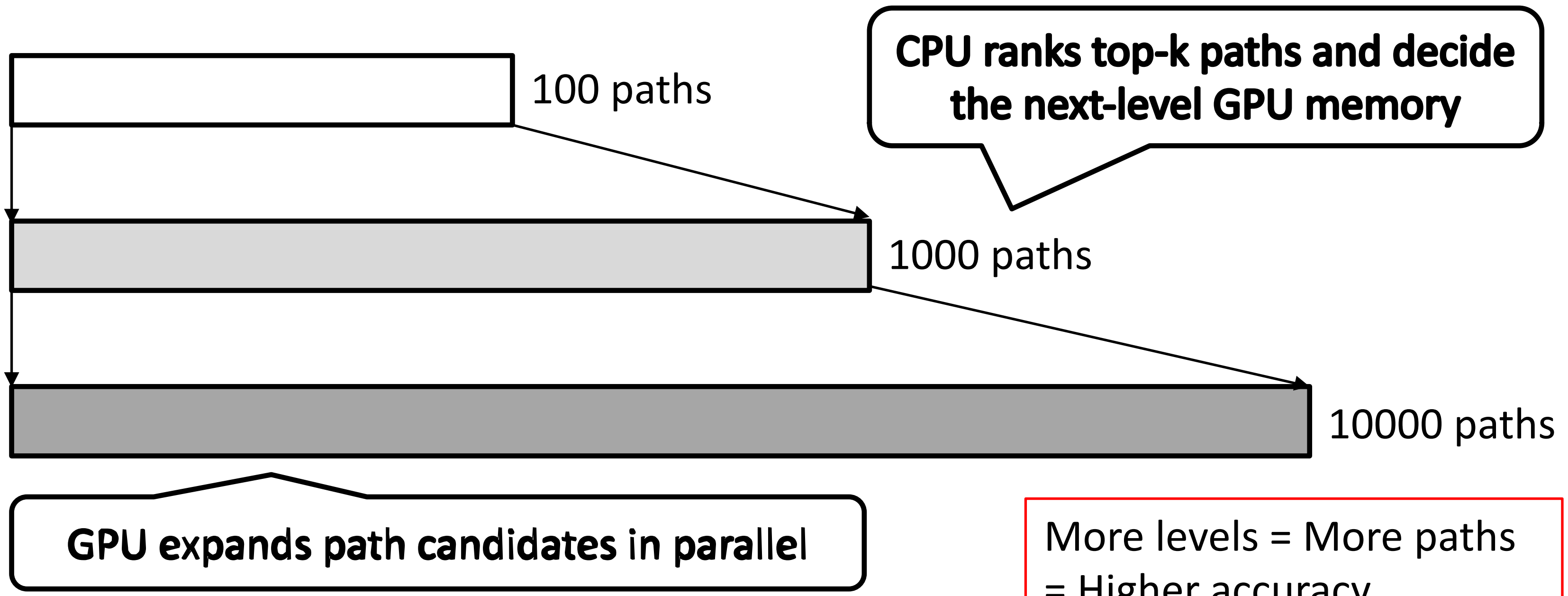
Bellman loop to find the shortest path tree (each thread per edge)

Step #2: Expand Prefix Tree on GPU



Step #2: Expand Prefix Tree on GPU (cont'd)

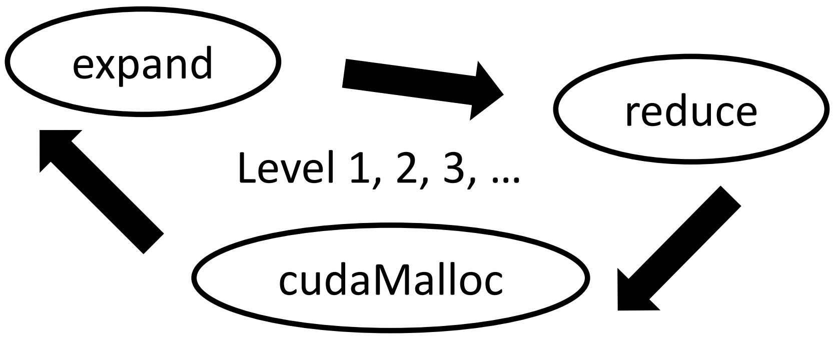
- ❑ **Iteratively grow GPU memory at each expansion**
  - ❑ Each iteration uses GPU to decide path candidates
  - ❑ Each iteration uses CPU to prune path candidates
  - ❑ Each path candidate takes O(1) space "deviation edge"


100 paths
CPU ranks top-k paths and decide the next-level GPU memory
1000 paths
10000 paths
GPU expands path candidates in parallel

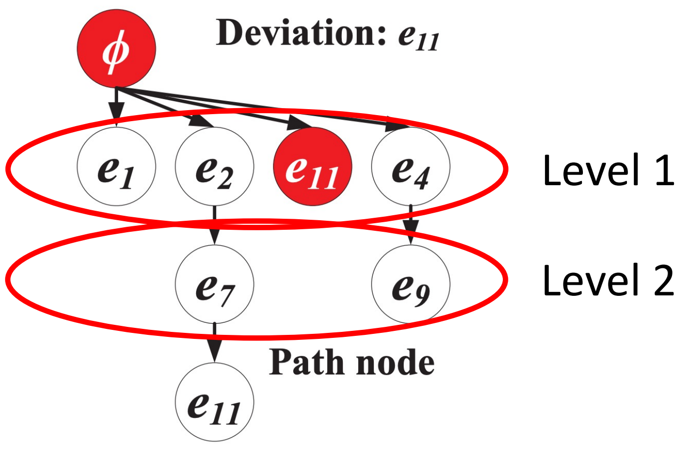
More levels = More paths = Higher accuracy

Step #2: Expand Prefix Tree Kernel

```
Input  : $G^+$ in CSR format, $N$ as #vertices, $M$ as #edges,
         vertices[$N$], edges[$M$], weights[$M$]
Input  : Shortest path forest, forest[$N$] as edge array,
         distances[$N$] as distance array
Input  : currLevel as current level
Input  : levelSize as the number of entries in current level
Result: Explore critical path candidates in next level
tid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
if tid ≥ levelSize then
    return;
end
offset ← (tid == 0) ? 0 : currLevel[tid-1].childOffset;
level ← currLevel[tid].level;
slack ← currLevel[tid].slack;
v ← currLevel[tid].to;
while v is not endpoint do
    edgeStart ← vertices[v];
    edgeEnd ← (v == N-1) ? M : vertices[v+1];
    for eid ← edgeStart to edgeEnd do
        neighbor ← edges[eid];
        weight ← weights[eid];
        if eid is deviation edge then
            /* Fill out child path data      */
            newPath ← nextLevel[offset];
            newPath.level ← level+1;
            newPath.from ← v;
            newPath.to ← neighbor;
            newPath.parent ← tid;
            newPath.childOffset ← 0;
            newPath.slack ← slack + distances[neighbor] +
              weight - distances[v];
            offset ← offset + 1;
        end
    end
    /* Traverse along the shortest path
       forest                            */
    v = forest[v];
end
return;
```



Key idea: levelized prefix tree

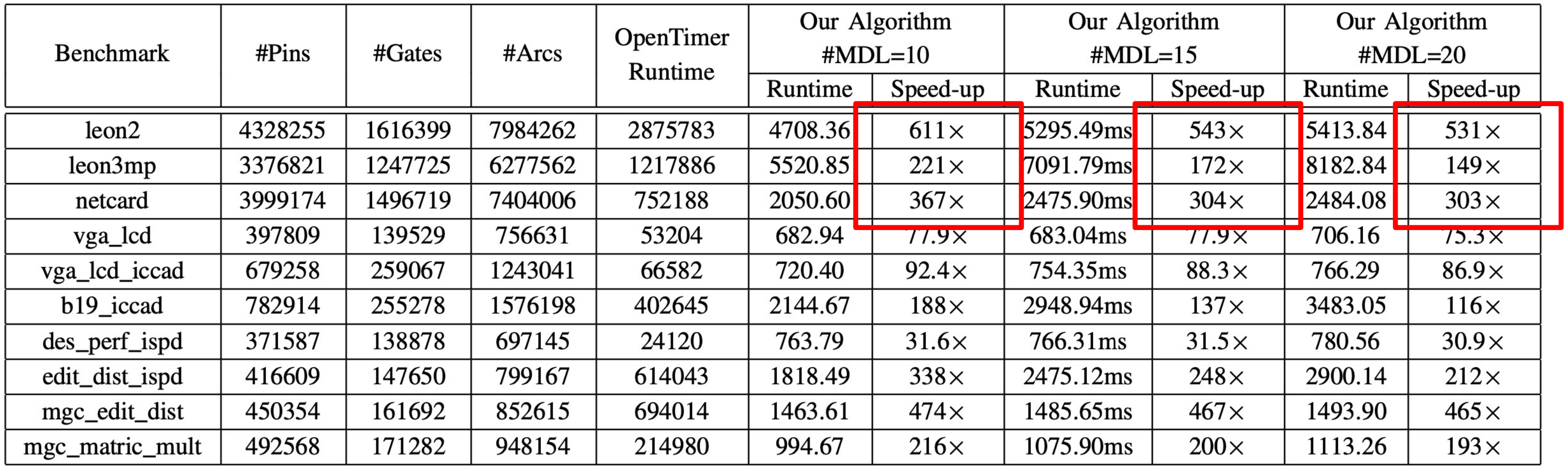


Experimental Setting

- **Machine configuration**
  - Nvidia CUDA, RTX 2080
  - 40 Intel Xeon Gold 6138 CPU cores
- **Measure the accuracy-runtime tradeoff**
  - “MDL” stands for maximum deviation level
- **Execution parameters for GPU kernels**
  - Suffix tree kernel
    - 1024 threads per block
  - Prefix tree kernel
    - 1024 threads per block

Overall Performance

- ❑ **Compare with OpenTimer's CPU-based PBA**
  - ❑ Report speed-up at different MDLs

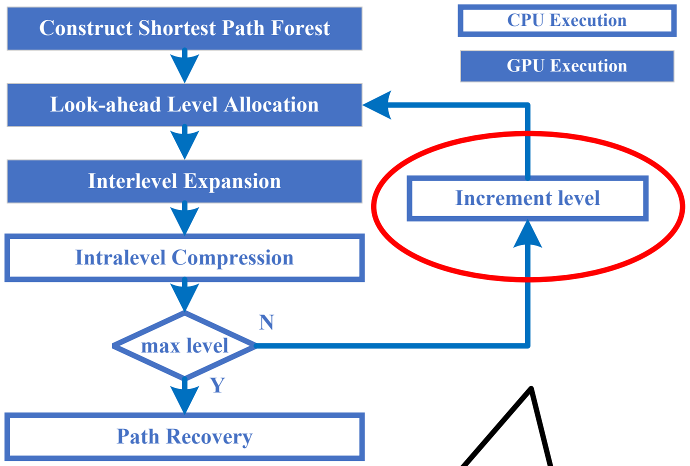


Benchmark	#Pins	#Gates	#Arcs	OpenTimer Runtime	Our Algorithm #MDL=10		Our Algorithm #MDL=15		Our Algorithm #MDL=20	
					Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	7984262	2875783	4708.36	611×	5295.49ms	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×	7091.79ms	172×	8182.84	149×
netcard	3999174	1496719	7404006	752188	2050.60	367×	2475.90ms	304×	2484.08	303×
vga_lcd	397809	139529	756631	53204	682.94	77.9×	683.04ms	77.9×	706.16	75.3×
vga_lcd_iccad	679258	259067	1243041	66582	720.40	92.4×	754.35ms	88.3×	766.29	86.9×
b19_iccad	782914	255278	1576198	402645	2144.67	188×	2948.94ms	137×	3483.05	116×
des_perf_ispd	371587	138878	697145	24120	763.79	31.6×	766.31ms	31.5×	780.56	30.9×
edit_dist_ispd	416609	147650	799167	614043	1818.49	338×	2475.12ms	248×	2900.14	212×
mgc_edit_dist	450354	161692	852615	694014	1463.61	474×	1485.65ms	467×	1493.90	465×
mgc_matric_mult	492568	171282	948154	214980	994.67	216×	1075.90ms	200×	1113.26	193×

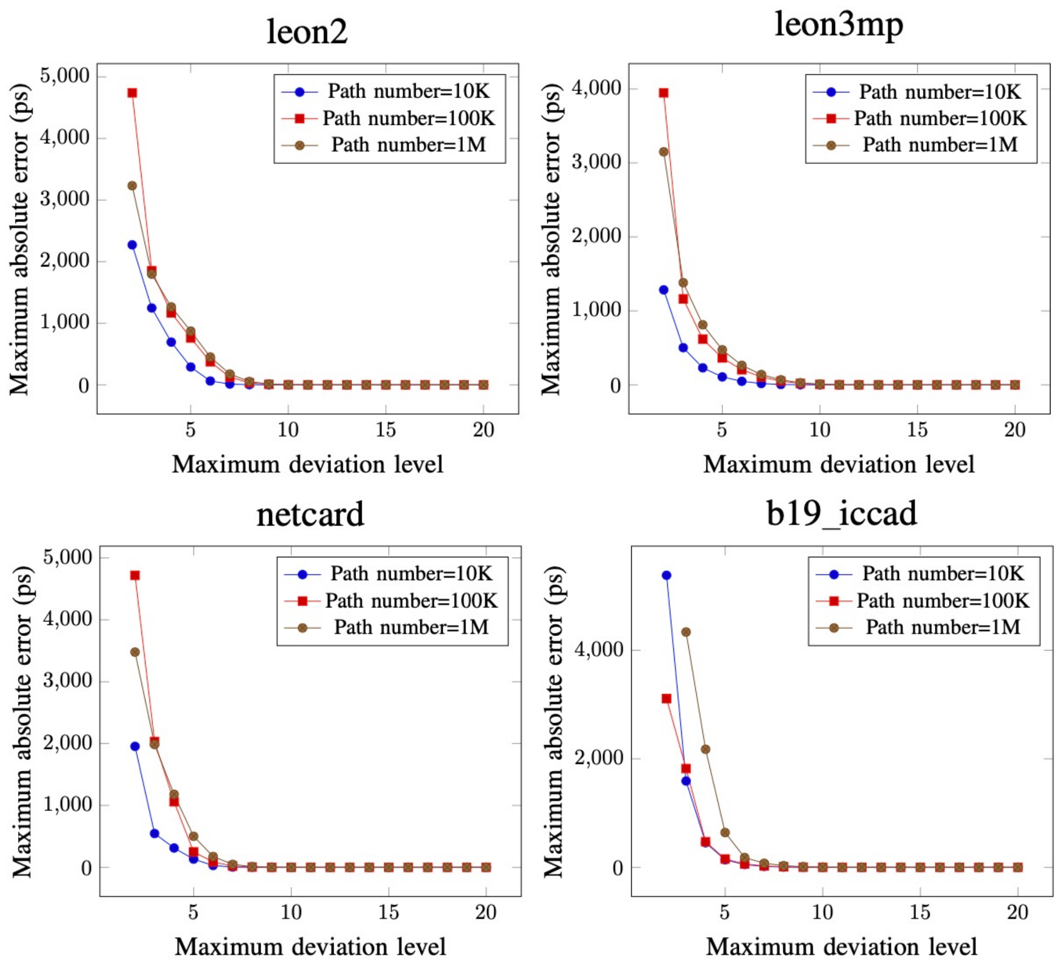
- ❑ **Achieve significant speed-up at large designs**
  - ❑ 611x speed-up in leon2 (1.3M gates)
  - ❑ 221x speed-up in leon3mp (1.2M gates)

Path Accuracy vs MDL

- ❑ Achieve decent accuracy at 10—12 GPU iterations



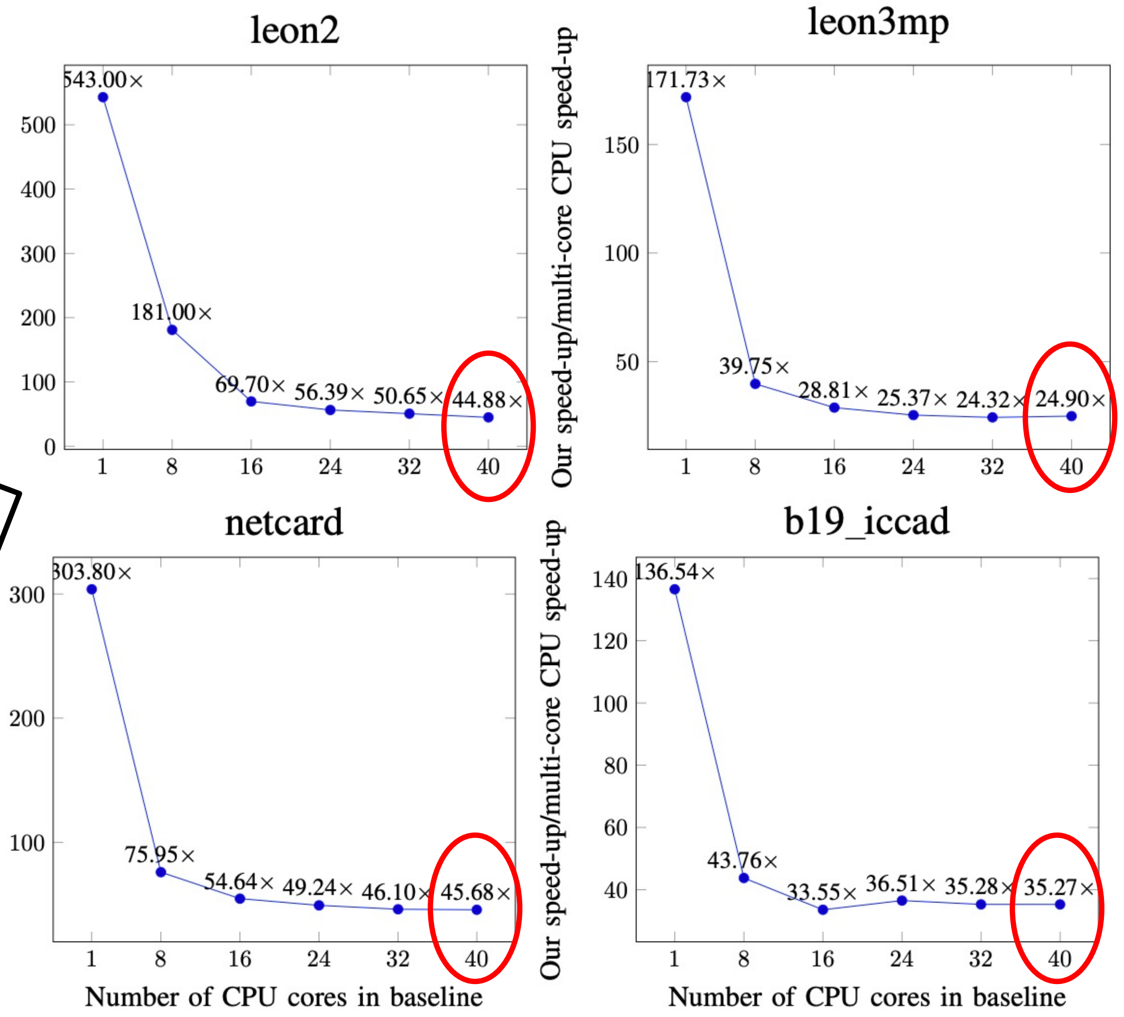
More GPU expansions (iterations) lead to higher numbers of paths and thus better accuracy



GPU Speed-up vs CPUs

- one GPU is even faster than OpenTimer with 40 CPUs
  - 44x on leon2
  - 25x on leon3mp
  - 46x on netcard
  - 35x on b19

In fact, according to our experiments, our GPU-accelerated PBA is always faster than OpenTimer's CPU baseline regardless of the core count



Conclusion

- **Introduced the runtime challenges of STA**
  - Introduced graph-based analysis
  - Introduced path-based analysis
- **Accelerated the graph-based analysis using GPU**
  - Achieved 4x speed-up over 40 CPUs on large designs
- **Accelerated the path-based analysis using GPU**
  - Achieved 600x speed-up over 40 CPUs on large designs
- **Future work**
  - Consider other important STA tasks (CPPR, CCS, etc.)
  - Leverage cudaMallocAsync to boost memory efficiency
  - Leverage cudaGraph to reduce kernel launch overheads
  - Collaborate with experts in Nvidia!

Thank You

Use right algorithms for GPU!

tsung-wei.huang@utah.edu