Taskflow: A General-purpose Parallel and Heterogeneous Task Computing System

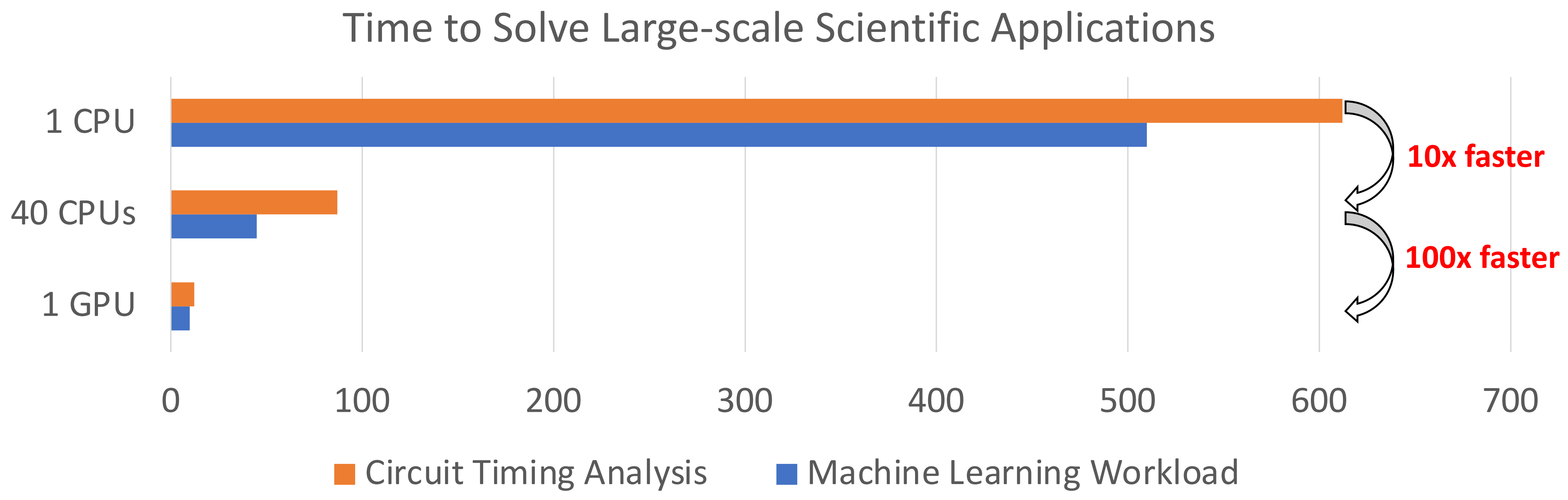
Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

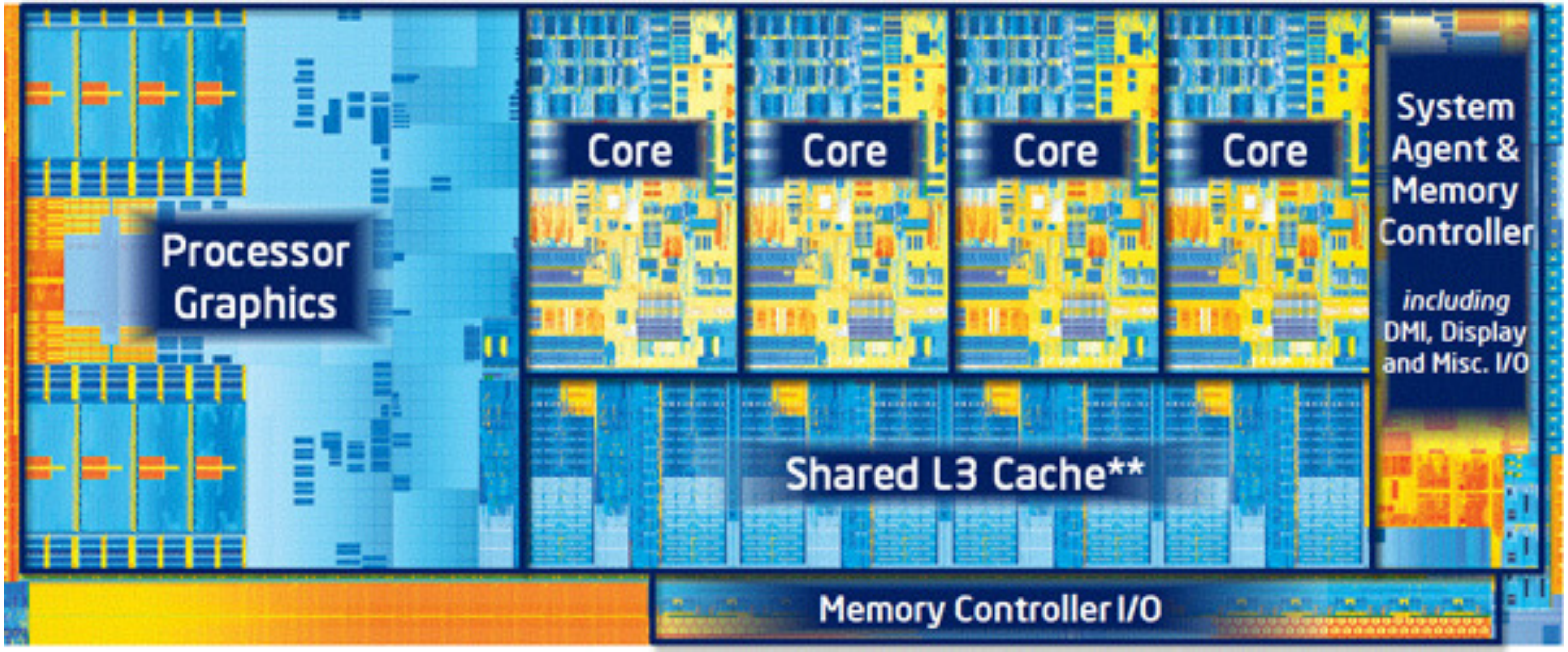
Why Parallel Computing?

- It's critical to advance your application performance

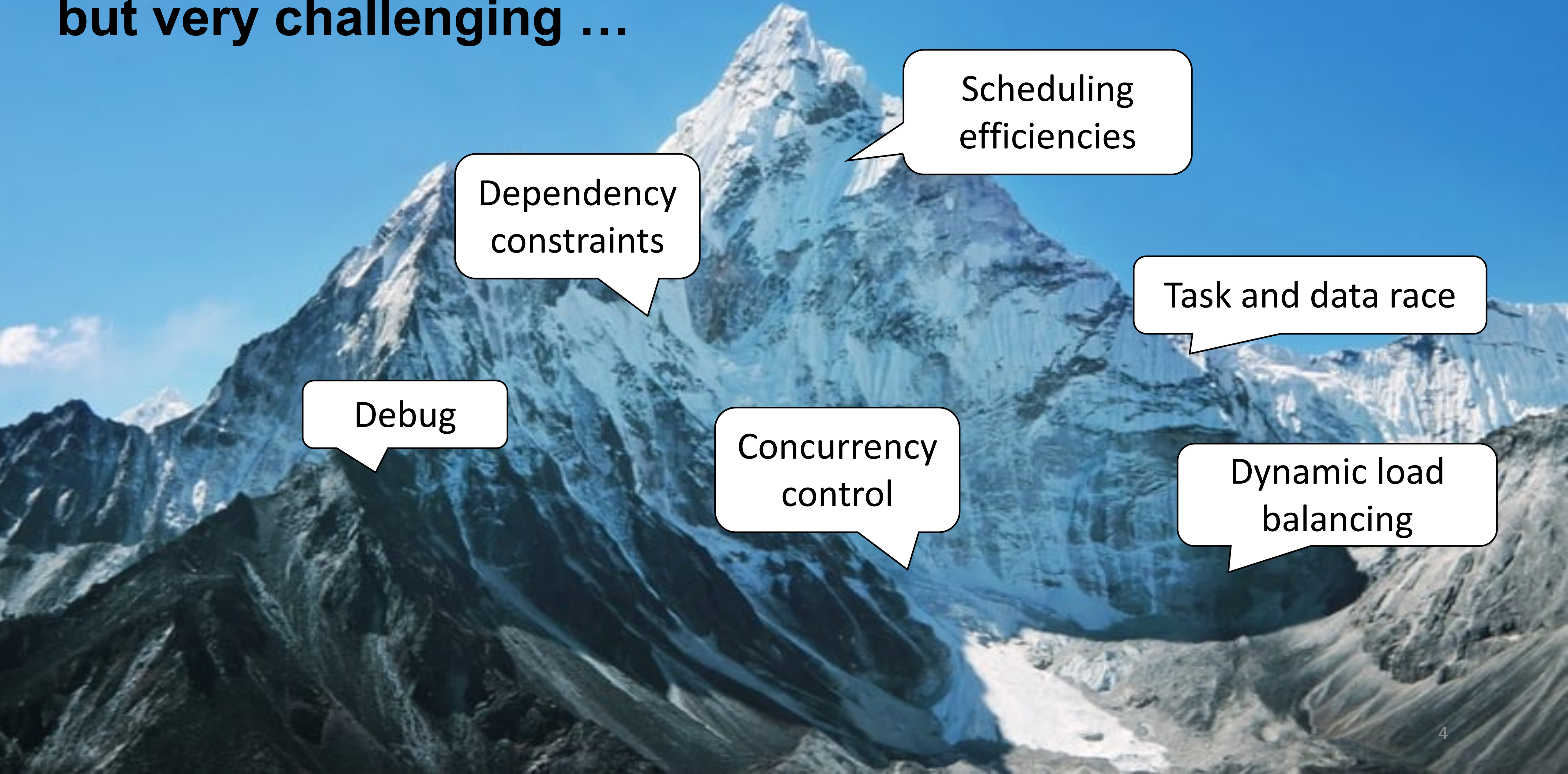


Your Computer is Already Parallel

- Intel i7-377K CPU (four cores to run your jobs in parallel)



Parallel programming is crucial but very challenging …

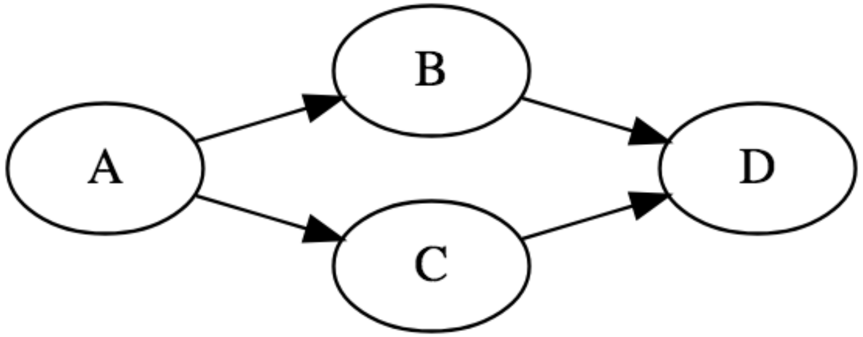


Taskflow offers a solution

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;
}
```



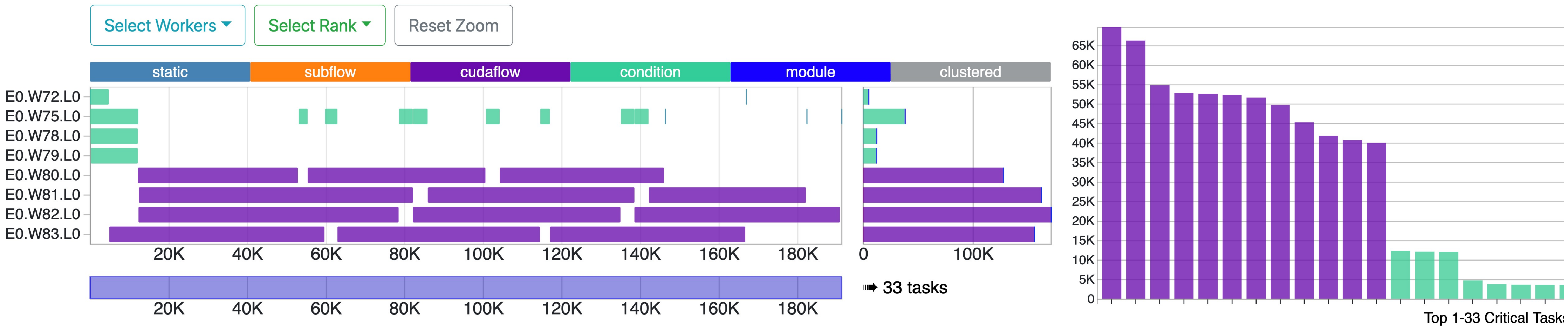
Drop-in Integration

- 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

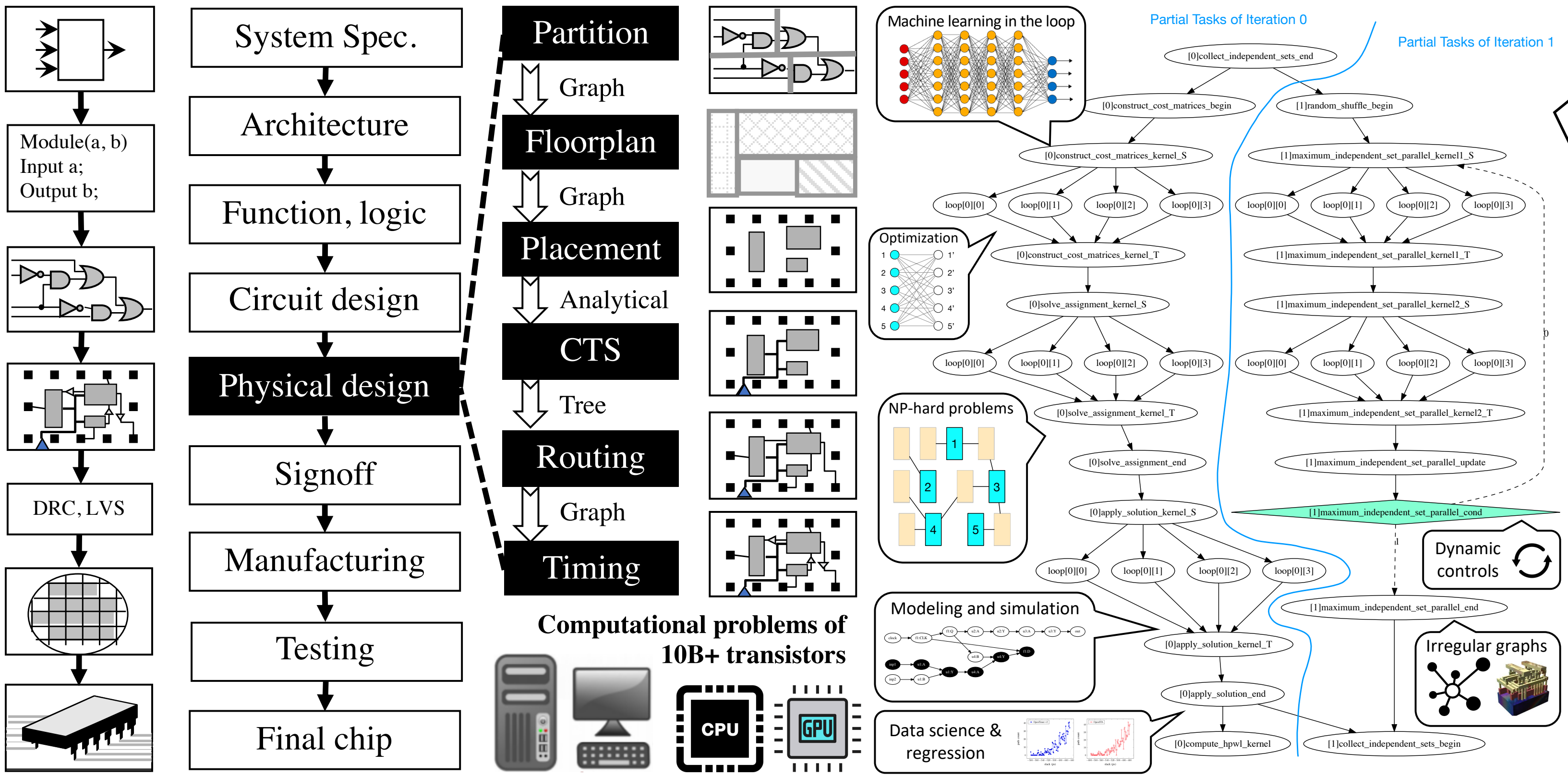
- Express your parallelism in the right way
- Parallelize your applications using Taskflow
- Understand our scheduling algorithm
- Boost performance in real applications
- Collaborate on using Taskflow in your applications

Agenda

- Express your parallelism in the right way
- Parallelize your applications using Taskflow
- Understand our scheduling algorithm
- Boost performance in real applications
- Collaborate on using Taskflow in your applications

Motivation: Parallelizing VLSI CAD Tools

- Billions of tasks with diverse computational patterns



"How can we write efficient C++ parallel programs for this *monster computational task graph* with ***millions of CPU-GPU dependent tasks along with algorithmic control flow***"

We Invested a lot in Existing Tools …

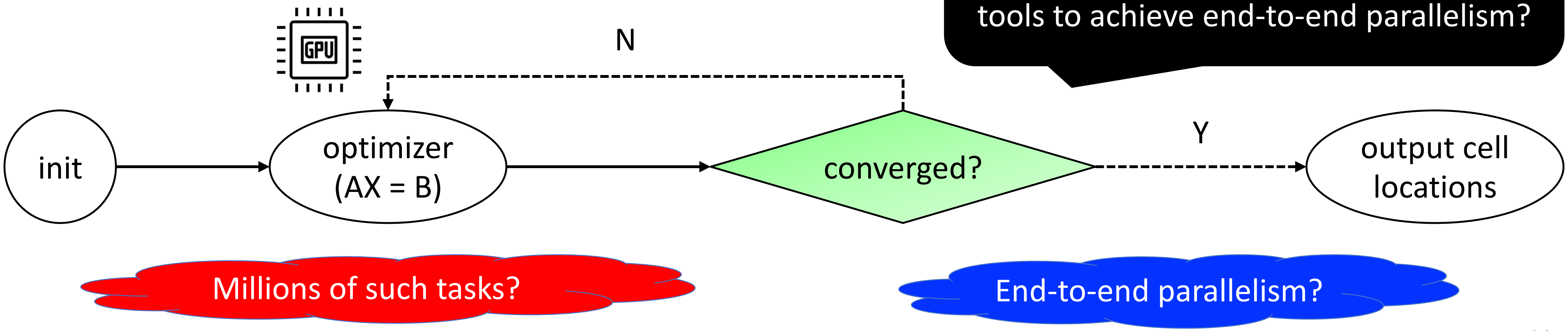
Two Big Problems of Existing Tools

- 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
    - Combinatorial optimization, analytical methods
  - **Problem**: existing tools are *directed acyclic graph* (DAG)-based and do not anticipate cycles or conditional dependencies, lacking *end-to-end* parallelism

Example: An Iterative Placement Optimizer

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

How can we easily describe this workload of *dynamic control flow* using existing tools to achieve end-to-end parallelism?



Need a New C++ 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
- Understand our scheduling algorithm
- Boost performance in real applications
- Collaborate on using Taskflow in your applications

WARNING

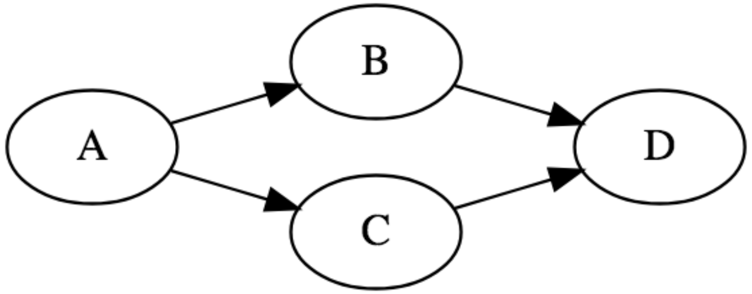
Code Ahead

#1: Static Task ("Hello World" Revisited)

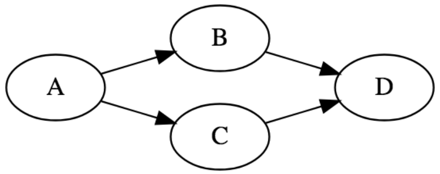
```
#include <taskflow/taskflow.hpp>   // Taskflow is header-only
int main(){
  tf::Taskflow taskflow;
  tf::Executor executor;
  auto [A, B, C, D] = taskflow.emplace(
    [] () { std::cout << "TaskA\n"; }
    [] () { std::cout << "TaskB\n"; },
    [] () { std::cout << "TaskC\n"; },
    [] () { std::cout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

This talk focuses on three task types:

1. static task
2. cudaFlow task
3. condition task



“Hello World” in OpenMP




```
#include <omp.h>  // OpenMP is a lang ext to describe parallelism using compiler directives
int main(){
   #omp parallel num_threads(std::thread::hardware_concurrency())
   {
      int A_B, A_C, B_D, C_D;
      #pragma omp task depend(out: A_B, A_C)
      {
         s t d : : c o u t << ”TaskA\n” ;
      }
      #pragma omp task depend(in: A_B; out: B_D)
      {
         s t d : : c o u t << ” TaskB\n” ;
      }
      #pragma omp task depend(in: A_C; out: C_D)
      {
         s t d : : c o u t << ” TaskC\n” ;
      }
      #pragma omp task depend(in: B_D, C_D)
      {
         s t d : : c o u t << ”TaskD\n” ;
      }
   }
   return 0;
}
```

Task dependency clauses

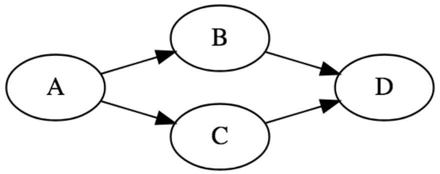
Task dependency clauses

Task dependency clauses

Task dependency clauses

OpenMP task clauses are static and explicit;
Programmers are responsible for a proper order of writing tasks consistent with sequential execution

“Hello World” in TBB




```
#include <tbb.h>  // Intel’s TBB is a general-purpose parallel programming library in C++
int main(){
   using namespace tbb;
   using namespace tbb:flow;
   int n = task_scheduler init::default_num_threads () ;
   task scheduler_init init(n);
   graph g;
   continue_node<continue_msg> A(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskA” ;
   }) ;
   continue_node<continue_msg> B(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskB” ;
   }) ;
   continue_node<continue_msg> C(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskC” ;
   }) ;
   continue_node<continue_msg> C(g, [] (const continue msg &) {
      s t d : : c o u t << “TaskD” ;
   }) ;
   make_edge(A, B);
   make_edge(A, C);
   make_edge(B, D);
   make_edge(C, D);
   A.try_put(continue_msg());
   g.wait_for_all();
}
```

Use TBB’s FlowGraph for task parallelism

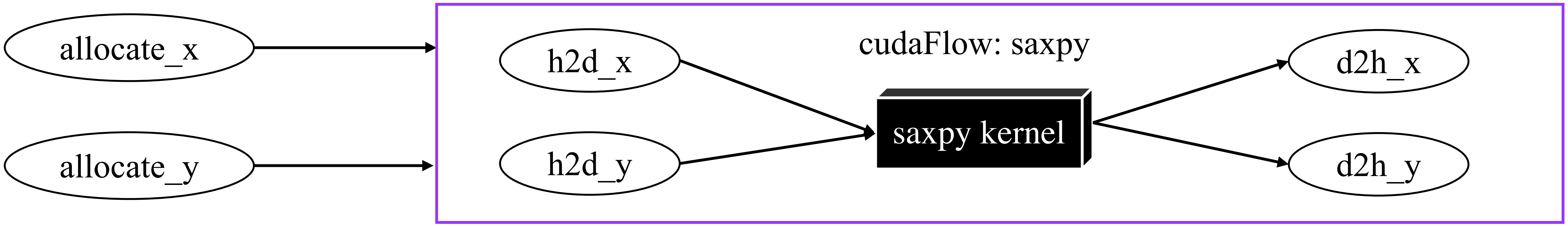
Declare a task as a continue_node

TBB has excellent performance in generic parallel computing. Its drawback is mostly in the ease-of-use standpoint (simplicity, expressivity, and programmability).

TBB FlowGraph: https://software.intel.com/content/www/us/en/develop/home.html

#2: cudaFlow Task

- Single Precision AX + Y (“SAXPY”)
  - Get x and y vectors on CPU (allocate_x, allocate_y)
  - Copy x and y to GPU (h2d_x, h2d_y)
  - Run saxpy kernel on x and y (saxpy kernel)
  - Copy x and y back to CPU (d2h_x, d2h_y)



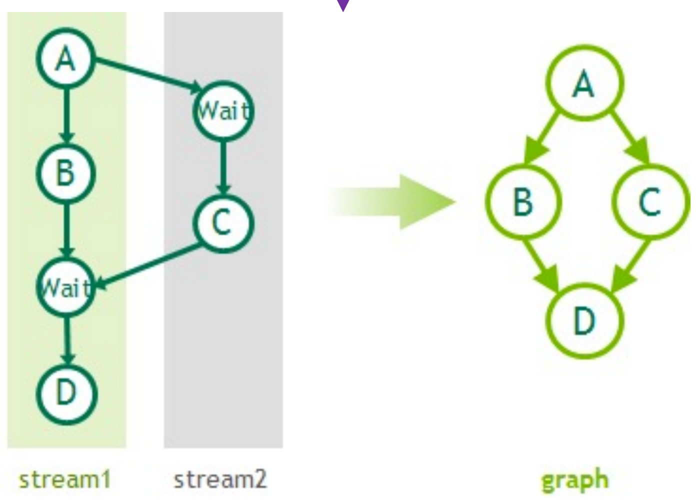
#2: cudaFlow Task (cont'd)

```
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();
```

To Nvidia
cudaGraph



Three Key Motivations of cudaFlow

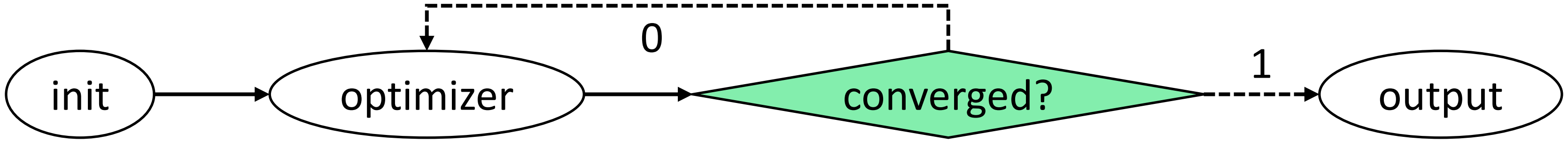
- Our closure enables stateful interface
  - Users capture data in reference to marshal data exchange between CPU and GPU tasks
- Our closure hides implementation details judiciously
  - We use cudaGraph (since cuda 10) due to its excellent performance, much faster than streams in large graphs
- Our 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)

#3: Condition Task (if-else)

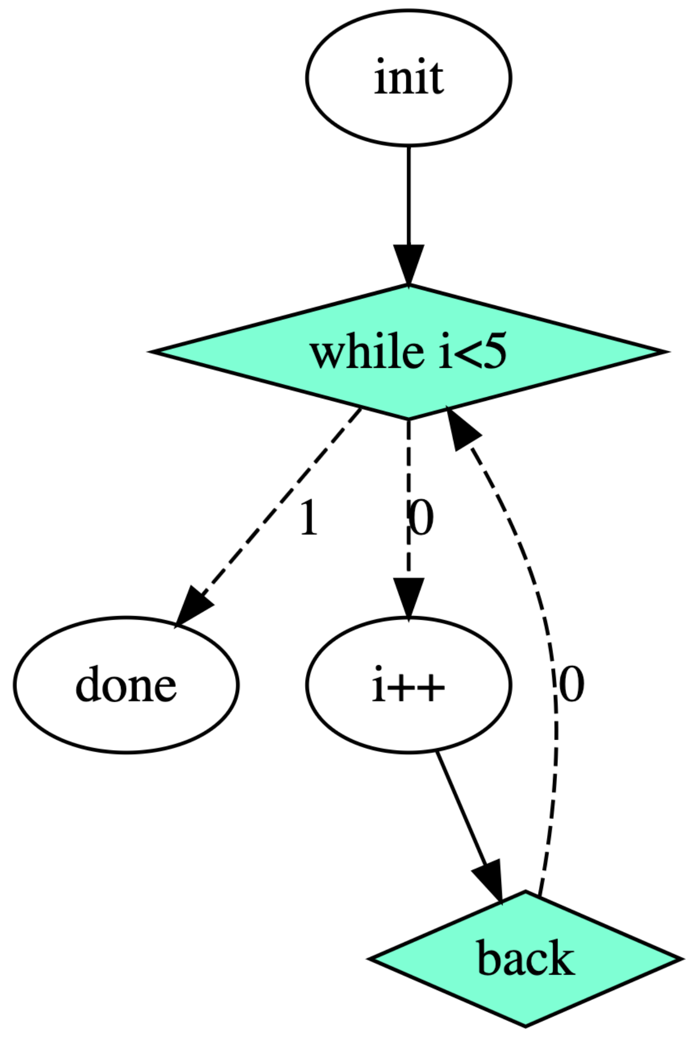
```
auto init        = taskflow.emplace([&](){ initialize_data_structure(); } )
                           .name(”init");
auto optimizer   = taskflow.emplace([&](){ matrix_solver(); } )
                           .name(”optimizer");
auto converged   = taskflow.emplace([&](){ return converged() ? 1 : 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 integrates control flow into a task graph to form **end-to-end** parallelism*

#3: Condition Task (While/For Loop)

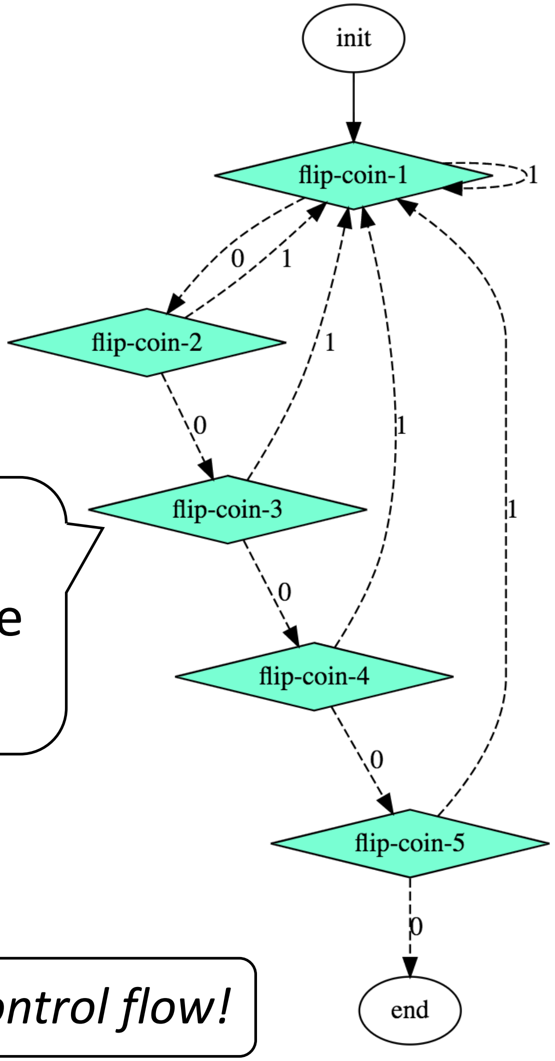
```
tf::Taskflow taskflow;
int i;
auto [init, cond, body, back, done] = taskflow.emplace(
  [&](){ std::cout << “i=0”; i=0; },
  [&](){ std::cout << "while i<5\n"; return i < 5 ? 0 : 1; },
  [&](){ std::cout << "i++=" << i++ << '\n'; },
  [&](){ std::cout << "back\n"; return 0; },
  [&](){ std::cout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



#3: Condition Task (Non-deterministic 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");
```

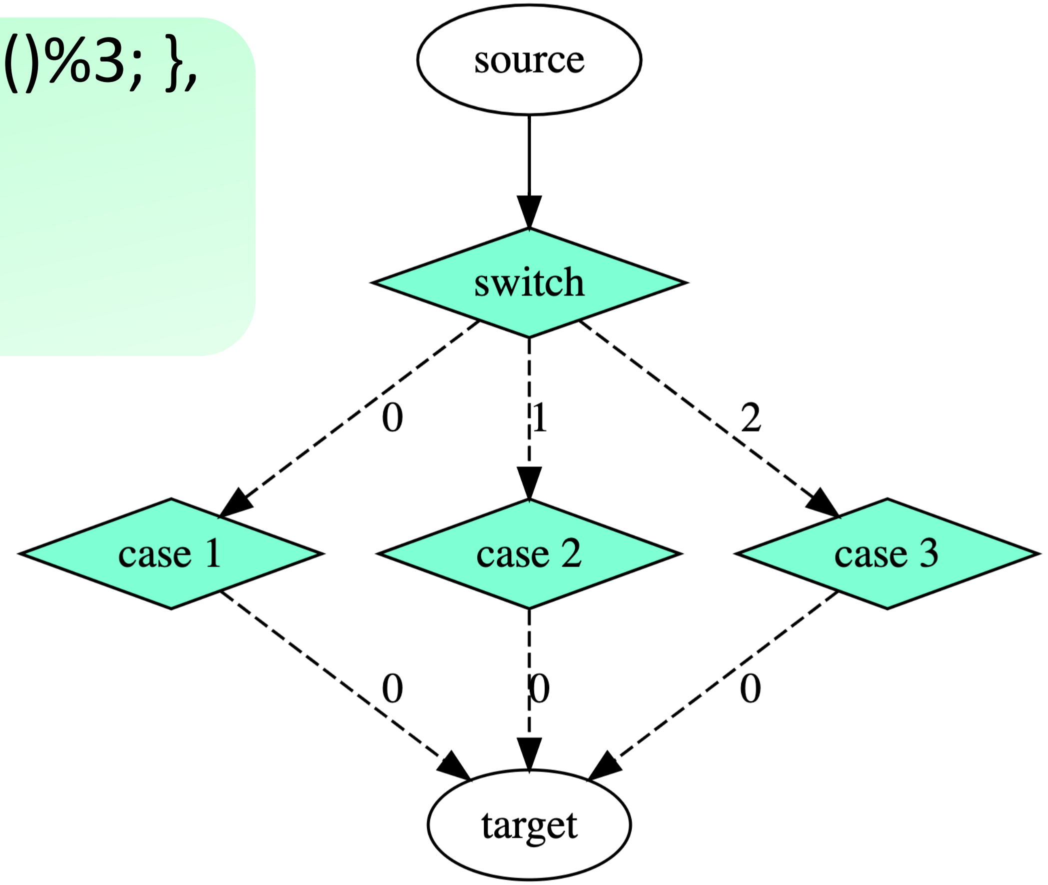
Each task flips a binary coin to decide the next path



You can describe non-deterministic, nested control flow!

#3: Condition Task (Switch)

```
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);
```



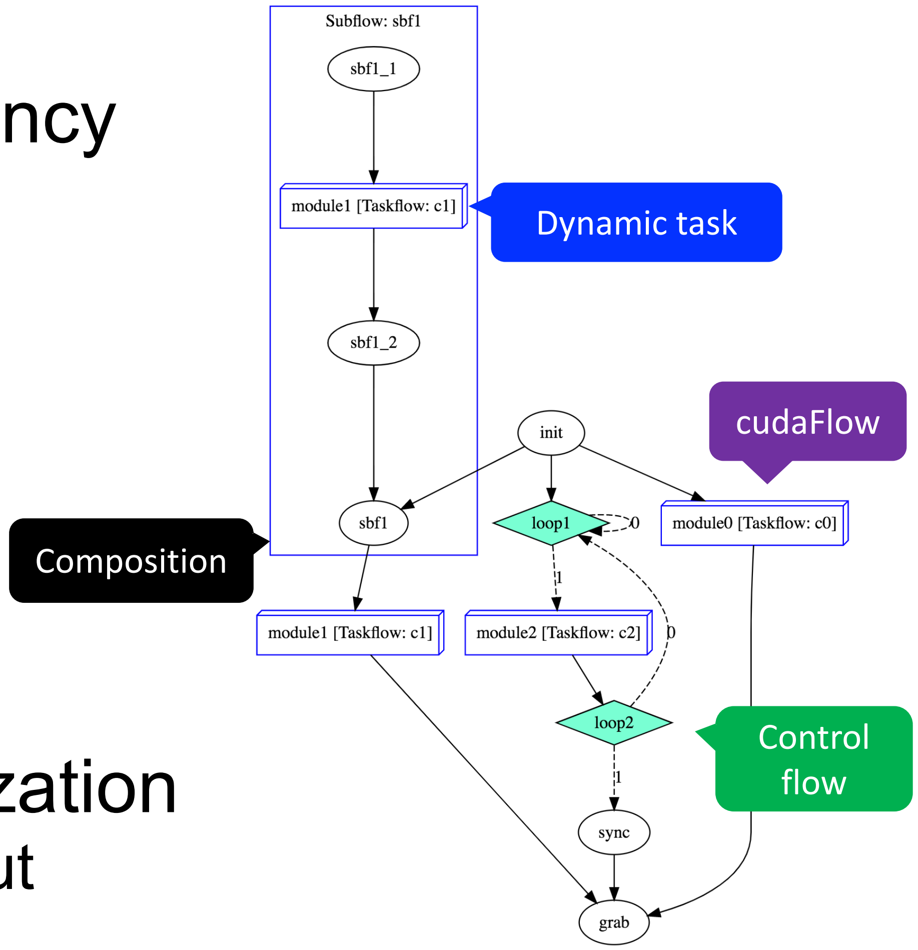
Existing Frameworks on Control Flow?

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

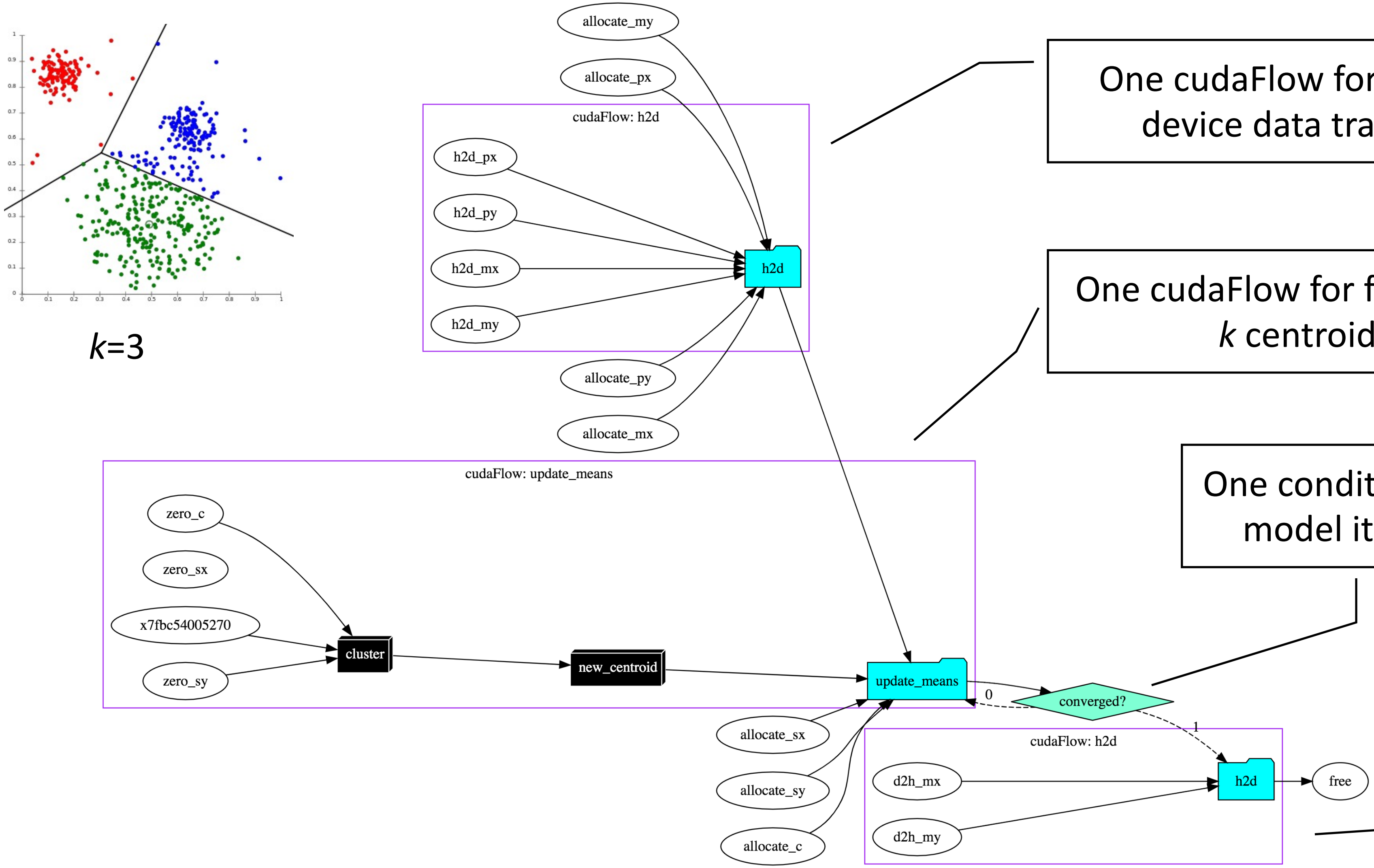
Existing frameworks on expressing conditional tasking or dynamic control flow suffer from exponential growth of code complexity

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



Example: *k*-means Clustering



k=3

One cudaFlow for host-to-device data transfers

One cudaFlow for finding the *k* centroids

One condition task to model iterations

One cudaFlow for device-to-host data transfers

Agenda

- Express your parallelism in the right way
- Parallelize your applications using Taskflow
- Understand our scheduling algorithm
- Boost performance in real applications
- Collaborate on using Taskflow in your applications

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*

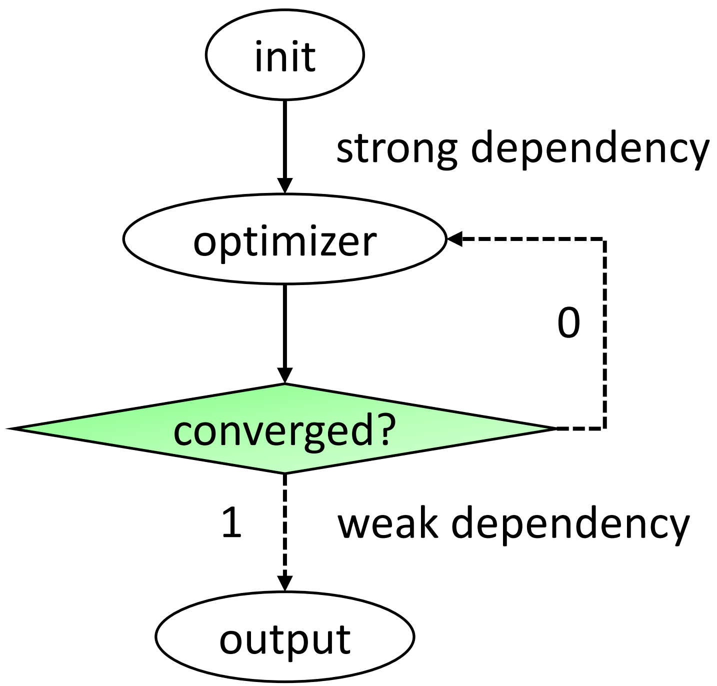
```
{
  tf::Taskflow taskflow1, taskflow2, taskflow3;
  tf::Executor executor;
  // create tasks and dependencies
  // …
  auto  future1 = executor.run(taskflow1);
  auto  future2 = executor.run_n(taskflow2, 1000);
  auto  future3 = executor.run_until(taskflow3, [i=0](){ return i++>5 });
  executor.async([](){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

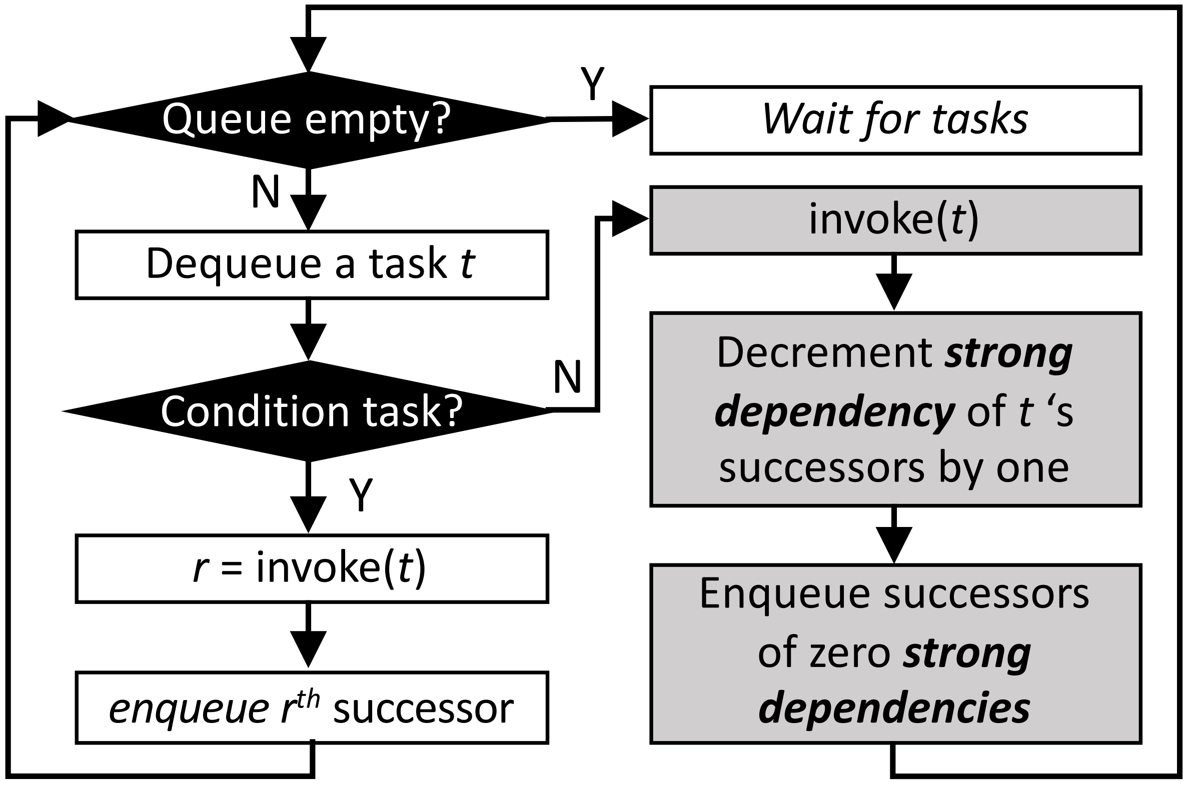
Executor Scheduling Algorithm

- Task-level scheduling
  - Decides how tasks are enqueued under control flow
    - Goal #1: ensures a feasible path to carry out control flow
    - Goal #2: avoids task race under cyclic and conditional execution
    - Goal #3: maximizes the capability of conditional tasking
- Worker-level scheduling
  - Decides how tasks are executed by which workers
    - Goal #1: adopts work stealing to dynamically balance load
    - Goal #2: adapts workers to available task parallelism
    - Goal #3: maximizes performance, energy, and throughput

Task-level Scheduling

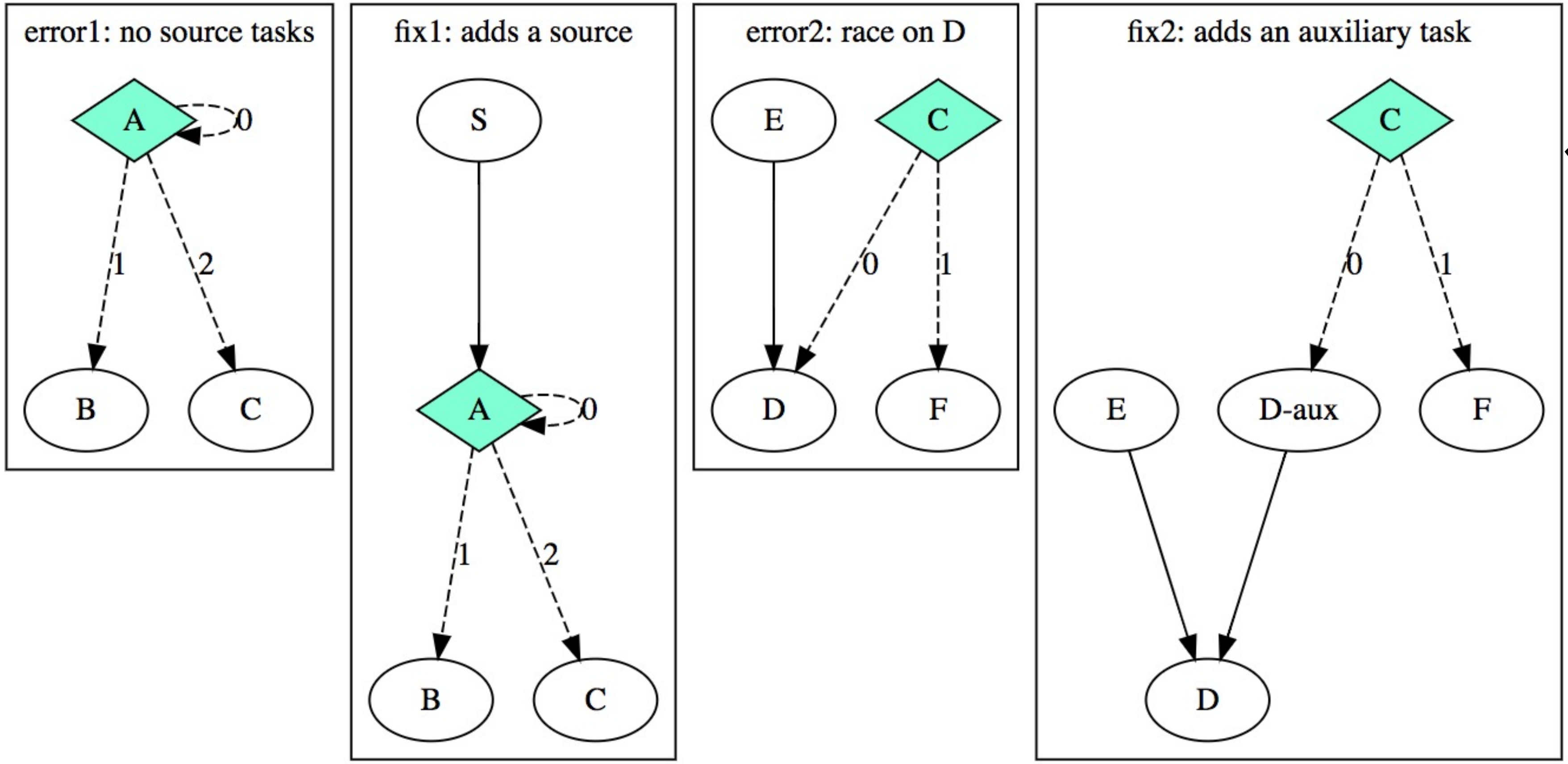
- *"Strong dependency"* versus "*Weak dependency*"
  - Weak dependency: dependencies out of condition tasks
  - Strong dependency: others else





Task-level Scheduling (cont'd)

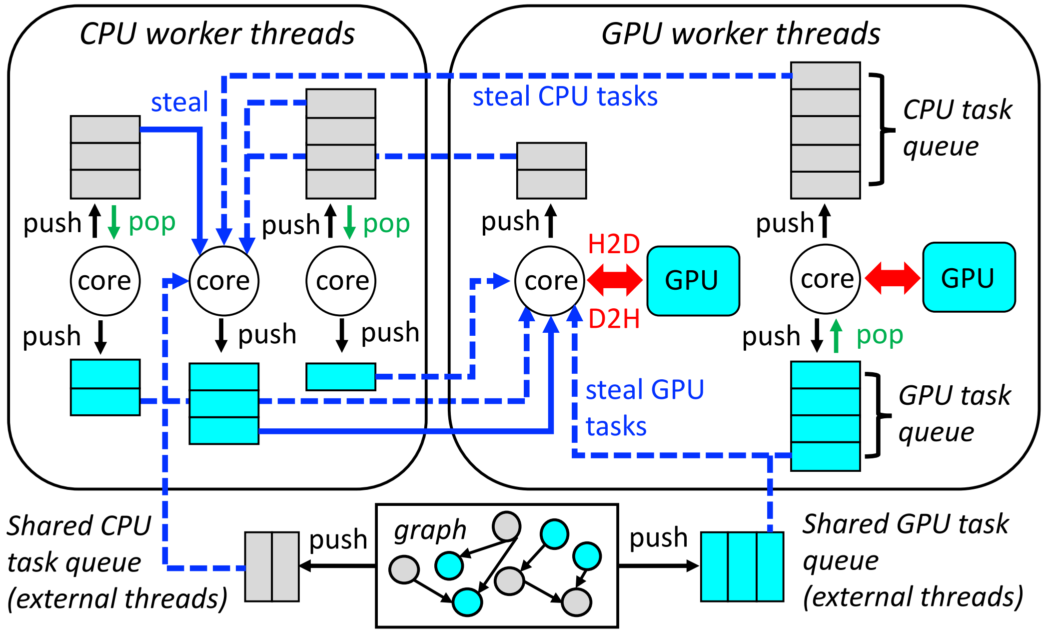
- Condition task is powerful but prone to mistakes …

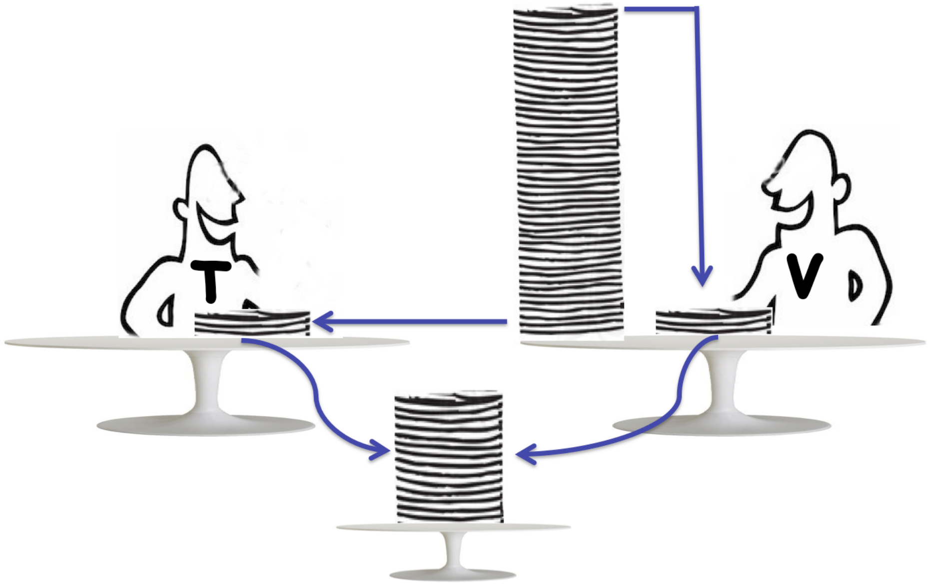


It is users' responsibility to ensure a taskflow is properly conditioned, i.e., no task race under our task-level scheduling policy

Worker-level Scheduling

- Taskflow adopts *work stealing* to gain *dynamic load balancing*
- What is work stealing? Why?
  - I finish my jobs first, and then steal jobs from you
  - So, we can improve performance and balance our loads



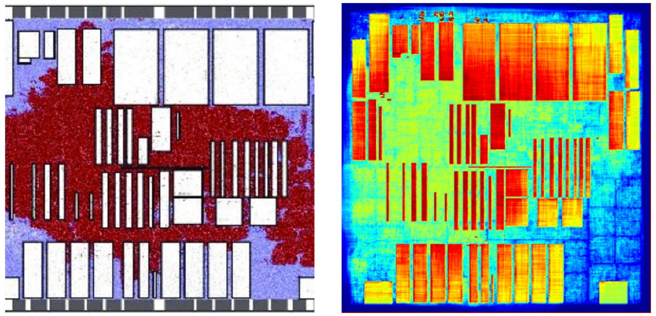


Agenda

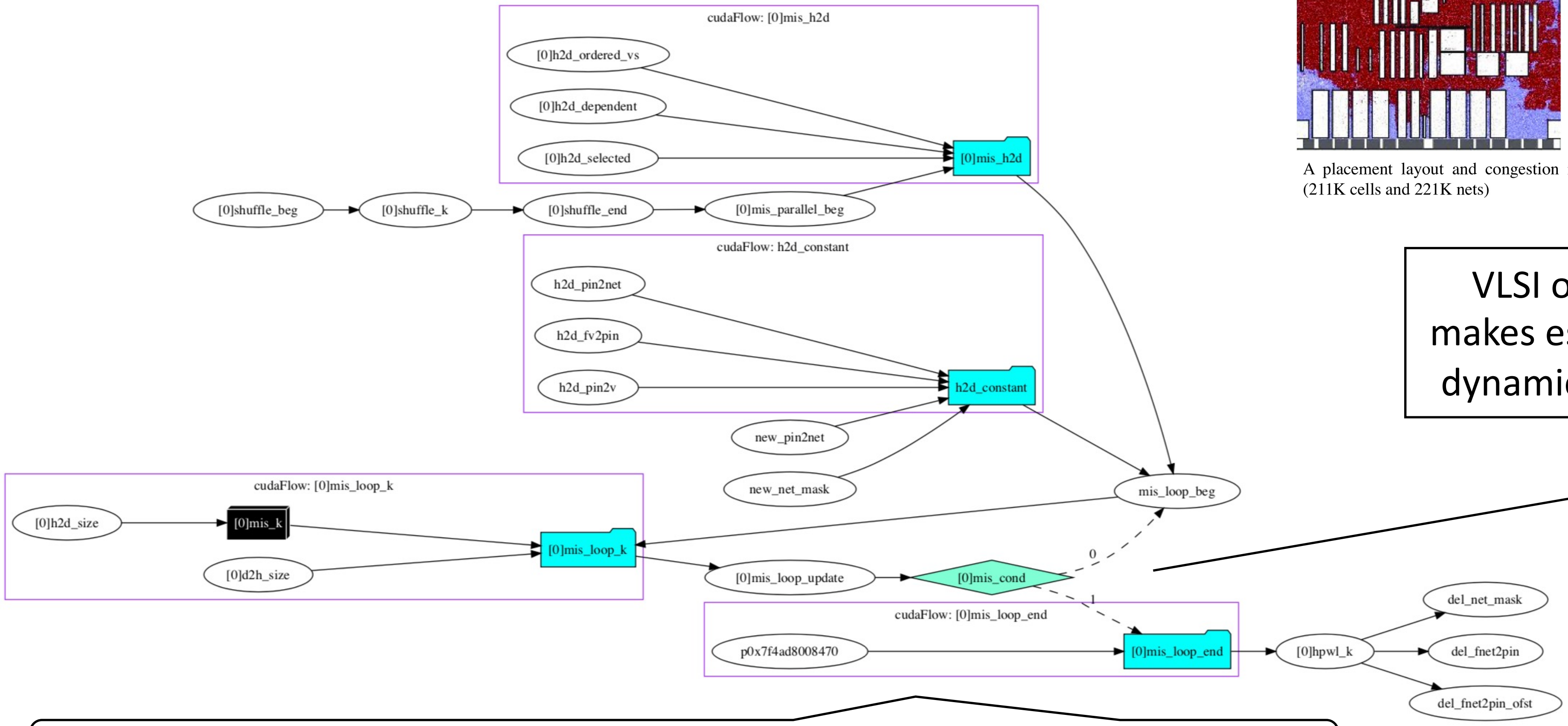
- Express your parallelism in the right way
- Parallelize your applications using Taskflow
- Understand our scheduling algorithm
- Boost performance in real applications
- Collaborate on using Taskflow in your applications

Application 1: VLSI Placement

- Optimize cell locations on a chip



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

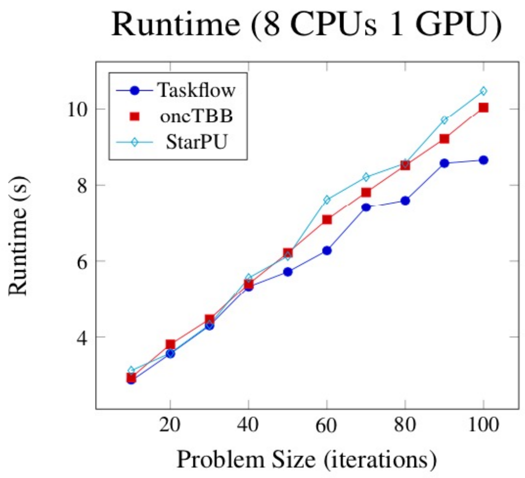


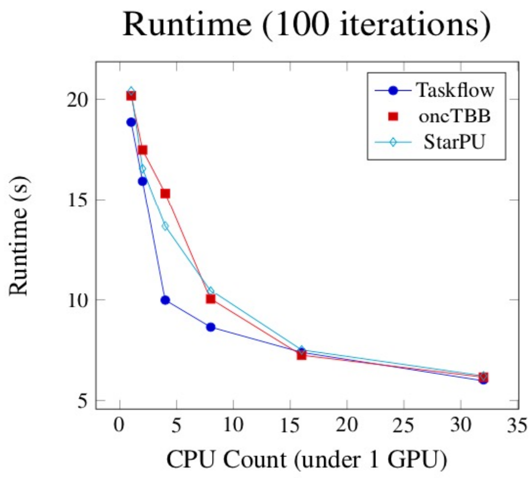
VLSI optimization makes essential use of dynamic control flow

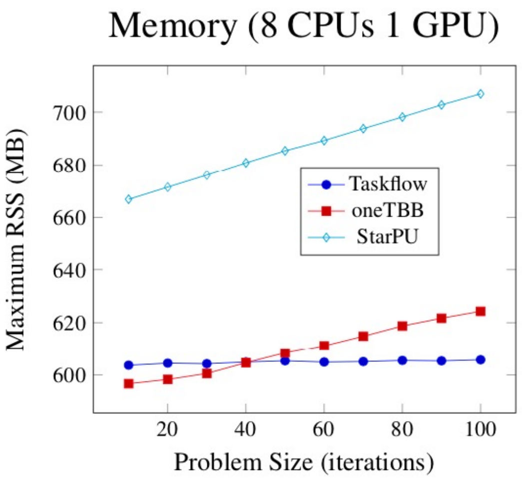
A partial TDG of 4 cudaFlows, 1 conditioned cycle, and 12 static tasks

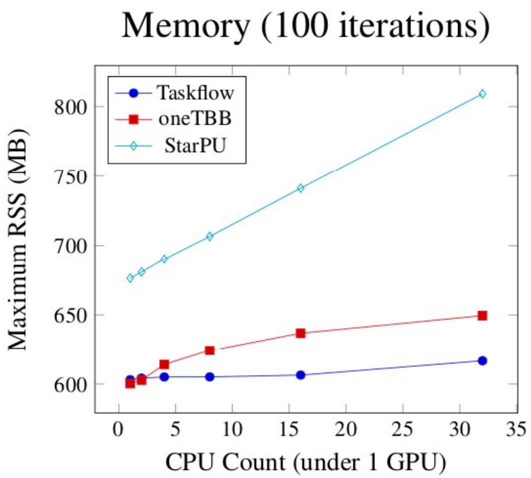
Application 1: VLSI Placement (cont'd)

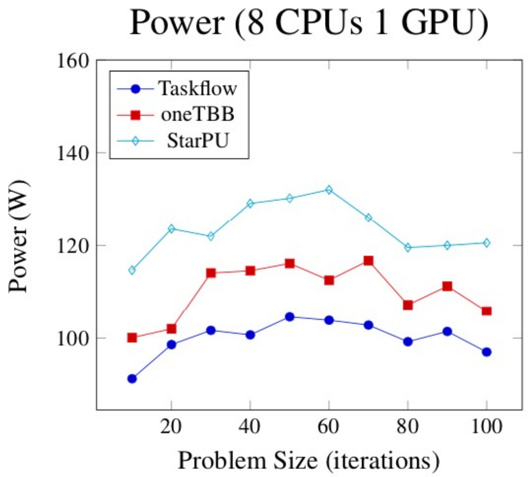
- Runtime, memory, power, and throughput

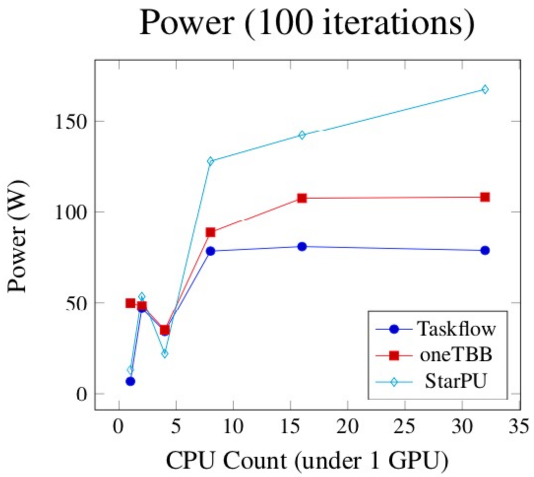








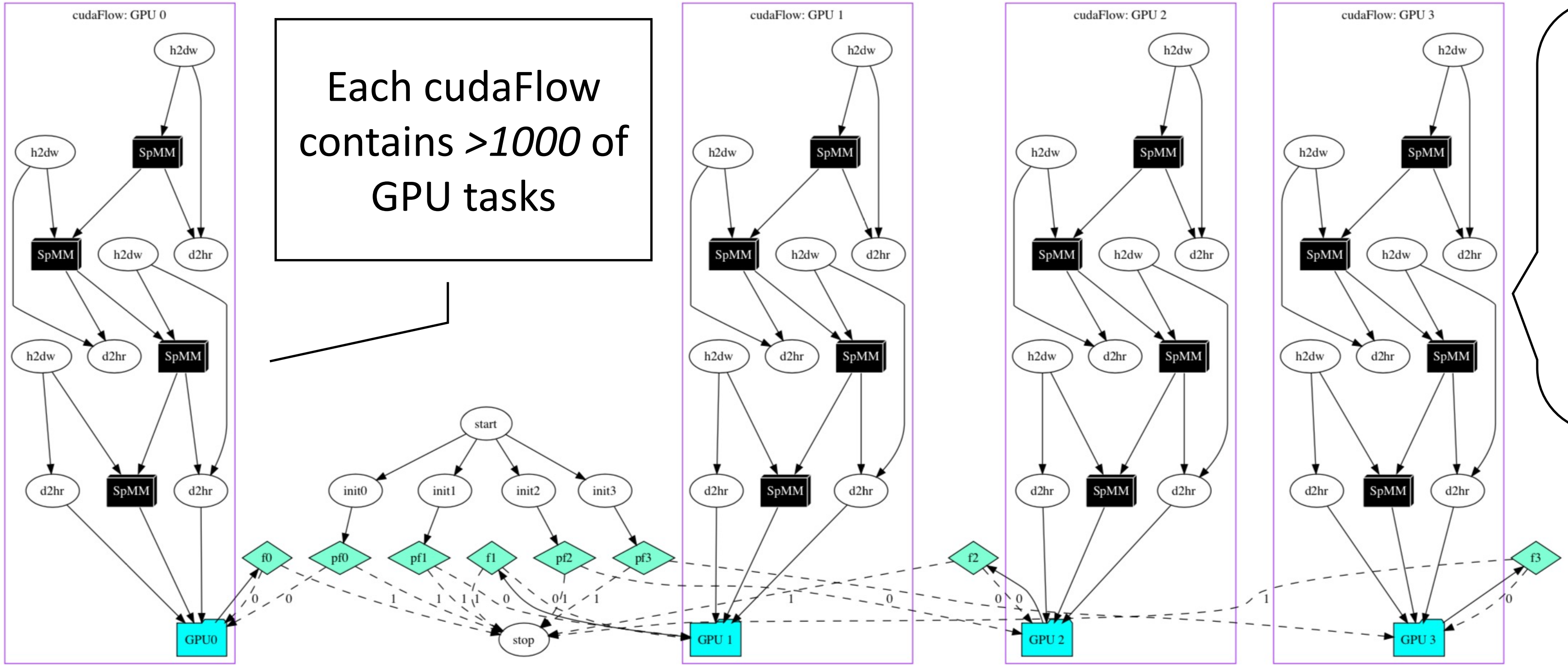




Performance improvement comes from the *end-to-end* expression of CPU-GPU dependent tasks using condition tasks

Application 2: Machine Learning

- IEEE HPEC/MIT/Amazon Sparse DNN Challenge
  - Compute a 1920-layer DNN each of 65536 neurons



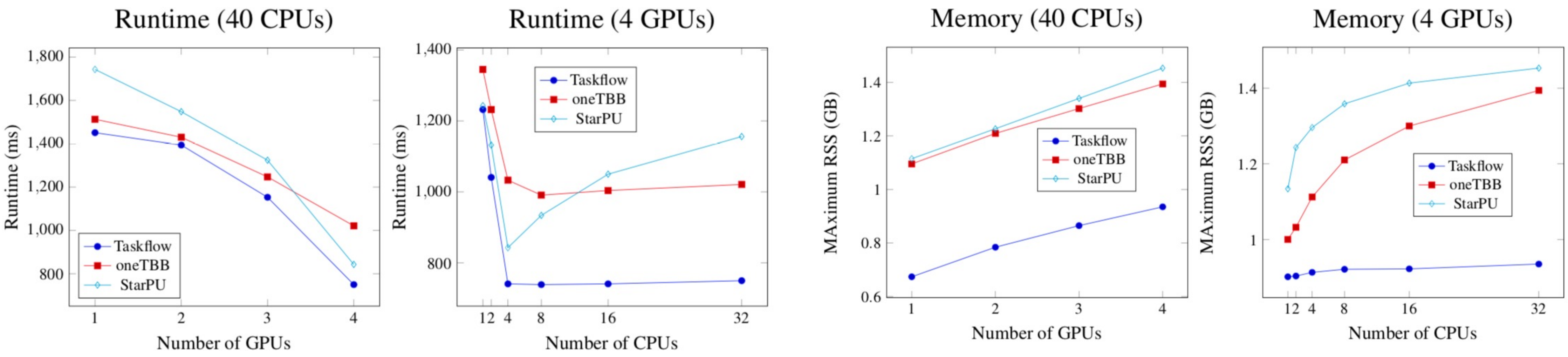
Each cudaFlow contains >*1000* of GPU tasks

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

Champions of HPEC 2020 Graph Challenge: https://graphchallenge.mit.edu/champions

Application 2: Machine Learning (cont'd)

- Comparison with TBB and StarPU



- Taskflow's runtime is up to 2x faster
  - Adaptive work stealing balances the worker count with task parallelism
- Taskflow's memory is up to 1.6x less
  - Conditional tasking allows efficient reuse of tasks

Summary of Experiments

- Parallel computing infrastructure matters

Different models give different implementations. The parallel code/algorithm 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

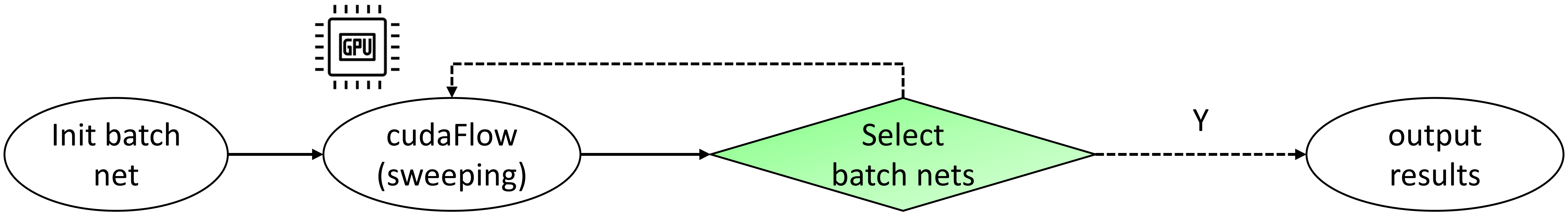
Agenda

- Express your parallelism in the right way
- Parallelize your applications using Taskflow
- Understand our scheduling algorithm
- Boost performance in real applications
- Collaborate on using Taskflow in your applications


Parallelism is
never standalone

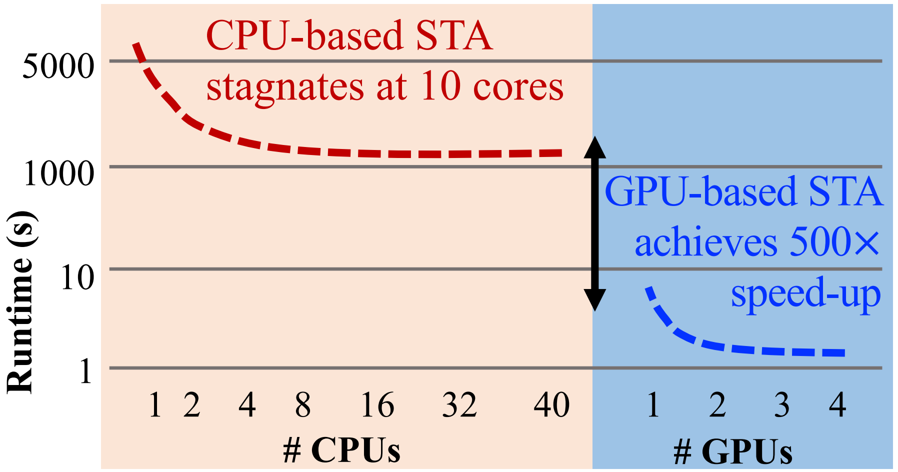
GAMER + Taskflow

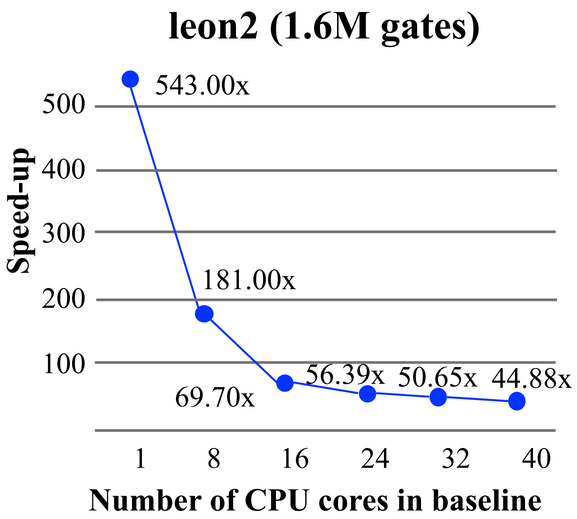
- GAMER accelerates maze routing using GPU
  - Primary innovation is GPU kernel – iterative parallel sweeping
- Potential collaboration topics for TCAD extension
  - Leverage Taskflow to achieve end-to-end heterogeneous parallelism
    - Condition tasks allow you to integrate control-flow decisions into dependencies
  - Leverage cudaFlow to reduce the kernel call overheads
  - Leverage multiple GPUs to handle a batch of nets

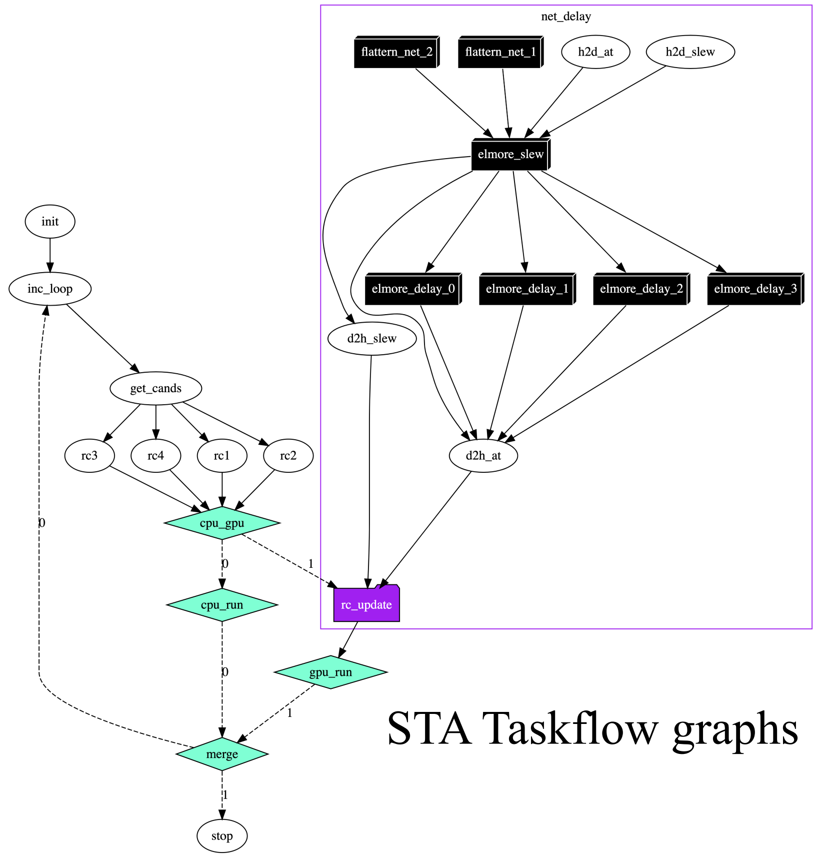


Prior Collaborative Results with Taskflow

- We collaborate with Peking U to accelerate STA with Taskflow
  - GPU-accelerated GBA [ICCAD'20]
  - GPU-accelerated PBA [DAC'21, ICCAD'21]
  - GPU-accelerated CPPR [ICCAD'21]
  - Taskflow system [TCAD'21, TPDS'21]
  - …







STA Taskflow graphs

Conclusion

- Taskflow is a lightweight parallel task programming system
  - Simple, efficient, and transparent tasking models
  - Efficient heterogeneous work-stealing executor
  - Promising performance in large-scale ML and VLSI CAD
- Taskflow is not to replace anyone but to
  - Complement the current state-of-the-art
  - Leverage modern C++ to express task graph parallelism
- Taskflow is very open to collaboration
  - We want to provide more higher-level algorithms
  - We want to broaden real use cases
  - We want to enhance the core functionalities (e.g., pipeline)

Thank You All Using Taskflow!

Use the right tool for the right job

Taskflow: https://taskflow.github.io

Thank You

Dr. Tsung-Wei Huang

tsung-wei.huang@utah.edu