A General-purpose Parallel and Heterogeneous Task Programming System at Scale

Dr. Tsung-Wei (TW) Huang

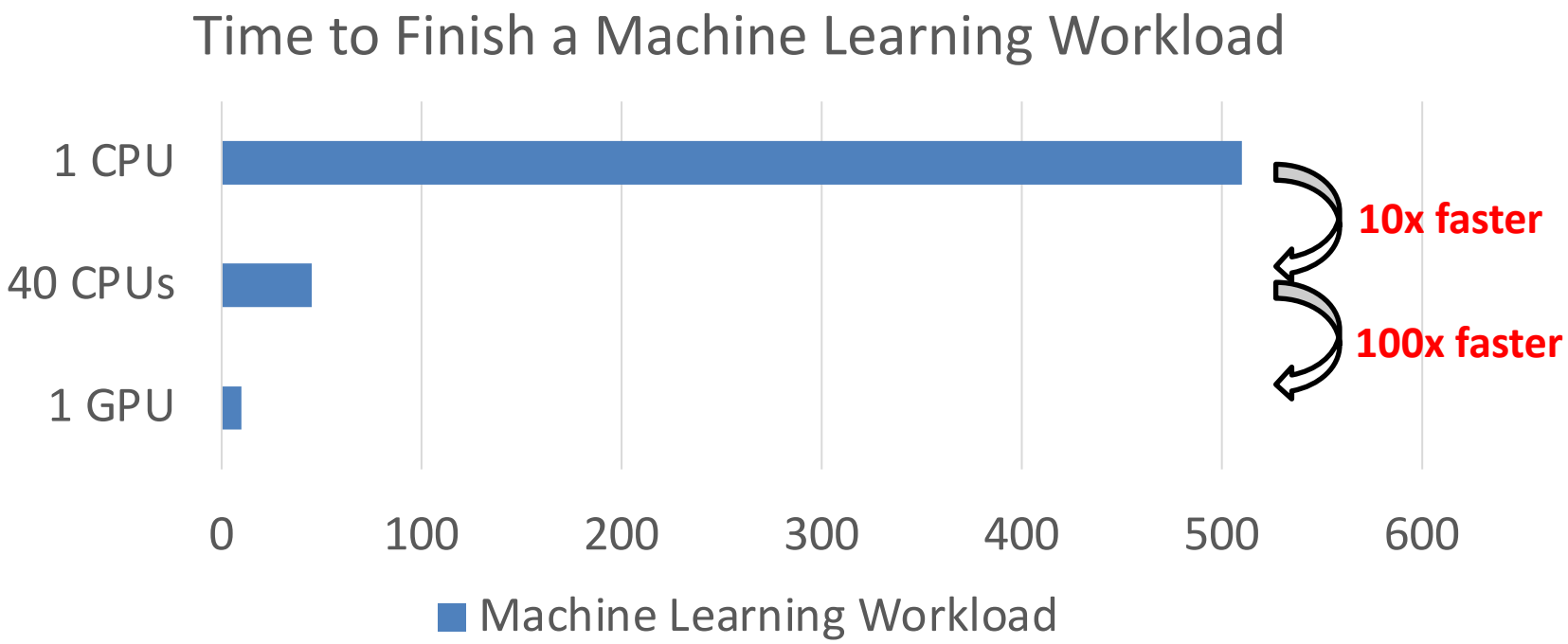
Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

https://taskflow.github.io/

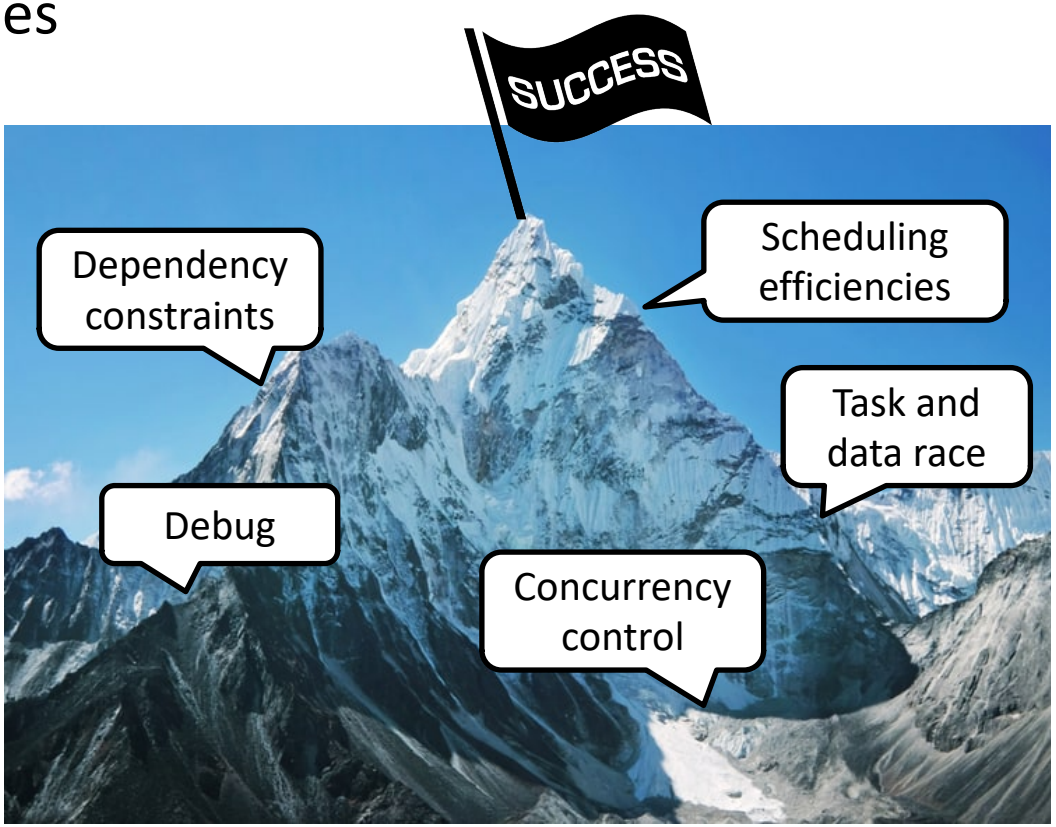
Why Parallel Computing?

- It's critical to advance your application performance



Parallel Programming is Not Easy, Yet

- **You need to deal with many difficult technical details**
  - Standard concurrency control
  - Task dependencies
  - Scheduling
  - Data race
  - … (more)



Many developers have hard time in getting them right!

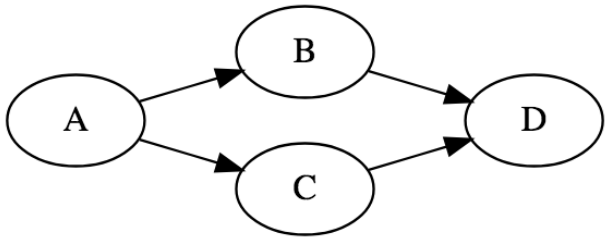
Taskflow offers a solution

How can we make it easier for 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;
}
```

*Only **15 lines** of code to get a parallel task execution!*



Agenda

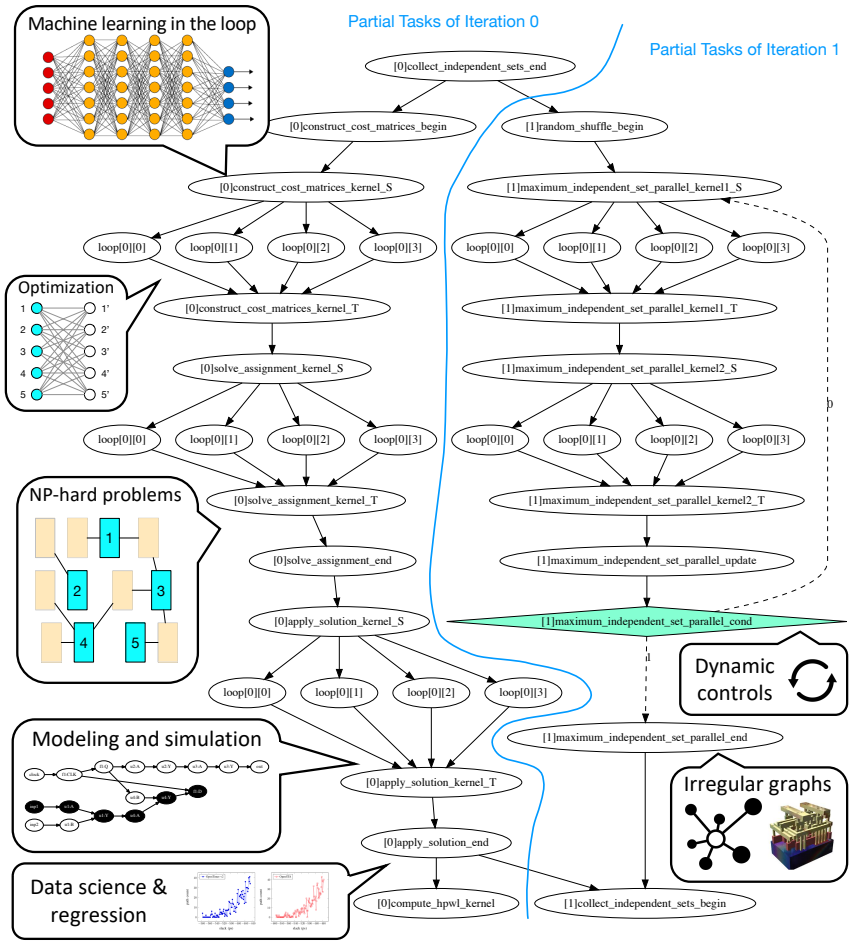
- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Boost performance in real applications

Agenda

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

Building ML-centric CAD Software …

- How can we program a task graph like this?



Broad mix of algorithms (graph, analytics, simulation)

Domain-specific heuristics (branch bound, approximation)

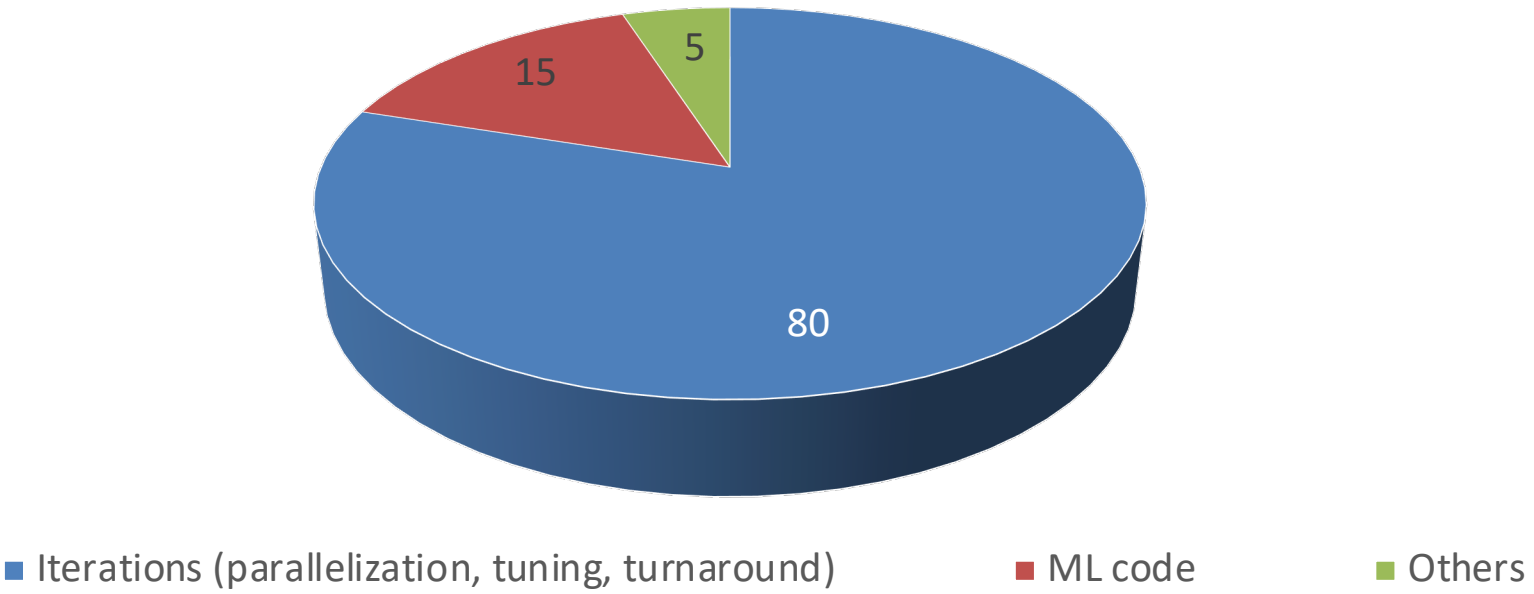
Enormous parameter tuning (million lines of scripts)

Vast task/data dependencies (terabytes of design files)

Machine learning in the loop (numerous flow trajectories)

ML Programming Landscape

How much time did you spend on writing your ML applications?



#1 concern: "*ML code is very cheap to write. The difficult part is the surrounding tasks, notably* ***parallelization, tuning, and turnaround time***"

Why Not Using Existing Tools?

Two Big Problems of Existing Tools

- **Our problems define *complex task dependencies***
  - **Example**: analysis algorithms compute the circuit network of million of node and dependencies
  - **Problem**: existing tools are often good at loop parallelism but weak in expressing heterogeneous task graphs at this large scale
- **Our problems define *complex control flow***
  - **Example**: optimization algorithms make essential use of *dynamic control flow* to implement various patterns
    - ML is essentially an optimization for making decisions
  - **Problem**: existing tools are *directed acyclic graph* (DAG)-based and do not anticipate cycles or conditional dependencies, lacking *end-to-end* parallelism

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 an algorithmic control flow and schedule them across heterogeneous computing resources***"

Agenda

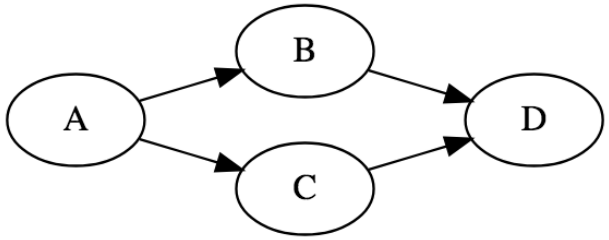
- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Boost performance in real applications

“Hello World” in Taskflow (Revisited)

```
#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;
}
```

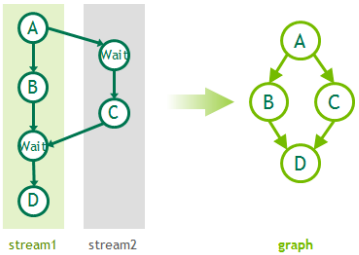
Taskflow defines five tasks:

1. static task (this talk)
2. dynamic task
3. cudaFlow task (this talk)
4. condition task (this talk)
5. module task



Heterogeneous Tasking (cudaFlow)

```
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);});
```

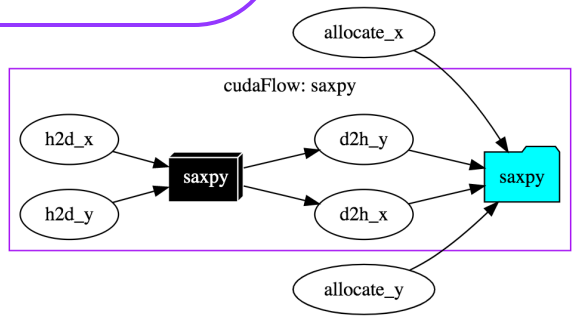



```
auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
  auto h2d_x = cf.copy(dx, hx.data(), N);  // CPU-GPU data transfer
  auto h2d_y = cf.copy(dy, hy.data(), N);
  auto d2h_x = cf.copy(hx.data(), dx, N);  // GPU-CPU data transfer
  auto d2h_y = cf.copy(hy.data(), dy, N);
  auto kernel = cf.kernel((N+255)/256,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

To Nvidia ***cudaGraph***

Single kernel launch

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



Three Key Advantages

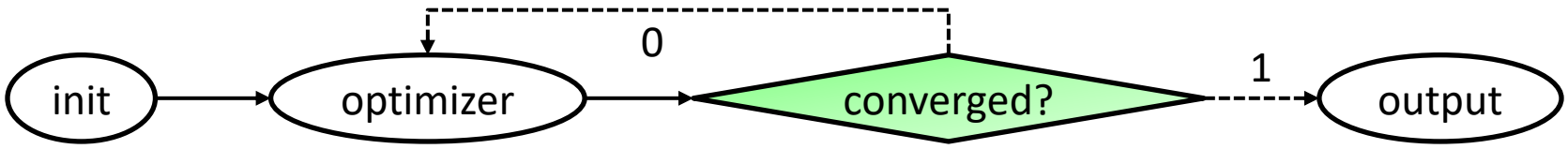
- **Our closure enables stateful interface**
  - Users capture data in reference to marshal data exchange between CPU and GPU tasks
- **Our closure hides implementation details judiciously**
  - We use cudaGraph (since cuda 10) due to its excellent performance, much faster than streams in large graphs
- **Our closure extend to new accelerator types**
  - syclFlow, openclFlow, coralFlow, tpuFlow, fpgaFlow, etc.

```
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);
});
```

We do not simplify kernel programming but **focus on *CPU-GPU tasking* that affects the performance to a large extent!** (same for data abstraction)

Conditional Tasking

```
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
```



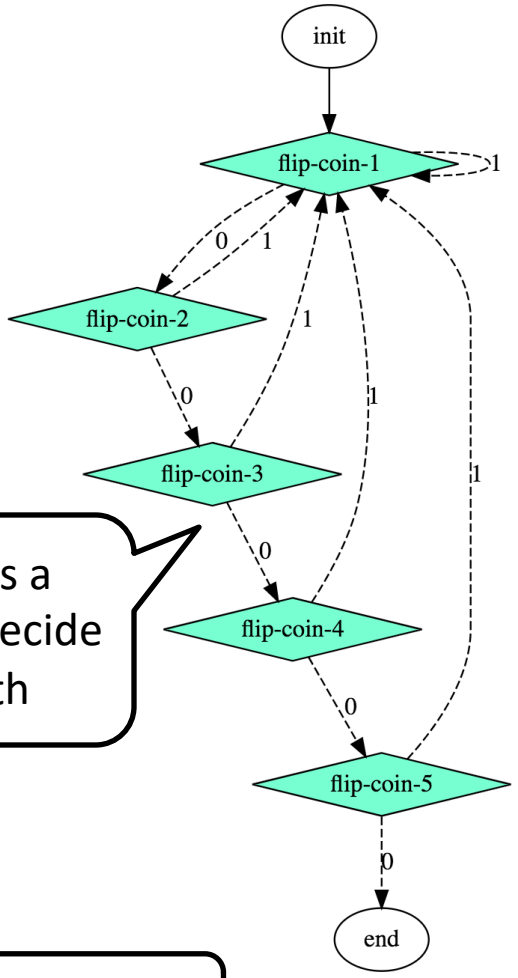
Tip!

Condition task integrates control flow into a task graph to form end-to-end parallelism; in this example, there are ultimately four tasks ever created

Conditional Tasking (cont'd)

```
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

Tip!

You can describe non-deterministic, nested control flow!

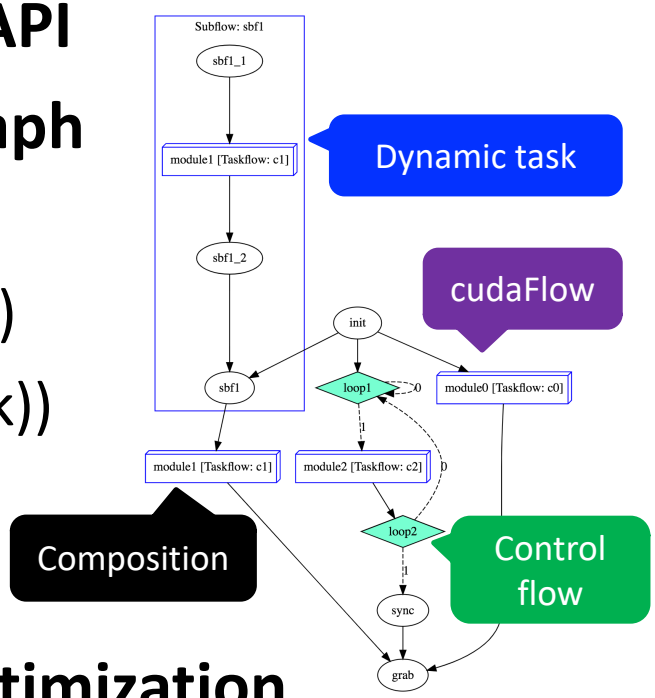
Existing Frameworks on Control Flow?

- **Expand a task graph across fixed-length iterations**
  - Graph size is linearly proportional to decision points
- **Unknown iterations? Non-deterministic conditions?**
  - Complex dynamic tasks executing "if" on the fly
- **Dynamic control flows and dynamic tasks?**
- **… (resort to client-side decision)**

Existing frameworks on expressing conditional tasking or dynamic control flow suffer from exponential growth of code complexity

Everything is Unified in Taskflow

- Use "`emplace`" to create a task
- Use "`precede`" to add a task dependency
- No need to learn different sets of API
- You can create a really complex graph
  - Subflow(ConditionTask(cudaFlow))
  - ConditionTask(StaticTask(cudaFlow))
  - Composition(Subflow(ConditionTask))
  - Subflow(ConditionTask(cudaFlow))
  - …
- Scheduler performs end-to-end optimization
  - Runtime, energy efficiency, and throughput

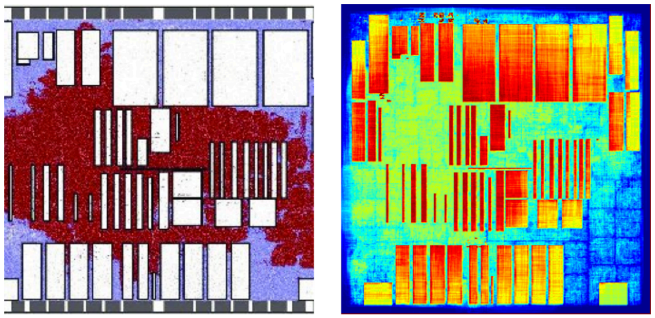


Agenda

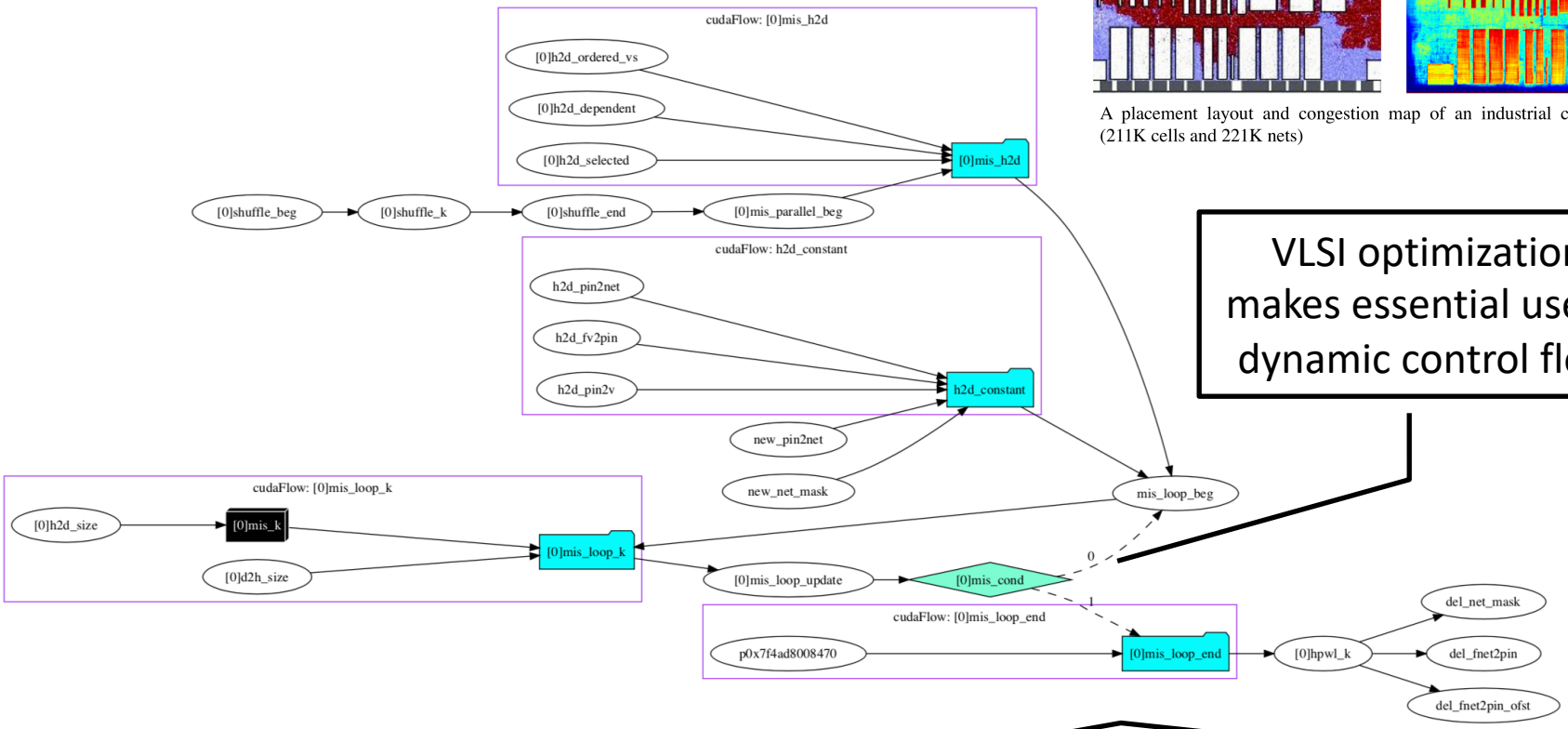
- ❑ Express your parallelism in the right way
- ❑ Parallelize your applications using Taskflow
- ❑ Boost performance in real applications

Application 1: VLSI Placement

- Optimize cell locations on a chip
  - GPU + CPUs



A placement layout and congestion map of an industrial circuit, adaptec1 (211K cells and 221K nets)

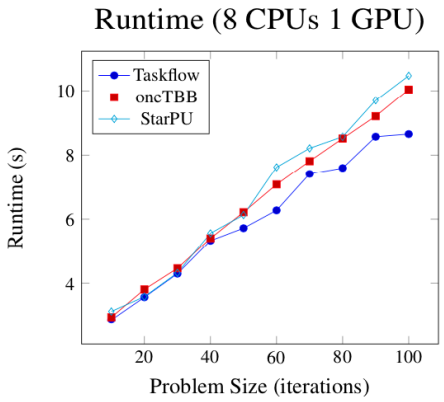


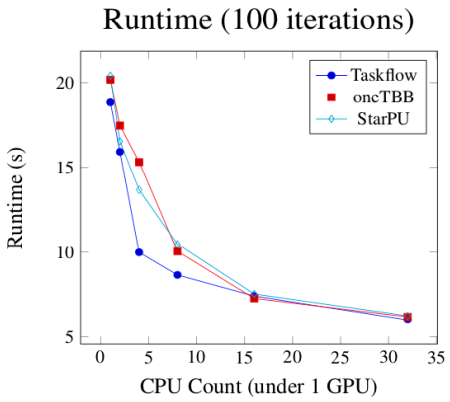
VLSI optimization makes essential use of dynamic control flow

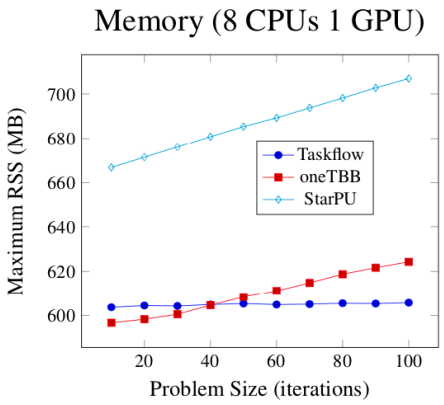
A partial TDG of 4 cudaFlows, 1 conditioned cycle, and 12 static tasks

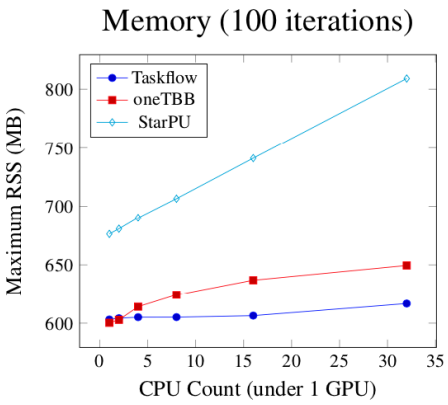
Application 1: VLSI Placement (cont'd)

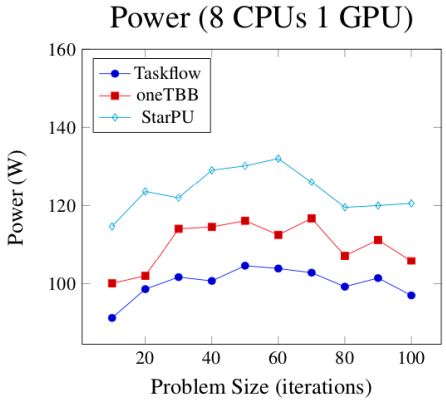
- **Runtime, memory, power, and throughput**

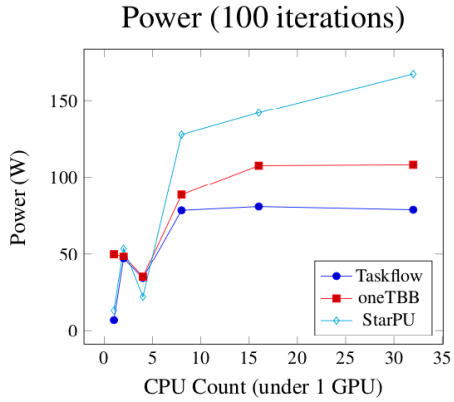








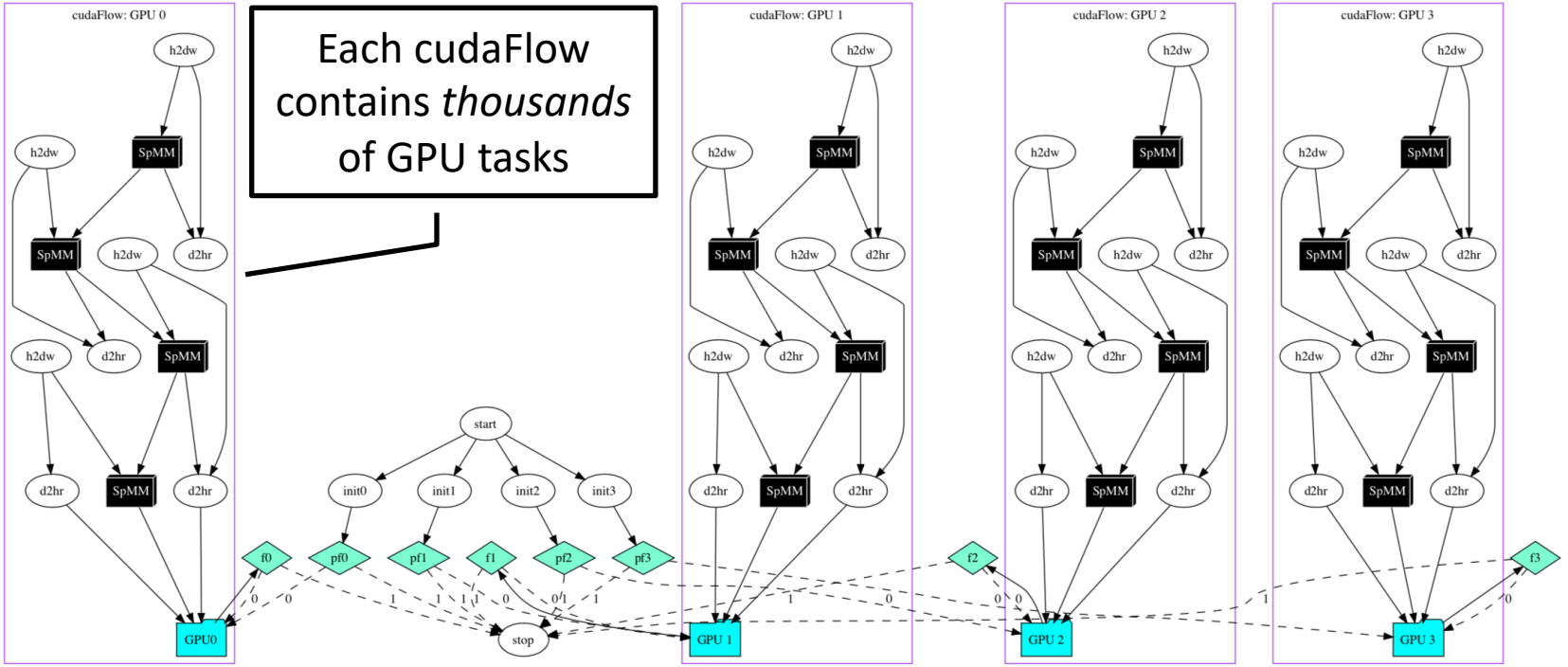




Performance improvement comes from *end-to-end* expression of CPU-GPU dependent tasks using condition tasks

Application 2: Machine Learning

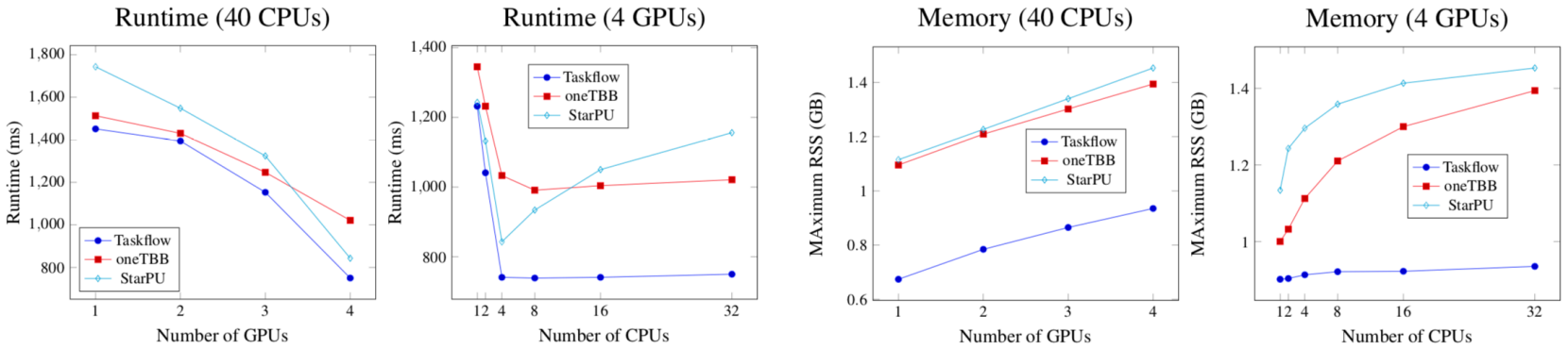
- **Compute a 1920-layer DNN each of 65536 neurons**
  - IEEE HPEC 2020 Neural Network Challenge Compute



A partial taskflow graph of 4 cudaFlows, 6 static tasks, and 8 conditioned cycles for this workload

Application 2: Machine Learning (cont'd)

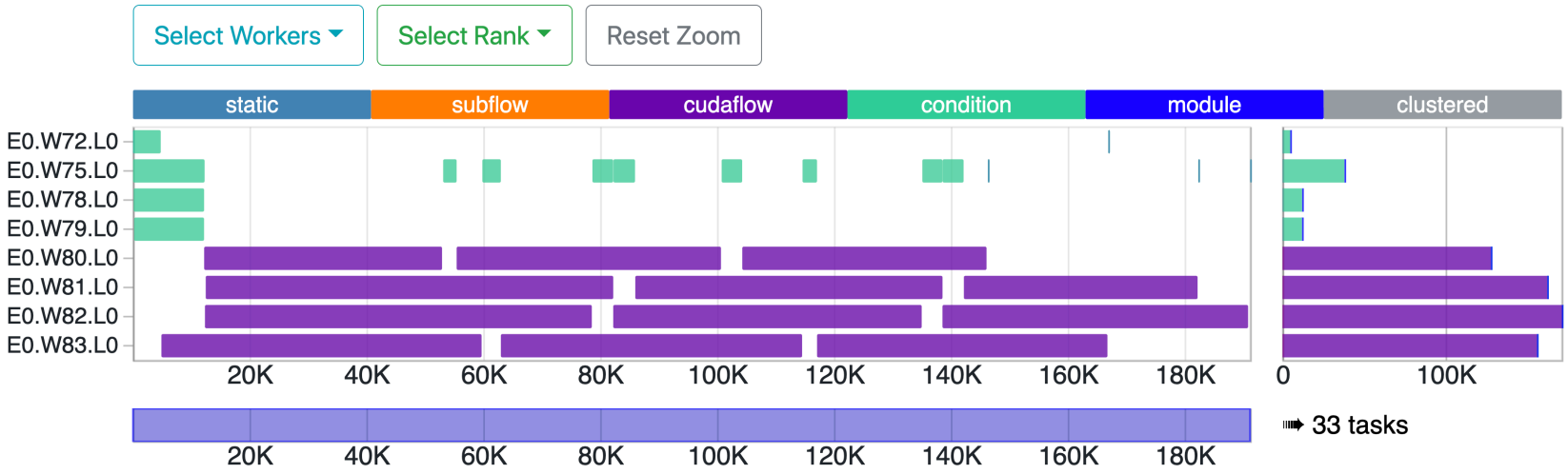
- **Comparison with TBB and StarPU**
  - Unroll task graphs across iterations found in hindsight
  - Implement cudaGraph for all

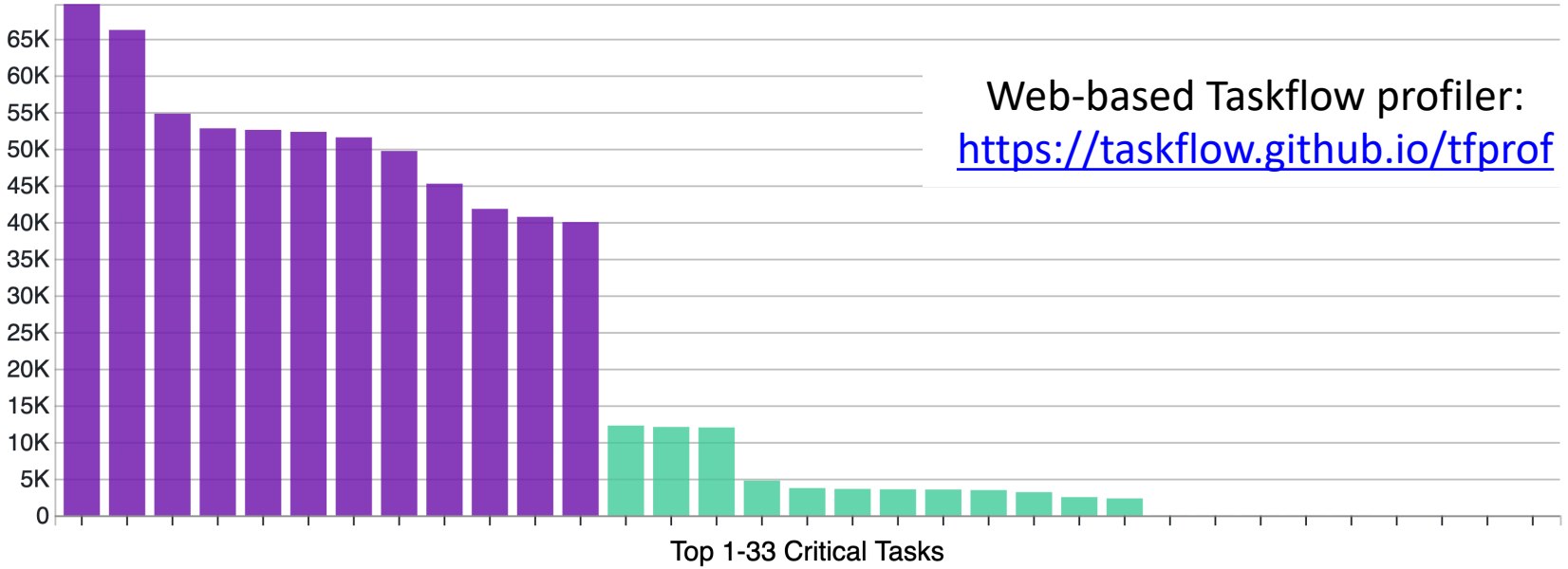


- **Taskflow's runtime is up to 2x faster**
- **Taskflow's memory is up to 1.6x less**

Our solutions received the champion award of IEEE MIT/Amazon/HPEC 2020 Graph Challenge: https://graphchallenge.mit.edu/champions

Application 2: Machine Learning (cont'd)





Web-based Taskflow profiler:
https://taskflow.github.io/tfprof

Parallel programming infrastructure matters

Different models give different implementations. The parallel algorithm itself may run fast, yet the parallel computing infrastructure to support that algorithm may dominate the entire performance.

Taskflow enables *end-to-end* expression of CPU-GPU dependent tasks along with algorithmic control flow

Thank You for Using Taskflow!

Taskflow: https://taskflow.github.io