Taskflow: A General-purpose Task-parallel Programming System

Dr. Tsung-Wei (TW) Huang, Assistant Professor

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

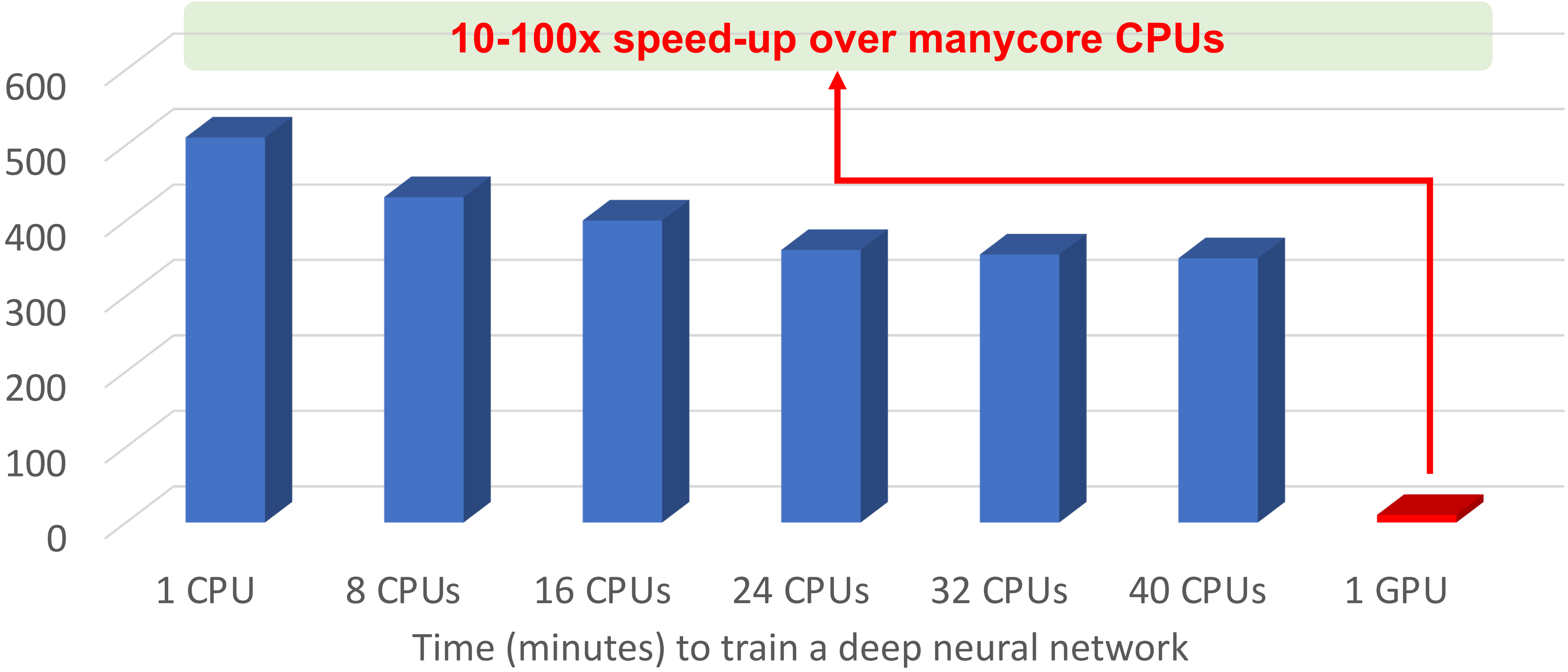
https://tsung-wei-huang.github.io/

Agenda

- Understand the challenges of parallel computing
- Introduce our new task-parallel programming system
- Dive into our system runtime
- Apply our system to computer engineering problems

Why Parallel Computing?

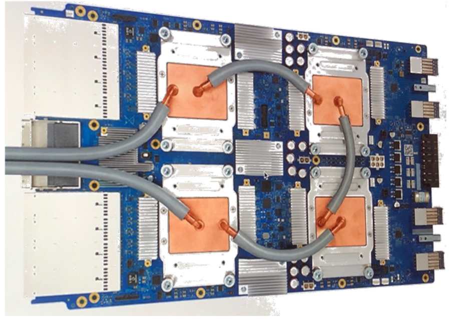
- **Advances performance to a new level previously out of reach**



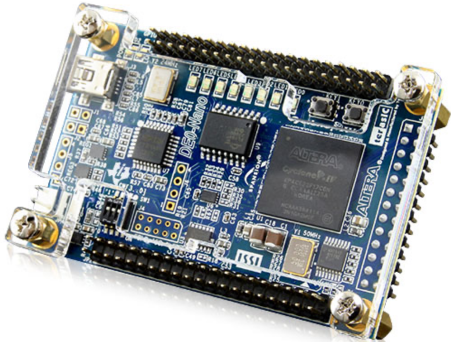
Increasing Heterogeneity in Computers …

Central Processing Unit (CPU)

Graphics Processing Unit (GPU)



Tensor Processing Unit (TPU)



FPGA

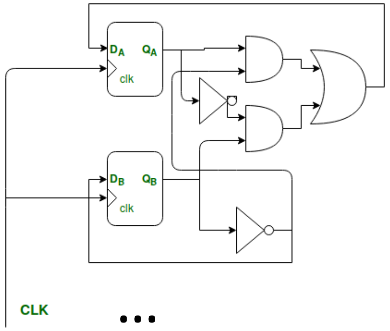
Neuromorphic Devices

Quantum Accelerator

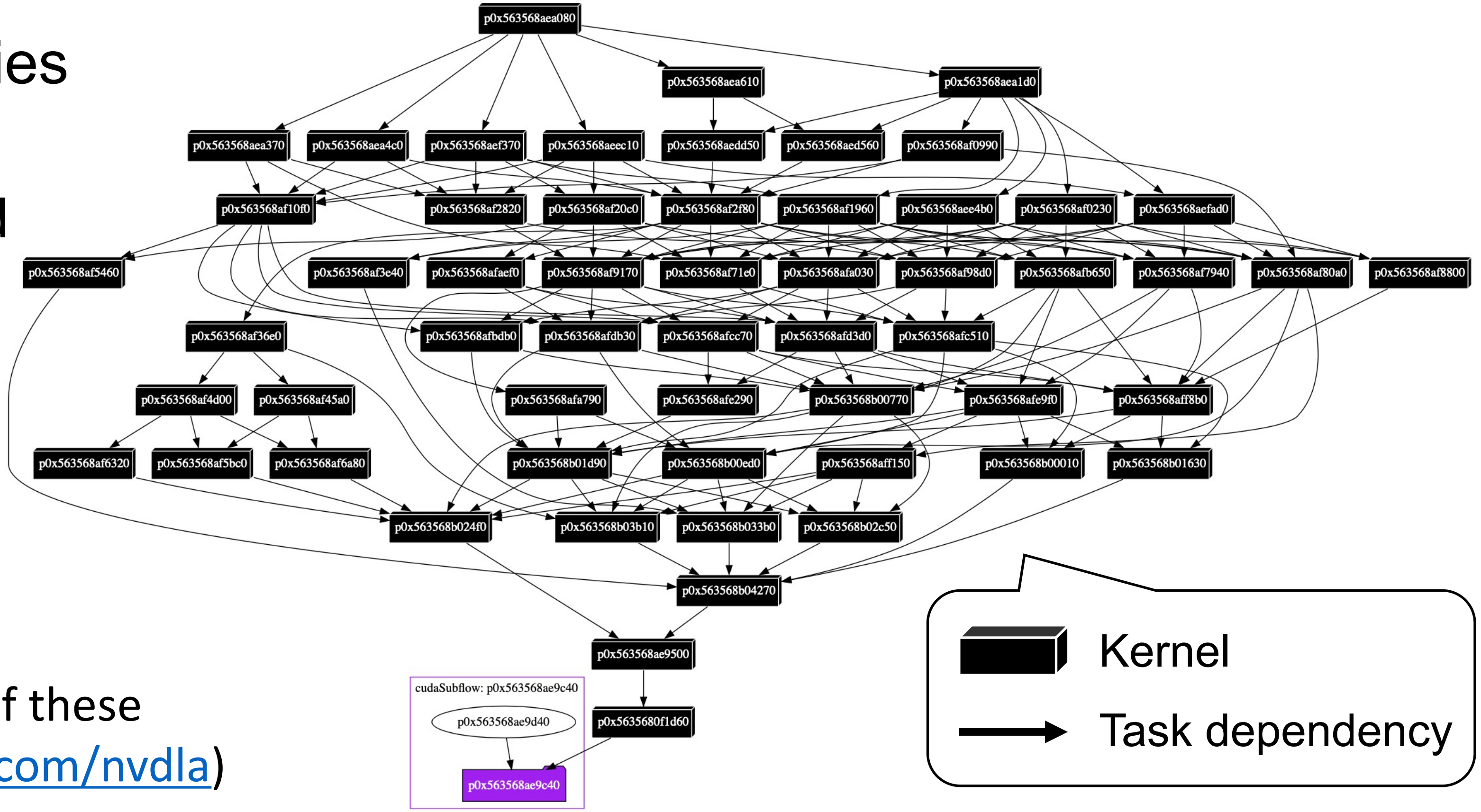
How do we program these accelerators? – *DARPA ERI, DOE, NSF PPoSS, Jump 2.0*

Today's Workloads are Very Complex …

- **GPU-accelerated circuit analysis on a design of <u>500M</u> gates**
  - >100 kernels
  - >100 dependencies
  - >500s to finish
  - >10 hrs turnaround



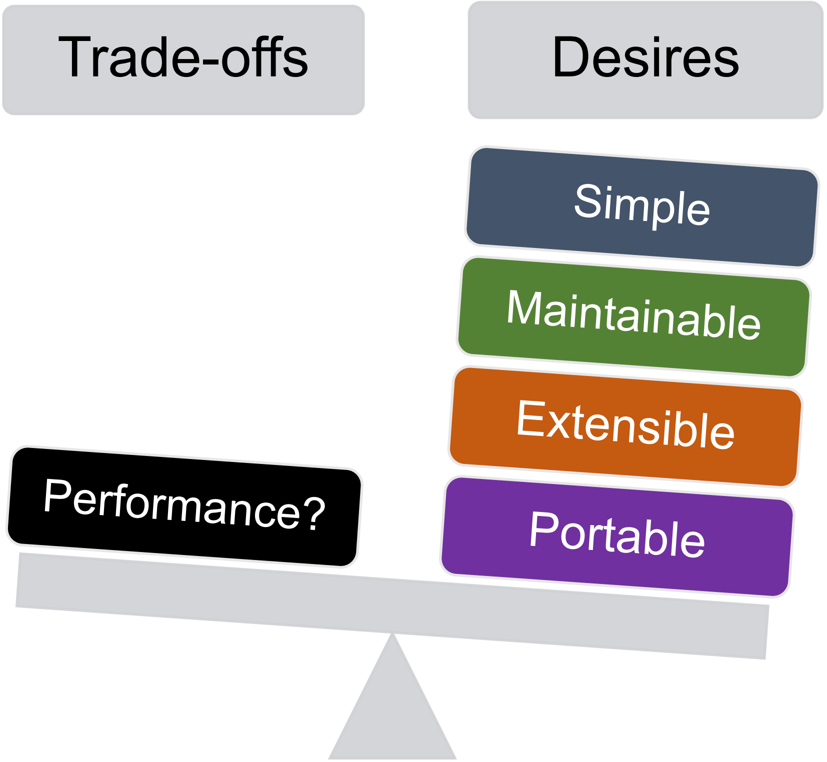
Simulation



What are the output values of these 500M gates? (https://github.com/nvdla)

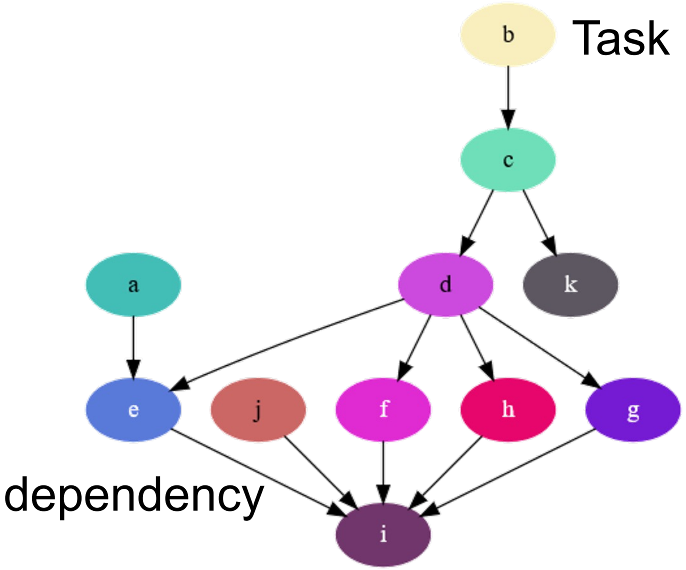
Parallel Programming is a “Big” Challenge

- **You need to deal with A LOT OF parallelization details**
  - Parallelism abstraction (software + hardware)
  - Concurrency control
  - Task and data race avoidance
  - Dependency constraints
  - Scheduling efficiencies (load balancing)
  - Performance portability
  - …
- **And, don’t forget about trade-offs**
  - Desires vs Performance



Need a New Programming Solution

- **Why existing parallel programming systems are not sufficient?**
  - Good at loop parallelism but weak in large and irregular task parallelism
  - Count on directed acyclic graph (DAG) model that cannot handle control flow
- **Envisioning from the evolution of parallel programming:**
  - Task parallelism is the best model for heterogeneous computing
- **Plenty of challenges to be solved …**
  - New applications demand new tasking models
    - Cost of control flow becomes more important
  - New accelerators demand new schedulers
    - Must value performance portability
  - Sustainability over hardware generations
  - …

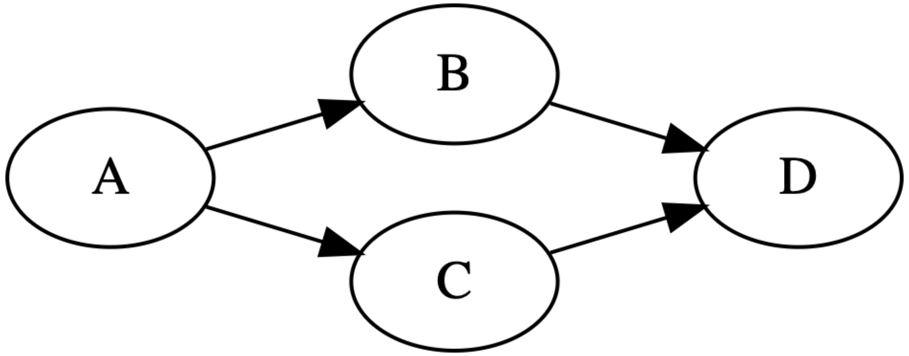


Agenda

- Understand the challenges of parallel computing
- Introduce our new task-parallel programming system
- Dive into our system runtime
- Apply our system to computer engineering problems

Our DARPA ERI/IDEA Project[1]: Taskflow

```
#include <taskflow/taskflow.hpp>   // Taskflow is header-only, no wrangle with installation
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);  // A runs before B and C
  D.succeed(B, C);  // D runs after    B and C
  executor.run(taskflow).wait();
  return 0;
}
```



[1]: “OpenTimer and DtCraft,” $427K, 06/2018-07/2019, DARPA Intelligent Design of Electronic Assets (IDEA) Program, FA 8650-18-2-7843

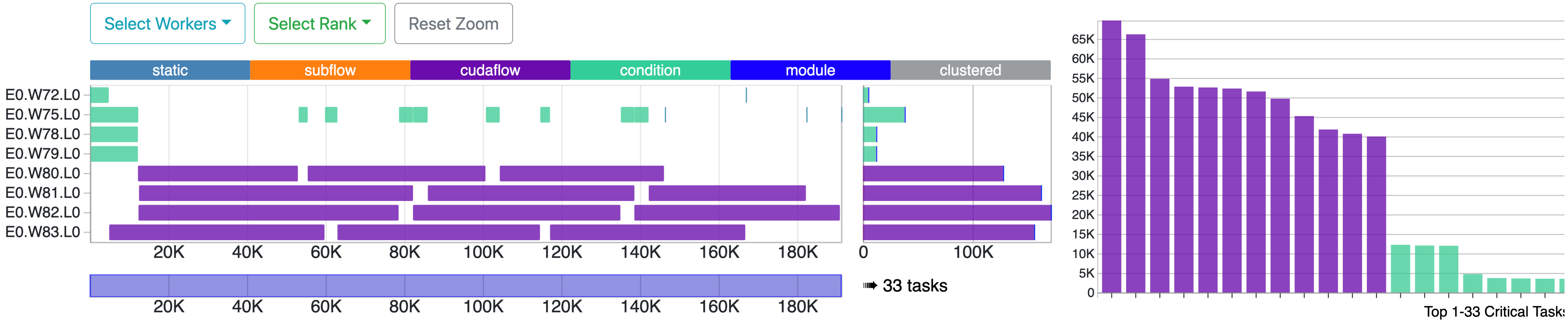
Drop-in Integration

- **Taskflow is header-only – *no wrangle with installation***
  - Include Taskflow to your project and tell your compiler where to find it

```
# Compile your program with Taskflow
~$ git clone https://github.com/taskflow/taskflow.git
~$ g++ -std=c++17 simple.cpp –I taskflow/ –O2 –pthread –o simple
~$ ./simple
TaskA
TaskC
TaskB
TaskD
```

Built-in Visualizer using a Browser

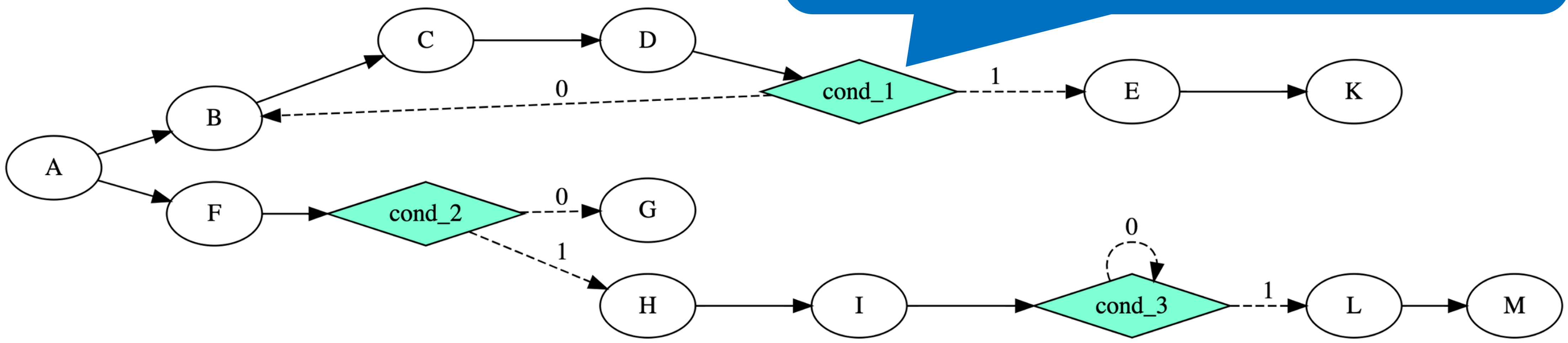
```
# Enable the environment variable TF_ENABLE_PROFILER for visualizer
~$ TF_ENABLE_PROFILER=simple.json ./simple
~$ cat simple.json
[
{“executor”:”0”, “data”:[{“worker”:0, “level”:0, “data”: …}]}
]
# Paste the JSON to https://taskflow.github.io/tfprof/
```



Control Taskflow Graph Programming (CTFG)

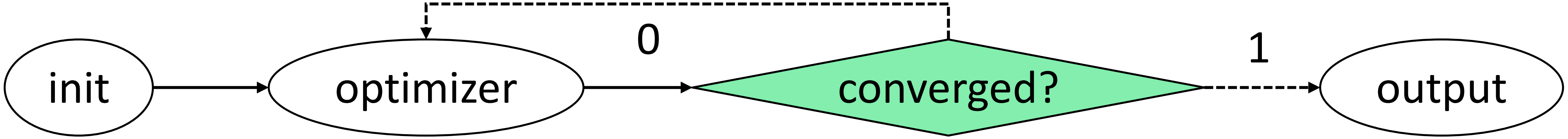
```
// CTFG goes beyond the limitation of traditional DAG
auto cond_1 = taskflow.emplace([](){ return decision1(); });
auto cond_2 = taskflow.emplace([](){ return decision2(); });
auto cond_3 = taskflow.emplace([](){ return decision3(); });
cond_1.precede(B, E);          // cycle
cond_2.precede(G, H);          // 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 …



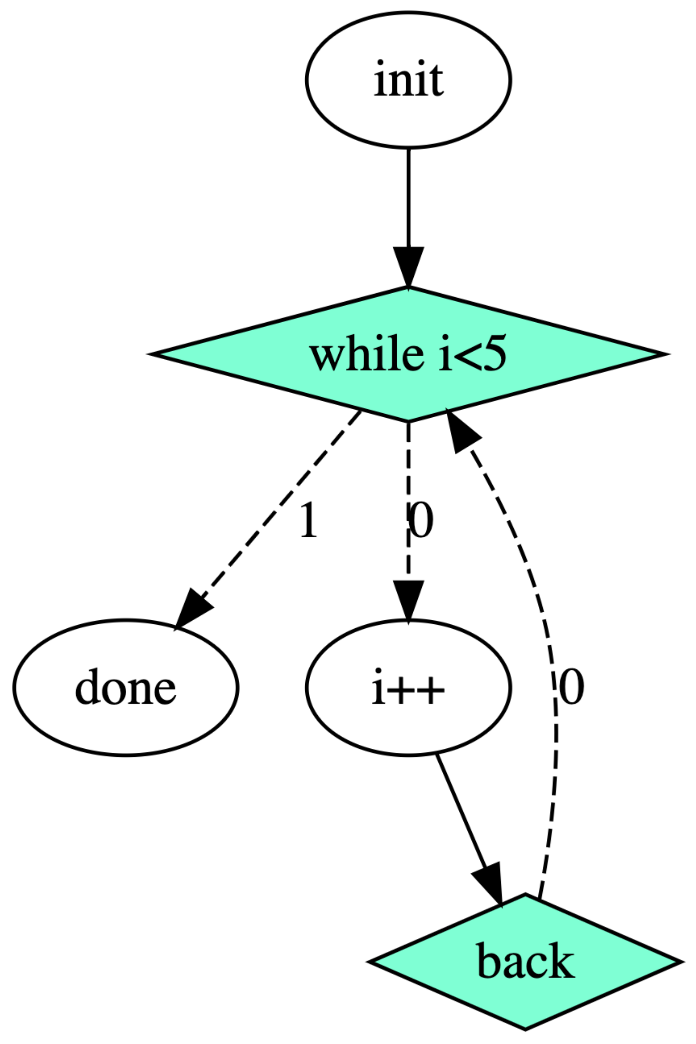
If-Else Control Flow

```
auto init          = taskflow.emplace([&](){ initialize_data_structure(); } )
                              .name(”init");
auto optimizer     = taskflow.emplace([&](){ matrix_solver(); } )
                             .name(”optimizer");
auto converged     = taskflow.emplace([&](){ return converged() ? 1 : 0 } )
                                     .name(”converged");
auto output        = taskflow.emplace([&](){ std::cout << ”done!\n"; } );
                              .name(”output");
init.precede(optimizer);
optimizer.precede(converged);
converged.precede(optimizer, output);  // return 0 to the optimizer again
```



Iterative Control Flow via Cycle

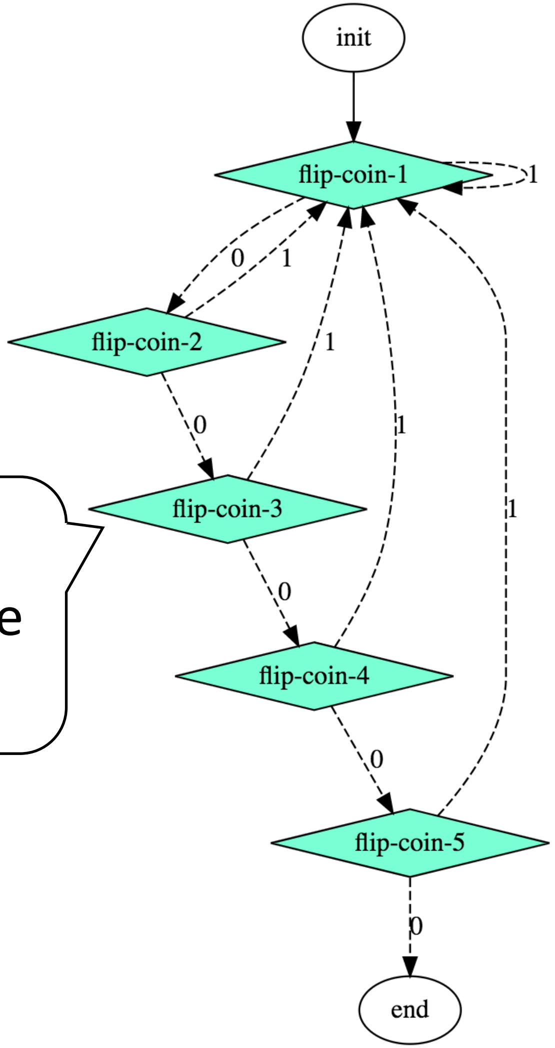
```
tf::Taskflow taskflow;
int i;
auto [init, cond, body, back, done] = taskflow.emplace(
  [&](){ std::cout << “i=0”; i=0; },
  [&](){ std::cout << "while i<5\n"; return i < 5 ? 0 : 1; },
  [&](){ std::cout << "i++=" << i++ << '\n'; },
  [&](){ std::cout << "back\n"; return 0; },
  [&](){ std::cout << "done\n"; }
);
init.precede(cond);
cond.precede(body, done);
body.precede(back);
back.precede(cond);
```



Non-deterministic Control Flow

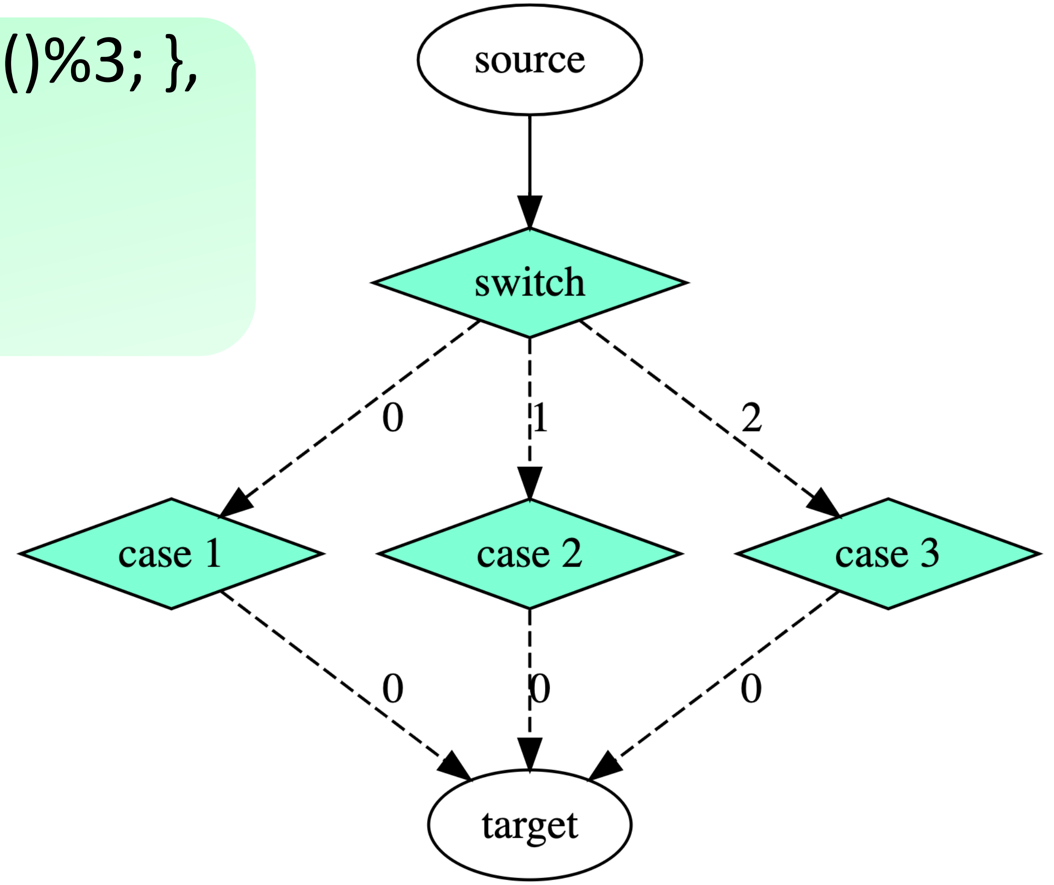
```
auto A = taskflow.emplace([&](){ } );
auto B = taskflow.emplace([&](){ return rand()%2; } );
auto C = taskflow.emplace([&](){ return rand()%2; } );
auto D = taskflow.emplace([&](){ return rand()%2; } );
auto E = taskflow.emplace([&](){ return rand()%2; } );
auto F = taskflow.emplace([&](){ return rand()%2; } );
auto G = taskflow.emplace([&](){});
A.precede(B).name("init");
B.precede(C, B).name("flip-coin-1");
C.precede(D, B).name("flip-coin-2");
D.precede(E, B).name("flip-coin-3");
E.precede(F, B).name("flip-coin-4");
F.precede(G, B).name("flip-coin-5");
G.name("end");
```

Each task flips a binary coin to decide the next path



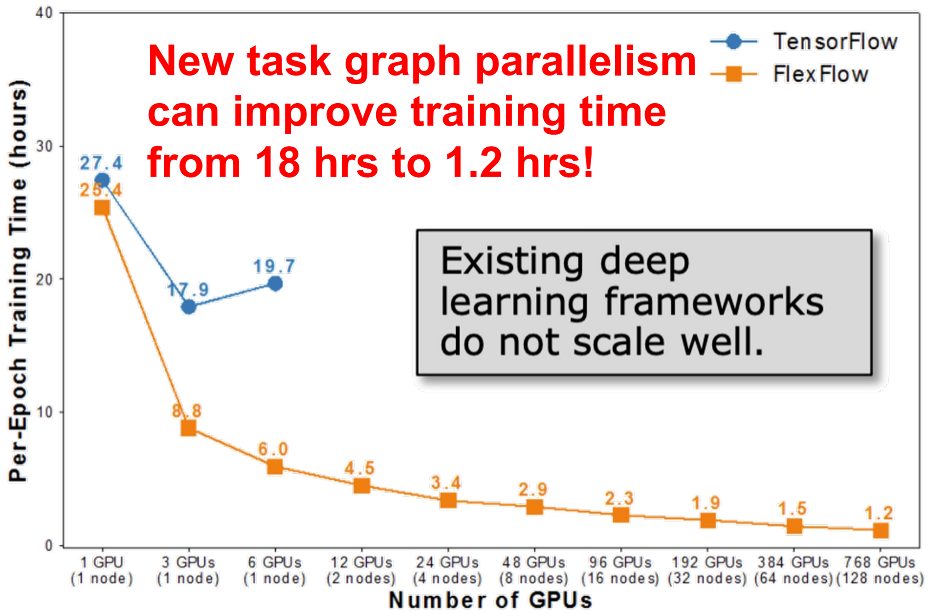
Switch-Case Control Flow

```
auto [source, swcond, case1, case2, case3, target] = taskflow.emplace(
  [](){ std::cout << "source\n"; },
  [](){ std::cout << "switch\n"; return rand()%3; },
  [](){ std::cout << "case 1\n"; return 0; },
  [](){ std::cout << "case 2\n"; return 0; },
  [](){ std::cout << "case 3\n"; return 0; },
  [](){ std::cout << "target\n"; }
);
source.precede(swcond);
swcond.precede(case1, case2, case3);
target.succeed(case1, case2, case3);
```



Existing Frameworks on Control Flow?

- **Expand a task graph across fixed-length iterations**
  - Large graph size linearly proportional to decision points
- **Unknown or non-deterministic iterations?**
  - Expensive dynamic tasks executing “if-else” on the fly
- **Dynamic control-flow tasks?**
  - Client-side partition
- **Same problem in large-scale ML**
  - TensorFlow with RNN (EuroSys’18)
  - FlexFlow (MLSys’19, ICML’18)
  - DGL (CoRR’19)
  - DOE 2022 funding preview (Dr. Finkel)



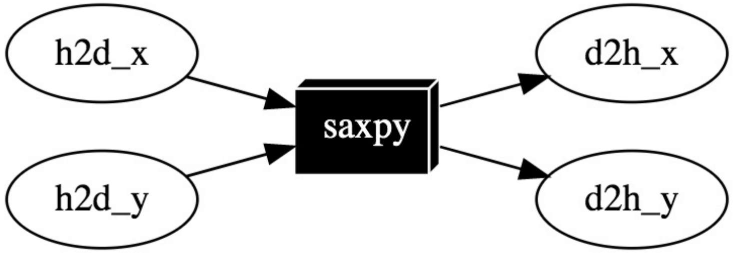
New task graph parallelism can improve training time from 18 hrs to 1.2 hrs!

Existing deep learning frameworks do not scale well.

Heterogeneous Tasking

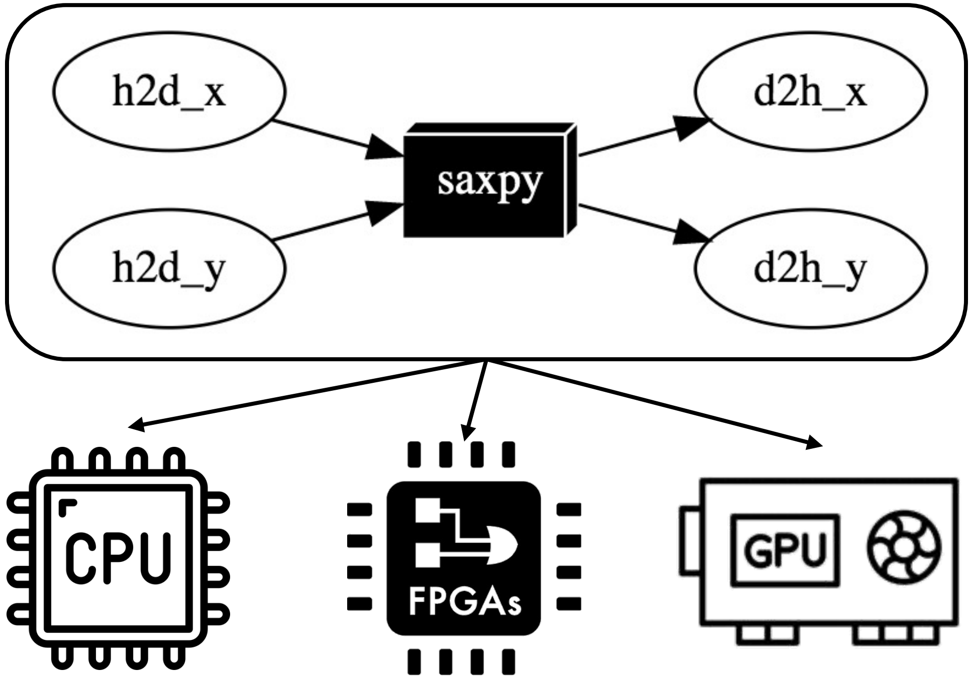
```
const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};
auto allocate_x = taskflow.emplace([&](){ cudaMalloc(&dx, 4*N);});
auto allocate_y = taskflow.emplace([&](){ cudaMalloc(&dy, 4*N);});

auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
   auto h2d_x = cf.copy(dx, hx.data(), N);  // CPU-GPU data transfer
   auto h2d_y = cf.copy(dy, hy.data(), N);
   auto d2h_x = cf.copy(hx.data(), dx, N);  // GPU-CPU data transfer
   auto d2h_y = cf.copy(hy.data(), dy, N);
   auto kernel = cf.kernel((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
   kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});

cudaflow.succeed(allocate_x, allocate_y);
executor.run(taskflow).wait();
```

cudaFlow automatically transforms an application GPU task graph to an optimized **"CUDA graph"**



Heterogeneous Tasking – SYCL

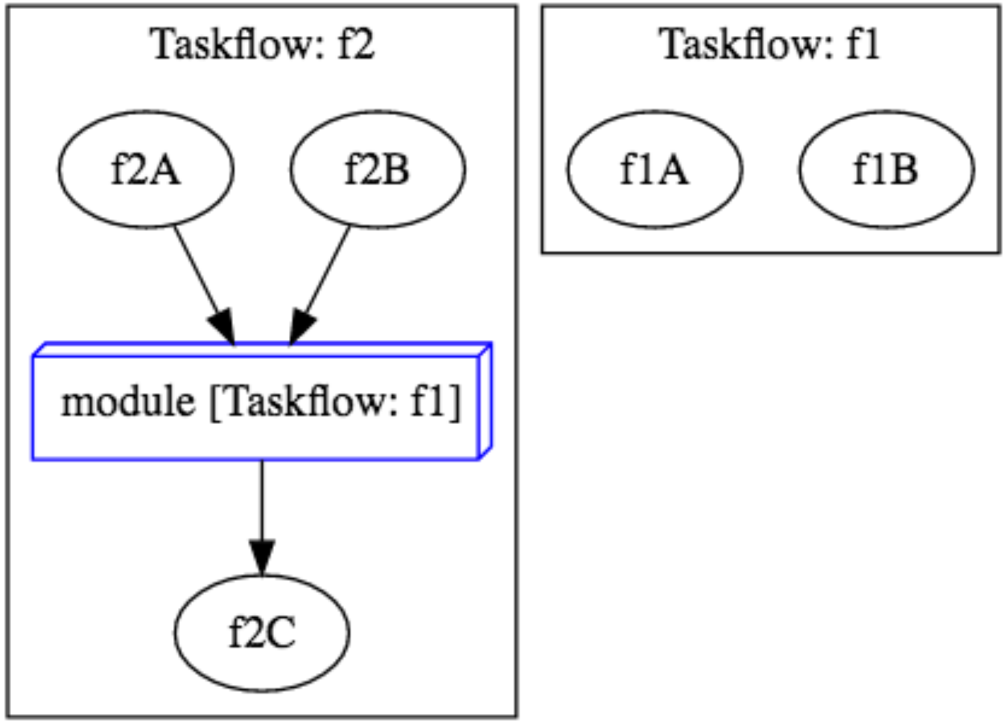
```
// leverage SYCL to enable single-source heterogeneous programming using completely C++
taskflow.emplace_on([&](tf::syclFlow& sf) {
  auto h2d_x = cf.copy(dx, hx.data(), N);
  auto h2d_y = cf.copy(dy, hy.data(), N);
  auto d2h_x = cf.copy(hx.data(), dx, N);
  auto d2h_y = cf.copy(hy.data(), dy, N);
  auto kernel = cf.parallel_for(
    sycl::range<1>(N), [=](sycl::id<1> id){
      dx[id] = 2.0f * dx[id] + dy[id];
    }
  );
  kernel.succeed(h2d_x, h2d_y)
        .precede(d2h_x, d2h_y);
}, queue);
```



SYCL-aware compilers create kernel executables to run on arbitrary architectures

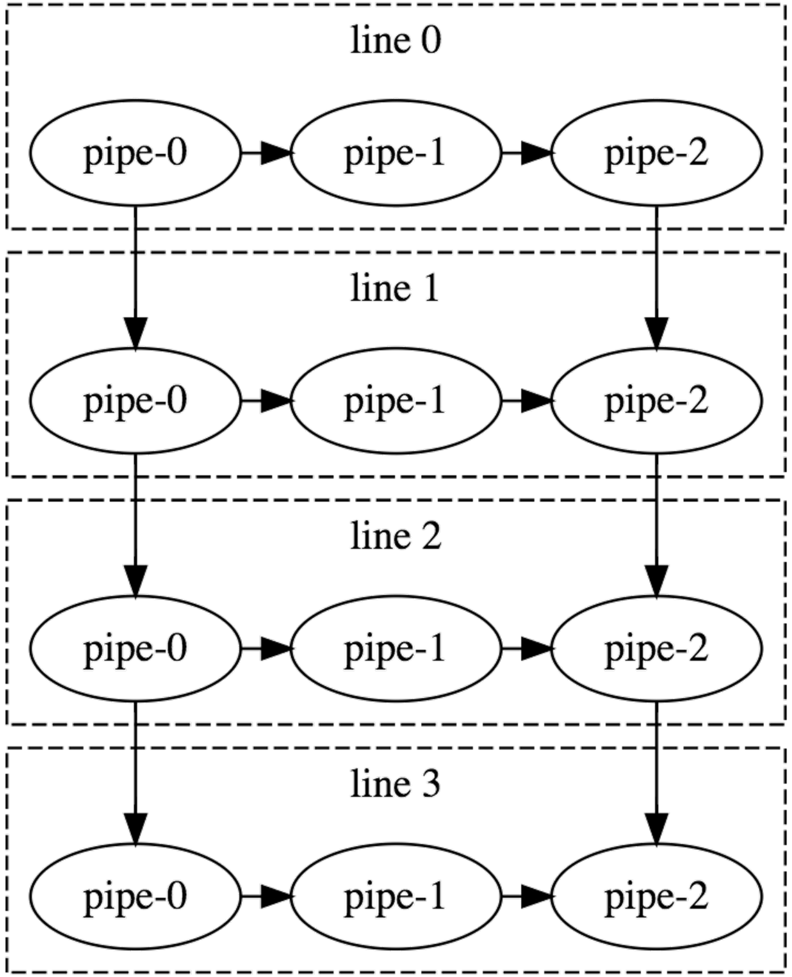
Composable Tasking

```
tf::Taskflow f1, f2;
auto [f1A, f1B] = f1.emplace(
  []() { std::cout << "Task f1A\n"; },
  []() { std::cout << "Task f1B\n"; }
);
auto [f2A, f2B, f2C] = f2.emplace(
  []() { std::cout << "Task f2A\n"; },
  []() { std::cout << "Task f2B\n"; },
  []() { std::cout << "Task f2C\n"; }
);
auto f1_module_task = f2.composed_of(f1);
f1_module_task.succeed(f2A, f2B)
                         .precede(f2C);
```



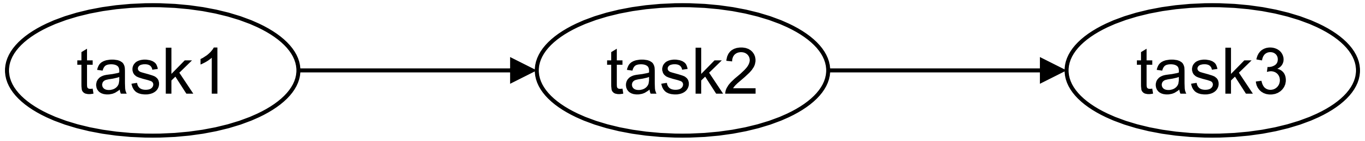
Task-parallel Pipeline

```
std::array <int, 4> buffer;
tf::Pipeline pl(4,
  tf::Pipe {tf::PipeType::SERIAL, [&buffer](tf::Pipeflow & pf) {
      if (pf.token() == 5) {
        pf.stop();
        return;
      }
      buffer[pf.line()] = pf.token();
  }},
  tf::Pipe {tf::PipeType::PARALLEL, [&buffer](tf::Pipeflow & pf) {
    buffer[pf.line()] = buffer[pf.line()] + 1;
  }},
  tf::Pipe {tf::PipeType::SERIAL, [&buffer](tf::Pipeflow & pf) {
    buffer[pf.line()] = buffer[pf.line()] + 1;
  }}
);
auto task = taskflow.composed_of(pl);
executor.run(taskflow).wait();
```



Standard Algorithms

```
// parallel iterations over a range of items
auto task1 = taskflow.for_each(first, last, [](auto i){ std::cout << “item” << i; });

// parallel reduction/summation over a range of items
auto task2 = taskflow.reduce(first, last, init, [](auto i, auto j){ return i + j; });

// parallel sort over a range of items
auto task3 = taskflow.sort(first, last, [](auto i, auto j){ return a < b; });

// build up dependencies for these algorithm tasks
task1.precede(task2);
task2.precede(task3);
```

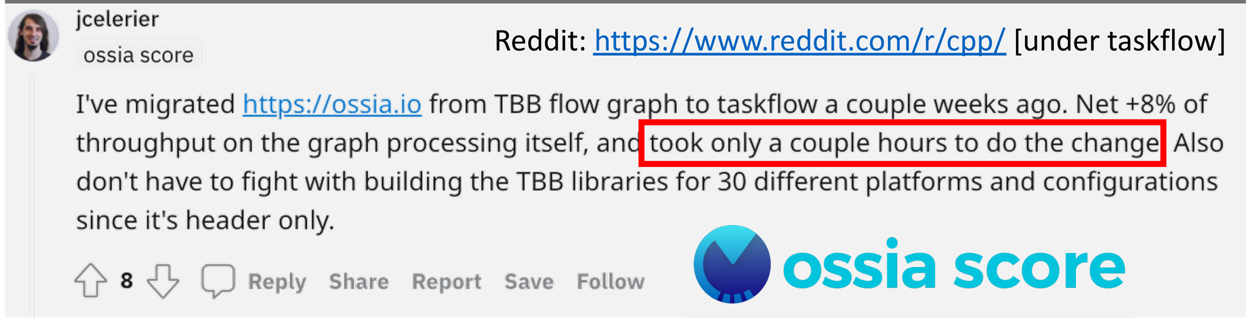


Everything is Composable in Taskflow

- **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!**

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

Industrial use-case of productivity improvement using Taskflow



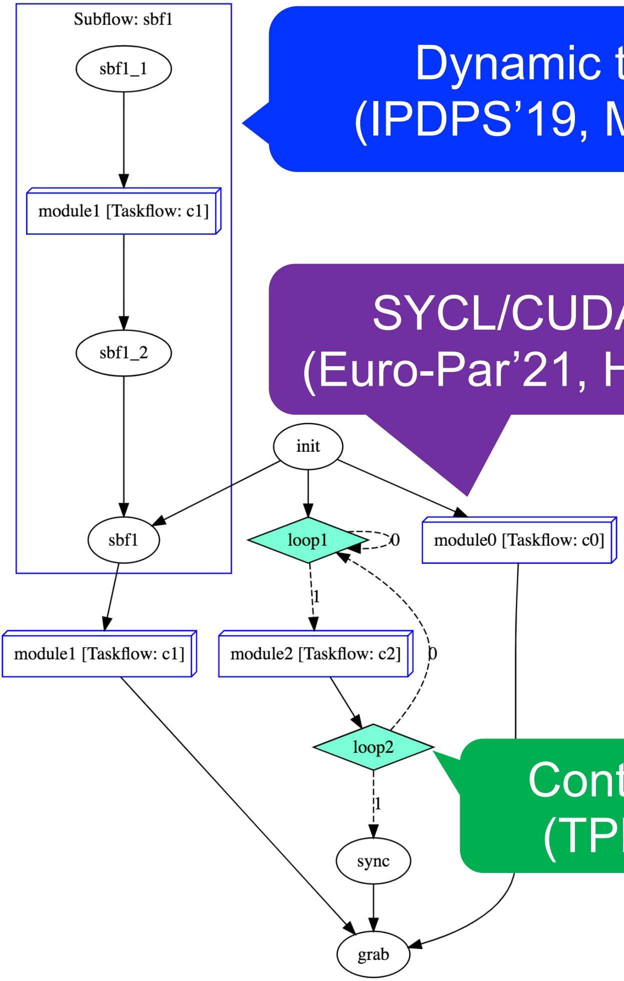
jcelerier
ossia score

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



Dynamic task (IPDPS'19, MM'19)

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

Control flow (TPDS'22)

We Value Research Impacts for Sustainability

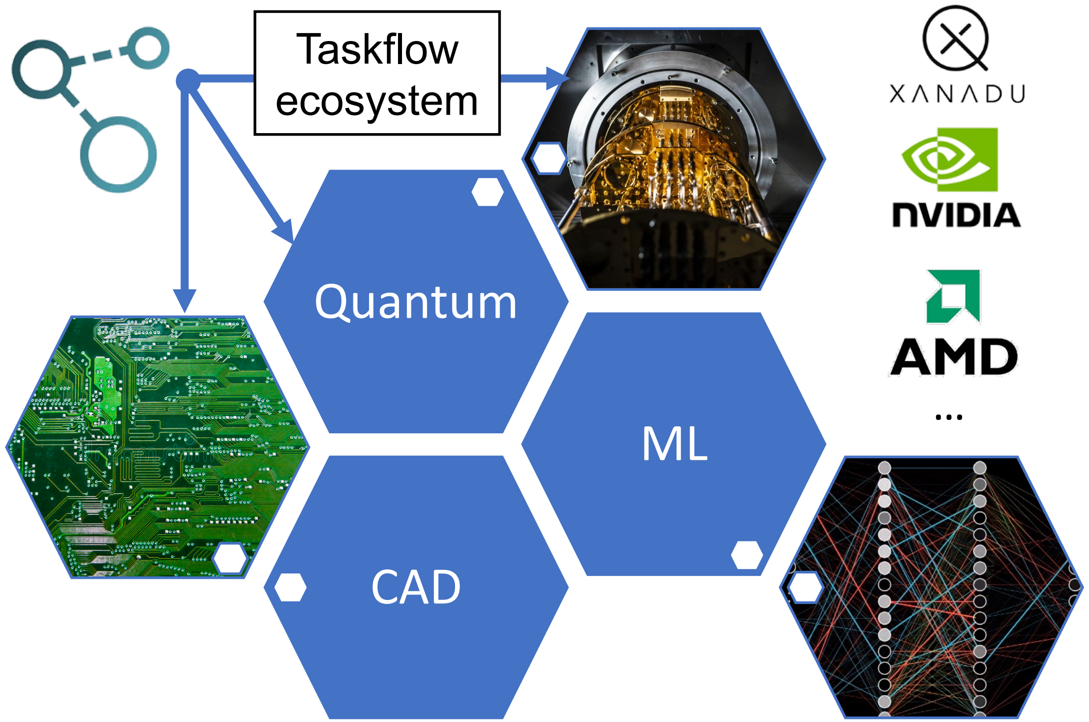
- **Taskflow**[1] has been downloaded thousands of times

OT
ossia
deal.II
MYDATAMODELS
XANADU
LiveHD
ROS industrial
REVEAL
SNIG
Inference Engine
24h
OPERON
NVIDIA
Explosion
METHANE
powered graphics
Chainblocks
Rapid Fuzz
JET BRAINS
RESEARCH
atomicDEX
ReAgent
RAPIDS
XILINX

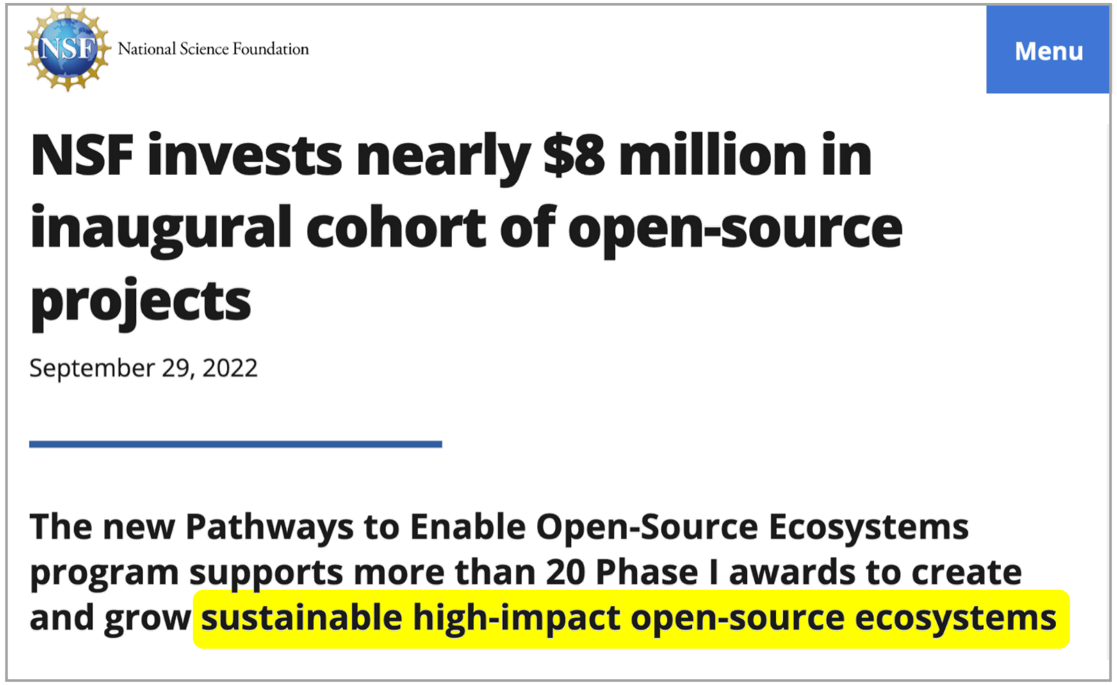
[1]: Tsung-Wei Huang, et al., "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," IEEE TPDS, vol. 33, no. 6, pp. 1303-1320, June 2022

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



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

Agenda

- Understand the challenges of parallel computing
- Introduce our new task-parallel programming system
- Dive into our system runtime
- Apply our system to computer engineering problems

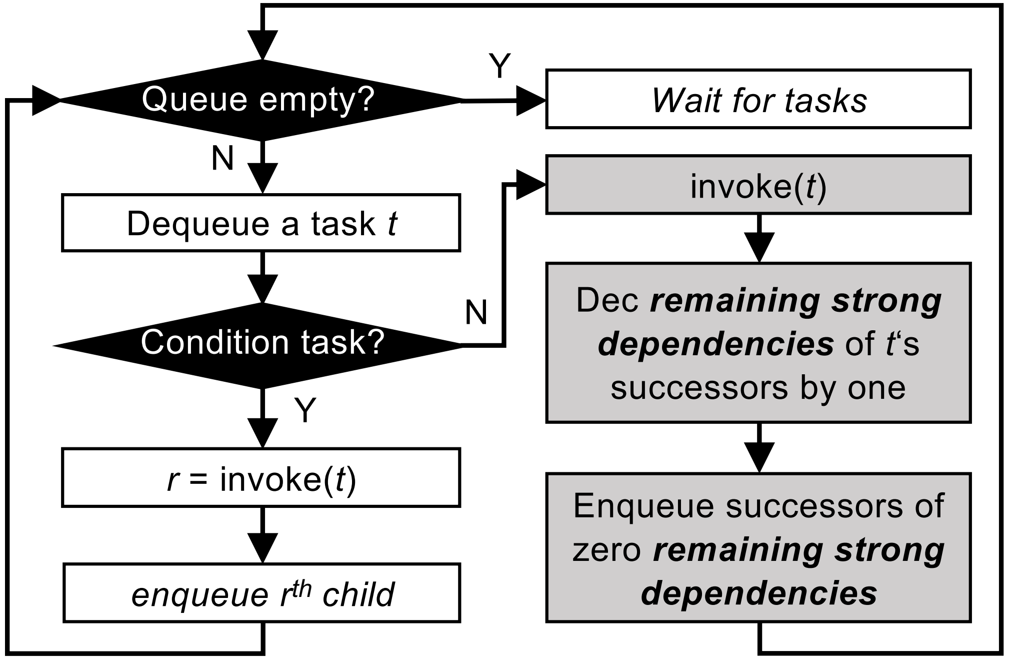
Submit a Taskflow to Executor

- **Executor manages a set of threads to run a taskflow**
  - All execution methods are *non-blocking*
  - All execution methods are *thread-safe*

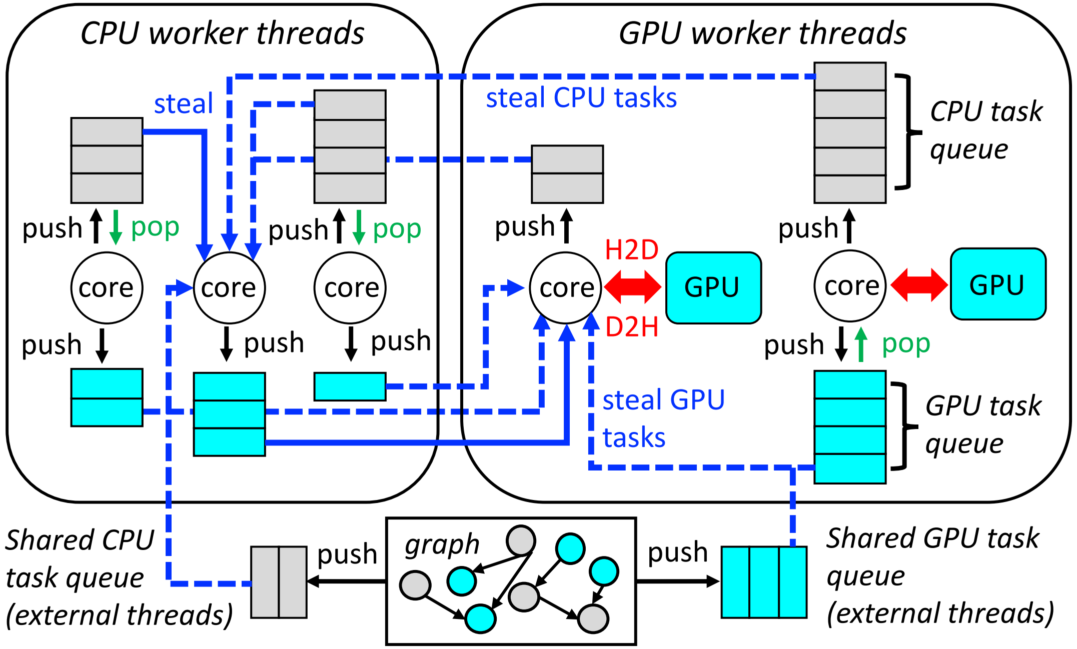
```
{
  tf::Taskflow taskflow1, taskflow2, taskflow3;
  tf::Executor executor;
  // create tasks and dependencies
  // …
  auto  future1 = executor.run(taskflow1);
  auto  future2 = executor.run_n(taskflow2, 1000);
  auto  future3 = executor.run_until(taskflow3, [i=0](){ return i++ > 5; });
  executor.async([](){ std::cout << “async task\n"; });
  executor.wait_for_all(); // wait for all the above tasks to finish
}
```

Taskflow Runtime (ICPADS'20, TPDS'22)

- **Task-level scheduling**
- **Worker-level scheduling**



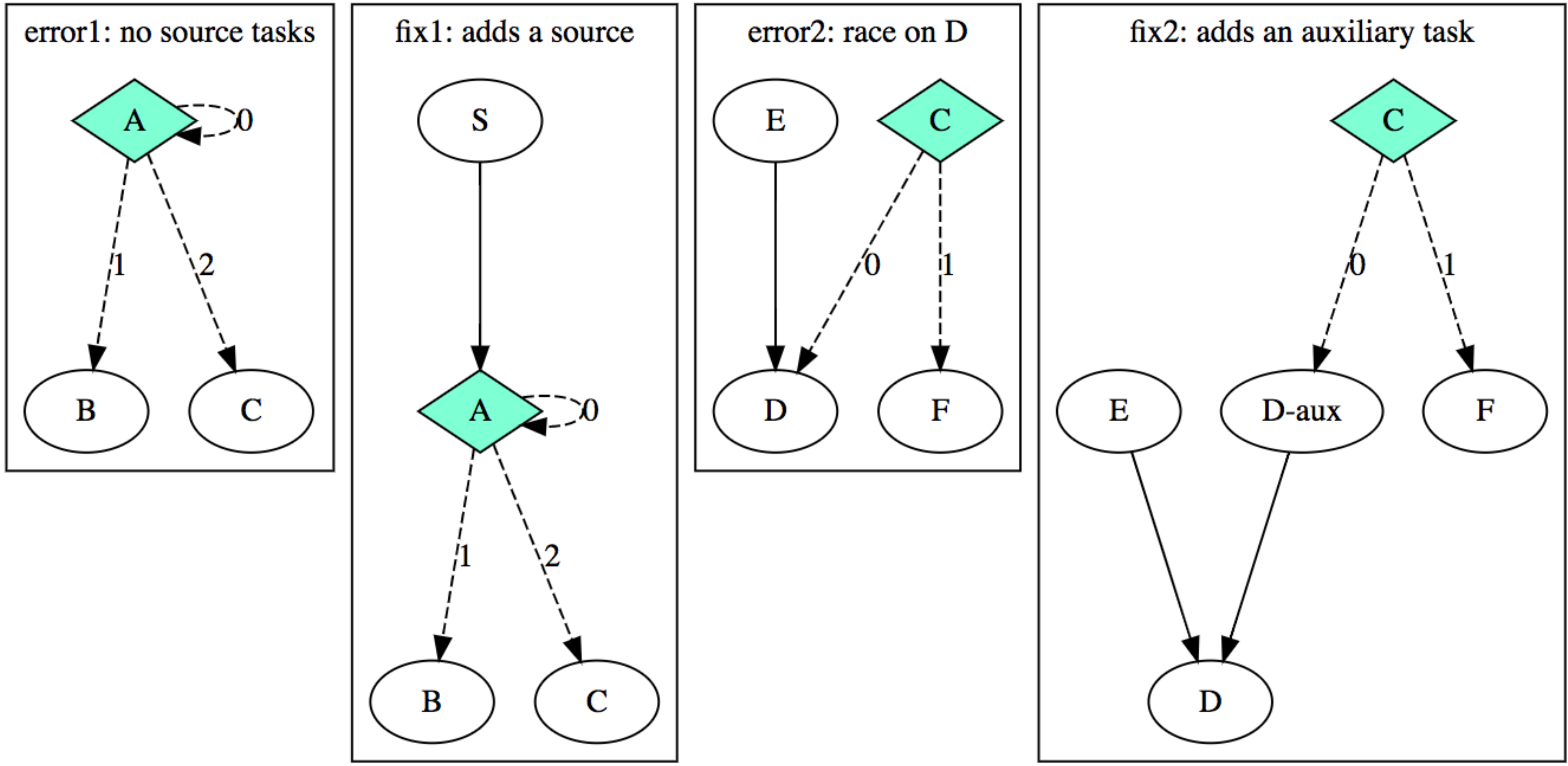
Queue empty?
Y
Wait for tasks
N
Dequeue a task *t*
invoke(*t*)
N
Condition task?
Dec ***remaining strong dependencies*** of *t*'s successors by one
Y
r = invoke(*t*)
Enqueue successors of zero ***remaining strong dependencies***
enqueue r^{th} child



Key results: schedule tasks with in-graph control flow with a strong balance between the number of active workers and dynamically generated tasks – generalized to any heterogeneous domains

Task-level Scheduling Pitfalls …

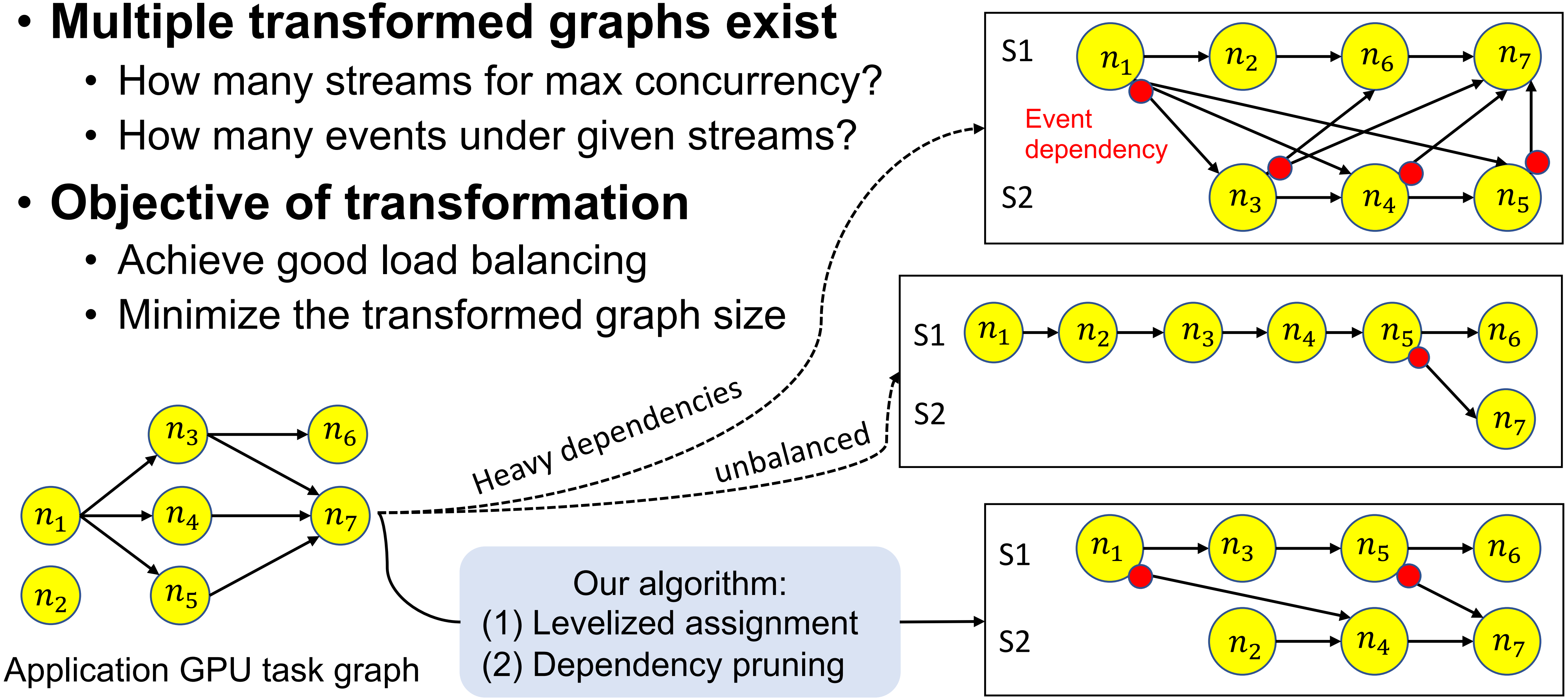
- **Condition task is powerful but prone to mistakes!**



It is users' responsibility to ensure a taskflow is properly conditioned, i.e., no task race under our task-level scheduling policy

GPU Task Graph Scheduling (EuroPar'21)

- **Multiple transformed graphs exist**
  - How many streams for max concurrency?
  - How many events under given streams?
- **Objective of transformation**
  - Achieve good load balancing
  - Minimize the transformed graph size



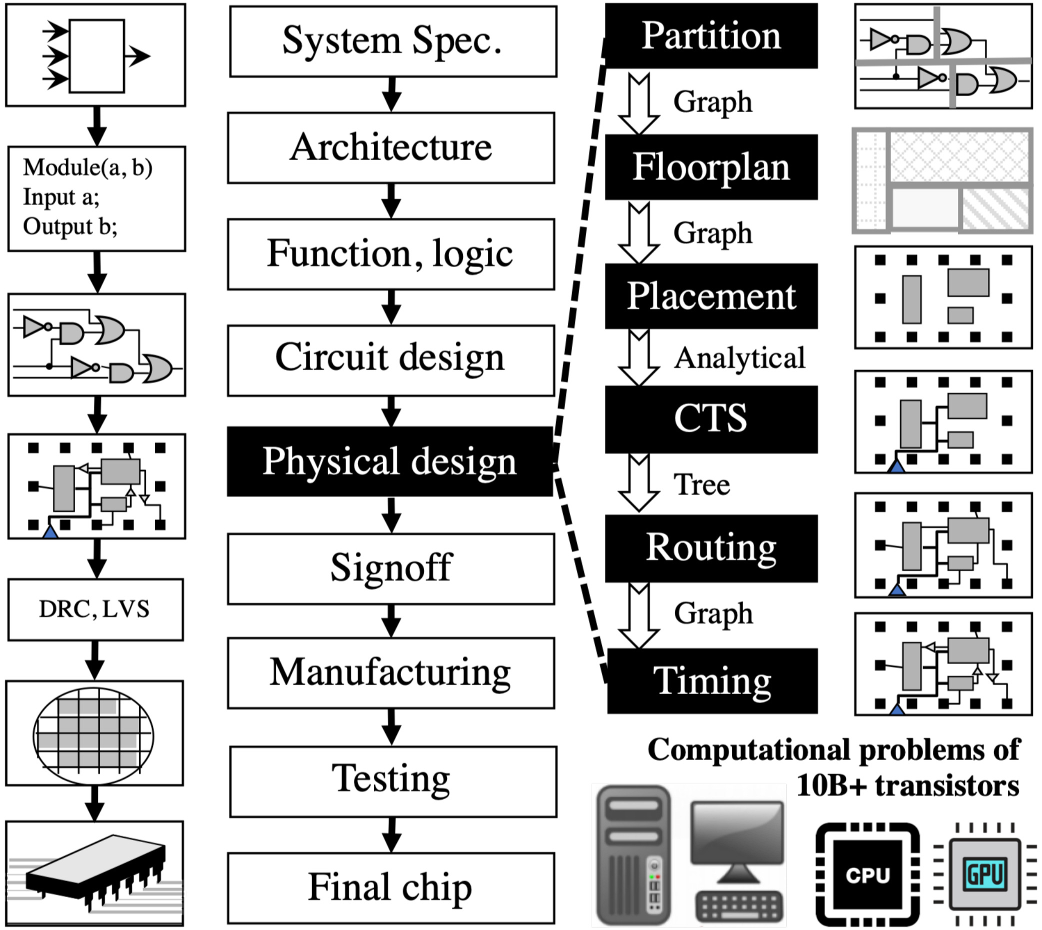
Application GPU task graph

Our algorithm:
(1) Levelized assignment
(2) Dependency pruning

Agenda

- Understand the challenges of parallel computing
- Introduce our new task-parallel programming system
- Dive into our system runtime
- Apply our system to computer engineering problems

Our NSF CCF Project[1]: Parallelizing CAD



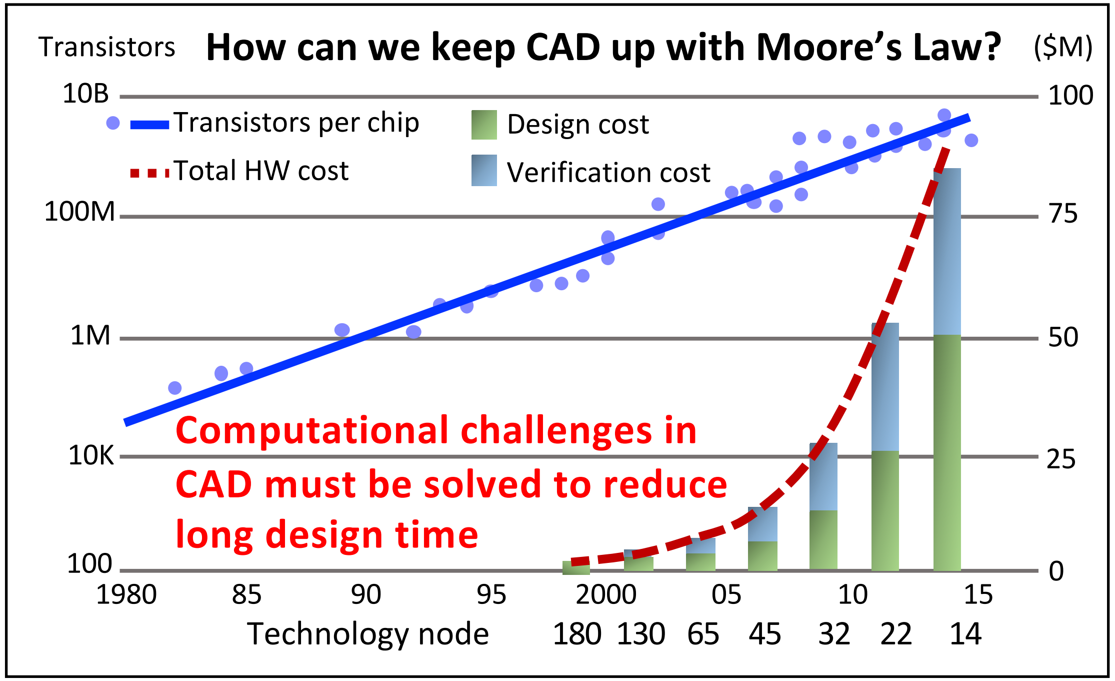
Module(a, b)
Input a;
Output b;

DRC, LVS

System Spec. → Architecture → Function, logic → Circuit design → Physical design → Signoff → Manufacturing → Testing → Final chip

Partition → Graph → Floorplan → Graph → Placement → Analytical → CTS → Tree → Routing → Graph → Timing

Computational problems of 10B+ transistors



How can we keep CAD up with Moore's Law?

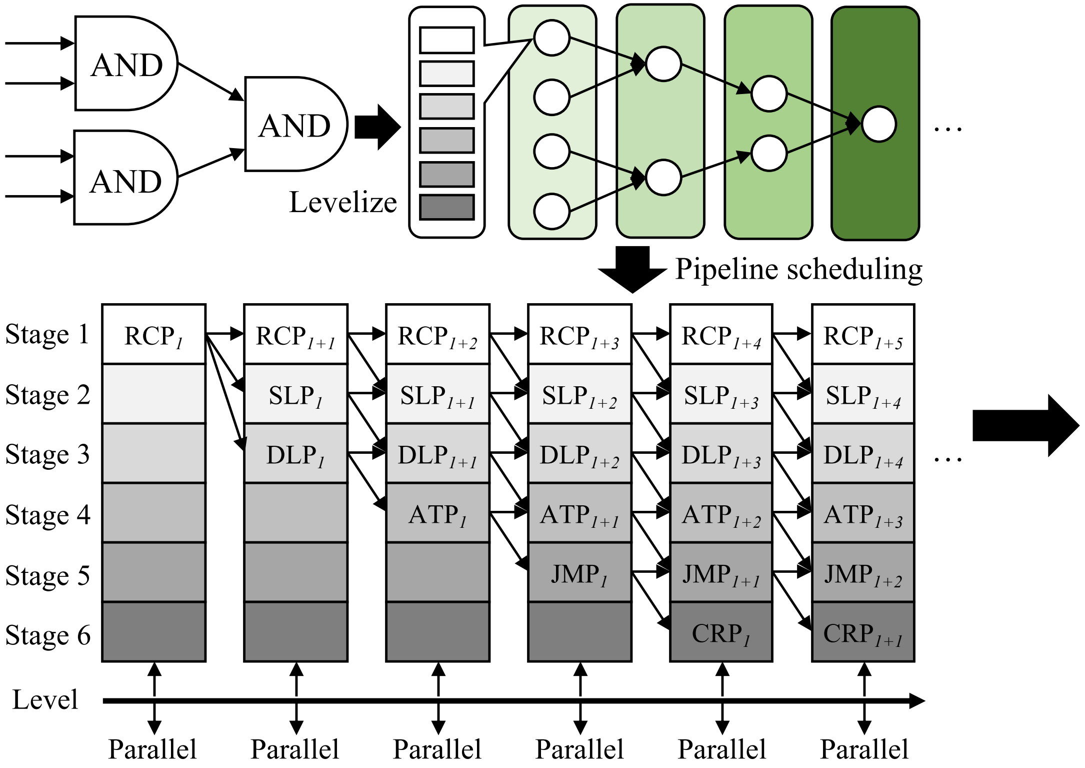
Computational challenges in CAD must be solved to reduce long design time

DARPA Electronic Resurgence Initiative (ERI): https://eri-summit.darpa.mil/

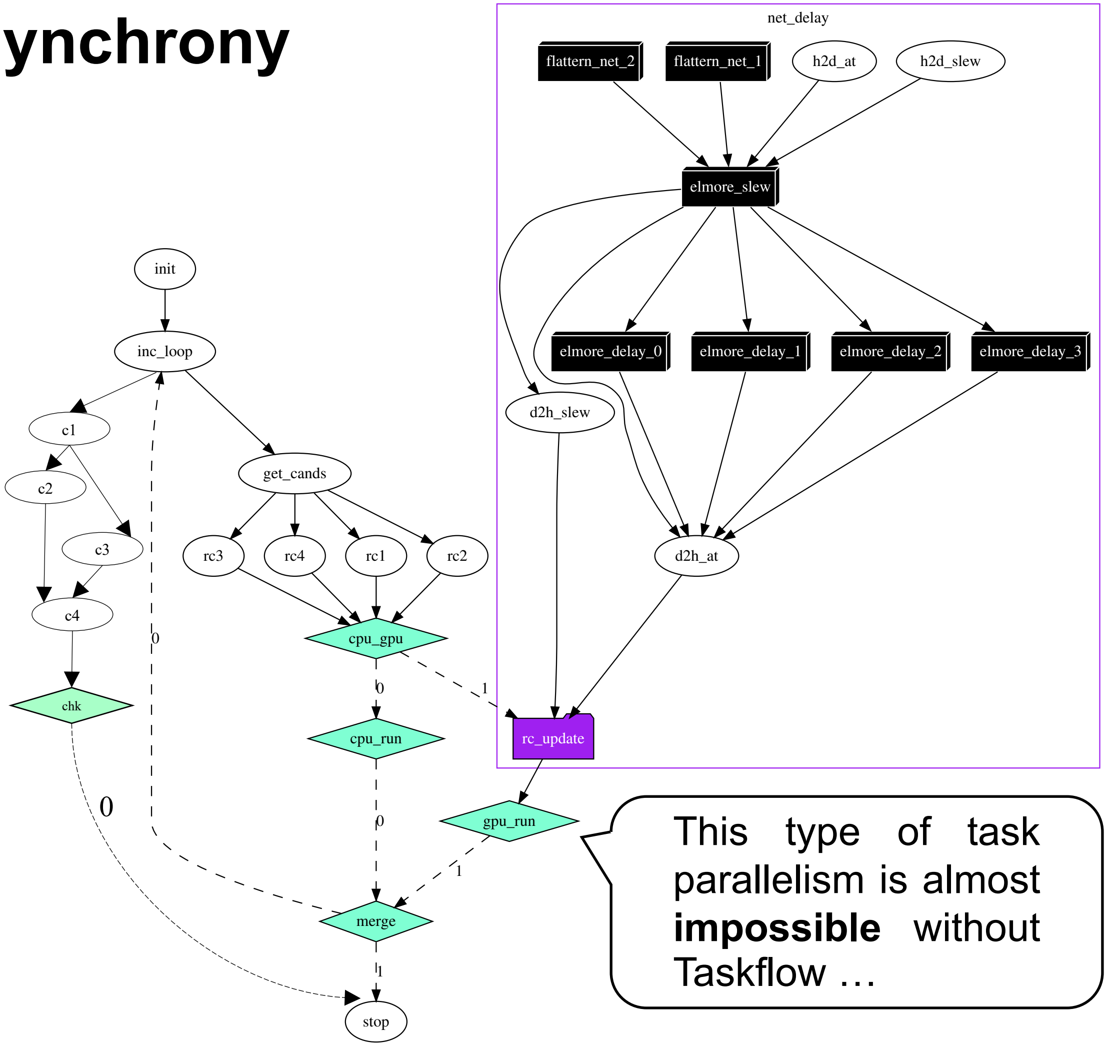
[1]: "A General-purpose Parallel and Heterogeneous Task Graph Computing System for VLSI CAD," $403K, 10/2021—10/2024, NSF CISE, CCF-2126672

Case Study 1: Timing Analysis (TCAD’21)

- **Taskflow largely improves task asynchrony**



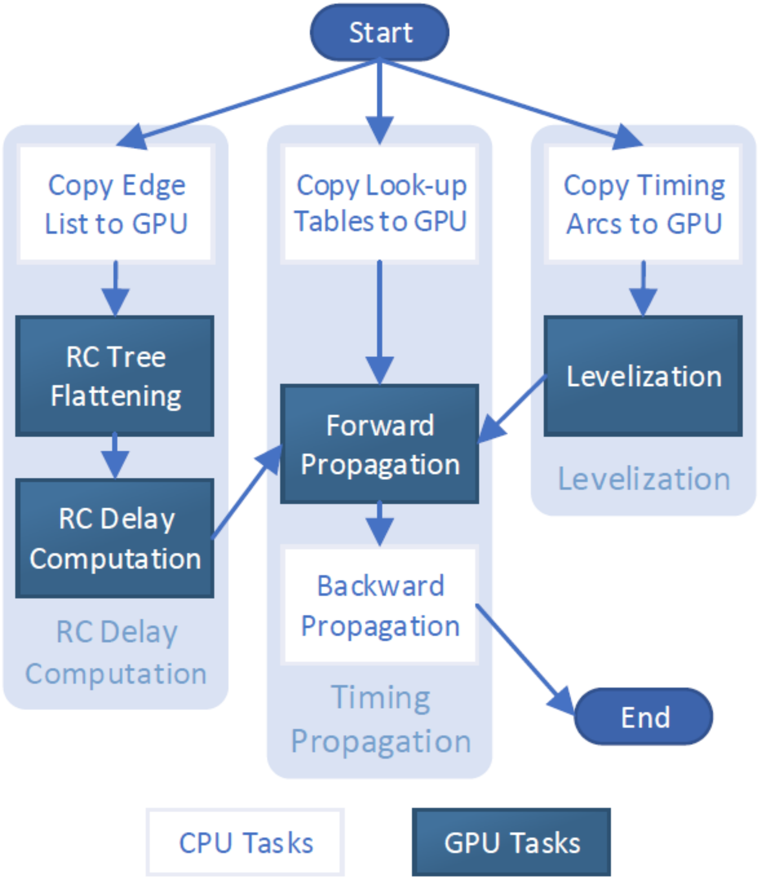
Stage 1	RCP_l	RCP_{l+1}	RCP_{l+2}	RCP_{l+3}	RCP_{l+4}	RCP_{l+5}
Stage 2		SLP_l	SLP_{l+1}	SLP_{l+2}	SLP_{l+3}	SLP_{l+4}
Stage 3		DLP_l	DLP_{l+1}	DLP_{l+2}	DLP_{l+3}	DLP_{l+4}
Stage 4			ATP_l	ATP_{l+1}	ATP_{l+2}	ATP_{l+3}
Stage 5				JMP_l	JMP_{l+1}	JMP_{l+2}
Stage 6					CRP_l	CRP_{l+1}



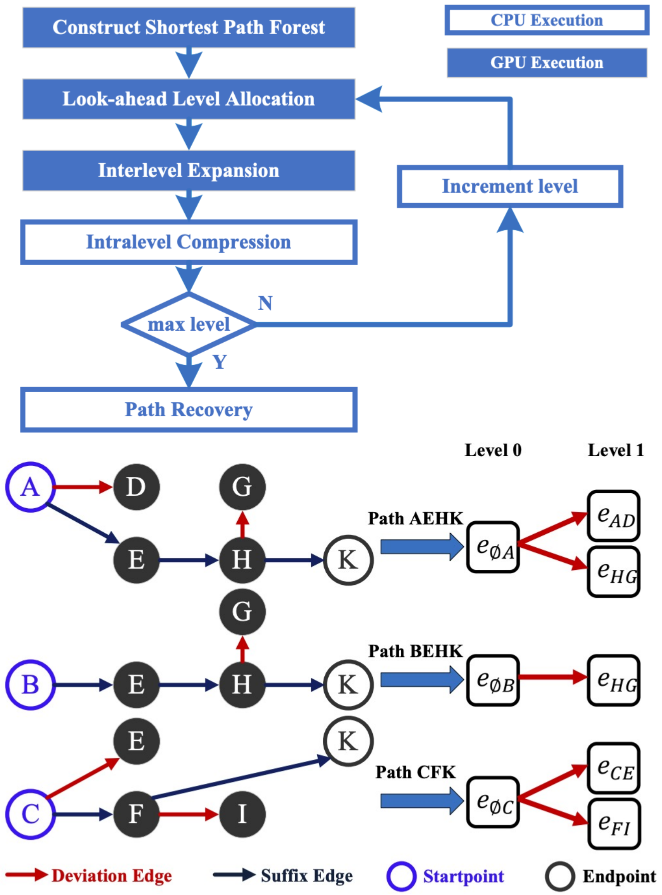
This type of task parallelism is almost **impossible** without Taskflow …

Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2021

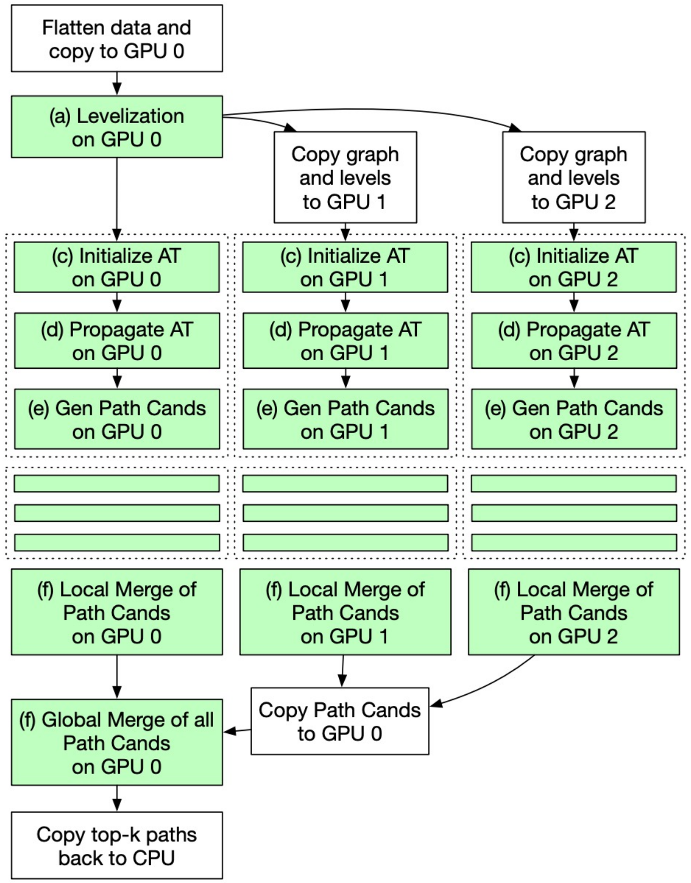
Case Study 1: Timing Analysis (cont'd)



GPU-based graph analysis (ICCAD'20)



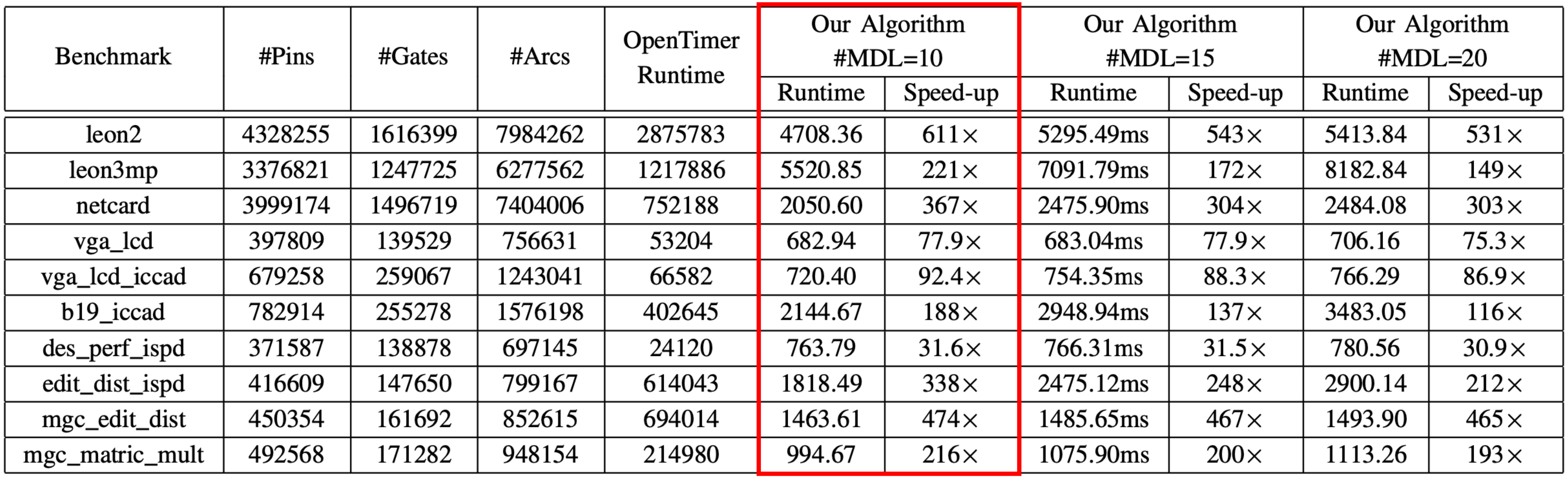
GPU-based path analysis (DAC'21)


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)

Case Study 1: Timing Analysis (DAC’21)

- **Applied Taskflow to accelerate path-based analysis on GPU**
  - Ex: leon3mp (1.6M gates): **611x speed-up** over 1 CPU (**44x** over 40 CPUs)
  - **Best paper award in TAU 2021**

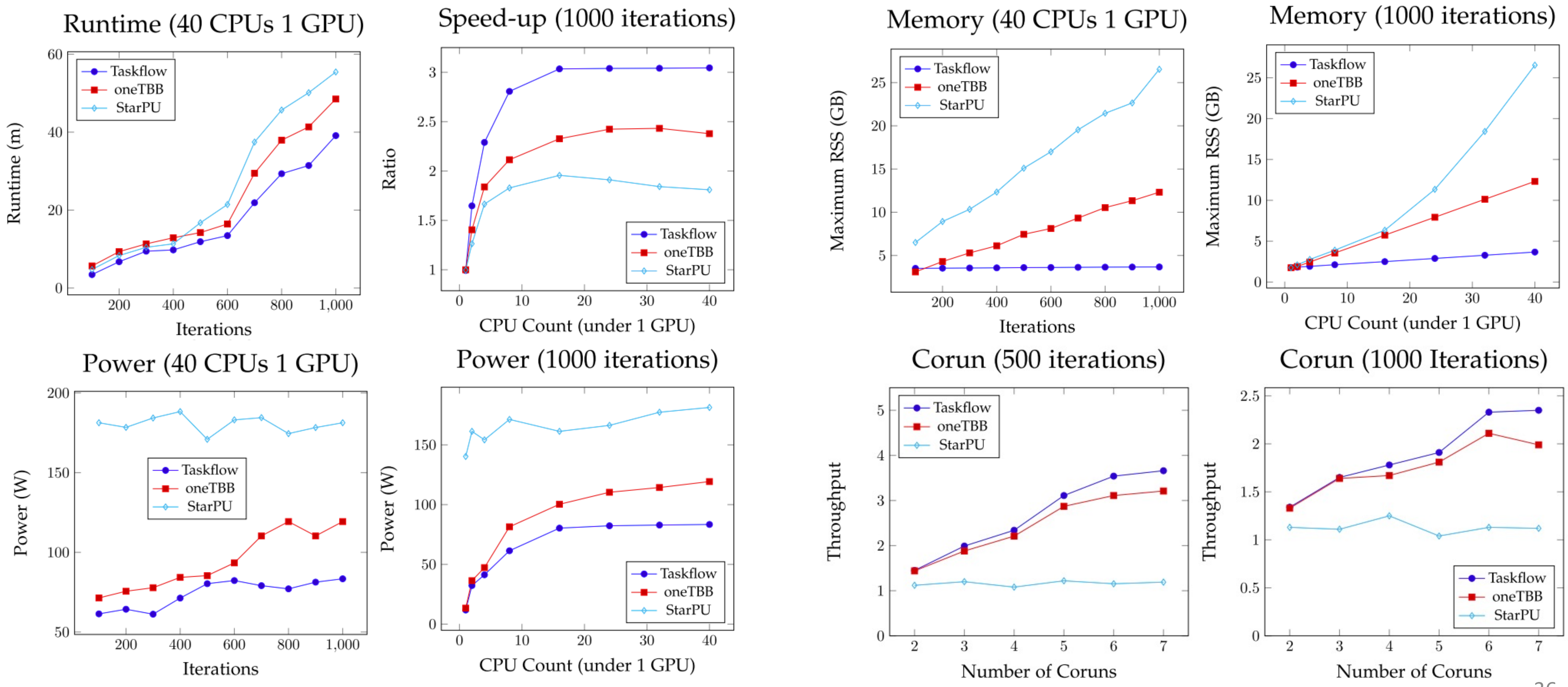


Benchmark	#Pins	#Gates	#Arcs	OpenTimer Runtime	Our Algorithm #MDL=10		Our Algorithm #MDL=15		Our Algorithm #MDL=20	
					Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	7984262	2875783	4708.36	611×	5295.49ms	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×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

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

Case Study 1: Timing Analysis (cont’d)

- **Comparison to existing high-performance computing systems**



Runtime (40 CPUs 1 GPU)

Speed-up (1000 iterations)

Memory (40 CPUs 1 GPU)

Memory (1000 iterations)

Power (40 CPUs 1 GPU)

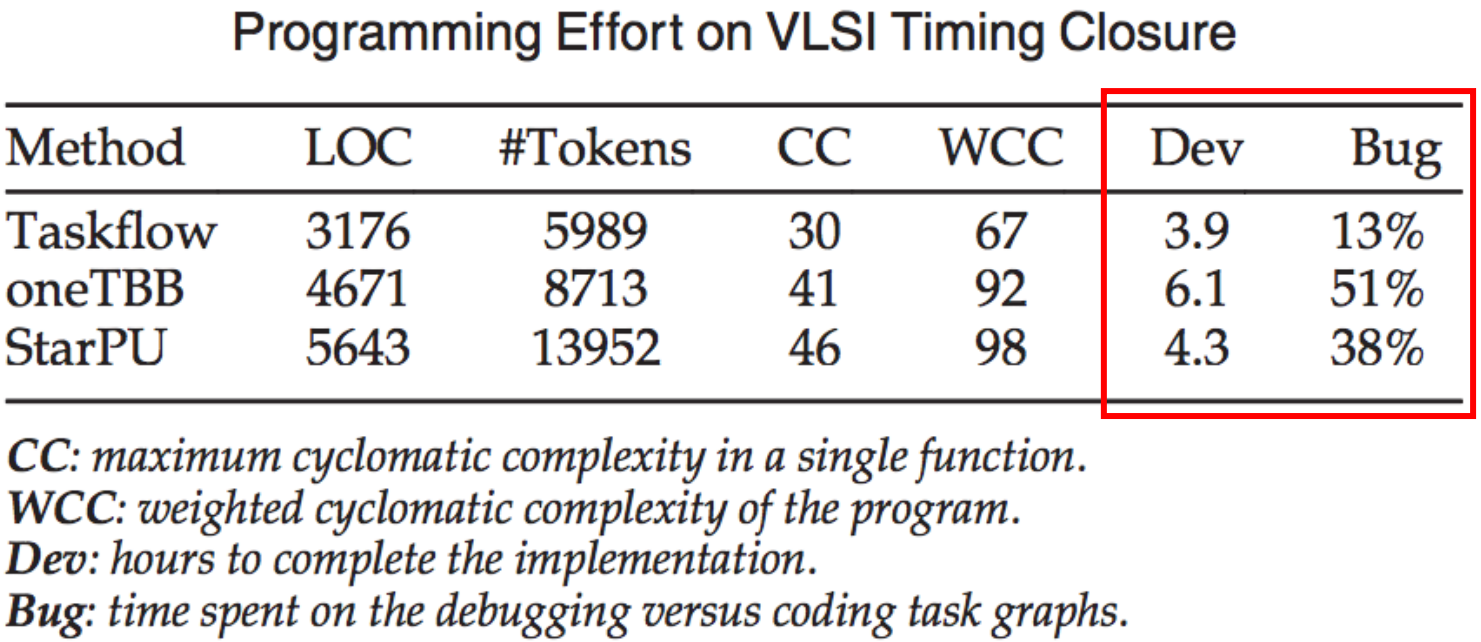
Power (1000 iterations)

Corun (500 iterations)

Corun (1000 Iterations)

Case Study 1: Timing Analysis (cont’d)

- **Implement a task-parallel VLSI timing analysis workload**
  - Taskflow vs industrial HPC systems (oneTBB and OpenMP)
  - Testimonials (10 ECE/CS PhD) have no prior background with Taskflow
  - Testimonials have OK knowledge about heterogeneous parallelism



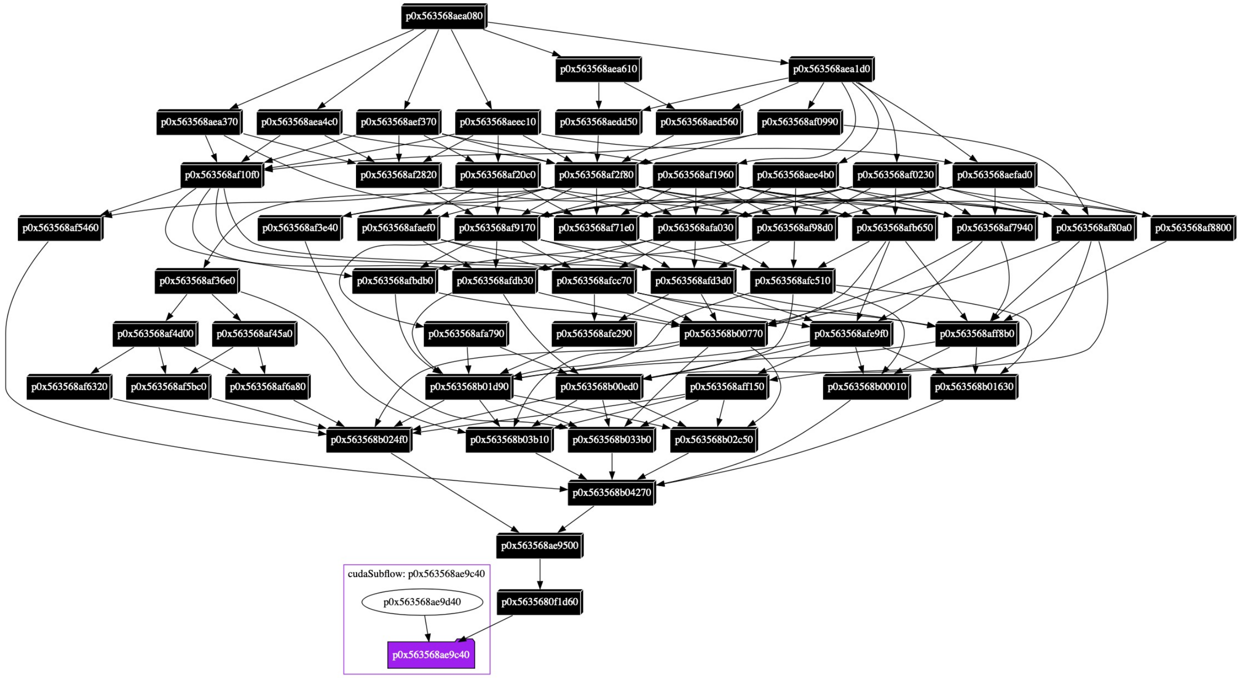
Programming Effort on VLSI Timing Closure

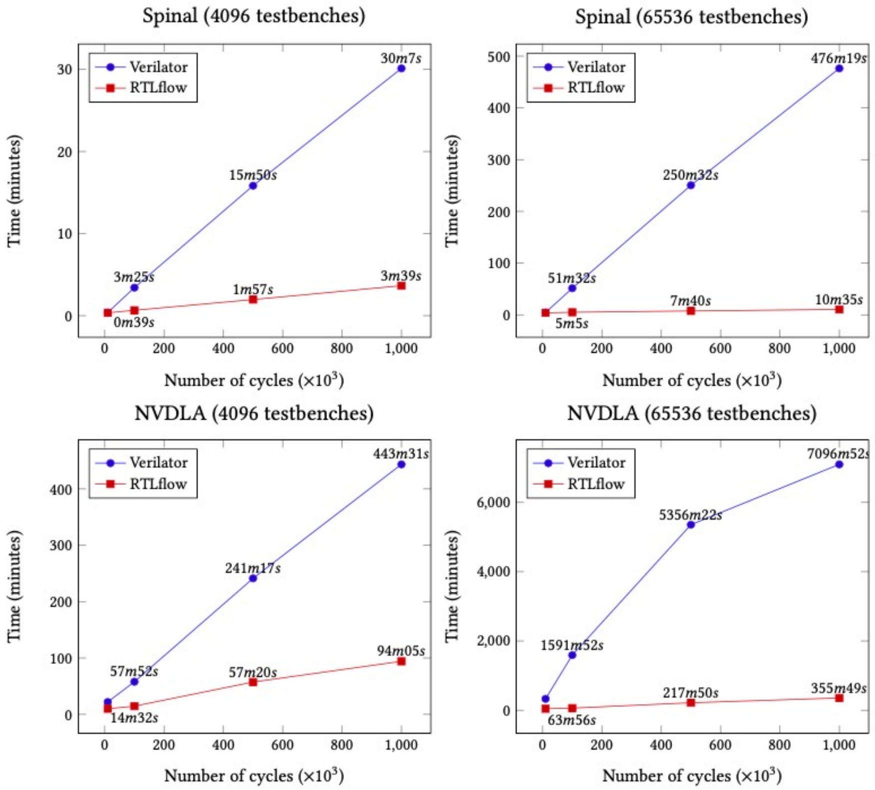
Method	LOC	#Tokens	CC	WCC	Dev	Bug
Taskflow	3176	5989	30	67	3.9	13%
oneTBB	4671	8713	41	92	6.1	51%
StarPU	5643	13952	46	98	4.3	38%

***CC**: maximum cyclomatic complexity in a single function.*
***WCC**: weighted cyclomatic complexity of the program.*
***Dev**: hours to complete the implementation.*
***Bug**: time spent on the debugging versus coding task graphs.*

Case Study 2: RTL Simulation

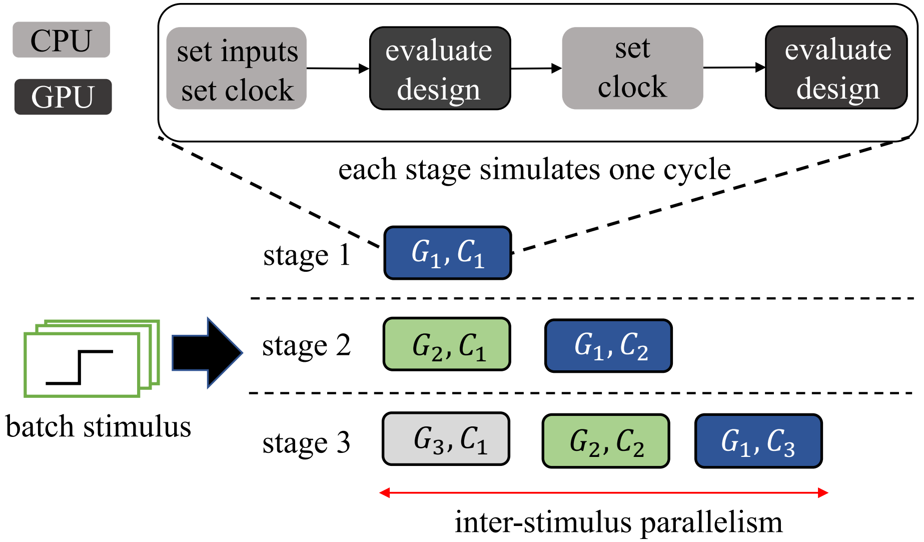
- **Leverage task graph and pipeline parallelisms (i.e., RTLflow)**
  - **10–500x** faster over existing RTL simulator for multiple simulation batches

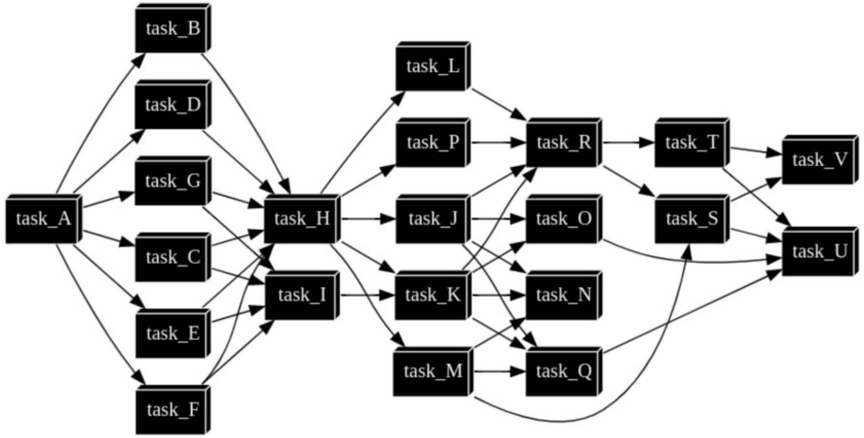


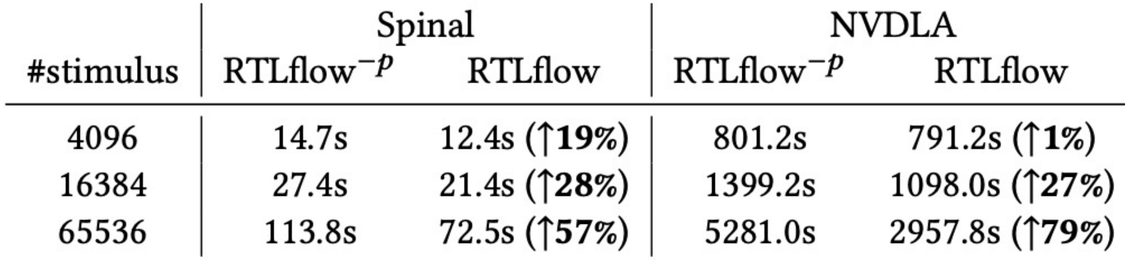


Dian-Lun Lin, et al, “From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus,” *ACM ICPP*, Bordeaux, France, 2022

Case Study 2: RTL Simulation (cont'd)

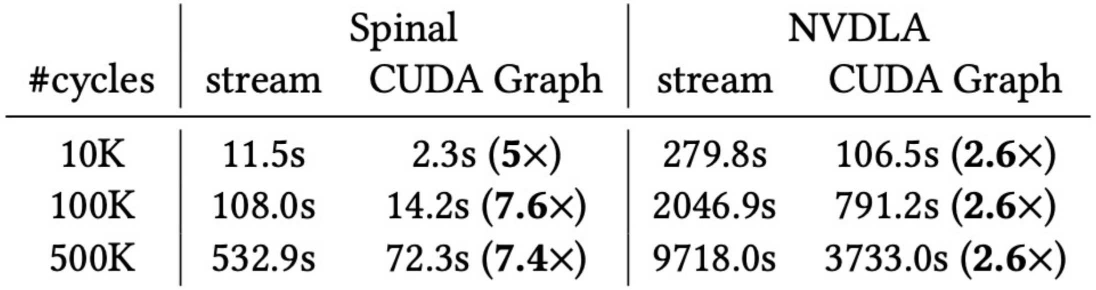






#stimulus	Spinal		NVDLA	
	RTLflow^{-p}	RTLflow	RTLflow^{-p}	RTLflow
4096	14.7s	12.4s (↑**19%**)	801.2s	791.2s (↑**1%**)
16384	27.4s	21.4s (↑**28%**)	1399.2s	1098.0s (↑**27%**)
65536	113.8s	72.5s (↑**57%**)	5281.0s	2957.8s (↑**79%**)

Table 5: Runtime comparison in terms of improvement (↑) between RTLflow with and without pipeline scheduling (RTLflow^{-p}) for Spinal and NVDLA with 100K cycles at different numbers of stimulus.

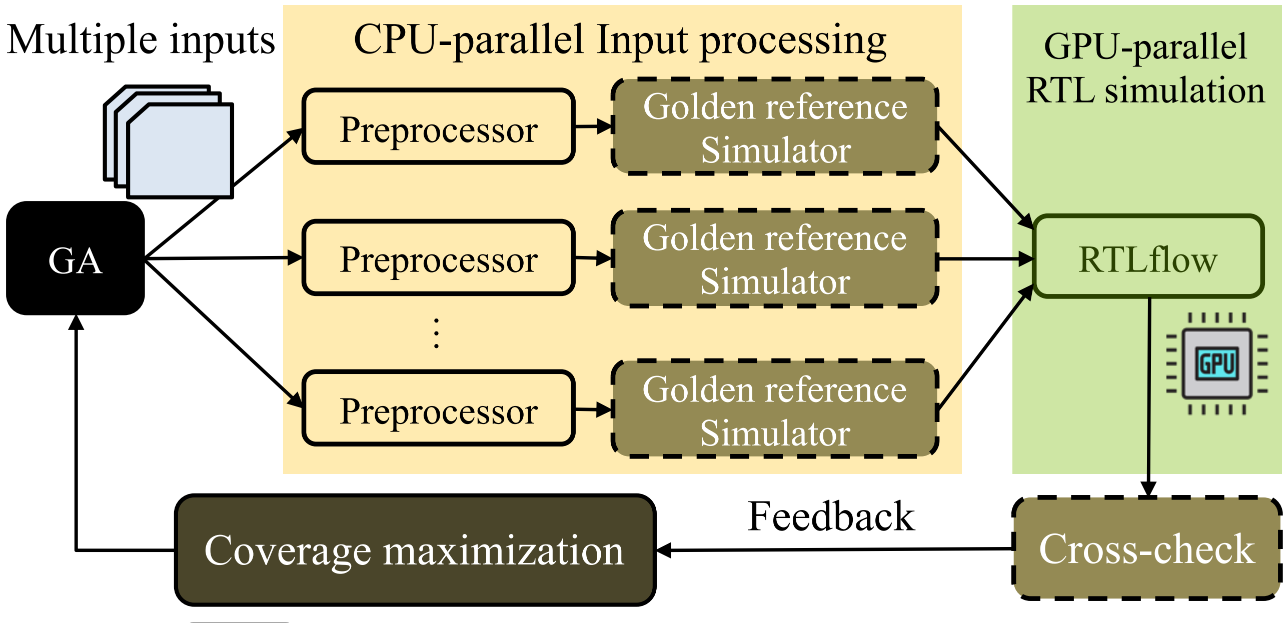


#cycles	Spinal		NVDLA	
	stream	CUDA Graph	stream	CUDA Graph
10K	11.5s	2.3s (**5×**)	279.8s	106.5s (**2.6×**)
100K	108.0s	14.2s (**7.6×**)	2046.9s	791.2s (**2.6×**)
500K	532.9s	72.3s (**7.4×**)	9718.0s	3733.0s (**2.6×**)

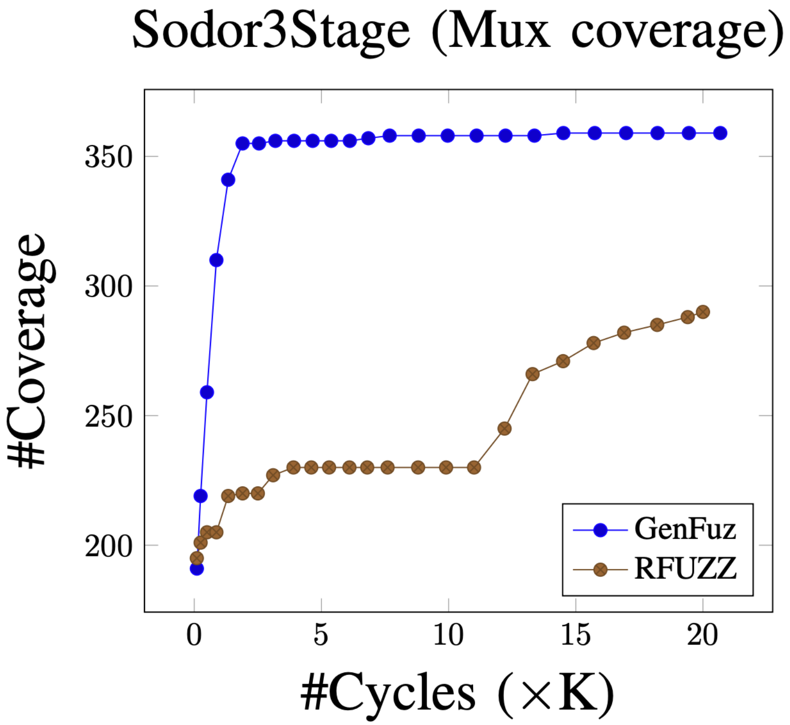
Table 4: Performance advantage of CUDA Graph execution in multi-stimulus simulation workloads, measured on Spinal and NVDLA with 4096 stimulus under different numbers of cycles.

Case Study 3: Hardware Fuzzing (DAC'23)

- **Applied our RTL simulator to accelerate hardware fuzzing**
  - A new genetic algorithm to largely improve coverage quality using GPU
  - **10–80x** faster over existing fuzzers and found undiscovered hardware bugs



Multiple inputs
CPU-parallel Input processing
GPU-parallel RTL simulation
GA
Preprocessor
Golden reference Simulator
Preprocessor
Golden reference Simulator
⋮
Preprocessor
Golden reference Simulator
RTLflow
GPU
Feedback
Coverage maximization
Cross-check

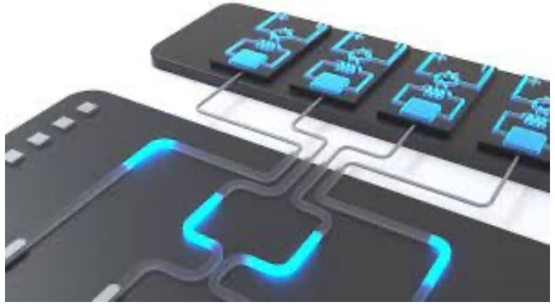


Dian-Lun Lin, et al, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," *ACM/IEEE DAC*, CA, 2023

Other Industrial Applications of Taskflow

- **Quantum computing**
  - Xanadu uses Taskflow in their quantum computing cloud
- **3D graphics and rendering engines**
  - Methane uses Taskflow in their renderer
- **Numerical analysis**
  - Deal.II uses Taskflow for advanced parallelism
- **Computer vision**
  - RevealTech uses Taskflow for real-time vision devices
- **Linear algebra**
  - JetBrains uses Taskflow in their sparse matrix libraries
- … **(ME, Biochips, Imaging, FinTech, etc.)**

METHANE
powered graphics



https://www.xanadu.ai/

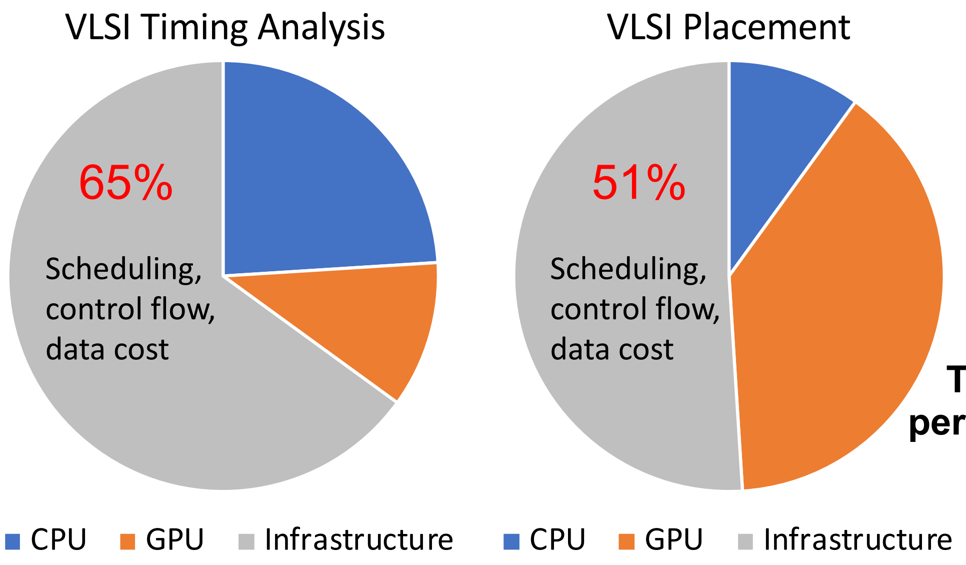
deal.II parameter GUI

https://www.dealii.org/

https://www.revealtech.ai/

Parallel Computing Infrastructure Matters

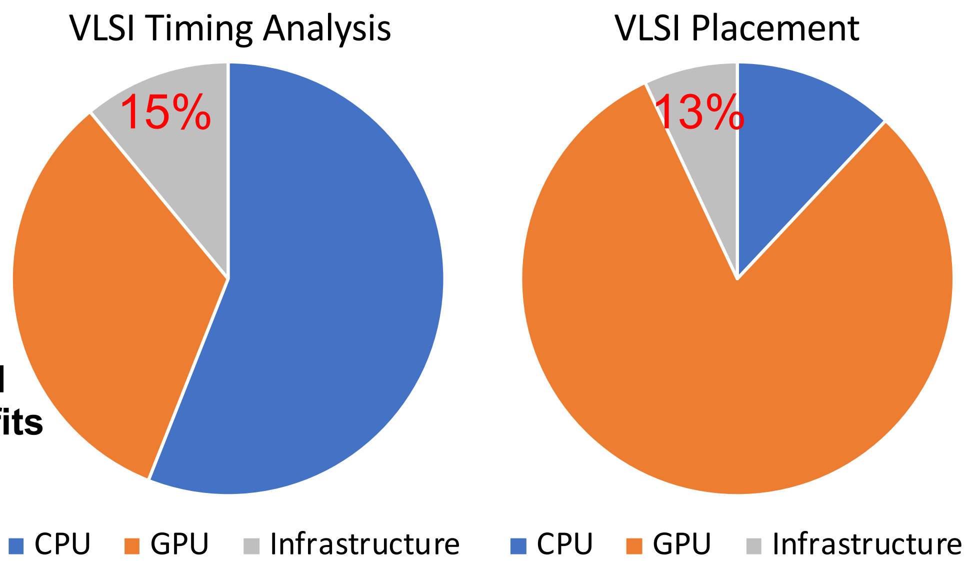
Different models give you different implementation results. The parallel algorithm itself may run fast, **but** ***the parallel computing infrastructure you use to implement that algorithm may dominate the entire performance***.



(ICCAD'15, TCAD'17, DAC'19)

With Taskflow

Transformational performance benefits (incl. GPUs)

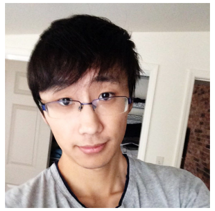


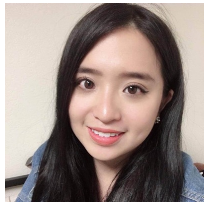
(IPDPS'19, ICPADS'20, HPEC'20, ICCAD'20-21, DAC'21-22, TCAD'21, TCAD'22, TPDS'22, HPDC'22)

Conclusion

- Understood the challenges of parallel computing
- Introduced our new task-parallel programming system
- Dived into our system runtime
- Applied our system to computer engineering problems
- We are very open to collaborate!

Acknowledgement: PhD Students & Sponsors





NSF

DARPA

NVIDIA

NUMFOCUS
OPEN CODE = BETTER SCIENCE

U
ECE
ELECTRICAL AND COMPUTER ENGINEERING

Google Summer of Code

intel®

Use the right tool for the right job

Taskflow: https://taskflow.github.io

Thank You

Dr. Tsung-Wei Huang

tsung-wei.huang@utah.edu