Taskflow: Programming System for building High-performance EDA Applications

How can we make it easier to program heterogeneous EDA applications?

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

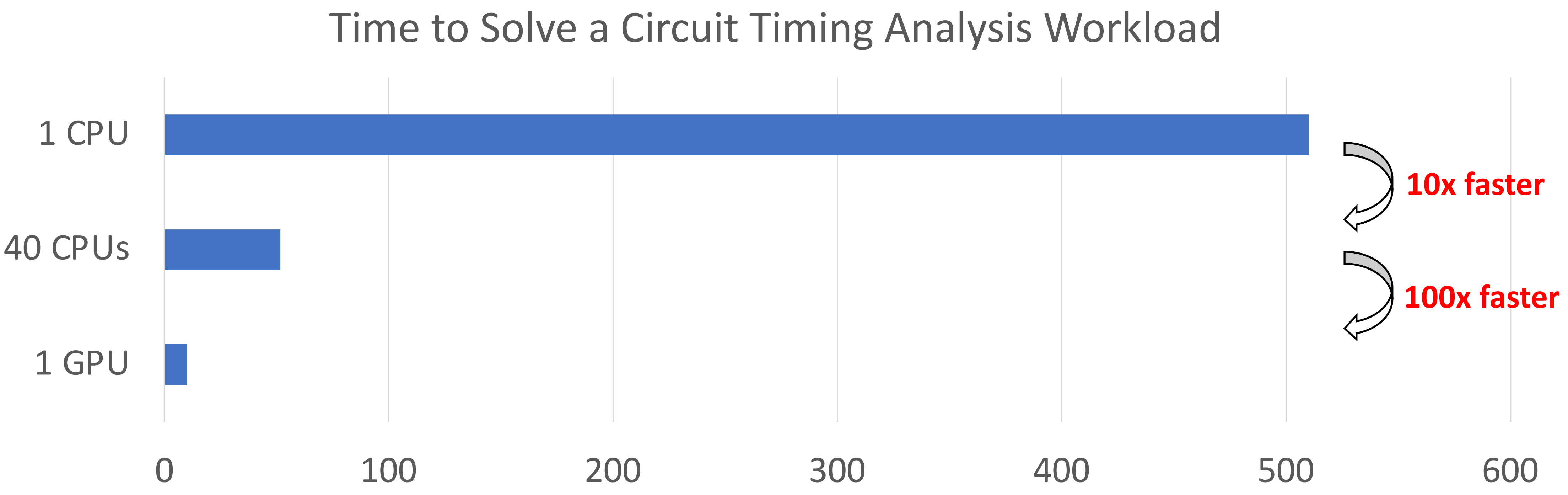
Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

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

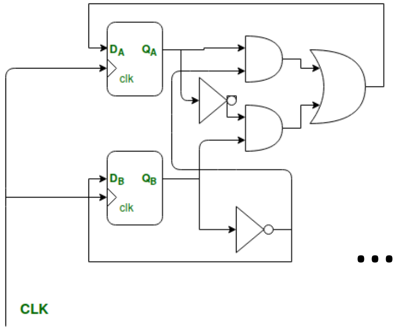
Why Parallel Computing?

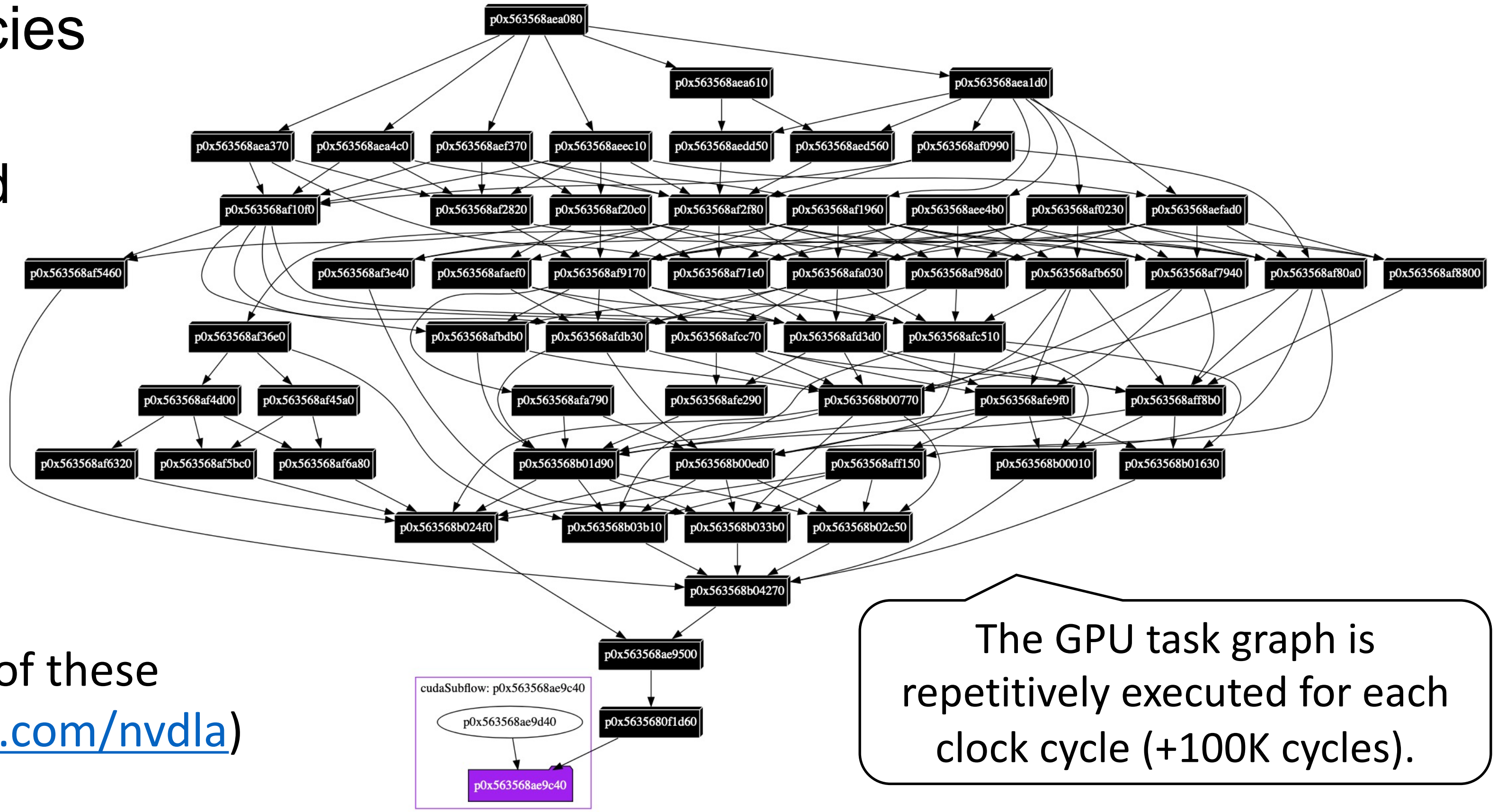
- It's critical to advance your application performance



Today’s Workload is Very Complex

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





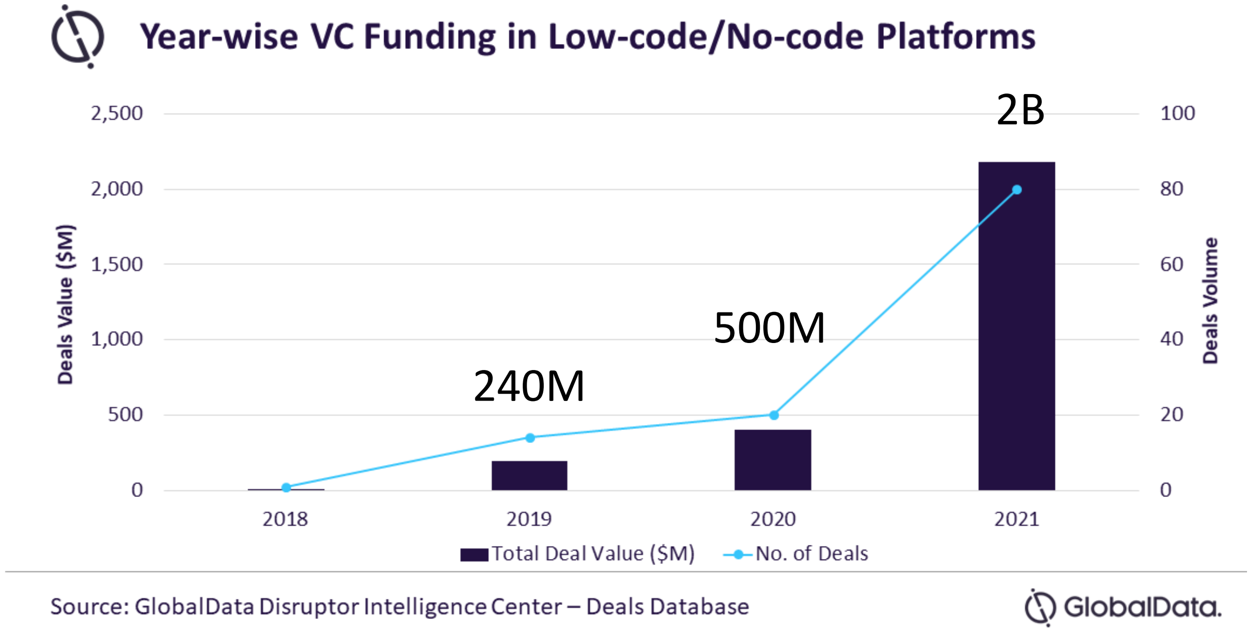
The GPU task graph is repetitively executed for each clock cycle (+100K cycles).

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

Parallel Programming is VERY Challenging

- **You need to deal with A LOT OF technical details**
  - Parallelism abstraction (task, data, concurrent data structures, etc.)
  - Concurrency control
  - Task/data race avoidance
  - Dependency constraints
  - Load balancing
  - Scheduling efficiencies
  - …

How can we make scientific software researchers' (your) lives easier?

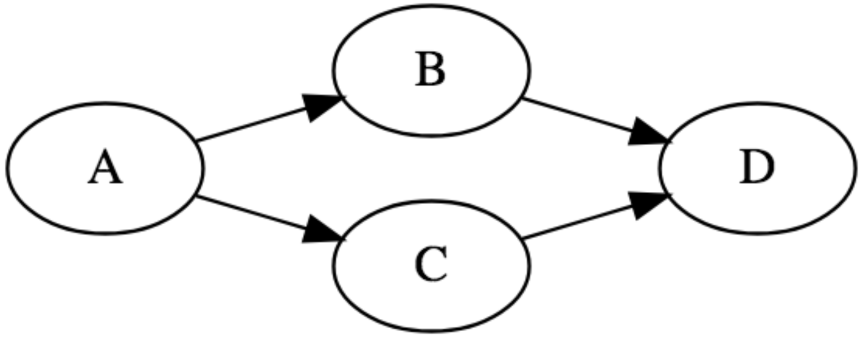


Taskflow offers a solution

How can we make it easier for C++ developers to quickly write parallel and heterogeneous programs with high performance scalability and simultaneous high productivity?

“Hello World” in Taskflow

```
#include <taskflow/taskflow.hpp>   // Taskflow is header-only
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(); // submit the taskflow to the executor
    return 0;
}
```



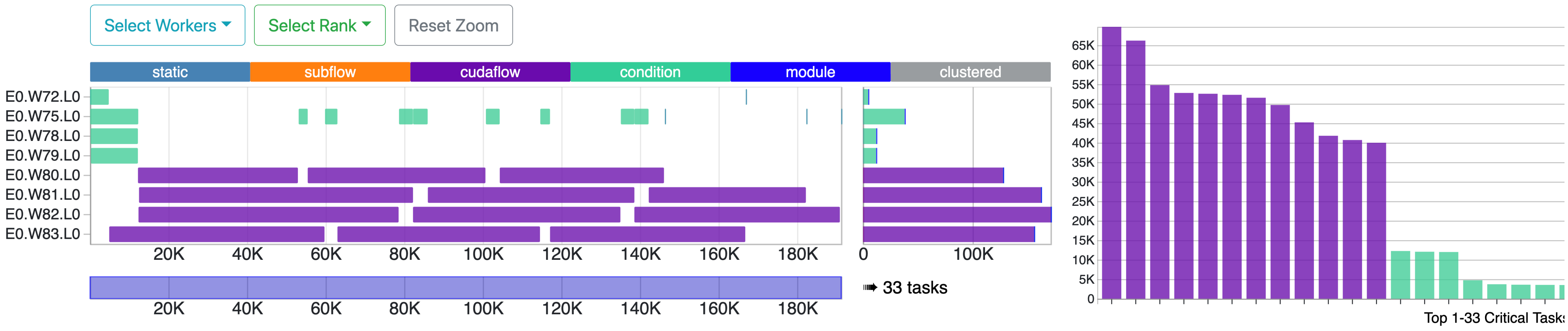
Drop-in Integration

- Taskflow is header-only – *no wrangle with installation*

```
~$ git clone https://github.com/taskflow/taskflow.git  # clone it only once
~$ g++ -std=c++17 simple.cpp -I taskflow/taskflow -O2 -pthread -o simple
~$ ./simple
TaskA
TaskC
TaskB
TaskD
```

Built-in Profiler/Visualizer

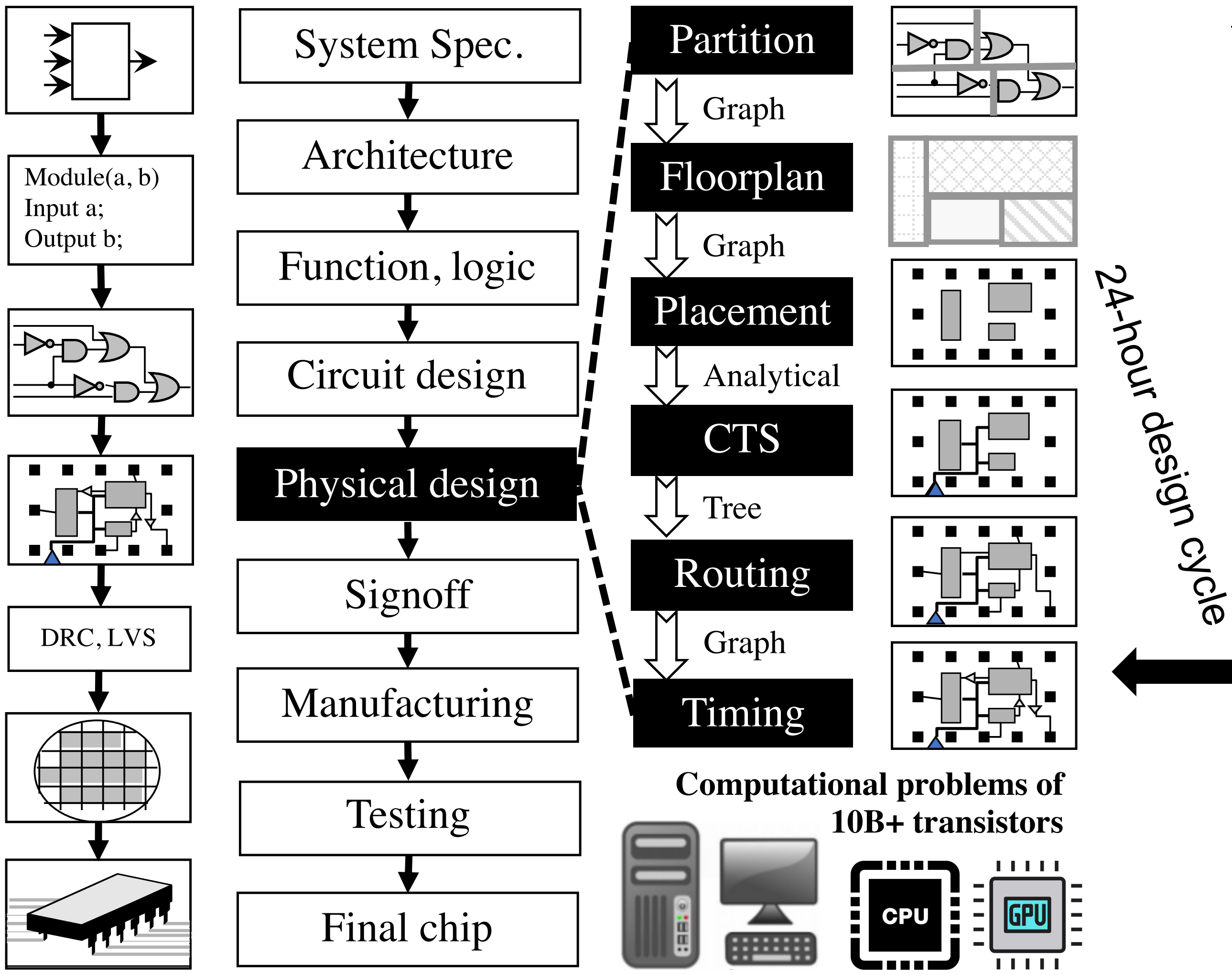
```
# run the program with the environment variable TF_ENABLE_PROFILER enabled
~$ TF_ENABLE_PROFILER=simple.json ./simple
~$ cat simple.json
[
{"executor":"0","data":[{"worker":0,"level":0,"data":[{"span":[172,186],"name
]
# paste the profiling json data to https://taskflow.github.io/tfprof/
```

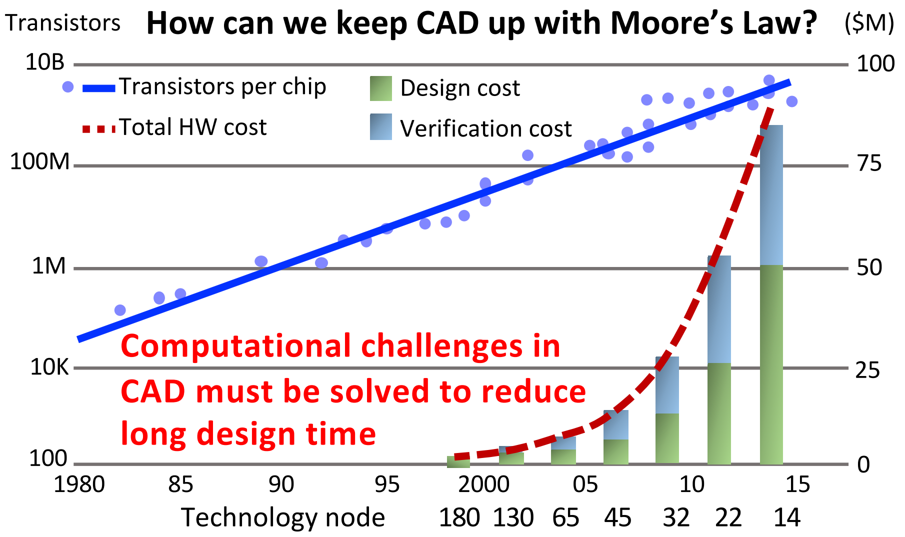


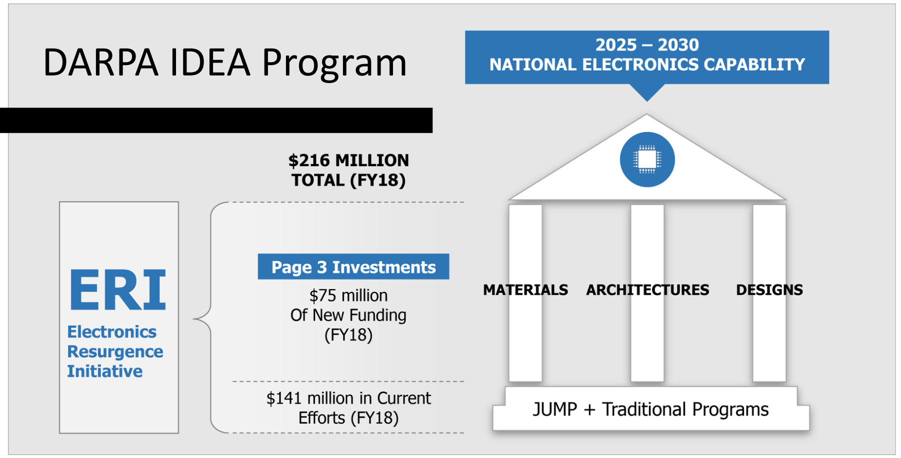
Agenda

- **Express your parallelism in the right way**
- Parallelize your applications using Taskflow
- Boost performance in CAD applications

Motivation: Parallelizing CAD/EDA Tools







We Invested a lot in Existing Tools …

pthreads

OpenMP

C++

TBB

NVIDIA CUDA

SYCL

PaRSEC

Open MPI

STE||AR GROUP

StarPU

ECP EXASCALE COMPUTING PROJECT

OpenCL

RAJA

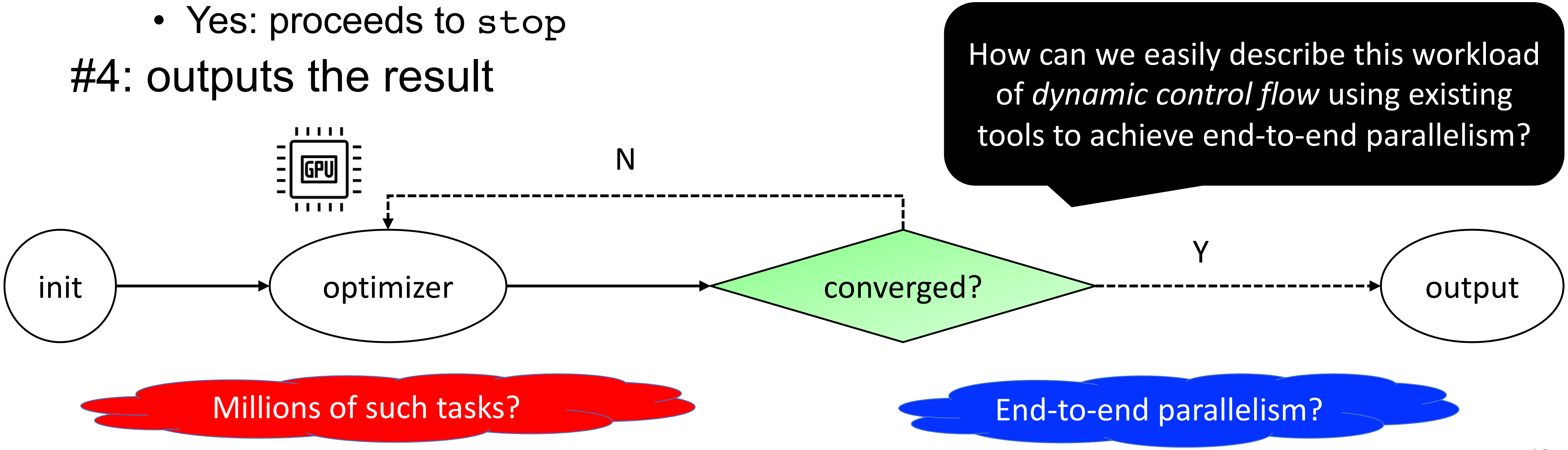
kokkos

Two Big Problems of Existing Tools

- **EDA has *complex task dependencies***
  - **Example**: analysis algorithms compute the circuit network of multi-millions of nodes and dependencies
  - **Problem**: existing tools are often good at loop parallelism but weak in expressing heterogeneous task graphs at this large scale
- **EDA has *complex control flow***
  - **Example**: synthesis algorithms make essential use of *dynamic control flow* to implement various patterns
    - Combinatorial optimization (e.g., graph algorithms, discrete math)
    - analytical methods (e.g., physical synthesis)
  - **Problem**: existing tools are *directed acyclic graph* (DAG)-based and do not anticipate control flow in the graph, lacking *end-to-end* parallelism

Example: An Iterative Optimizer

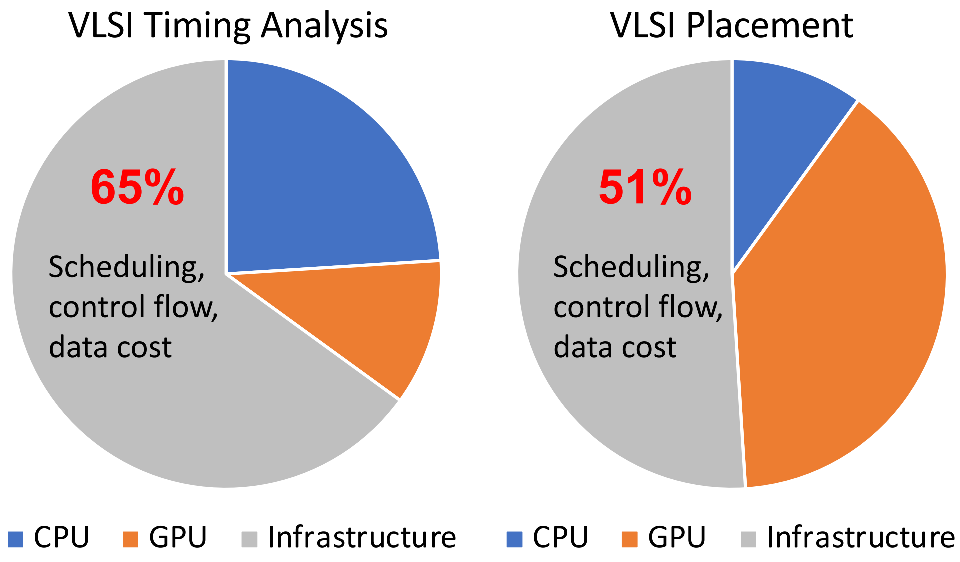
- Four computational tasks with dynamic control flow
  - #1: starts with `init` task
  - #2: enters the `optimizer` task (e.g., GPU math solver)
  - #3: checks if the optimization converged
    - No: loops back to `optimizer`
    - Yes: proceeds to `stop`
  - #4: outputs the result



Need a New Parallel Programming System

While designing parallel algorithms is non-trivial, what makes parallel programming an enormous challenge is the **infrastructure work** of "***how to efficiently express dependent tasks along with algorithmic control flow and schedule them across heterogeneous computing resources***"

- **VLSI timing analysis (ICCAD'20)**
  - up to **65%** runtime on infrastructure
  - 24% on CPU and 11% on GPU
- **VLSI placement (TCAD'21)**
  - up to **51%** runtime on infrastructure
  - 10% on CPU and 39% on GPU



Agenda

- Express your parallelism in the right way
- **Parallelize your applications using Taskflow**
- Boost performance in CAD applications

Revisit “Hello World” in Taskflow (TPDS’22)

```
#include <taskflow/taskflow.hpp>   // Taskflow is header-only
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(); // submit the taskflow to the executor
  return 0;
}
```

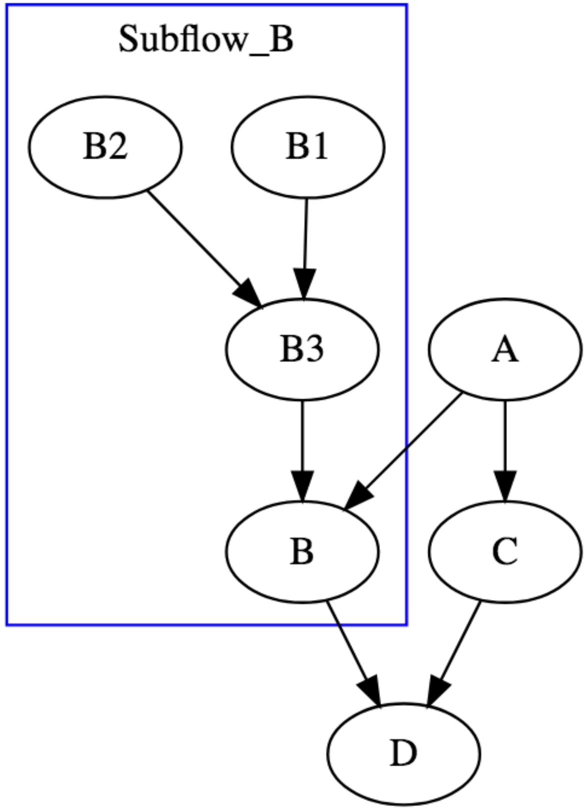
A new **control taskflow graph (CTFG)** programming model

1. Static tasking
2. Dynamic tasking
3. Conditional tasking
4. Heterogeneous tasking
5. Pipeline tasking

… (more on https://taskflow.github.io/)

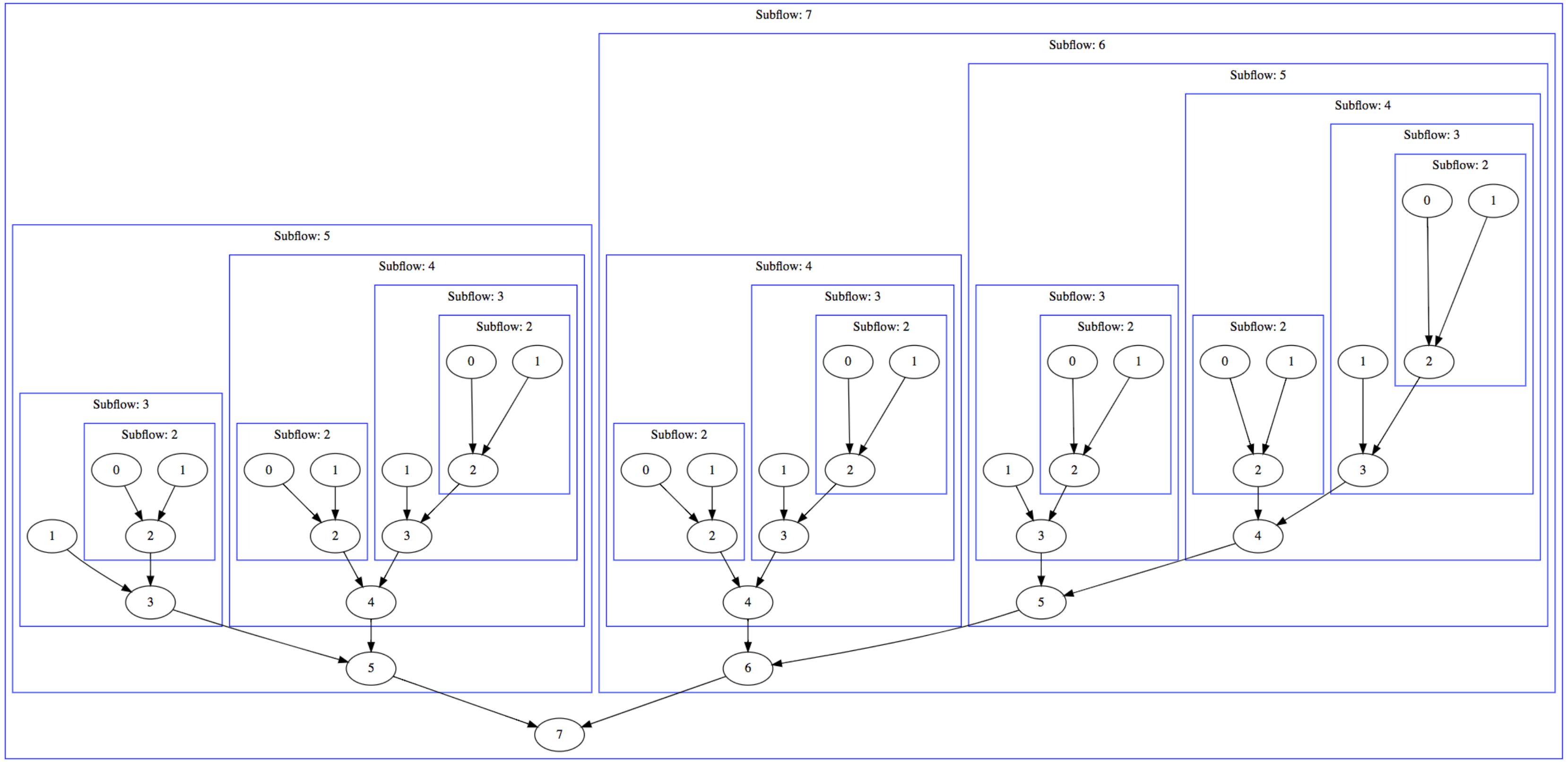
#2: Dynamic Tasking (Subflow)

```
// create three regular tasks
tf::Task A = tf.emplace([](){}).name("A");
tf::Task C = tf.emplace([](){}).name("C");
tf::Task D = tf.emplace([](){}).name("D");

// create a subflow graph (dynamic tasking)
tf::Task B = tf.emplace([] (tf::Subflow& subflow) {
  tf::Task B1 = subflow.emplace([](){}).name("B1");
  tf::Task B2 = subflow.emplace([](){}).name("B2");
  tf::Task B3 = subflow.emplace([](){}).name("B3");
  B1.precede(B3);
  B2.precede(B3);
}).name("B");

A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C
```



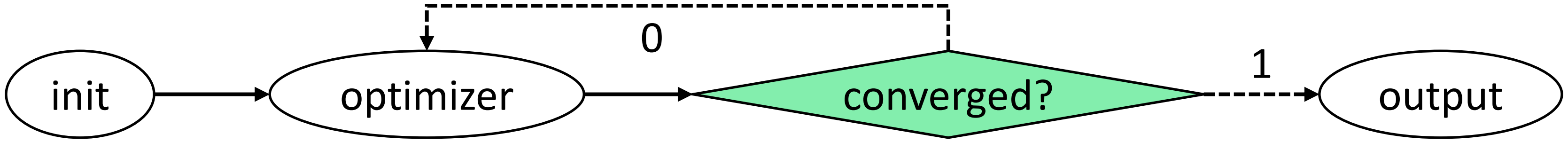
Subflow can be Nested and Recurive

- Find the 7th Fibonacci number using subflow
  - Fib(n) = Fib(n-1) + Fib(n-2)



#3: Conditional Tasking (if-else)

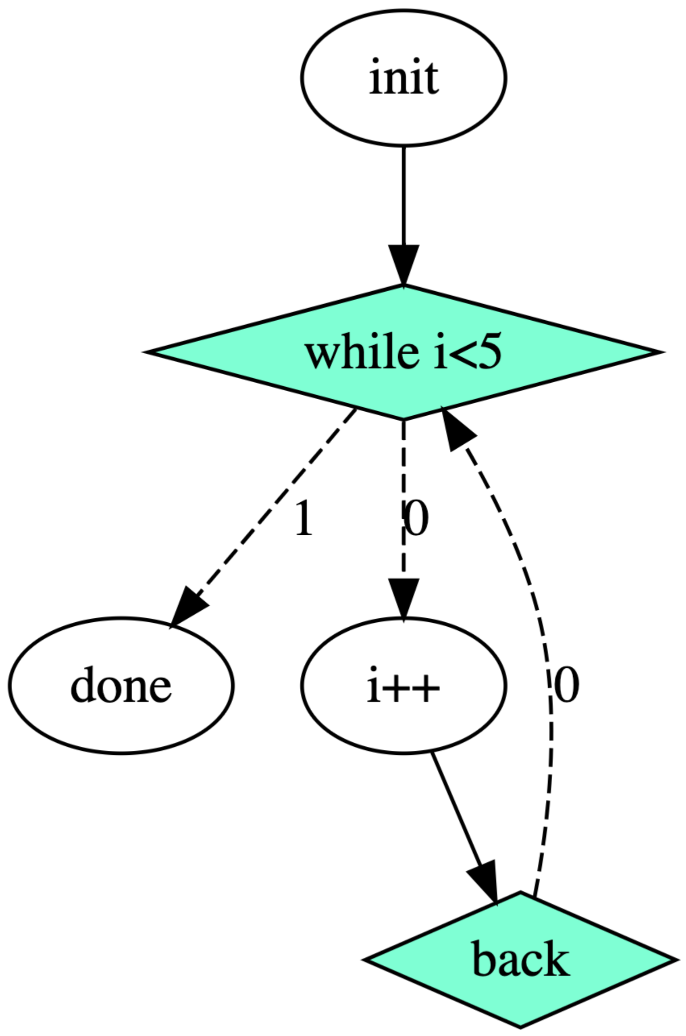
```
auto init          = taskflow.emplace([&](){ initialize_data_structure(); } )
                             .name(”init");
auto optimizer     = taskflow.emplace([&](){ matrix_solver(); } )
                             .name(”optimizer");
auto converged     = taskflow.emplace([&](){ return converged() ? 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
```



*Condition task enables in-graph control flow to achieve **end-to-end** parallelism*

#3: Conditional Tasking (iterative loop)

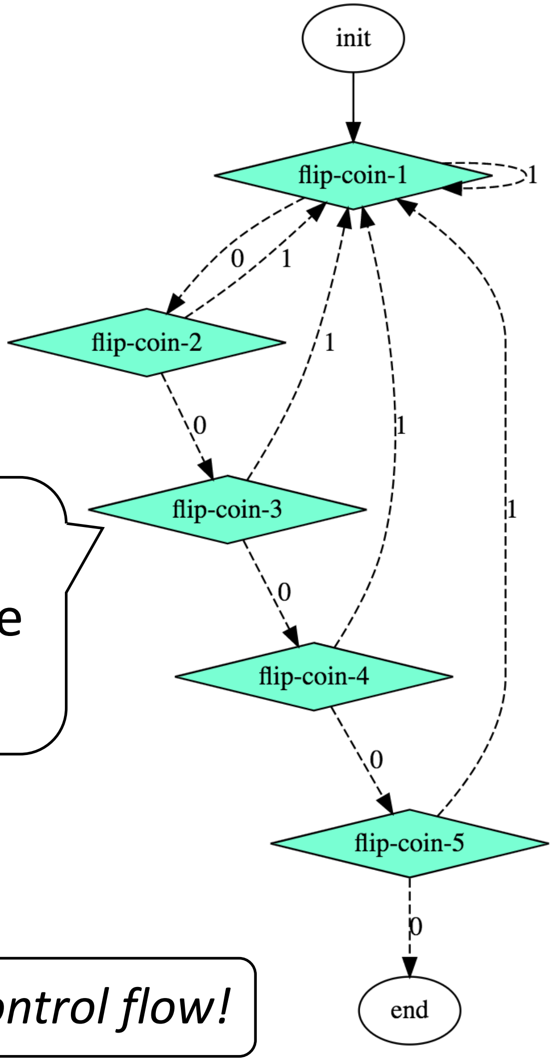
```
tf::Taskflow taskflow;
int i;
auto [init, cond, body, back, done] = taskflow.emplace(
  [&](){ std::cout << “i=0”; i=0; },
  [&](){ std::cout << "while i<5\n"; return i < 5 ? 0 : 1; },
  [&](){ std::cout << "i++=" << i++ << '\n'; },
  [&](){ std::cout << "back\n"; return 0; },
  [&](){ std::cout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



#3: Conditional Tasking (random loops)

```
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");
```

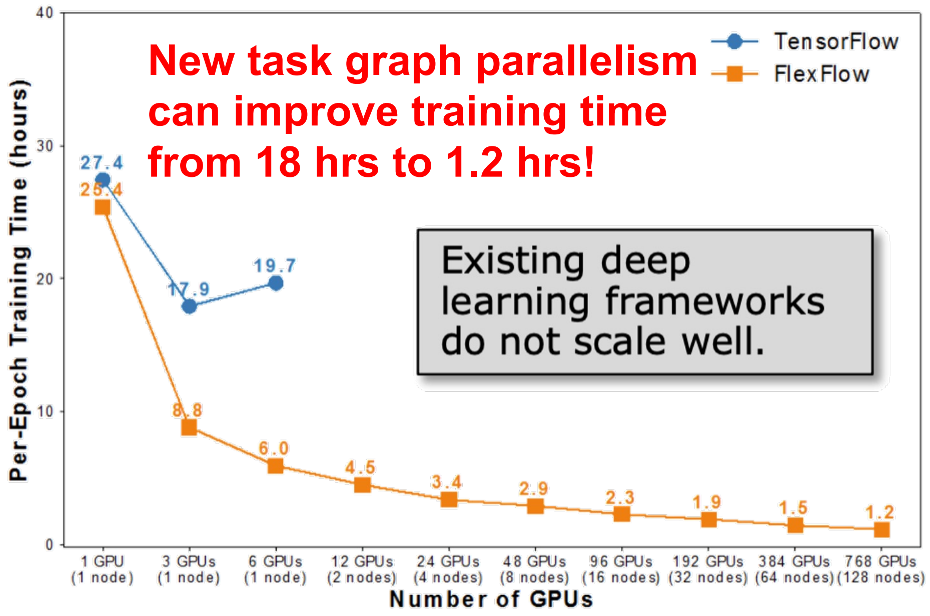
Each task flips a binary coin to decide the next path



You can describe non-deterministic, nested control flow!

Existing Frameworks on Control Flow?

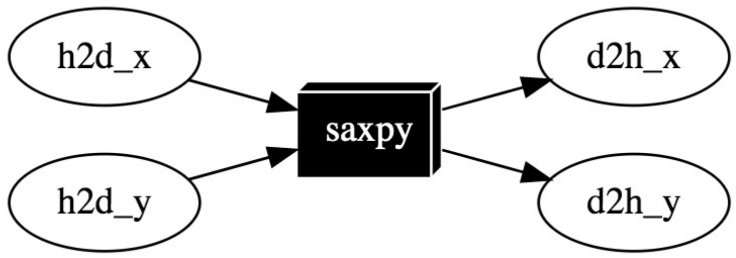
- **Expand a task graph across fixed-length iterations**
  - Large graph size linearly proportional to decision points
- **Unknown or non-deterministic iterations?**
  - Expensive dynamic tasks executing “if-else” on the fly
- **Dynamic control-flow tasks?**
  - Client-side partition
- **Same problem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



#4: Heterogeneous Tasking

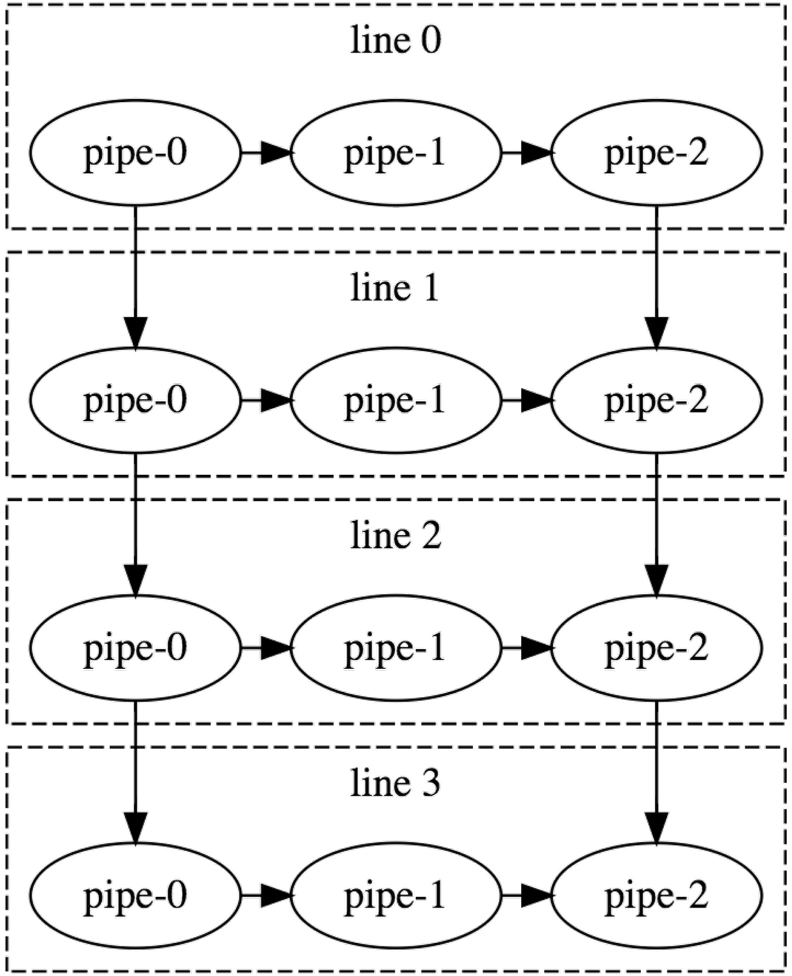
```
const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};
auto allocate_x = taskflow.emplace([&](){ cudaMalloc(&dx, 4*N);});
auto allocate_y = taskflow.emplace([&](){ cudaMalloc(&dy, 4*N);});

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"**



5: Pipeline Tasking (HPDC'22)

```
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();
```



Submit Taskflow to Executor

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

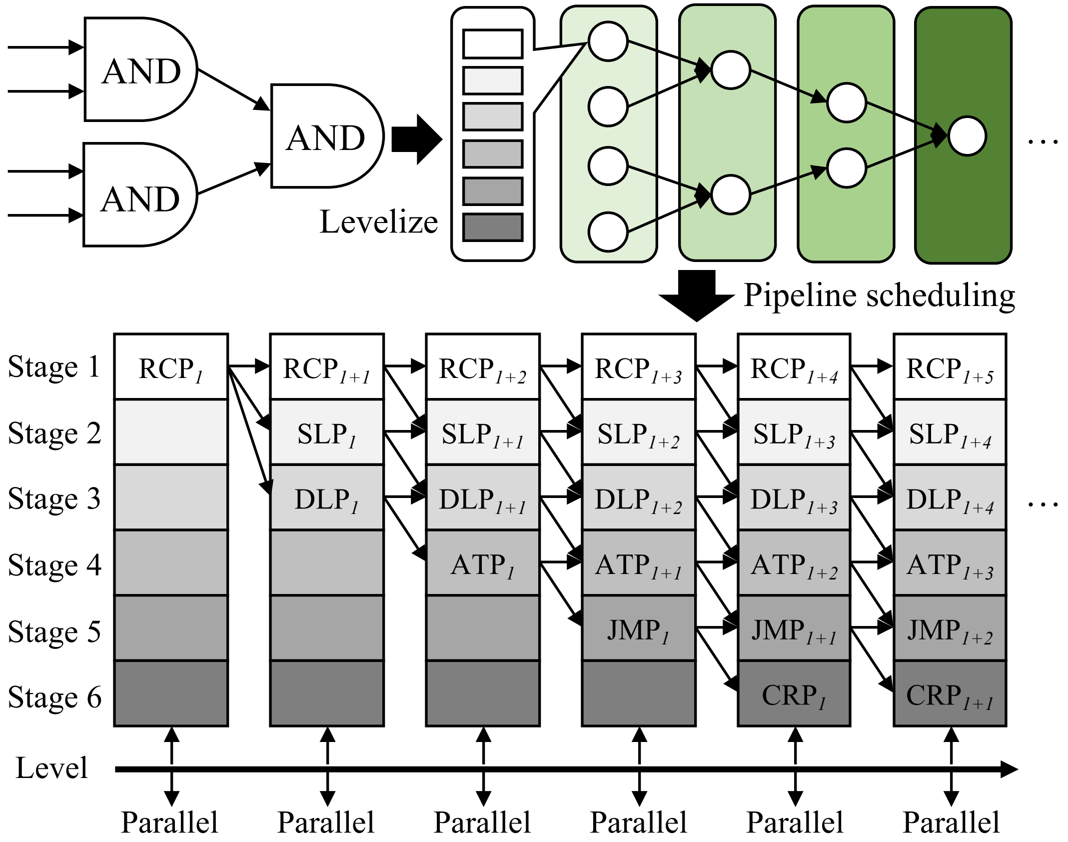
```
{
  tf::Taskflow taskflow1, taskflow2, taskflow3;
  tf::Executor executor;
  // create tasks and dependencies
  // …
  auto  future1 = executor.run(taskflow1);
  auto  future2 = executor.run_n(taskflow2, 1000);
  auto  future3 = executor.run_until(taskflow3, [i=0](){ return i++>5 });
  executor.wait_for_all();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

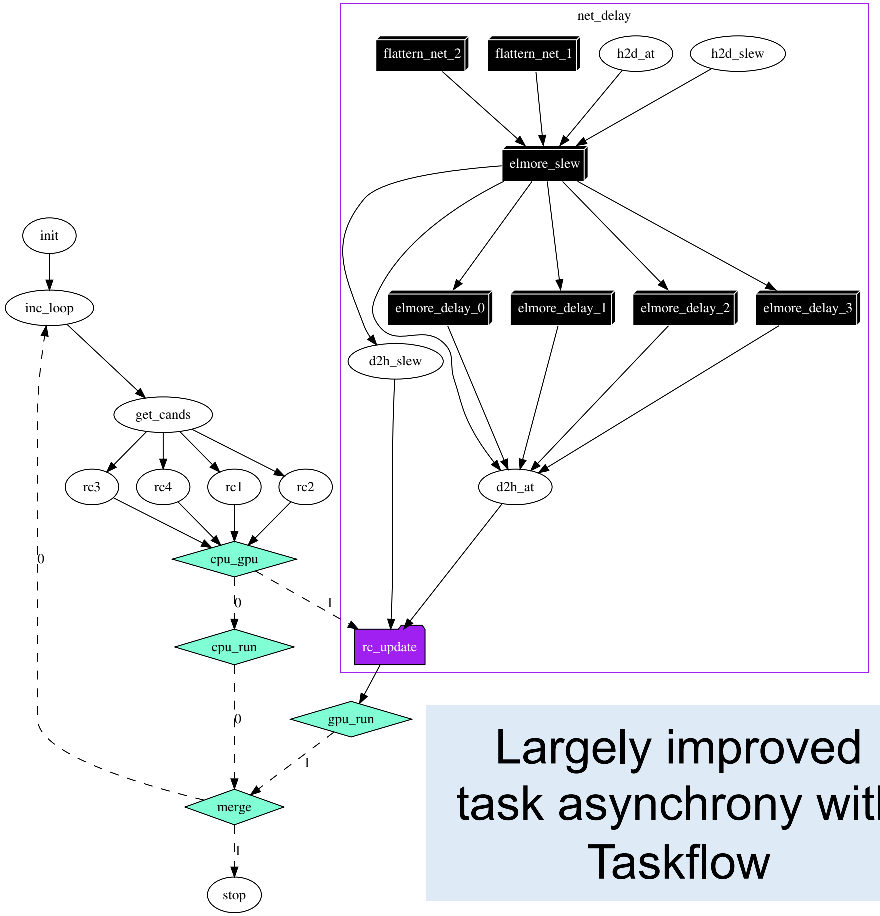
Agenda

- Express your parallelism in the right way
- Parallelize your applications using Taskflow
- **Boost performance in CAD applications**

Case Study 1: Timing Analysis (TCAD'21)

- **Analyzing the timing of large circuits can take *hours* to finish**

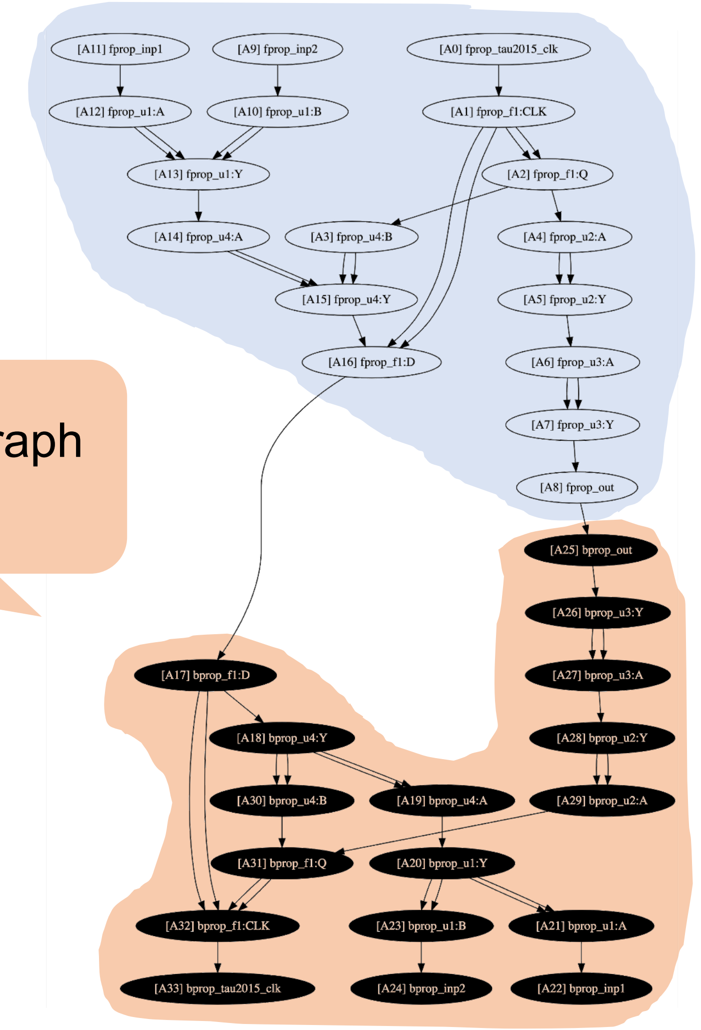




Largely improved task asynchrony with Taskflow

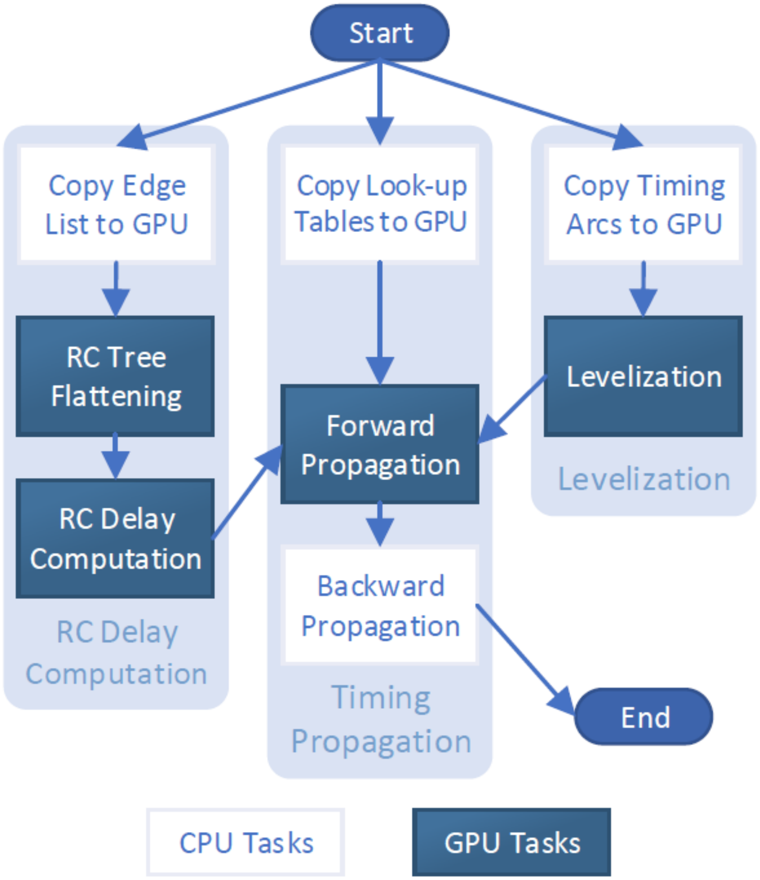
Case Study: Timing Analysis (cont'd)

Forward propagation task graph
(RC, arrival time, slew, etc.)

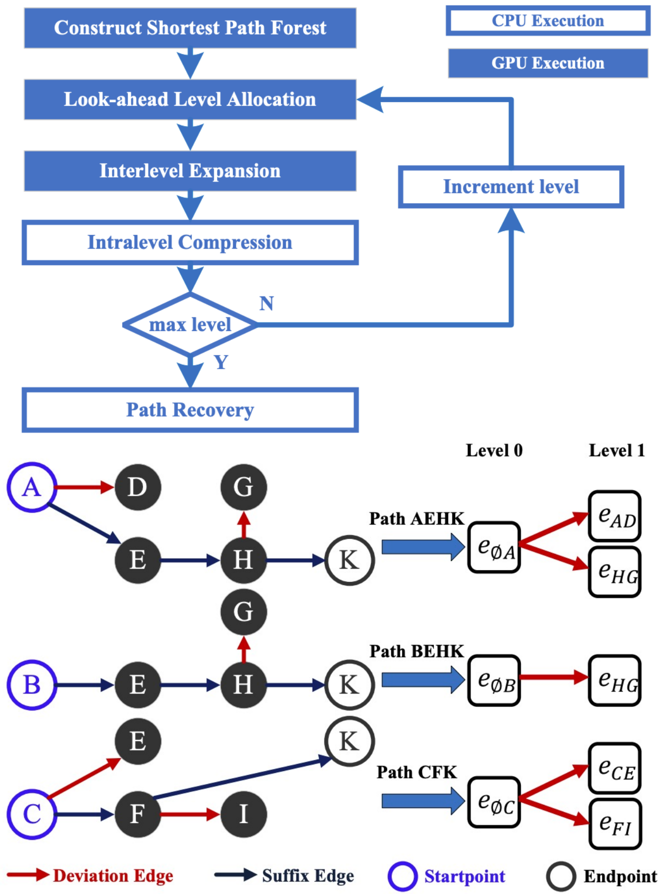


Backward propagation task graph
(constraints, slack, etc.)

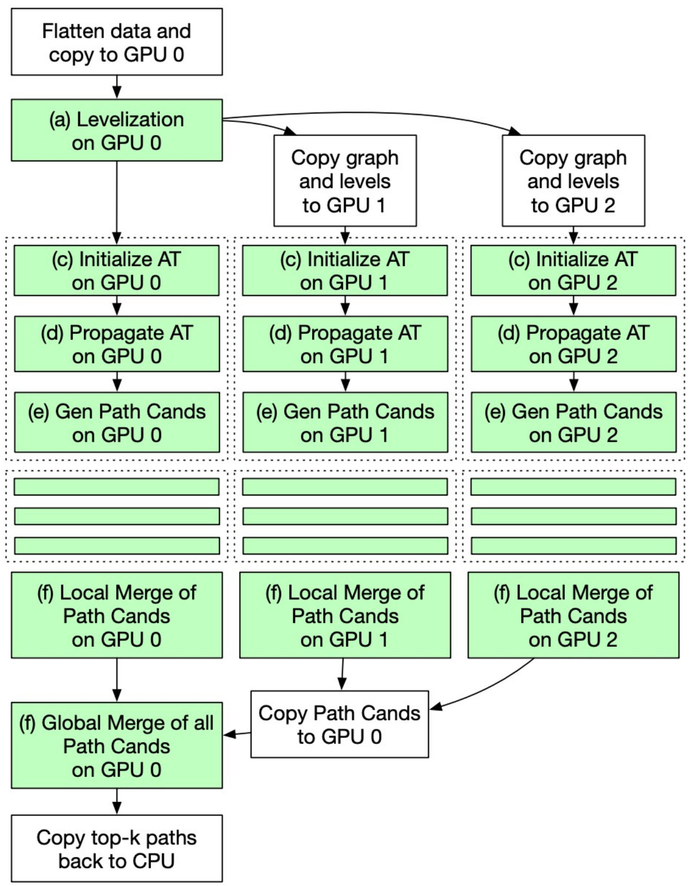
Case Study 1: Timing Analysis (cont'd)



GPU-based graph analysis (ICCAD'20)



GPU-based path analysis (DAC'21)



GPU-based CPPR (ICCAD'21)

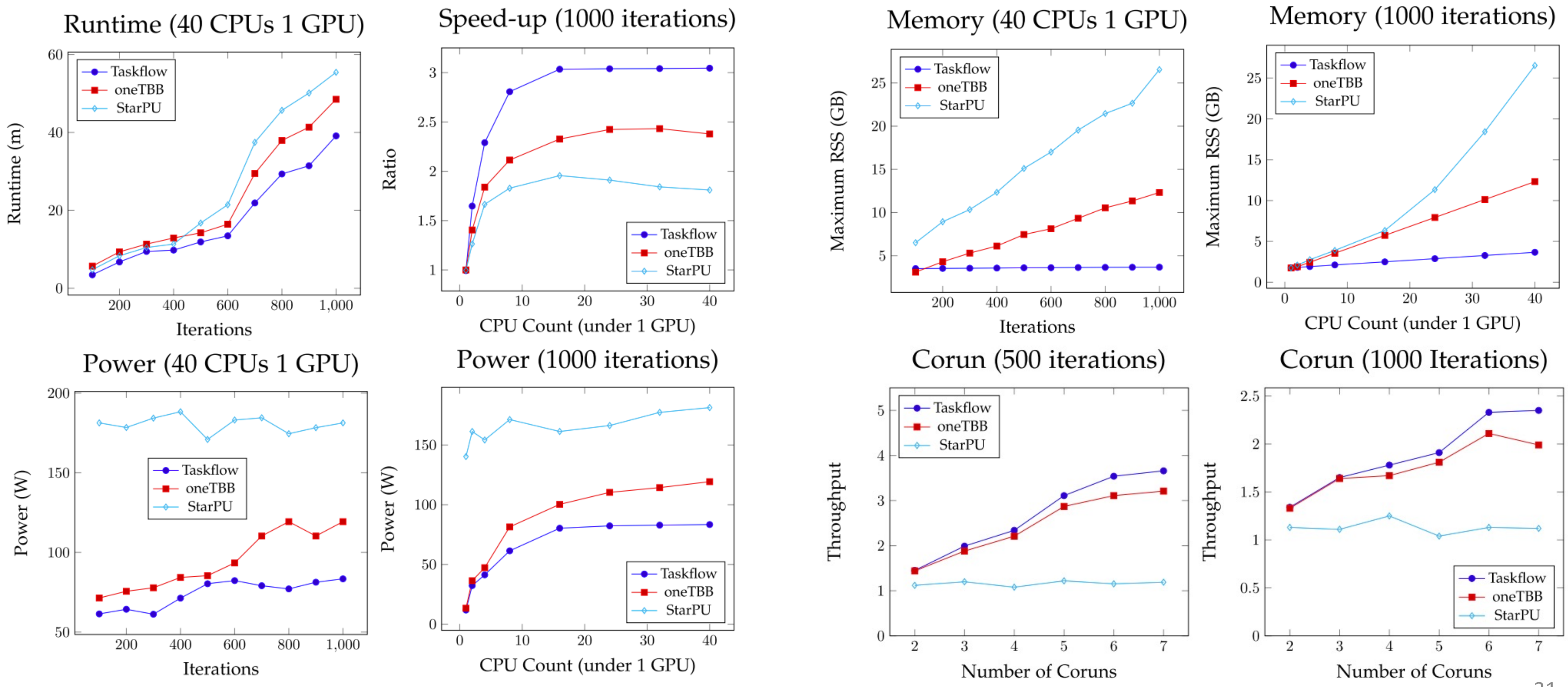
Case Study 1: Timing Analysis (cont'd)

- **Path-based timing analysis (DAC'21, Best Paper in TAU'21)**
  - leon3mp (1.6M gates): 611x speed-up over 1 CPU (44x over 40 CPUs)
  - netcard (1.5M gates): 367x speed-up over 1 CPU (46x over 40 CPUs)

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×

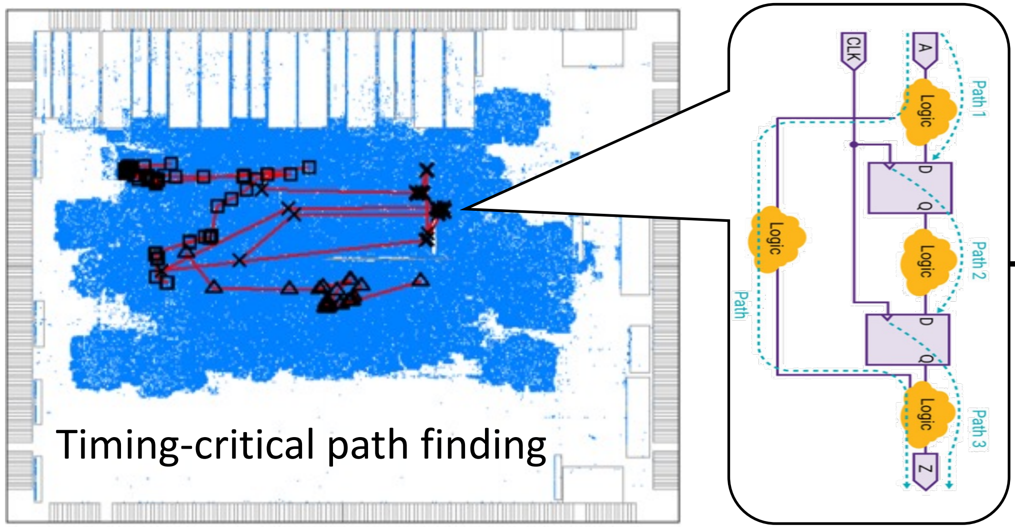
Case Study 1: Timing Analysis (cont'd)

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



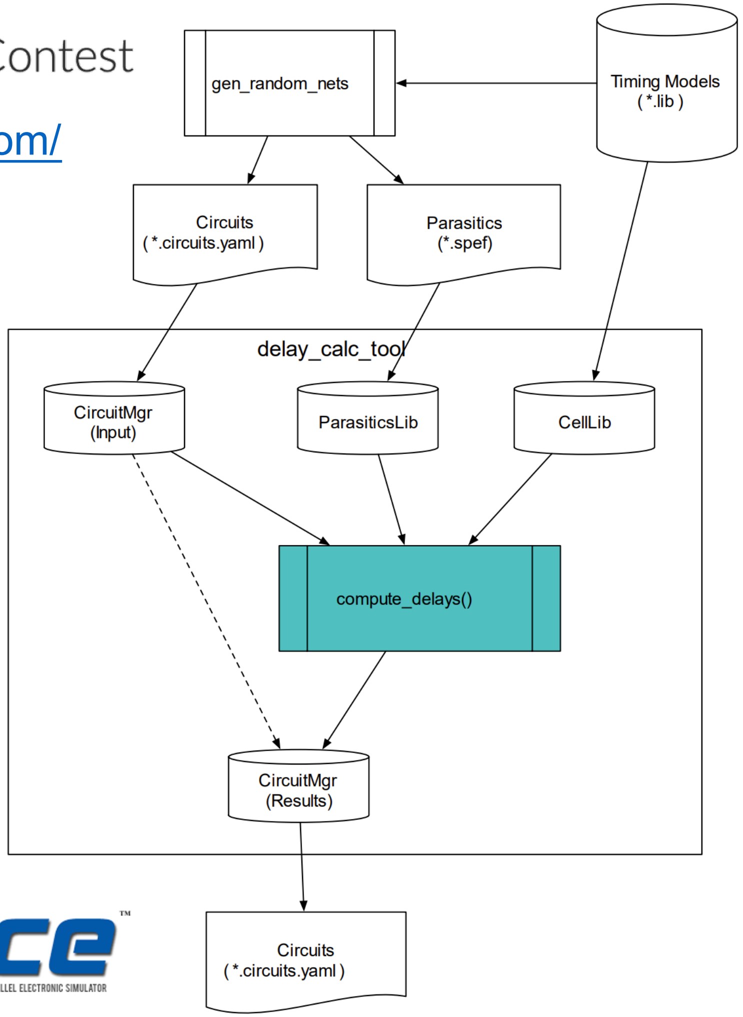
OpenTimer: Static Timing Analysis Engine

https://github.com/OpenTimer/OpenTimer



Tau 2021 Contest

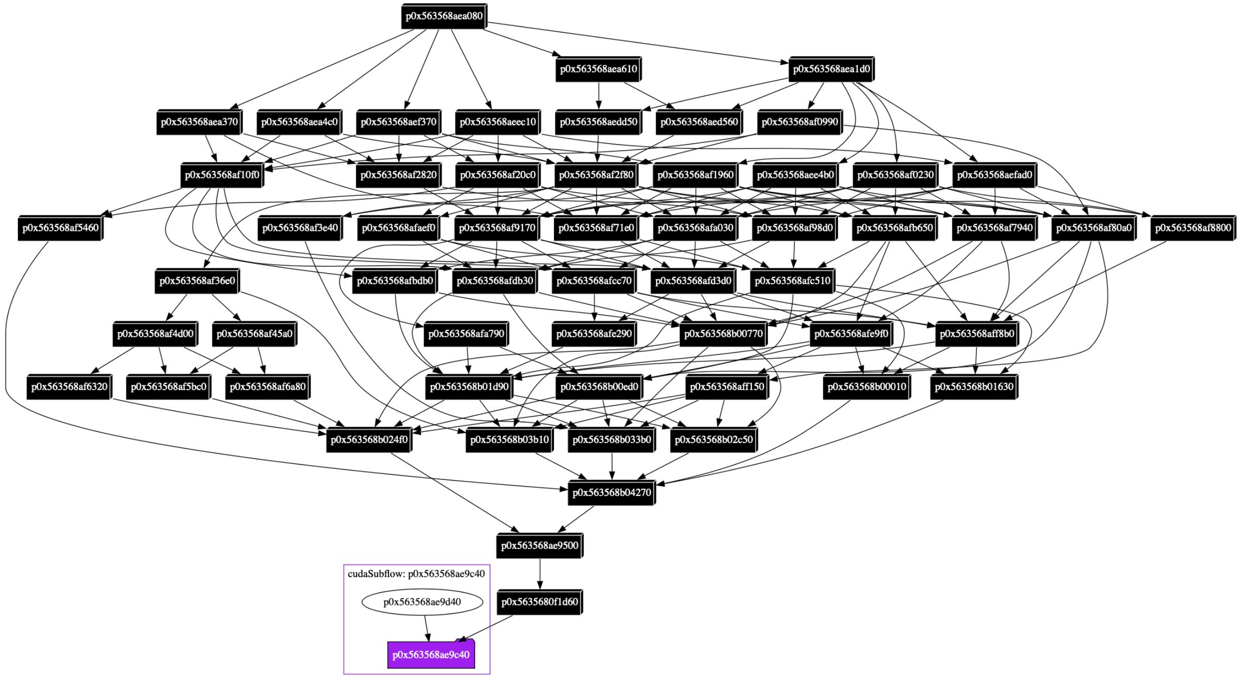
https://sites.google.com/view/tau-contest-2021/home

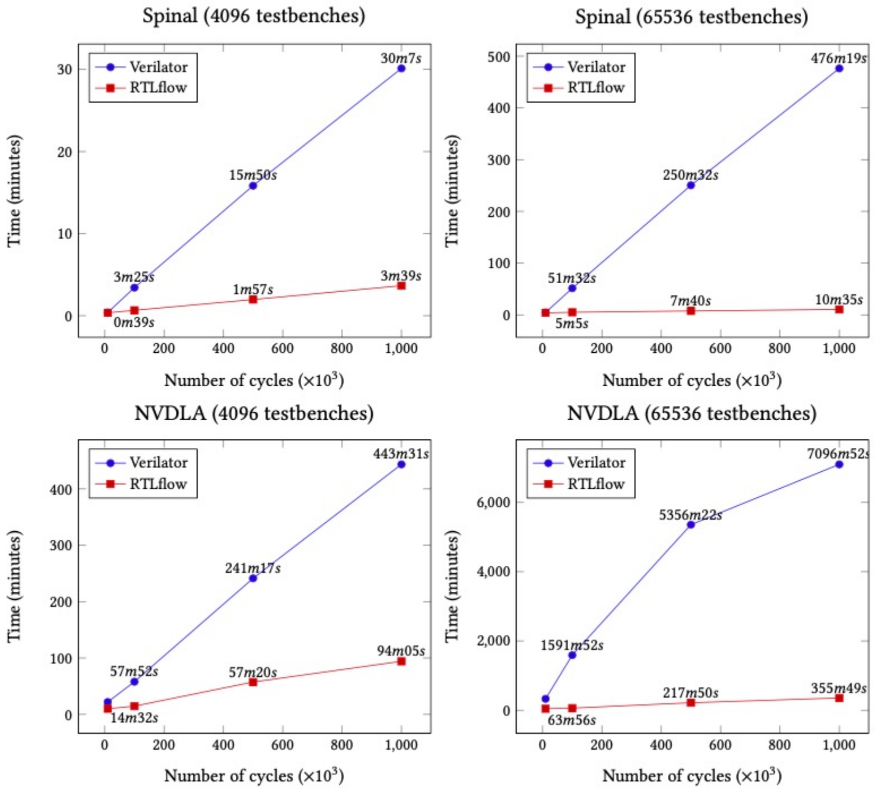


User community

Case Study 2: RTL Simulation

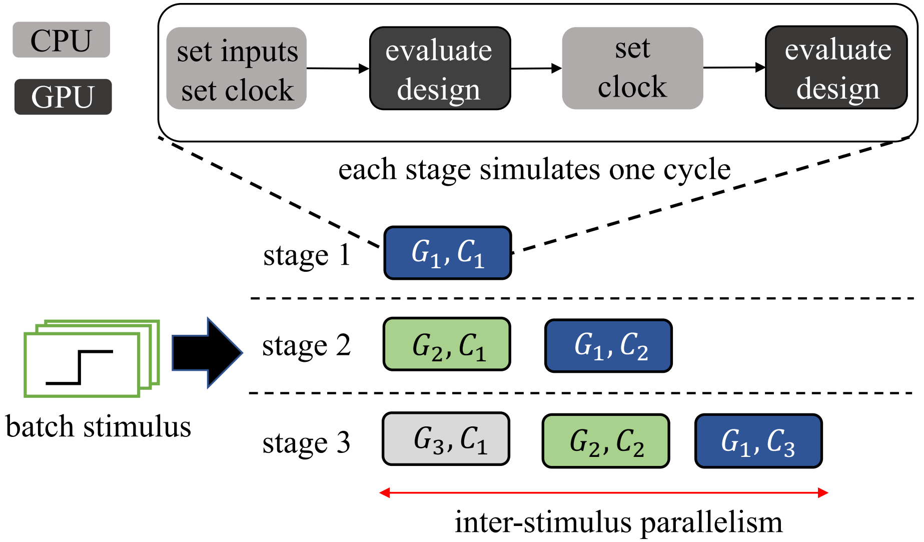
- **Leverage task graph and pipeline parallelisms (i.e., RTLflow)**
  - <u>**10–500x**</u> faster over existing RTL simulator for multiple simulation batches

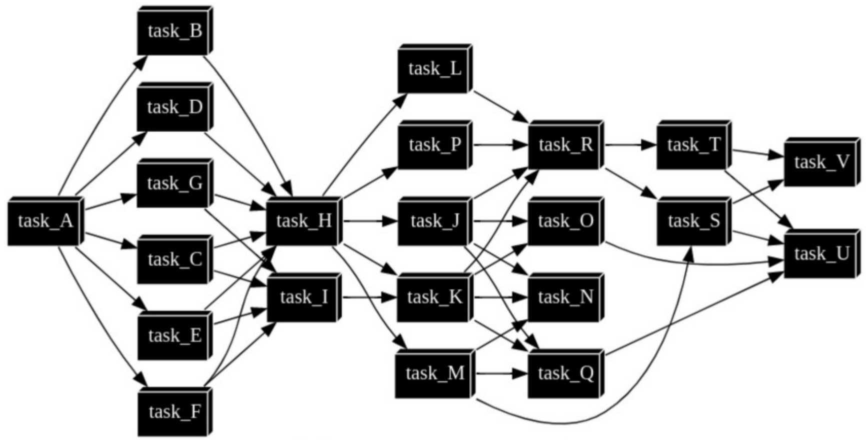




Dian-Lun Lin, et al, "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus," *ACM ICPP*, Bordeaux, France, 2022

Case Study 2: RTL Simulation (cont'd)





#stimulus	Spinal RTLflow^{-p}	Spinal RTLflow	NVDLA RTLflow^{-p}	NVDLA 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.

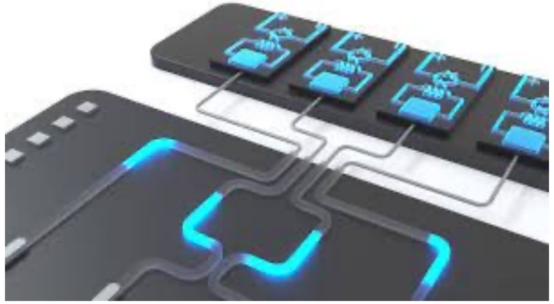
#cycles	Spinal stream	Spinal CUDA Graph	NVDLA stream	NVDLA CUDA Graph
10K	11.5s	2.3s (**5×**)	279.8s	106.5s (**2.6×**)
100K	108.0s	14.2s (**7.6×**)	2046.9s	791.2s (**2.6×**)
500K	532.9s	72.3s (**7.4×**)	9718.0s	3733.0s (**2.6×**)

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.

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/

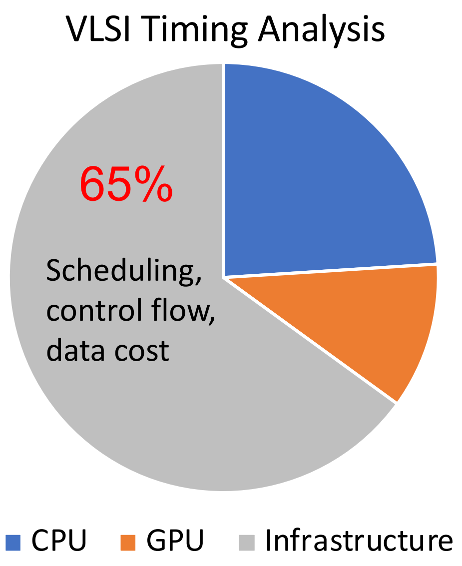
deal.II parameter GUI

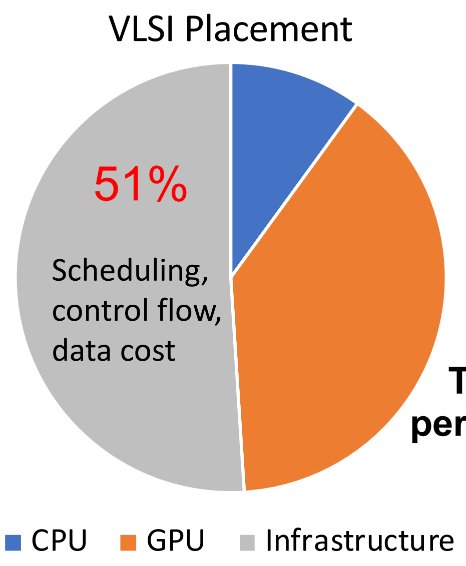
https://www.dealii.org/

https://www.revealtech.ai/

Parallel Computing Infrastructure Matters

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

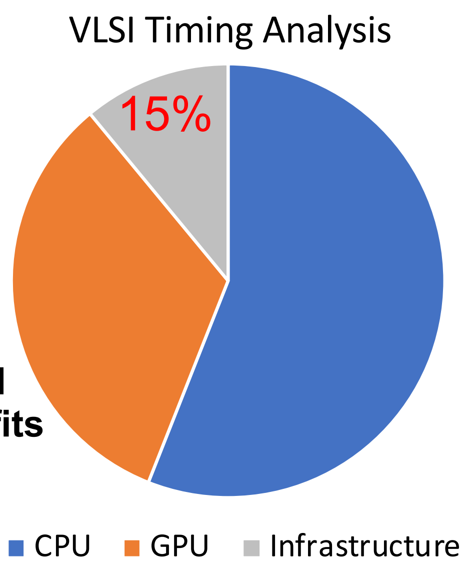


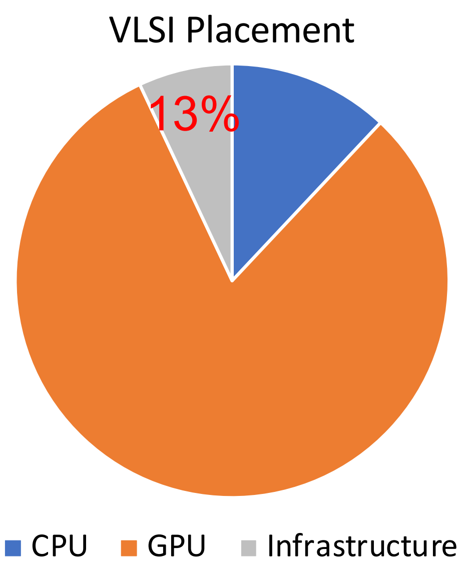


(ICCAD'15, TCAD'17, DAC'19)

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)

Use the right tool for the right job

Taskflow: https://taskflow.github.io

Thank You

Dr. Tsung-Wei Huang

tsung-wei.huang@utah.edu