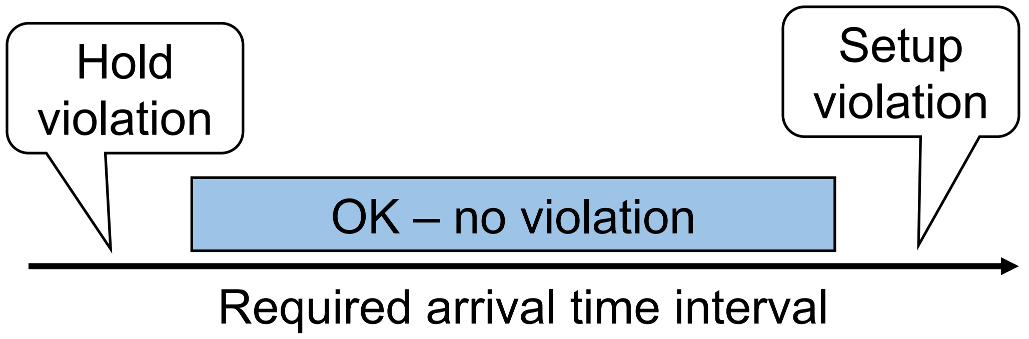
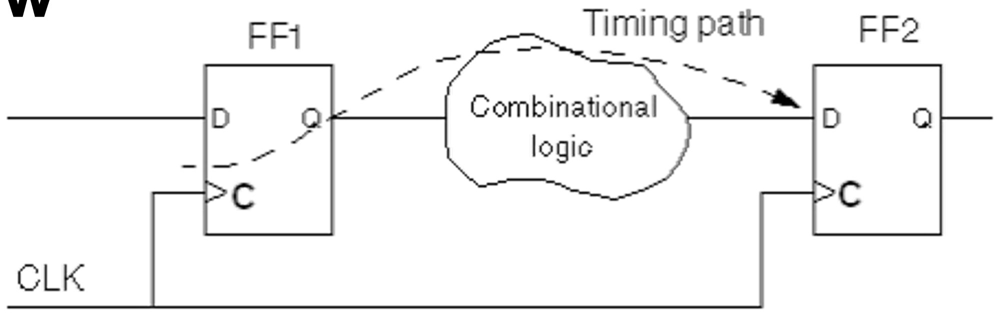
Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System

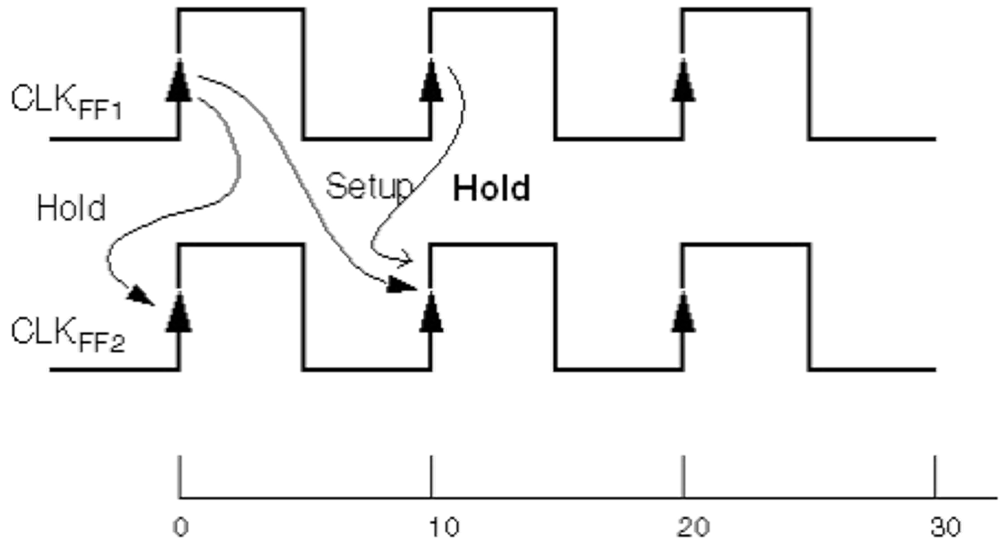
Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://tsung-wei-huang.github.io/

Static Timing Analysis (STA)

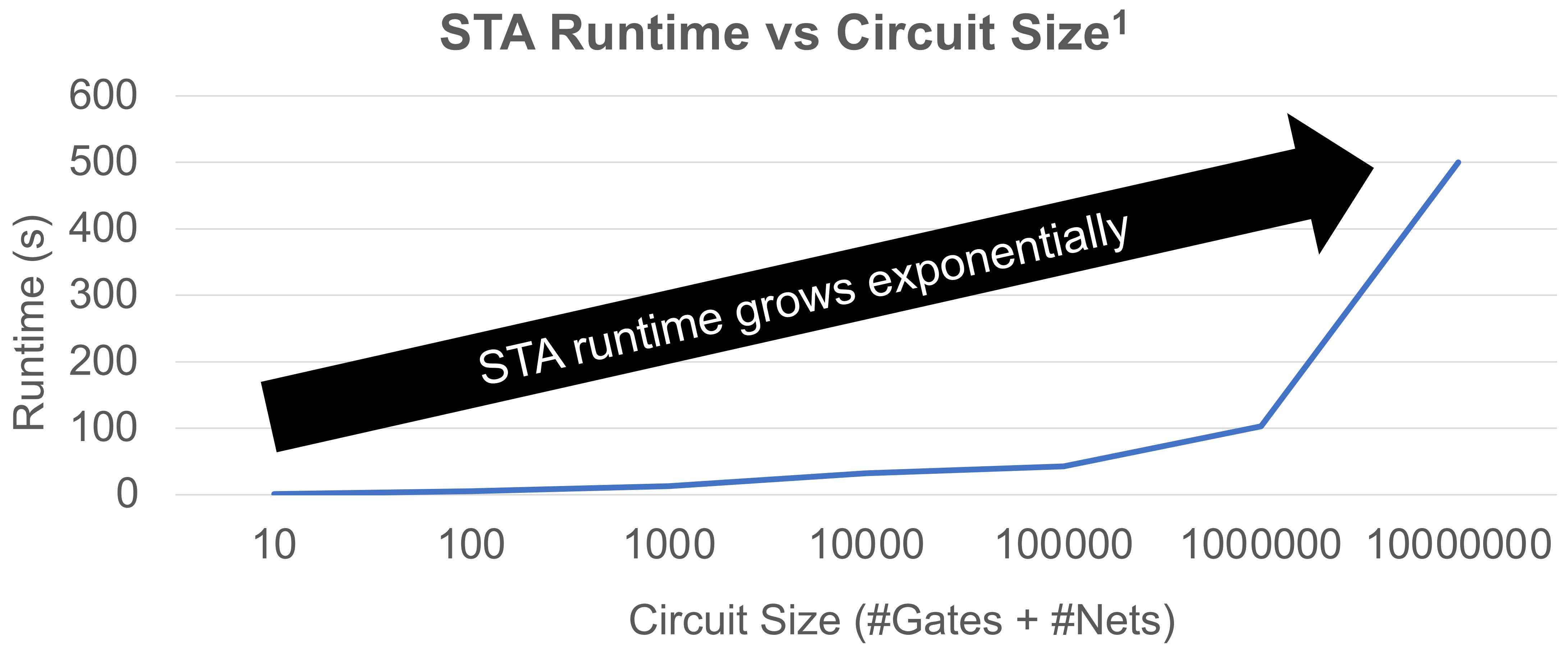
- **A key step in the overall design flow**
  - Verify the timing behavior of a circuit
  - Make sure signals can arrive on time
- **Analyze the worst-case scenario**
  - Before clock: *setup* time constraint
    - Latest required arrival time
  - After clock: *hold* time constraint
    - Earliest required arrival time







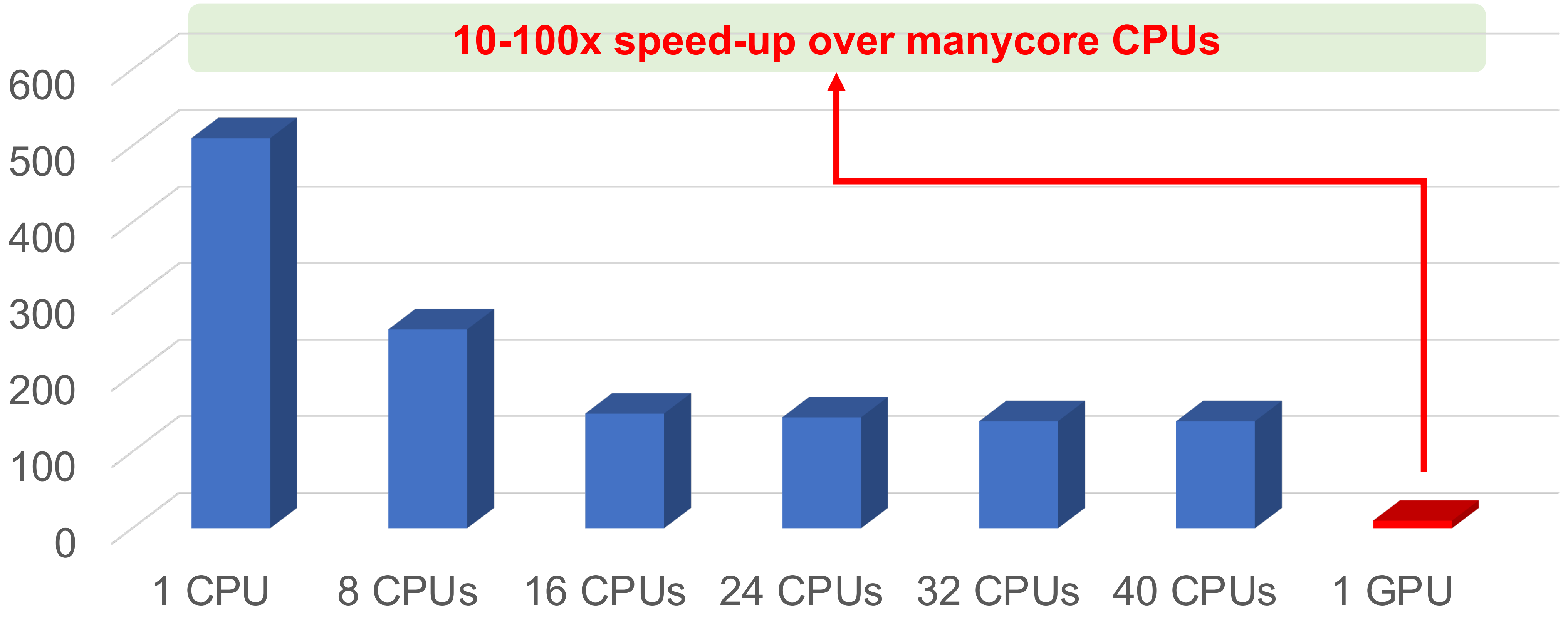
Challenge: Modern Circuits are Very Big



1: Tsung-Wei Huang, et al, “OpenTimer v2: A New Parallel Incremental Timing Analysis Engine,” *IEEE TCAD*, 2022

Why Do We Need to Parallelize STA?

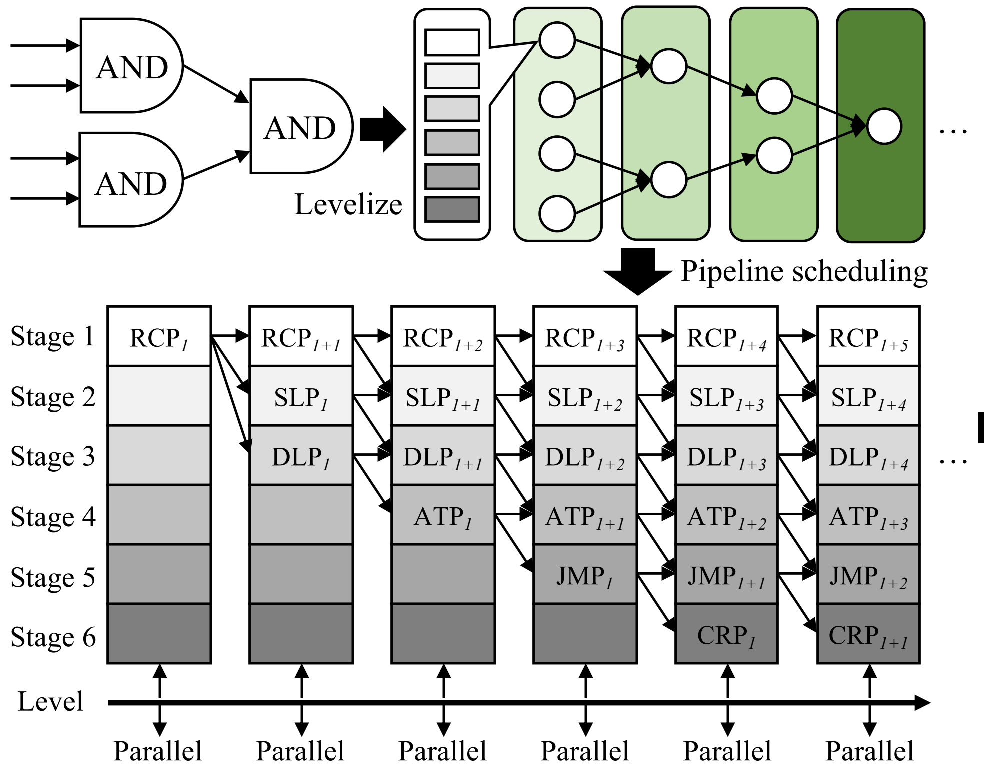
- **Advances the runtime performance to a new level**

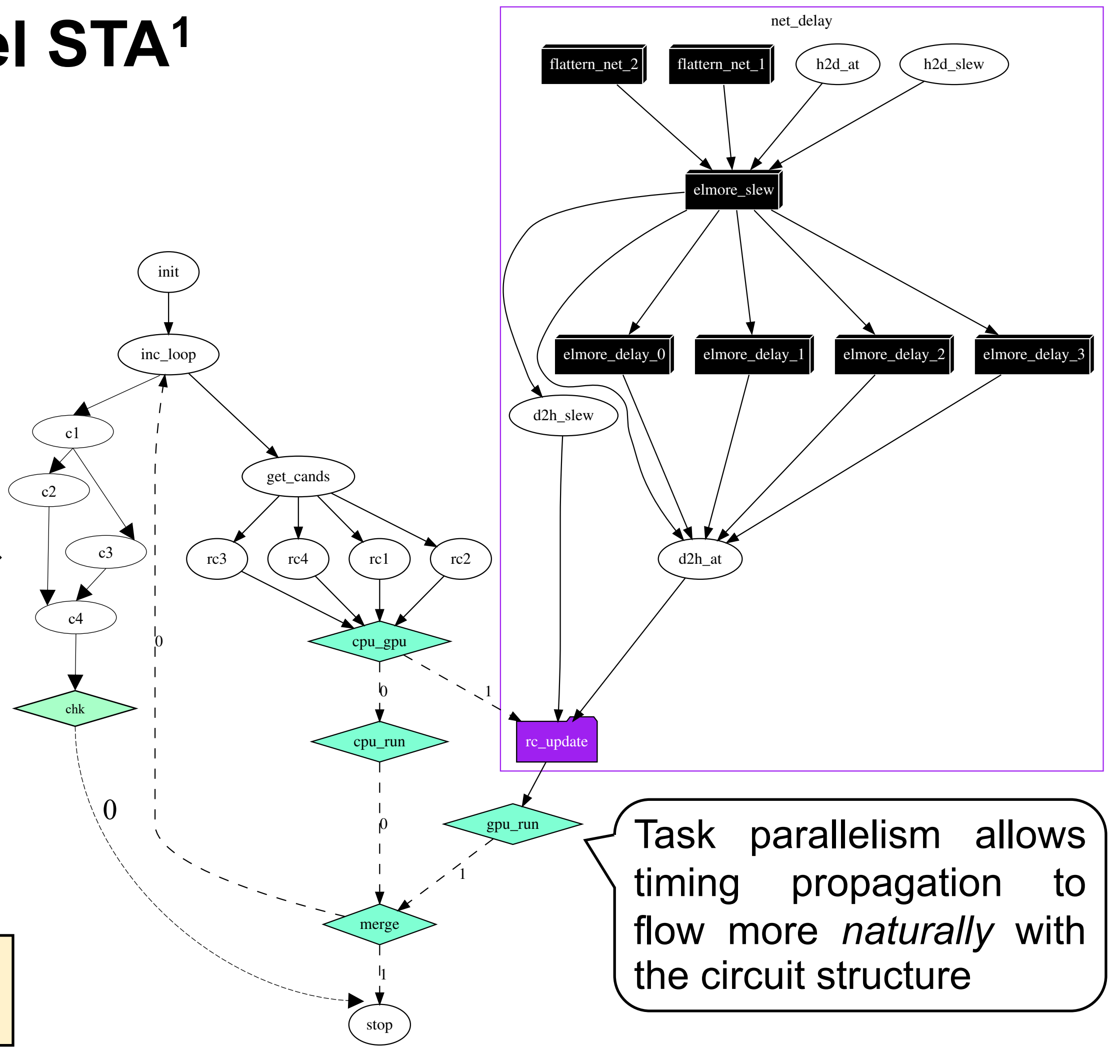


Time (minutes) to speed up a circuit timing analysis algorithm

CPU-parallel STA Algorithms

- **Levelization-based vs task-parallel STA[1]**



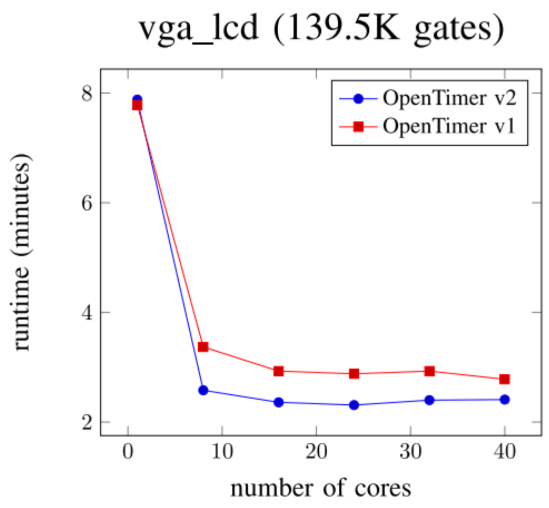


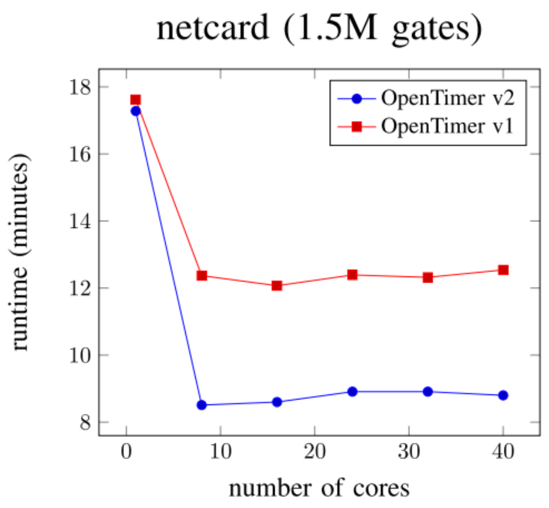
Task parallelism allows timing propagation to flow more *naturally* with the circuit structure

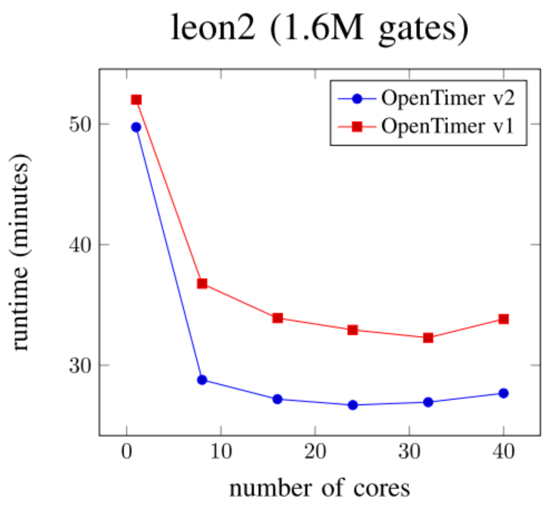
[1]: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2022

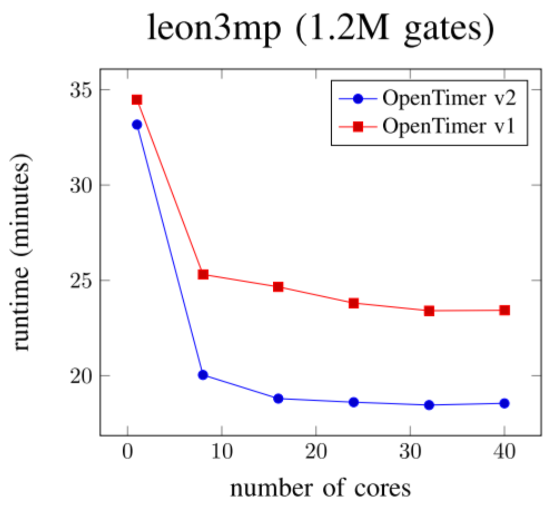
Levelization-based vs Task-parallel STA

- **OpenTimer v1: levelization-based parallel timing propagation**[1]
  - Implemented using OpenMP "parallel_for" primitive
- **OpenTimer v2: task-parallel timing propagation**[2]
  - Implemented using Taskflow (https://taskflow.github.io/)







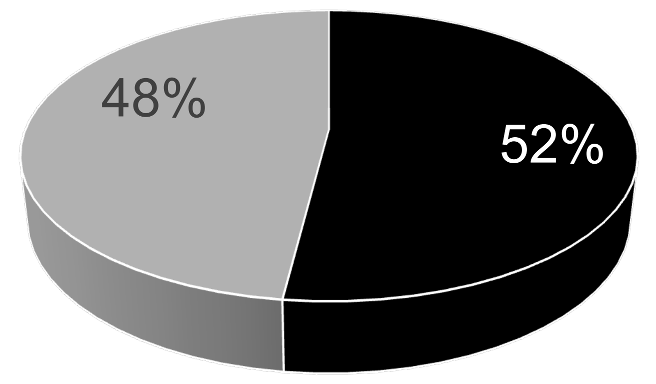


[1]: Tsung-Wei Huang and Martin Wong, "OpenTimer: A High-Performance Timing Analysis Tool," *IEEE/ACM ICCAD*, 2015
[2]: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD,* 2022

Overhead of Task-parallel STA

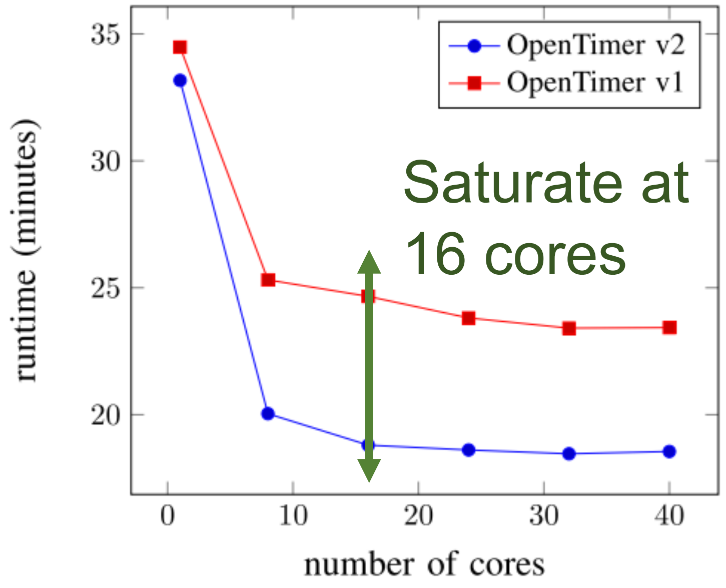
- **Task-parallel STA involves two runtime components**
  - Build a task dependency graph (TDG) – *often done in sequential*
  - Run the built TDG – *actual parallelization*
- **Large circuits induce big TDGs**
  - >10M tasks and >10M dependencies
- **Big TDG has a big scheduling cost**
  - 500–1000us for scheduling a task
  - Dependency breaking
  - Dynamic load balancing
  - Worker notification
  - …

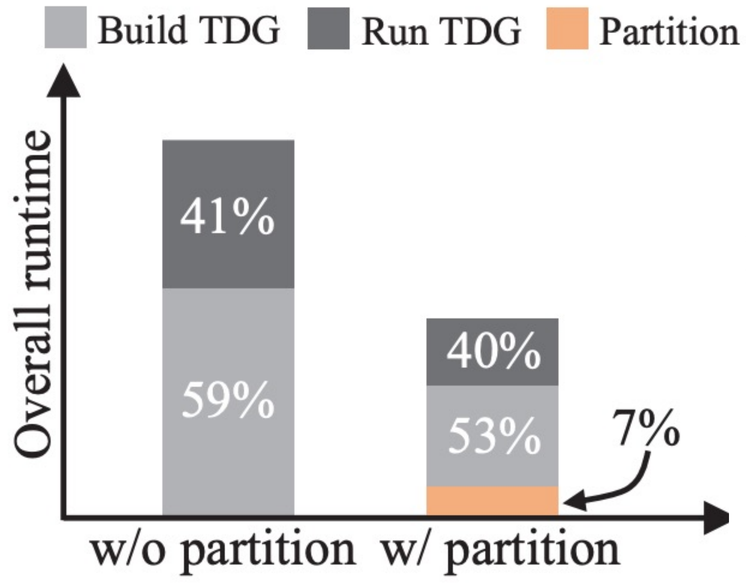
Runtime Breakdown of Task-parallel STA

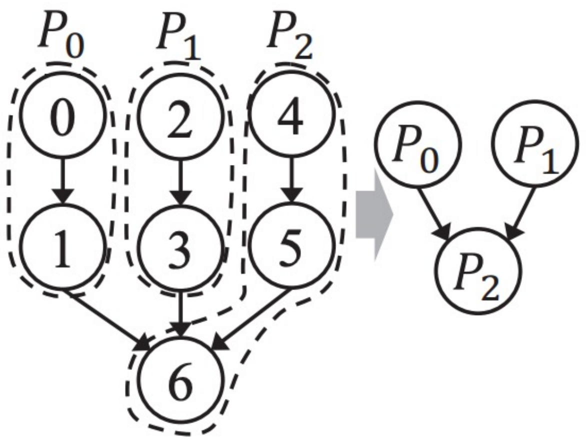


Need for a TDG Partitioning Algorithm

- **In practice, task-parallel STA saturates at 8–16 cores**
  - No need of a TDG of 10M tasks and 10M dependencies
- **We can partition a large TDG into a smaller version to**
  - Minimize TDG construction time (static overhead)
  - Minimize TDG scheduling overhead (dynamic overhead)

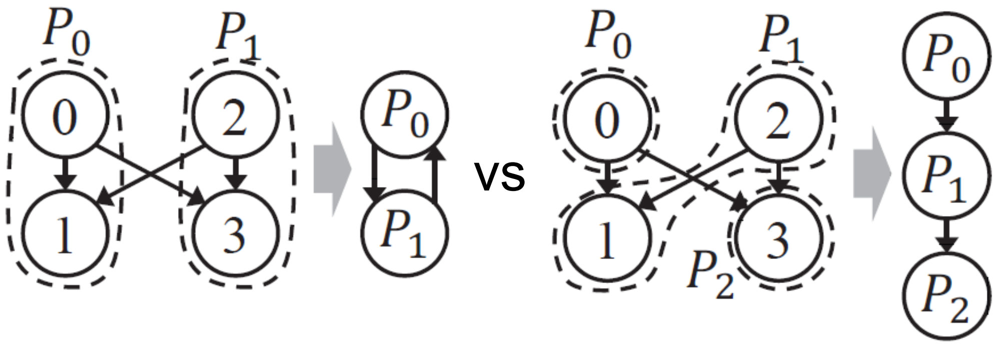


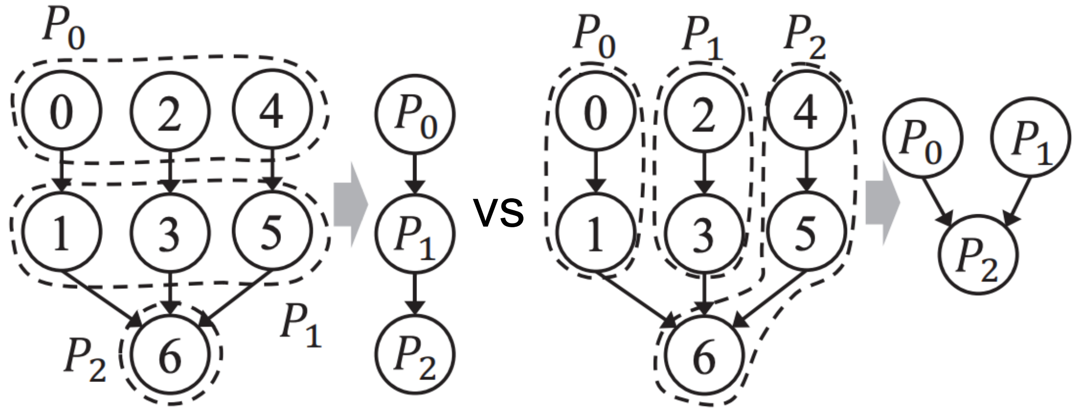




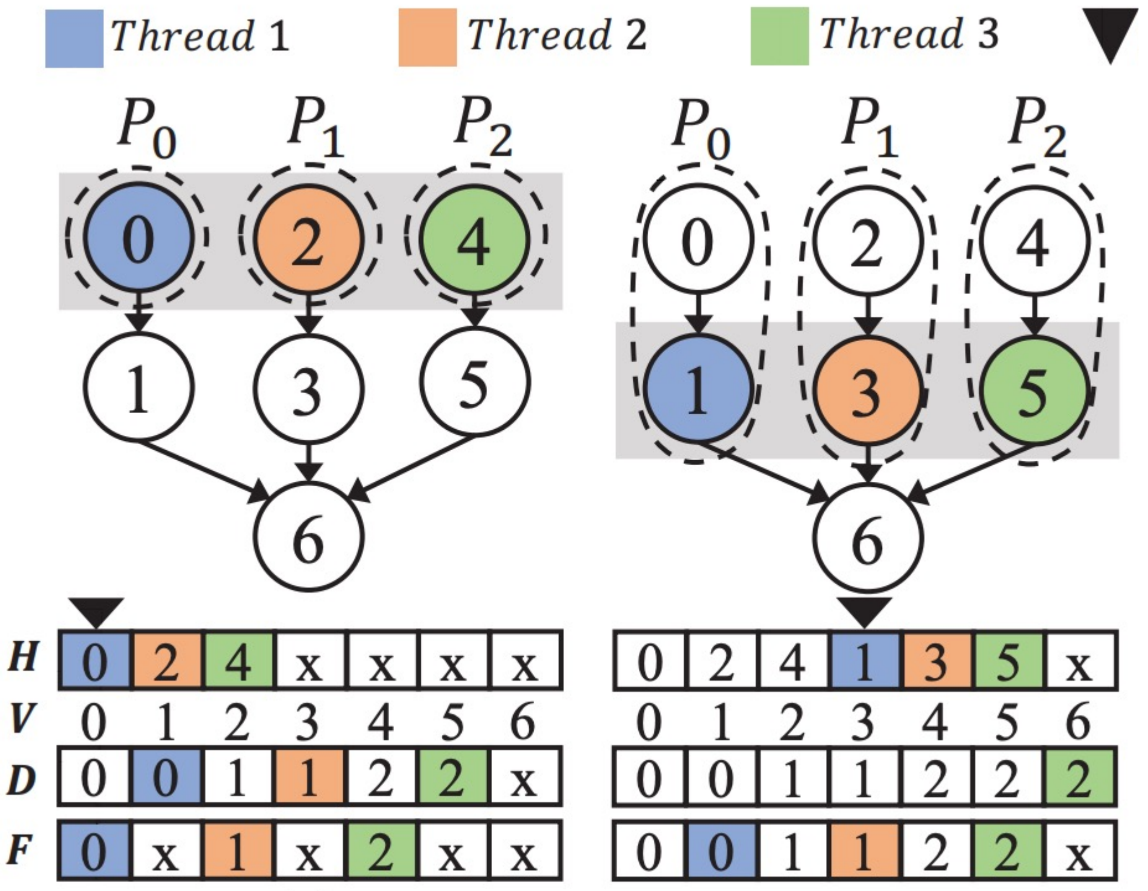
Challenges of TDG Partitioning

- **TDG partitioning is very different from circuit graph partitioning**
  - Circuit graph partitioning targets minimizing "cut"
  - TDG partitioning targets reducing the graph size without impacting too much its original task parallelism
- **TDG partitioning has other constraints to worry about …**
  - Cannot introduce too much time on TDG partitioning
  - Cannot introduce cyclic task dependencies
  - Cannot introduce too much sequential parallelism





G-PASTA: GPU-parallel TDG Partitioner[1]



Algorithm 1: G-PASTA partitioning kernel

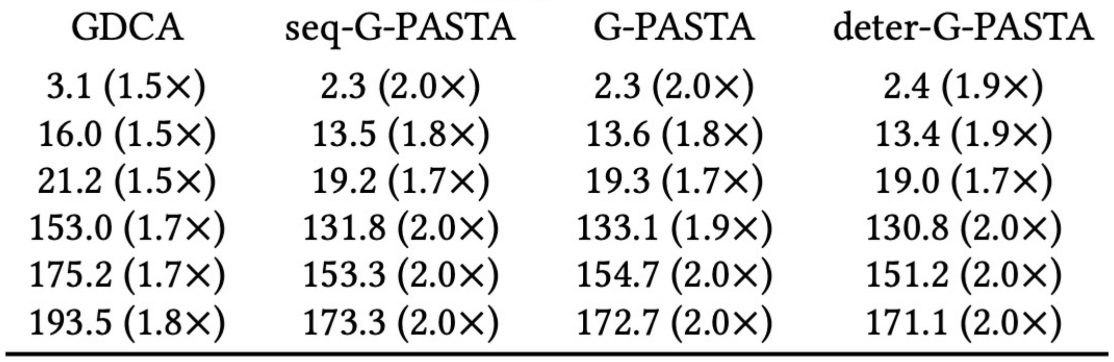
```
/* Step 1: assign f_pid for current-level tasks by d_pid */
parallel for each thread gid { /* gid < Rsize */
    cur = handle[Roffset + gid];
    cur_pid = d_pid[cur];
    if (atomicAdd(pid_cnt[cur_pid], 1) < Ps) then
        f_pid[cur] = cur_pid;
    else then
        new_pid = atomicAdd(max_pid, 1) + 1;
        f_pid[cur] = new_pid;
        pid_cnt[new_pid] ++;
}
/* Step 2: assign d_pid and release dependencies for neighbors */
parallel for each thread gid {
    for each n ∈ neighbors of cur
        /* cycle_free_clustering_algorithm() */
        atomicMax(d_pid[n], f_pid[cur]);
        if (atomicSub(dep_cnt[n], 1) == 1) then
            Woffset = atomicAdd(Wsize, 1);
            handle[Roffset + Rsize + Woffset] = n;
}
```

[1]: Boyang Zhang, et al. "G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis," *ACM/IEEE DAC*, 2024

Raw TDG vs Partitioned TDG

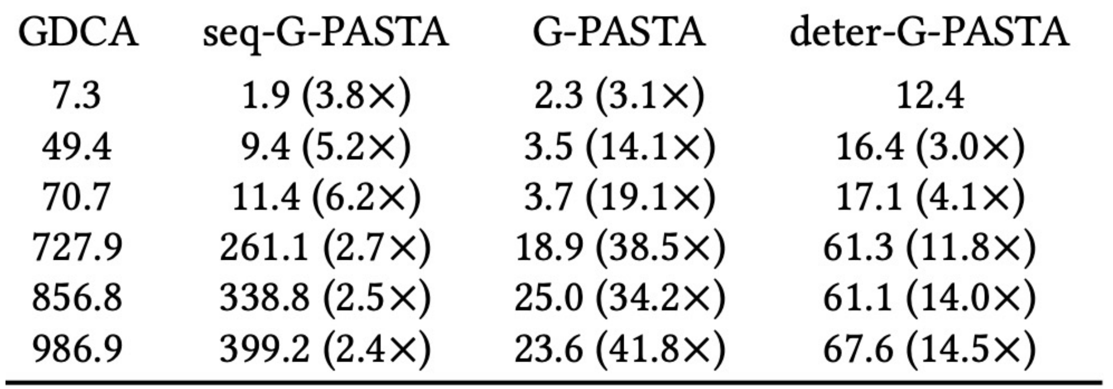
- **Baseline TDG partitioner: GDCA[1]**

TDG runtime after partitioning (ms/speed-up)

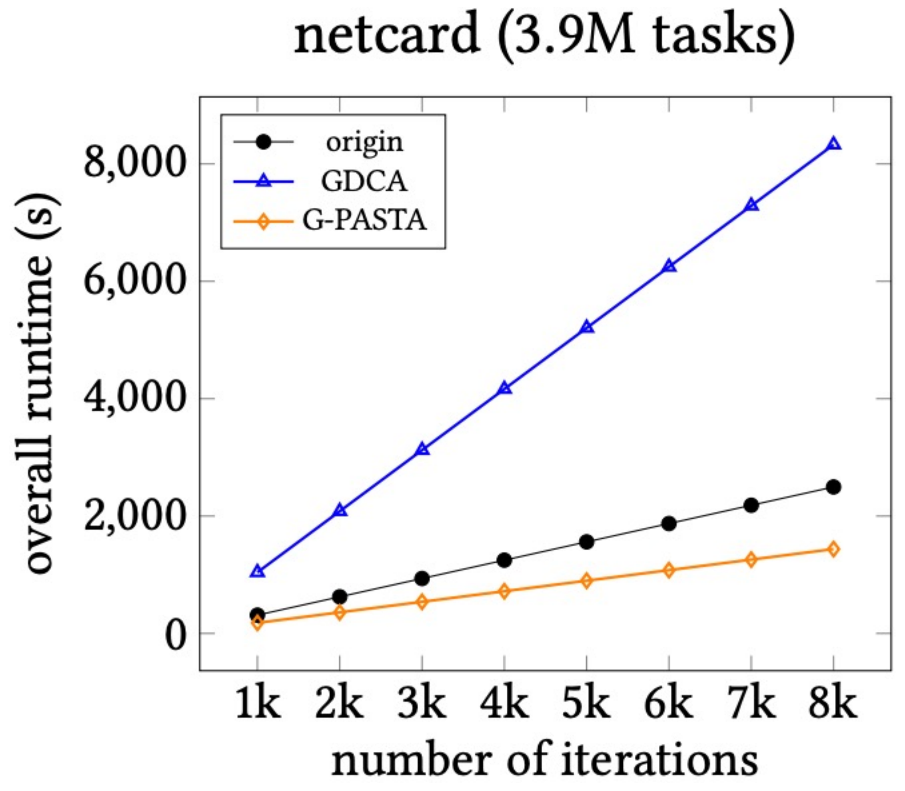


GDCA	seq-G-PASTA	G-PASTA	deter-G-PASTA
3.1 (1.5×)	2.3 (2.0×)	2.3 (2.0×)	2.4 (1.9×)
16.0 (1.5×)	13.5 (1.8×)	13.6 (1.8×)	13.4 (1.9×)
21.2 (1.5×)	19.2 (1.7×)	19.3 (1.7×)	19.0 (1.7×)
153.0 (1.7×)	131.8 (2.0×)	133.1 (1.9×)	130.8 (2.0×)
175.2 (1.7×)	153.3 (2.0×)	154.7 (2.0×)	151.2 (2.0×)
193.5 (1.8×)	173.3 (2.0×)	172.7 (2.0×)	171.1 (2.0×)

Partitioning runtime (ms/speed-up)



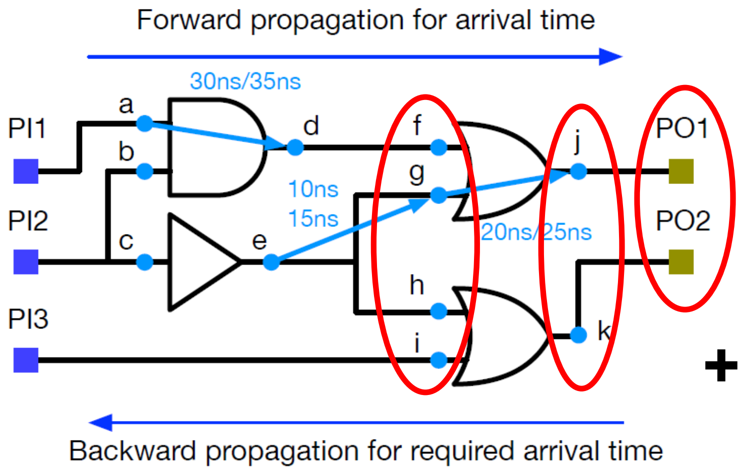
GDCA	seq-G-PASTA	G-PASTA	deter-G-PASTA
7.3	1.9 (3.8×)	2.3 (3.1×)	12.4
49.4	9.4 (5.2×)	3.5 (14.1×)	16.4 (3.0×)
70.7	11.4 (6.2×)	3.7 (19.1×)	17.1 (4.1×)
727.9	261.1 (2.7×)	18.9 (38.5×)	61.3 (11.8×)
856.8	338.8 (2.5×)	25.0 (34.2×)	61.1 (14.0×)
986.9	399.2 (2.4×)	23.6 (41.8×)	67.6 (14.5×)



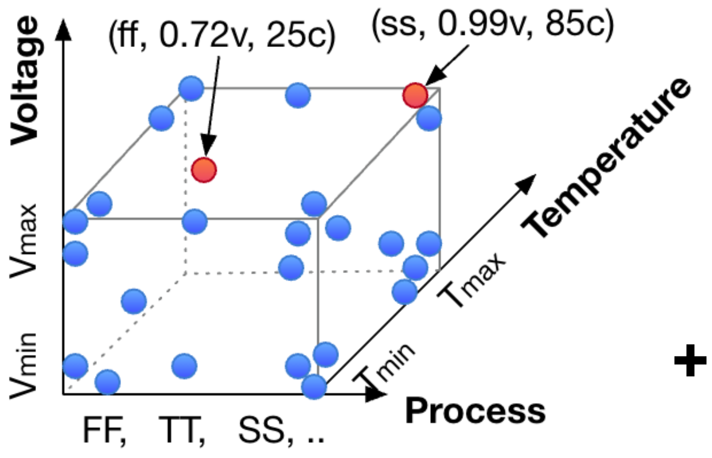
[1]: Bérenger Bramas and Alain Ketterlin. "Improving parallel executions by increasing task granularity in task-based runtime systems using acyclic DAG clustering," *Peer J Computer Science*, 2020

Why CPU-GPU Heterogeneous STA?

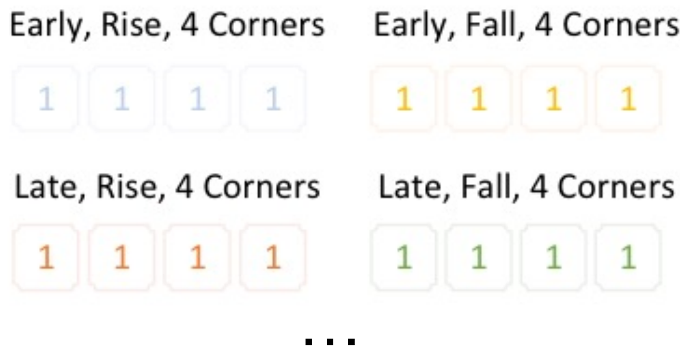
- **CPU-based parallelism does not scale beyond 16 threads**
  - CPU has limited thread count and memory bandwidth
  - Strong scalability is limited to the circuit structure itself
    - Amdahl's law: https://en.wikipedia.org/wiki/Amdahl%27s_law
- **Modern STA workloads exhibit a big volume of data parallelism**
  - Millions of dates to analyze (in parallel)
  - Hundreds of timing quantifies to propagate through millions of date


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

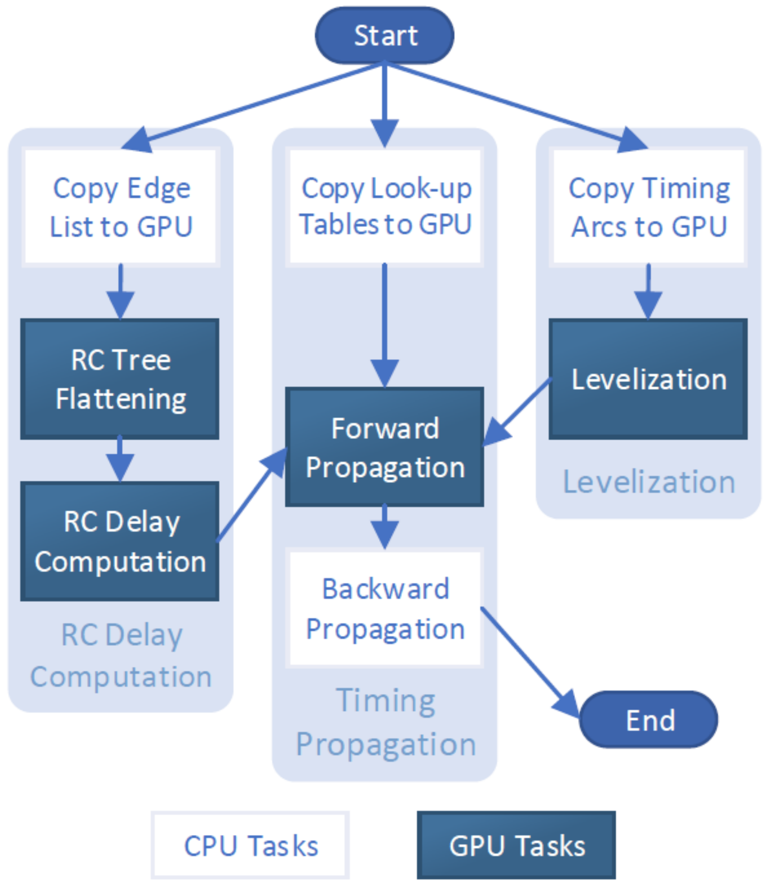
+



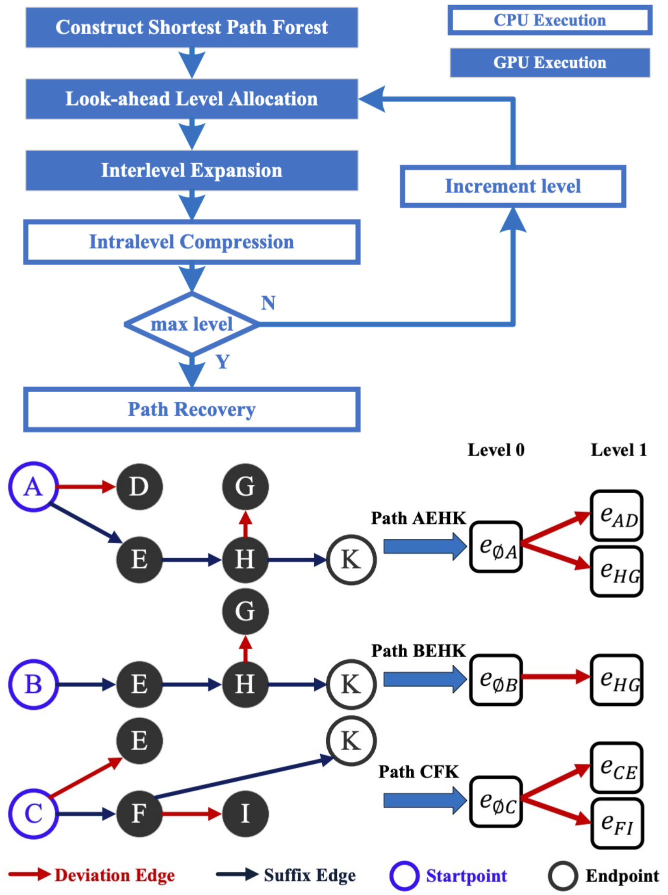
+



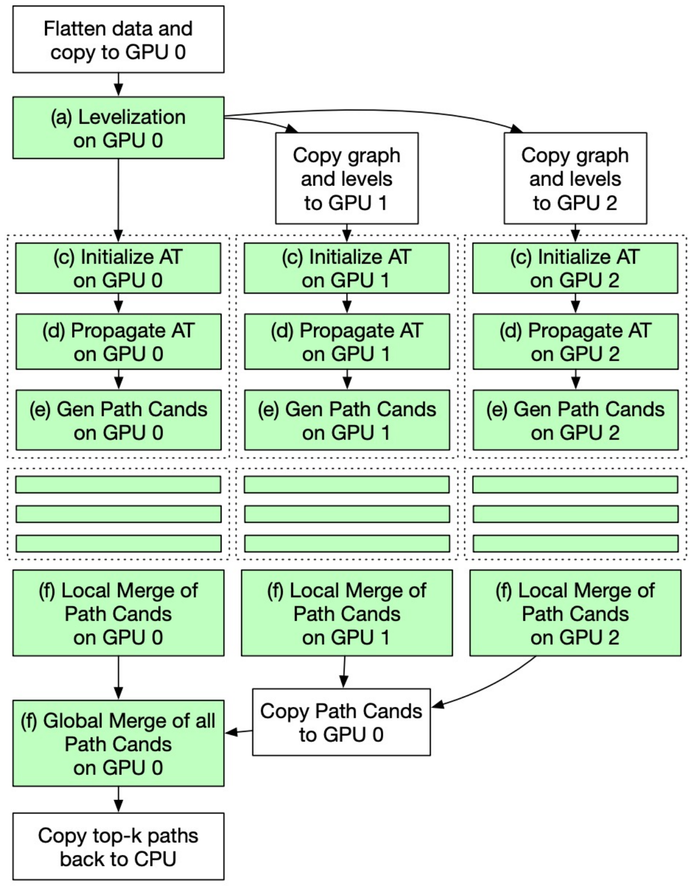
Our Research on GPU-accelerated STA



GPU-based graph analysis (ICCAD'20)



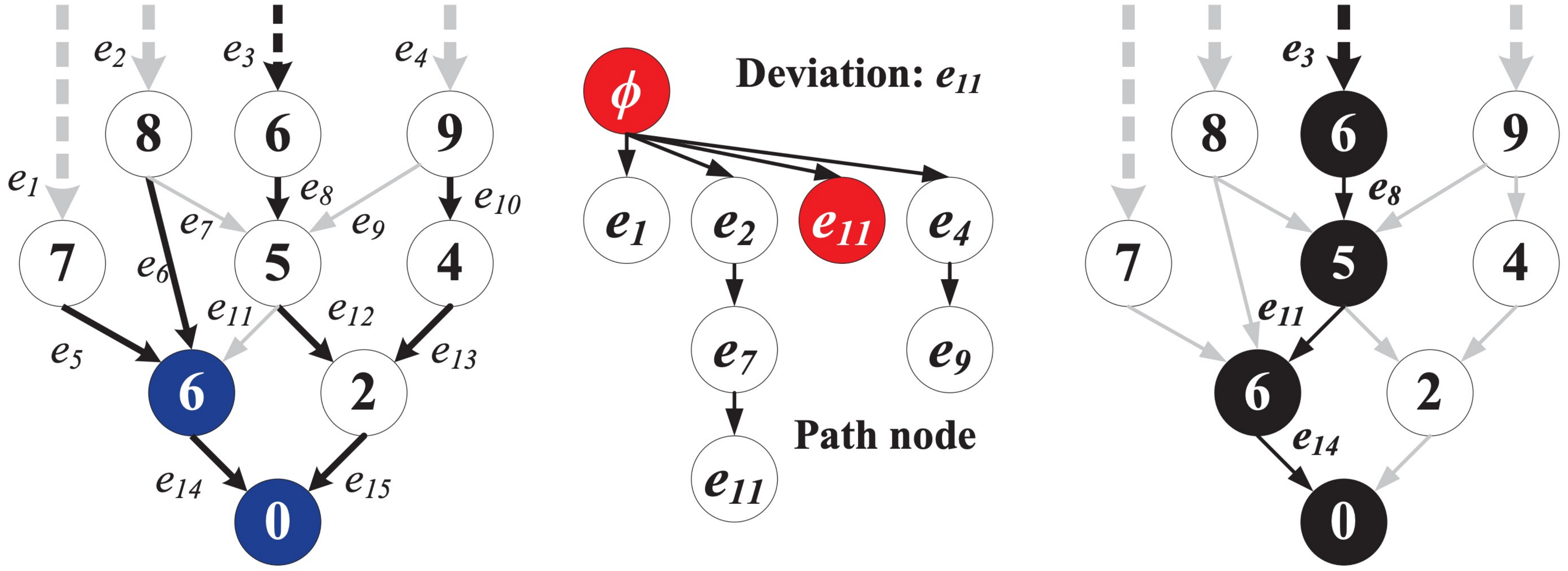
GPU-based path analysis (DAC'21)


Flatten data and copy to GPU 0
(a) Levelization on GPU 0
Copy graph and levels to GPU 1
Copy graph and levels to GPU 2
(c) Initialize AT on GPU 0
(c) Initialize AT on GPU 1
(c) Initialize AT on GPU 2
(d) Propagate AT on GPU 0
(d) Propagate AT on GPU 1
(d) Propagate AT on GPU 2
(e) Gen Path Cands on GPU 0
(e) Gen Path Cands on GPU 1
(e) Gen Path Cands on GPU 2
(f) Local Merge of Path Cands on GPU 0
(f) Local Merge of Path Cands on GPU 1
(f) Local Merge of Path Cands on GPU 2
(f) Global Merge of all Path Cands on GPU 0
Copy Path Cands to GPU 0
Copy top-k paths back to CPU

GPU-based CPPR (ICCAD'21)

GPU-accelerated Critical Path Generation

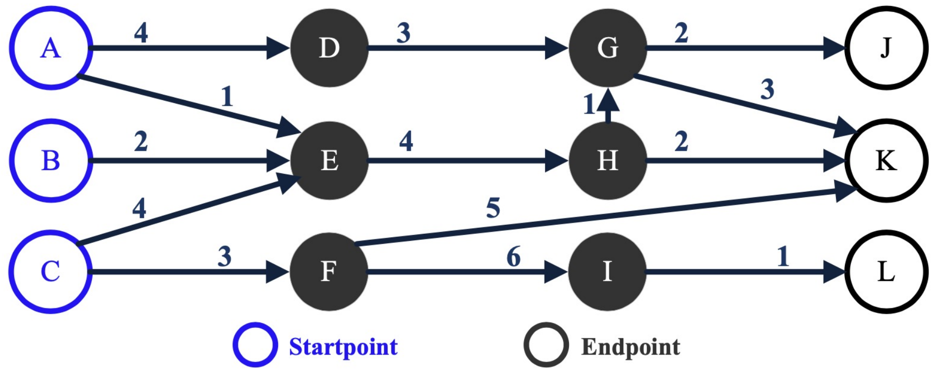
- GPU-friendly data structure for representing a critical path



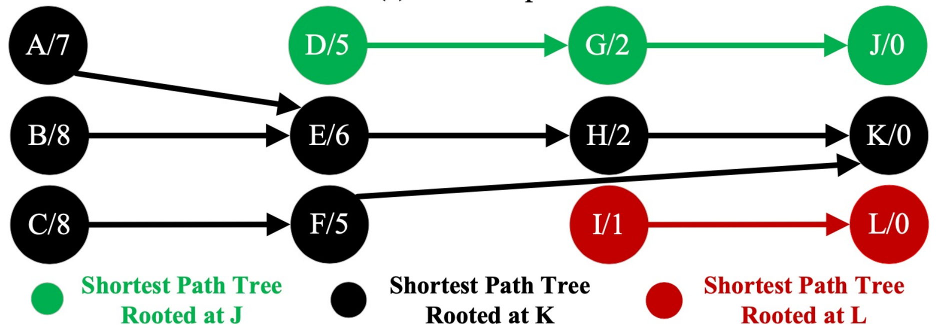
Path suffix: <e_{14}> + Path prefix: <e_3, e_8, e_{11}> = Path: <e_3, e_8, e_{11}, e_{14}>

GPU-parallel Suffix/Prefix Tree Building[1]

GPU-based suffix tree construction



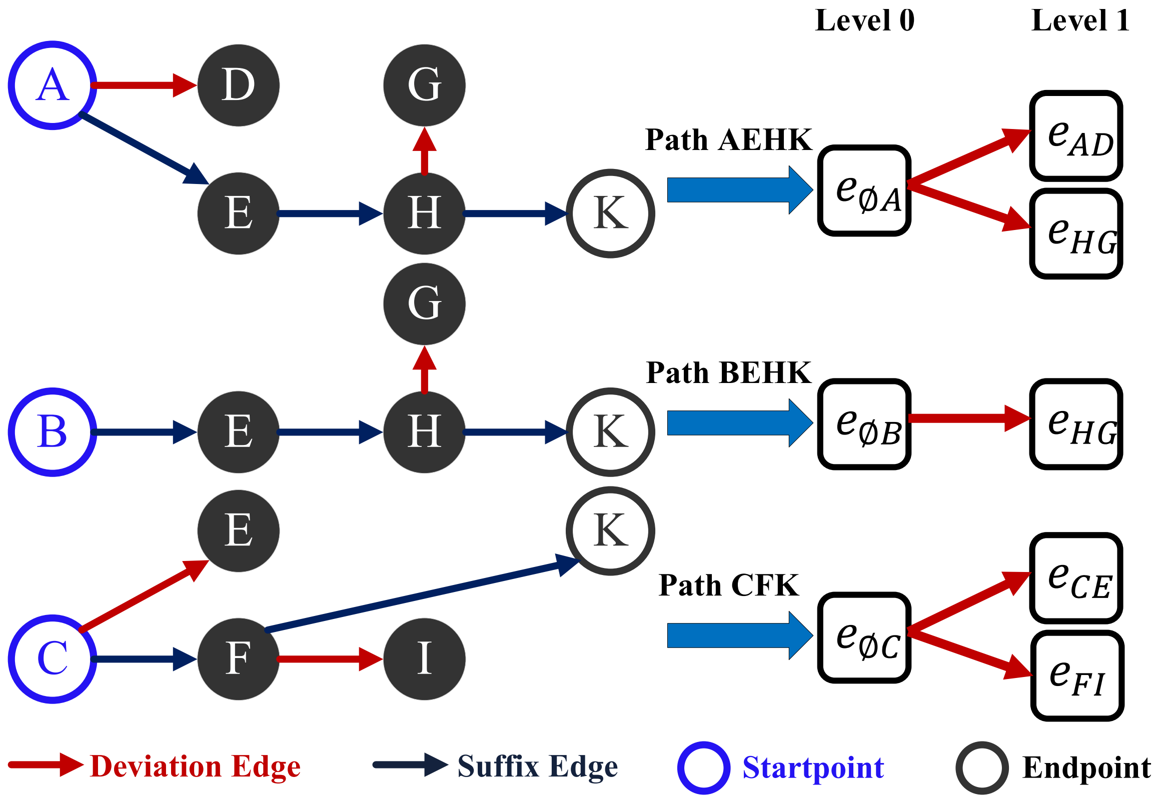
(a) STA Graph.



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.

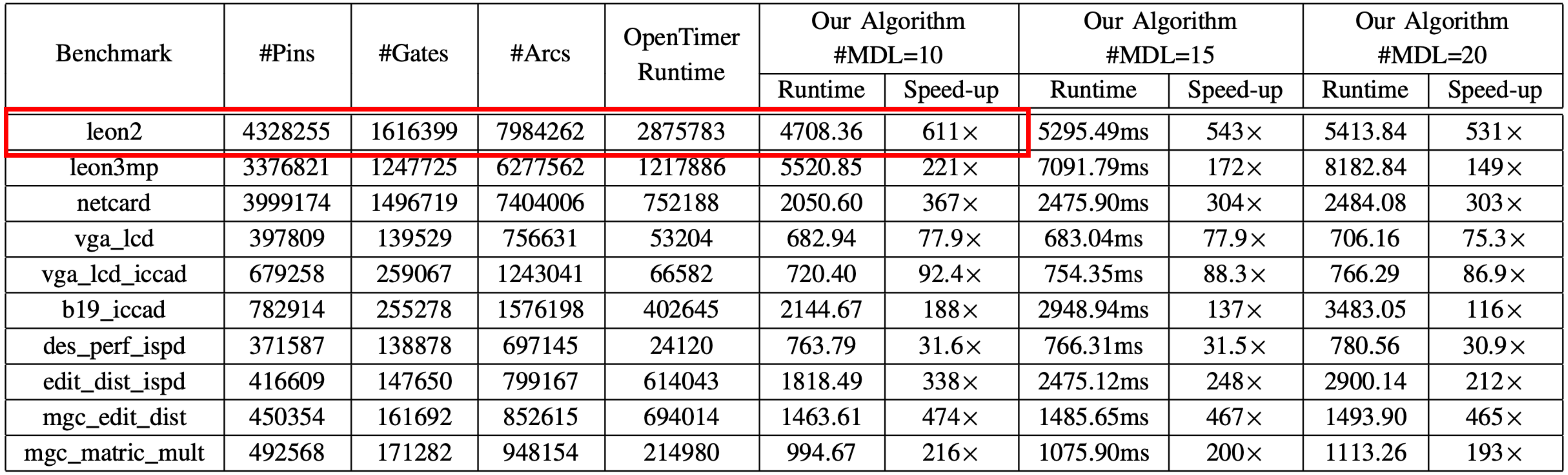
GPU-based prefix tree construction



[1]: Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong, "GPU-accelerated Path-based Timing Analysis," *IEEE/ACM Design Automation Conference (DAC)*, CA, 2021

GPU-accelerated Path-based Analysis

- **Example speed-up on a large design, leon2 (1.6M gates)**
  - **611x speed-up** over 1 CPU and **44x** over 40 CPUs
  - Evaluated on an Nvidia RTX 3090 GPU



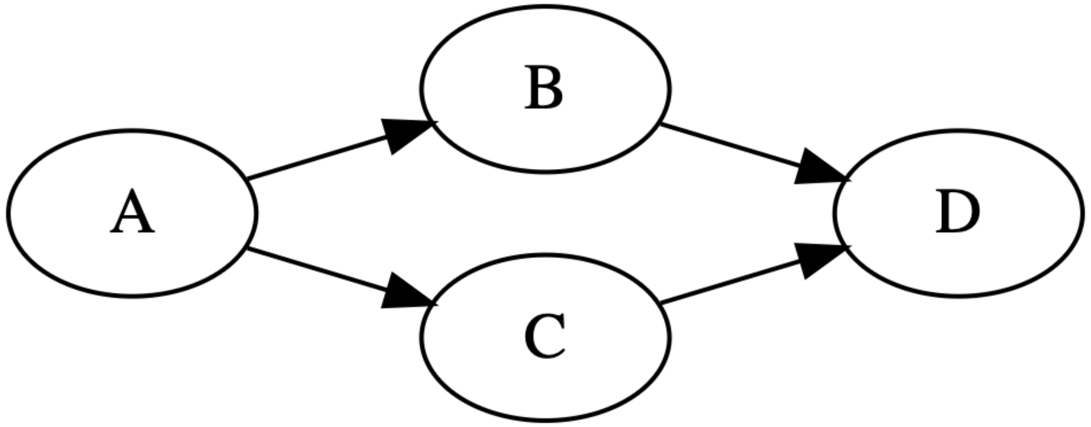
Benchmark	#Pins	#Gates	#Arcs	OpenTimer Runtime	Our Algorithm #MDL=10		Our Algorithm #MDL=15		Our Algorithm #MDL=20	
					Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	7984262	2875783	4708.36	611×	5295.49ms	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×	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×

[1]: Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong, "GPU-accelerated Path-based Timing Analysis," *IEEE/ACM Design Automation Conference (DAC)*, CA, 2021

Going Beyond STA – Taskflow System[1]

```
#include <taskflow/taskflow.hpp>
int main(){
    tf::Taskflow taskflow;
    tf::Executor executor;
    auto [A, B, C, D] = taskflow.emplace(
        [] () { std::cout << "TaskA\n"; }
        [] () { std::cout << "TaskB\n"; },
        [] () { std::cout << "TaskC\n"; },
        [] () { std::cout << "TaskD\n"; }
    );
    A.precede(B, C);
    D.succeed(B, C);
    executor.run(taskflow).wait();
    return 0;
}
```

// live: https://godbolt.org/z/j8hx3xnnx

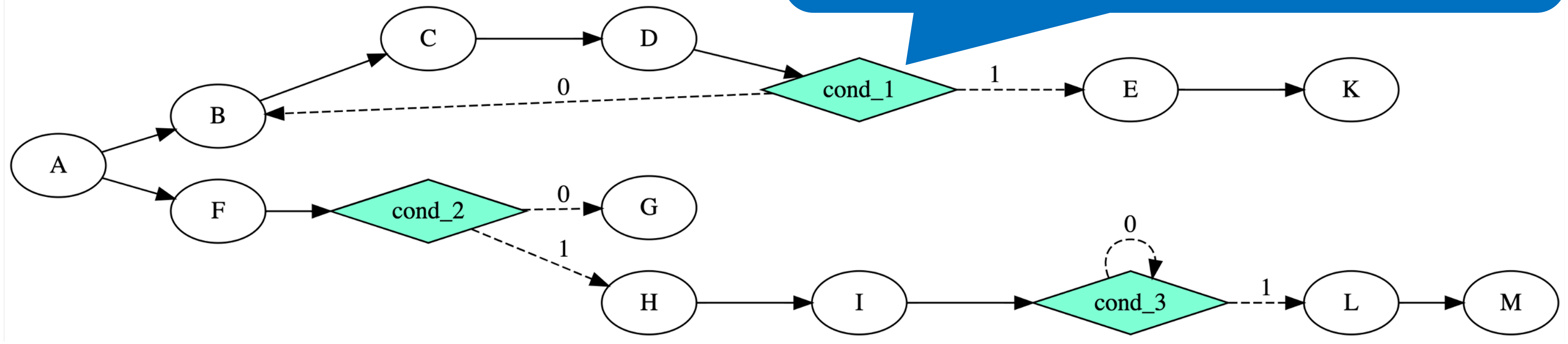

$$\begin{tikzcd} & B \arrow[dr] & \\ A \arrow[ur] \arrow[dr] & & D \\ & C \arrow[ur] & \end{tikzcd}$$

[1]: T.-W. Huang, et. al, “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System,” *IEEE TPDS*, vol. 33, no. 6, pp. 1303-1320, June 2022

Control Taskflow Graph (CTFG) Model

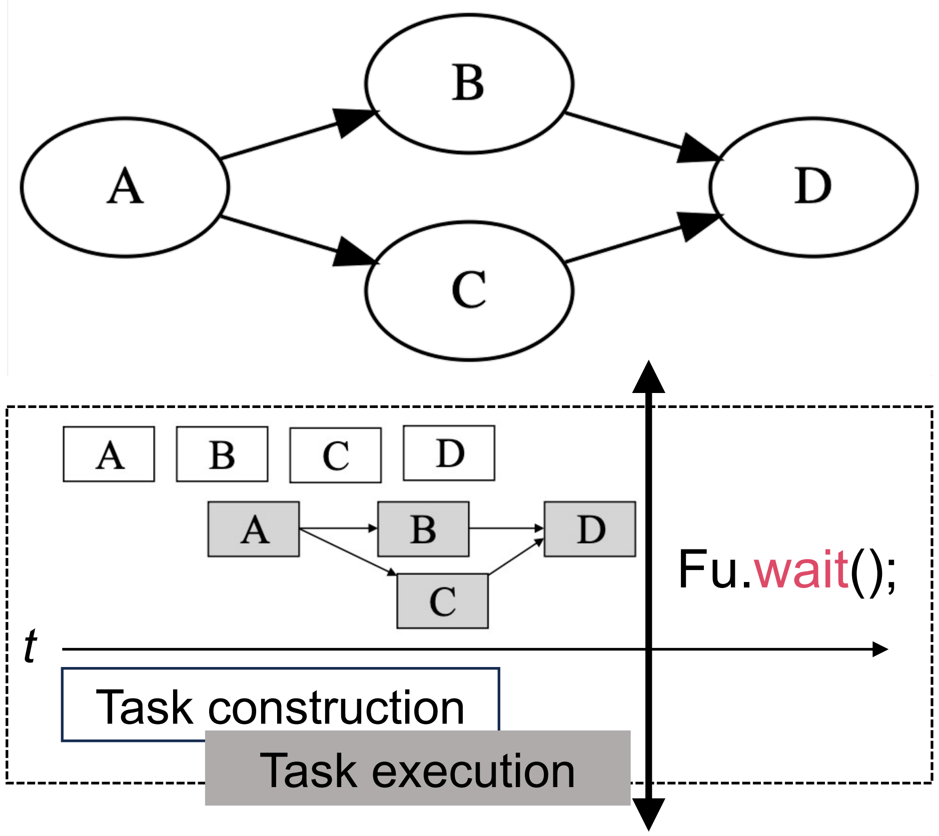
```
// CTFG goes beyond the limitation of traditional DAG-based models
auto cond_1 = taskflow.emplace([](){ return run_B() ? 0 : 1; }); // 0: is the index of B
auto cond_2 = taskflow.emplace([](){ return run_G() ? 0 : 1; }); // 0: is the index of G
auto cond_3 = taskflow.emplace([](){ return loop() ? 0 : 1; }); // 0: is the index of cond_3
cond_1.precede(B, E);           // cycle
cond_2.precede(G, H);           // if-else
cond_3.precede(cond_3, L);      // loop
```

Very difficult for existing DAG-based systems to express an efficient overlap between tasks and control flow …



Dynamic Task Graph in Taskflow

```
// Live: https://godbolt.org/z/j76ThGbWK
tf::Executor executor;
auto A = executor.silent_dependent_async([](){
    std::cout << "TaskA\n";
});
auto B = executor.silent_dependent_async([](){
    std::cout << "TaskB\n";
}, A);
auto C = executor.silent_dependent_async([](){
    std::cout << "TaskC\n";
}, A);
auto [D, Fu] = executor.dependent_async([](){
    std::cout << "TaskD\n";
}, B, C);
Fu.wait();
```



Specify arbitrary task dependencies using C++ variadic parameter pack

Everything is Composable in Taskflow

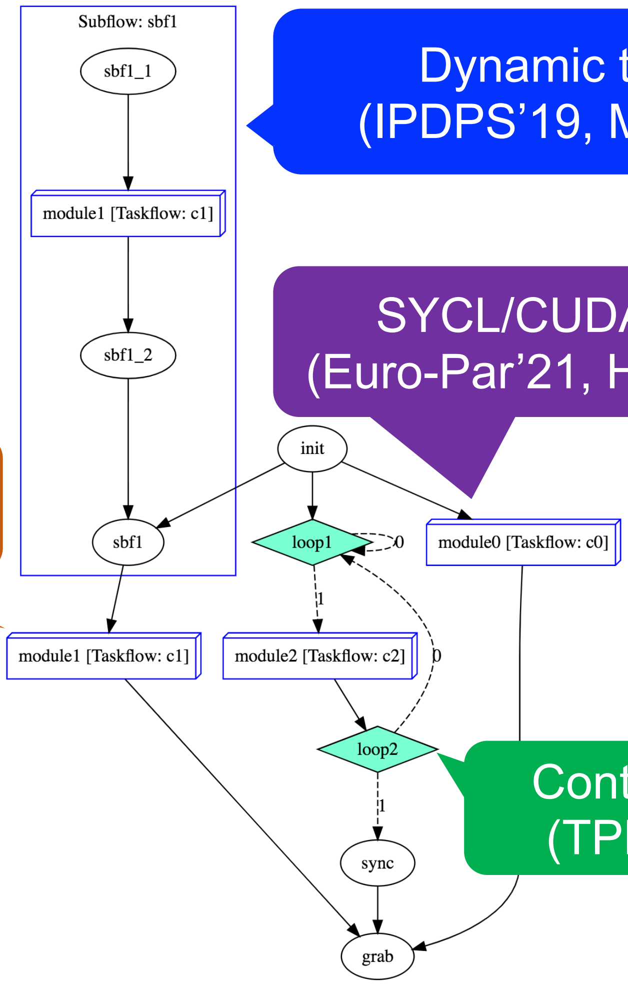
- **End-to-end parallelism in one graph**
  - Task, dependency, control flow all together
  - Scheduling with whole-graph optimization
  - Efficient overlap among heterogeneous tasks
- **Largely improved productivity!**

Dynamic task
(IPDPS'19, MM'19)

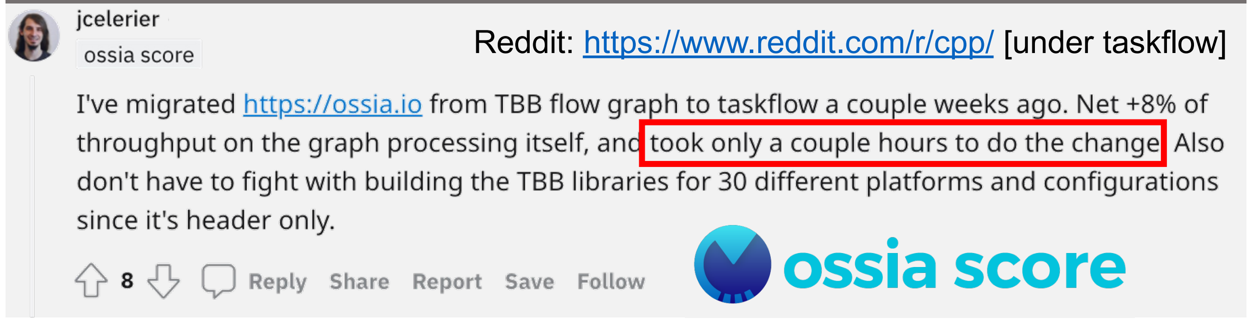
SYCL/CUDA task
(Euro-Par'21, HPEC'20)

Composition
(HPDC'22, ICPP'22, HPEC'19)

Control flow
(TPDS'22)



Industrial use-case of productivity improvement using Taskflow


jcelerier
ossia score

Reddit: https://www.reddit.com/r/cpp/ [under taskflow]

I've migrated https://ossia.io from TBB flow graph to taskflow a couple weeks ago. Net +8% of throughput on the graph processing itself, and took only a couple hours to do the change. Also don't have to fight with building the TBB libraries for 30 different platforms and configurations since it's header only.

8 Reply Share Report Save Follow

ossia score

Taskflow is Being Used by Many Projects

OT ossia deal.II MYDATAMODELS XANADU LiveHD

ROS industrial REVEAL SNIG Inference Engine 24h OPERON

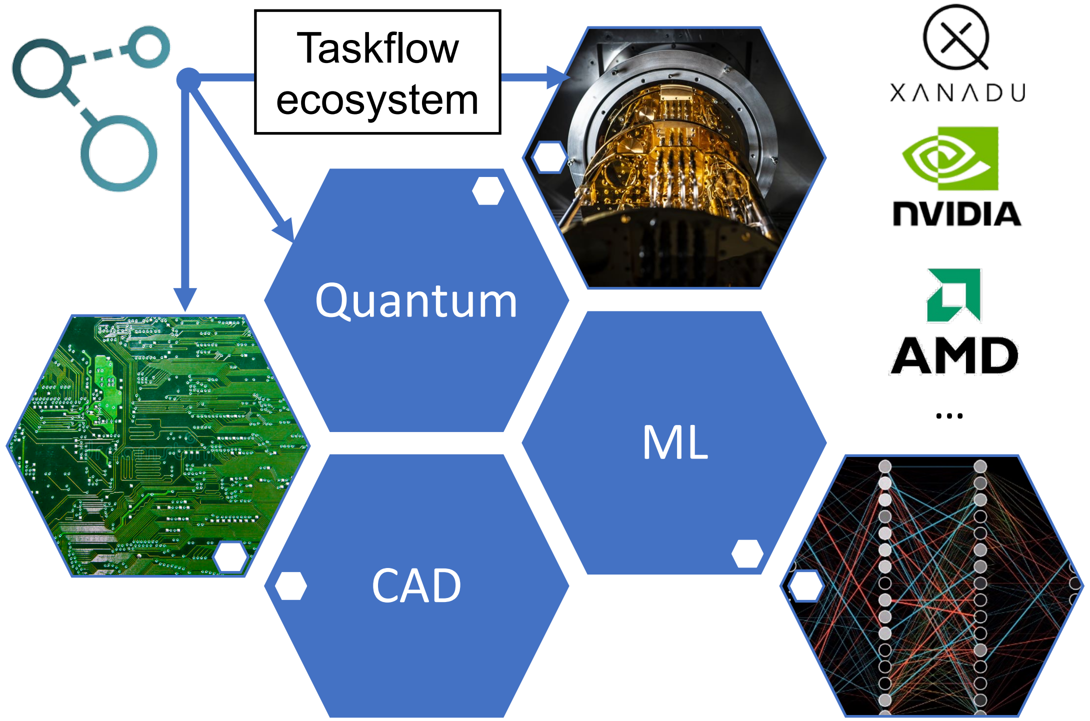
R NVIDIA. Explosion METHANE powered graphics Chainblocks Rapid Fuzz JET BRAINS RESEARCH

atomicDEX ReAgent RAPIDS XILINX TESSERACT

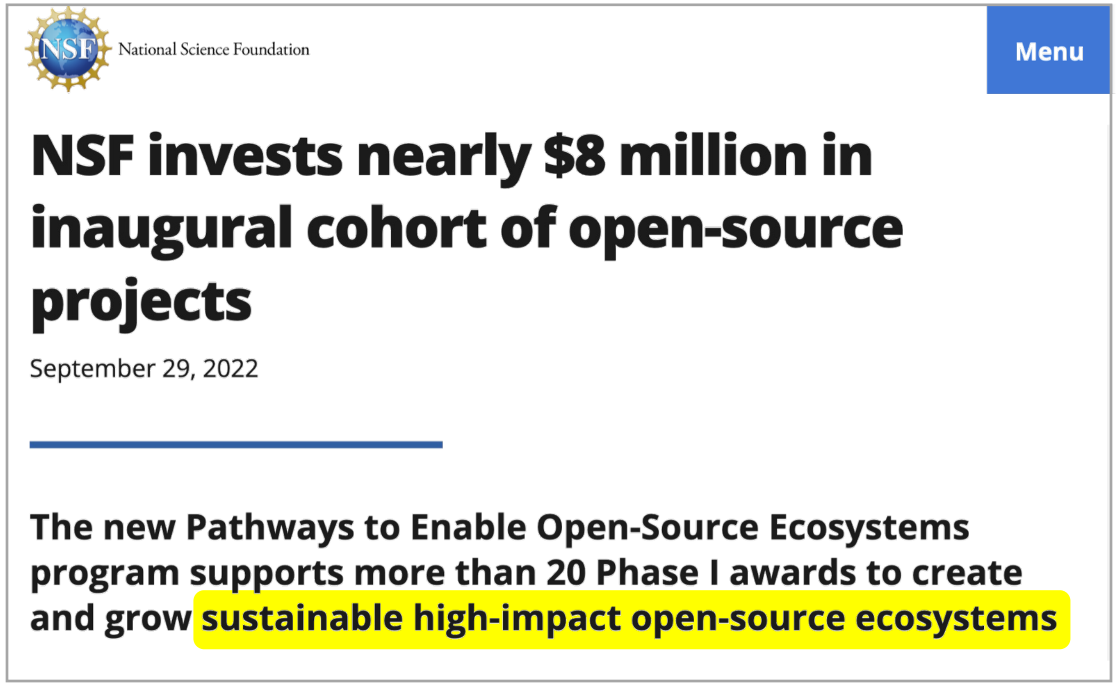
COCOS MUNICH QUANTUM TOOLKIT ModuleWorks

Our NSF POSE Project[1]: Sustainability

- **Create a sustainable Taskflow application ecosystem**



https://beta.nsf.gov/tip/updates/nsf-invests-nearly-8-million-inaugural-cohort-open



National Science Foundation

Menu

NSF invests nearly $8 million in inaugural cohort of open-source projects

September 29, 2022

The new Pathways to Enable Open-Source Ecosystems program supports more than 20 Phase I awards to create and grow sustainable high-impact open-source ecosystems

[1]: "POSE: Phase I: Toward a Task-Parallel Programming Ecosystem for Modern Scientific Computing," $298K, 09/15/2022—08/31/2023, NSF POSE, TI-2229304