Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System

Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin

Abstract—Taskflow aims to streamline the building of parallel and heterogeneous applications using a lightweight task graph-based approach. Taskflow introduces an expressive task graph programming model to assist developers in the implementation of parallel and heterogeneous decomposition strategies on a heterogeneous computing platform. Our programming model distinguishes itself as a very general class of task graph parallelism with in-graph control flow to enable end-to-end parallel optimization. To support our model with high performance, we design an efficient system runtime that solves many of the new scheduling challenges arising out of our models and optimizes the performance across latency, energy efficiency, and throughput. We have demonstrated the promising performance of Taskflow in real-world applications. As an example, Taskflow solves a large-scale machine learning workload up to 29% faster, 1.5× less memory, and 1.9× higher throughput than the industrial system, oneTBB, on a machine of 40 CPUs and 4 GPUs. We have opened the source of Taskflow and deployed it to large numbers of users in the open-source community.

Index Terms—Parallel programming, task parallelism, high-performance computing, modern C++ programming

1 Introduction

Task graph computing system (TGCS) plays an essential role in advanced scientific computing. Unlike loop-based models, TGCSs encapsulate function calls and their dependencies in a top-down task graph to implement *irregular* parallel decomposition strategies that scale to large numbers of processors, including manycore central processing units (CPUs) and graphics processing units (GPUs). As a result, recent years have seen a great deal amount of TGCS research, just name a few, oneTBB FlowGraph [2], StarPU [17], TPL [40], Legion [18], Kokkos-DAG [24], PaRSEC [20], HPX [34], and Fastflow [15]. These systems have enabled vast success in a variety of scientific computing applications, such as machine learning, data analytics, and simulation.

However, three key limitations prevent existing TGCSs from exploring the full potential of task graph parallelism. First, existing TGCSs closely rely on directed acyclic graph (DAG) models to define tasks and dependencies. Users implement *control-flow* decisions outside the graph description, which typically results in rather complicated implementations that lack *end-to-end parallelism*. For instance, when encountering an if-else block, users need to synchronize the graph execution with a TGCS runtime, which could otherwise be omitted if in-graph control-flow tasks are supported. Second, existing TGCSs do not align well with modern hardware. In particular, new GPU task graph parallelism, such as CUDA Graph, can bring significant yet largely untapped performance benefits. Third, existing TGCSs are good at either CPU- or GPU-focused workloads, but rarely both simultaneously. Consequently, we introduce in this paper *Taskflow*, a lightweight TGCS to overcome these limitations. We summarize three main contributions of Taskflow as follows:

- **Expressive programming model** – We design an expressive task graph programming model by leveraging modern C++ closure. Our model enables efficient implementations of parallel and heterogeneous decomposition strategies using the task graph model. The expressiveness of our model lets developers perform rather a lot of work with relative ease of programming. Our user experiences lead us to believe that, although it requires some effort to learn, a programmer can master our APIs needed for many applications in just a few hours.
- **In-graph control flow** – We design a new conditional tasking model to support *in-graph control flow* beyond the capability of traditional DAG models that prevail in existing TGCSs. Our condition tasks enable developers to integrate control-flow decisions, such as conditional dependencies, cyclic execution, and non-deterministic flows into a task graph of end-to-end parallelism. In case applications have frequent dynamic behavior, such as optimization and branch and bound, programmers can efficiently overlap tasks both inside and outside the control flow to hide expensive control-flow costs.
- **Heterogeneous work stealing** – We design an efficient work-stealing algorithm to adapt the number of workers to dynamically generated task parallelism at any time during the graph execution. Our algorithm prevents the graph execution from underutilized threads that is harmful to performance, while avoiding excessive waste of thread resources when available tasks are scarce. The result largely improves the overall system performance, including latency, energy usage, and throughput. We have derived theory results to justify the efficiency of our work-stealing algorithm.

- *Tsung-Wei Huang, and Dian-Lun Lin are with the Department of Electrical and Computer Engineering, the University of Utah, Salt Lake City, UT.*
- *Chun-Xun Lin is with MathWorks, USA.*
- *Yibo Lin is with the Department of Computer Science, Peking University, Beijing, China.*

We have evaluated Taskflow on real-world applications to demonstrate its promising performance. As an example, Taskflow solved a large-scale machine learning problem up to 29% faster, 1.5× less memory, and 1.9× higher throughput than the industrial system, oneTBB [2], on a machine of 40 CPUs and 4 GPUs. We believe Taskflow stands out as a unique system given the ensemble of software tradeoffs and architecture decisions we have made. Taskflow is open-source at GitHub under MIT license and is being used by many academic and industrial projects [10].

2 Motivations

Taskflow is motivated by our DARPA project to reduce the long design times of modern circuits [1]. The main research objective is to advance *computer-aided design* (CAD) tools with heterogeneous parallelism to achieve transformational performance and productivity milestones. Unlike traditional loop-parallel scientific computing problems, many CAD algorithms exhibit *irregular computational patterns* and *complex control flow* that require strategic task graph decompositions to benefit from heterogeneous parallelism [28]. This type of complex parallel algorithm is difficult to implement and execute efficiently using mainstream TGCS. We highlight three reasons below, *end-to-end tasking*, *GPU task graph parallelism*, and *heterogeneous runtimes*.

End-to-End Tasking – Optimization engines implement various graph and combinatorial algorithms that frequently call for iterations, conditionals, and dynamic control flow. Existing TGCSs [2], [7], [12], [17], [18], [20], [24], [34], [40], closely rely on DAG models to define tasks and their dependencies. Users implement control-flow decisions *outside* the graph description via either statically unrolling the graph across fixed-length iterations or dynamically executing an "if statement" on the fly to decide the next path and so forth. These solutions often incur rather complicated implementations that lack *end-to-end* parallelism using just one task graph entity. For instance, when describing an iterative algorithm using a DAG model, we need to repetitively wait for the task graph to complete at the end of each iteration. This wait operation is not cheap because it involves synchronization between the application code and the TGCS runtime, which could otherwise be totally avoided by supporting in-graph control-flow tasks. More importantly, developers can benefit by making in-graph control-flow decisions to efficiently overlap tasks both inside and outside control flow, completely decided by a dynamic scheduler.

GPU Task Graph Parallelism – Emerging GPU task graph acceleration, such as CUDA Graph [4], can offer dramatic yet largely untapped performance advantages by running a GPU task graph directly on a GPU. This type of GPU task graph parallelism is particularly beneficial for many large-scale analysis and machine learning algorithms that compose thousands of dependent GPU operations to run on the same task graph using iterative methods. By creating an executable image for a GPU task graph, we can iteratively launch it with extremely low kernel overheads. However, existing TGCSs are short of a generic model to express and offload task graph parallelism directly on a GPU, as opposed to a simple encapsulation of GPU operations into CPU tasks.

Heterogeneous Runtimes – Many CAD algorithms compute extremely large circuit graphs. Different quantities are often dependent on each other, via either logical relation or physical net order, and are expensive to compute. The resulting task graph in terms of encapsulated function calls and task dependencies is usually very large. For example, the task graph representing a timing analysis on a million-gate design can add up to *billions* of tasks that take several hours to finish [31]. During the execution, tasks can run on CPUs or GPUs, or more frequently *a mix*. Scheduling these heterogeneously dependent tasks is a big challenge. Existing runtimes are good at either CPU- or GPU-focused work but rarely both simultaneously.

Therefore, we argue that there is a critical need for a new heterogeneous task graph programming environment that supports in-graph control flow. The environment must handle new scheduling challenges, such as conditional dependencies and cyclic executions. To this end, Taskflow aims to (1) introduce a new programming model that enables end-to-end expressions of CPU-GPU dependent tasks along with algorithmic control flow and (2) establish an efficient system runtime to support our model with high performance across latency, energy efficiency, and throughput. Taskflow focuses on a single heterogeneous node of CPUs and GPUs.

3 Preliminary Results

Taskflow is established atop our prior system, *Cpp-Taskflow* [31], [33] which targets CPU-only parallelism using a DAG model, and extends its capability to heterogeneous computing using a new *heterogeneous task dependency graph* (HTDG) programming model beyond DAG. Since we opened the source of Cpp-Taskflow/Taskflow, it has been successfully adopted by much software, including important CAD projects [14], [30], [44], [55] under the DARPA ERI IDEA/POSH program [1]. Because of the success, we are recently invited to publish a 5-page TCAD brief to overview how Taskflow address the parallelization challenges of CAD workloads [32]. For the rest of the paper, we will provide comprehensive details of the Taskflow system from the top-level programming model to the system runtime, including several new technical materials for control-flow primitives, capturer-based GPU task graph parallelism, work-stealing algorithms and theory results, and experiments.

4 Taskflow Programming Model

This section discusses five fundamental task types of Taskflow, *static task*, *dynamic task*, *module task*, *condition task*, and *cudaFlow* task.

4.1 Static Tasking

Static tasking is the most basic task type in Taskflow. A static task takes a callable of no arguments and runs it. The callable can be a generic C++ lambda function object, binding expression, or a functor. Listing 1 demonstrates a simple Taskflow program of four static tasks, where `A` runs before `B` and `C`, and `D` runs after `B` and `C`. The graph is run by an *executor* which schedules dependent tasks across worker threads. Overall, the code explains itself.

```
tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = taskflow.emplace(
  [] () { std::cout << "Task A"; },
  [] () { std::cout << "Task B"; },
  [] () { std::cout << "Task C"; },
  [] () { std::cout << "Task D"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C
executor.run(tf).wait();
```

Listing 1: A task graph of four static tasks.

4.2 Dynamic Tasking

Dynamic tasking refers to the creation of a task graph during the execution of a task. Dynamic tasks are spawned from a parent task and are grouped to form a hierarchy called *subflow*. Figure 1 shows an example of dynamic tasking. The graph has four static tasks, `A`, `C`, `D`, and `B`. The precedence constraints force `A` to run before `B` and `C`, and `D` to run after `B` and `C`. During the execution of task `B`, it spawns another graph of three tasks, `B1`, `B2`, and `B3`, where `B1` and `B2` run before `B3`. In this example, `B1`, `B2`, and `B3` are grouped to a subflow parented at `B`.

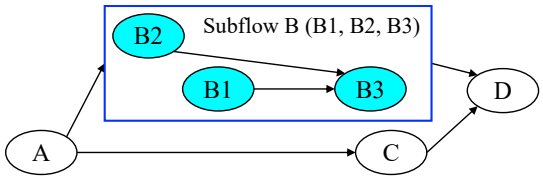


Fig. 1: A task graph that spawns another task graph (`B1`, `B2`, and `B3`) during the execution of task `B`.

```
auto [A, C, D] = taskflow.emplace(
  [] () { std::cout << "A"; },
  [] () { std::cout << "C"; },
  [] () { std::cout << "D"; }
);
auto B = tf.emplace([] (tf::Subflow& subflow) {
  std::cout << "B\n";
  auto [B1, B2, B3] = subflow.emplace(
    [] () { std::cout << "B1"; },
    [] () { std::cout << "B2"; },
    [] () { std::cout << "B3"; }
  );
  B3.succeed(B1, B2);
});
A.precede(B, C);
D.succeed(B, C);
```

Listing 2: Taskflow code of Figure 1.

Listing 2 shows the Taskflow code in Figure 1. A dynamic task accepts a reference of type `tf::Subflow` that is created by the executor during the execution of task `B`. A subflow inherits all graph building blocks of static tasking. By default, a spawned subflow joins its parent task (`B3` precedes its parent `B` implicitly), forcing a subflow to follow the subsequent dependency constraints of its parent task. Depending on applications, users can detach a subflow from its parent task using the method `detach`, allowing its execution to flow independently. A detached subflow will eventually join its parent taskflow.

4.3 Composable Tasking

Composable tasking enables developers to define task hierarchies and compose large task graphs from modular and reusable blocks that are easier to optimize. Figure 2 gives an example of a Taskflow graph using composition. The top-level taskflow defines one static task `C` that runs before a dynamic task `D` that spawns two dependent tasks `D1` and `D2`. Task `D` precedes a *module task* `E` that composes a taskflow of two dependent tasks `A` and `B`.

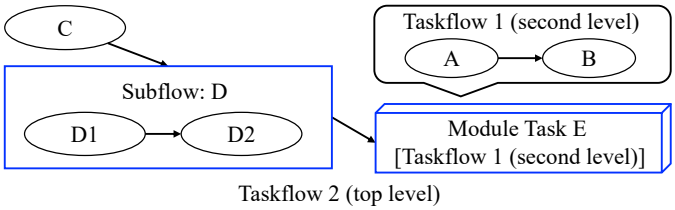


Fig. 2: An example of taskflow composition.

```
// file 1 defines taskflow1
tf::Taskflow taskflow1;
auto [A, B] = taskflow1.emplace(
  [] () { std::cout << "TaskA"; },
  [] () { std::cout << "TaskB"; }
);
A.precede(B);
// file 2 defines taskflow2
tf::Taskflow taskflow2;
auto [C, D] = taskflow2.emplace(
  [] () { std::cout << "TaskC"; },
  [] (tf::Subflow& sf) {
    std::cout << "TaskD";
    auto [D1, D2] = sf.emplace(
      [] () { std::cout << "D1"; },
      [] () { std::cout << "D2"; }
    );
    D1.precede(D2);
  }
);
auto E = taskflow2.composed_of(taskflow1); // module
D.precede(E);
C.precede(D);
```

Listing 3: Taskflow code of Figure 2.

Listing 3 shows the Taskflow code of Figure 2. It declares two taskflows, `taskflow1` and `taskflow2`. `taskflow2` forms a module task `E` by calling the method `composed_of` from `taskflow1`, which is then preceded by task `D`. Unlike a subflow task, a module task does not own the taskflow but maintains a soft mapping to its composed taskflow. Users can create multiple module tasks from the same taskflow but they must not run concurrently; on the contrary, subflows are created dynamically and can run concurrently. In practice, we use composable tasking to partition large parallel programs into smaller or reusable taskflows in separate files (e.g., `taskflow1` in file 1 and `taskflow2` in file 2) to improve program modularity and testability. Subflows are instead used for enclosing a task graph that needs stateful data referencing via lambda capture.

4.4 Conditional Tasking

We introduce a new *conditional tasking* model to overcome the limitation of existing frameworks in expressing *general control flow* beyond DAG. A condition task is a callable that

returns an integer index indicating the next successor task to execute. The index is defined with respect to the order of the successors preceded by the condition task. Figure 3 shows an example of if-else control flow, and Listing 4 gives its implementation. The code is self-explanatory. The condition task, `cond`, precedes two tasks, `yes` and `no`. With this order, if `cond` returns `0`, the execution moves on to `yes`, or `no` if `cond` returns `1`.

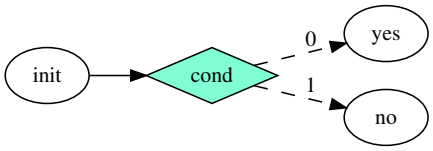


Fig. 3: A Taskflow graph of if-else control flow using one condition task (in diamond).

```
auto [init, cond, yes, no] = taskflow.emplace(
  [] () { std::cout << "init"; },
  [] () { std::cout << "cond"; return 0; },
  [] () { std::cout << "cond returns 0"; },
  [] () { std::cout << "cond returns 1"; }
);
cond.succeed(init)
    .precede(yes, no);
```

Listing 4: Taskflow program of Figure 3.

Our condition task supports iterative control flow by introducing a *cycle* in the graph. Figure 4 shows a task graph of *do-while* iterative control flow, implemented in Listing 5. The loop continuation condition is implemented by a single condition task, `cond`, that precedes two tasks, `body` and `done`. When `cond` returns `0`, the execution loops back to `body`. When `cond` returns `1`, the execution moves onto `done` and stops. In this example, we use only four tasks even though the control flow spans 100 iterations. Our model is more efficient and expressive than existing frameworks that count on dynamic tasking or recursive parallelism to execute condition on the fly [18], [20].

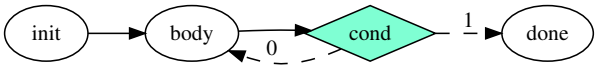


Fig. 4: A Taskflow graph of iterative control flow using one condition task.

```
int i;
auto [init, body, cond, done] = taskflow.emplace(
  [&](){ i=0; },
  [&](){ i++; },
  [&](){ return i<100 ? 0 : 1; },
  [&](){ std::cout << "done"; }
);
init.precede(body);
body.precede(cond);
cond.precede(body, done);
```

Listing 5: Taskflow program of Figure 4.

Furthermore, our condition task can model non-deterministic control flow where many existing models do not support. Figure 5 shows an example of nested non-deterministic control flow frequently used in stochastic optimization (e.g., VLSI floorplan annealing [54]). The graph consists of two regular tasks, `init` and `stop`, and three condition tasks, `F1`, `F2`, and `F3`. Each condition task forms a dynamic control flow to randomly go to either the next task or loop back to `F1` with a probability of 1/2. Starting from `init`, the expected number of condition tasks to execute before reaching `stop` is eight. Listing 6 implements Figure 5 in just 11 lines of code.

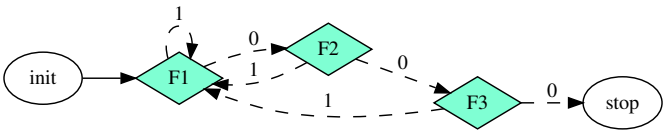


Fig. 5: A Taskflow graph of non-deterministic control flow using three condition tasks.

```
auto [init, F1, F2, F3, stop] = taskflow.emplace(
  [] () { std::cout << "init"; },
  [] () { return rand()%2 },
  [] () { return rand()%2 },
  [] () { return rand()%2 },
  [] () { std::cout << "stop"; }
);
init.precede(F1);
F1.precede(F2, F1);
F2.precede(F3, F1);
F3.precede(stop, F1);
```

Listing 6: Taskflow program of Figure 5.

The advantage of our conditional tasking is threefold. First, it is simple and expressive. Developers benefit from the ability to make *in-graph* control-flow decisions that are integrated within task dependencies. This type of decision making is different from dataflow [34] as we do not abstract data but tasks, and is more general than the primitive-based method [57] that is limited to domain applications. Second, condition tasks can be associated with other tasks to integrate control flow into a unified graph entity. Users ought not to partition the control flow or unroll it to a flat DAG, but focus on expressing dependent tasks and control flow. The later section will explain our scheduling algorithms for condition tasks. Third, our model enables developers to efficiently overlap tasks both inside and outside control flow. For example, Figure 6 implements a task graph of three control-flow blocks, and `cond_1` can run in parallel with `cond_2` and `cond_3`. This example requires only 30 lines of code.

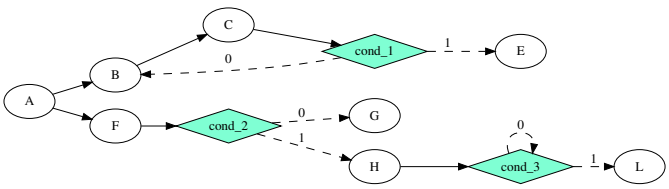


Fig. 6: A Taskflow graph of parallel control-flow blocks using three condition tasks.

4.5 Heterogeneous Tasking

We introduce a new heterogeneous task graph programming model by leveraging C++ closure and emerging GPU

task graph acceleration, *CUDA Graph* [4]. Figure 7 and Listing 7 show the canonical CPU-GPU saxpy (A•X plus Y) workload and its implementation using our model. Our model lets users describe a GPU workload in a *task graph* called *cudaFlow* rather than aggregated GPU operations using explicit CUDA streams and events. A cudaFlow lives inside a closure and defines methods for constructing a GPU task graph. In this example, we define two parallel CPU tasks (`allocate_x`, `allocate_y`) to allocate unified shared memory (`cudaMallocManaged`) and one `cudaFlow` task to spawn a GPU task graph consisting of two host-to-device (H2D) transfer tasks (`h2d_x`, `h2d_y`), one saxpy kernel task (`kernel`), and two device-to-host (D2H) transfer tasks (`d2h_x`, `d2h_y`), in this order of task dependencies. Task dependencies are established through `precede` or `succeed`. Apparently, `cudaFlow` must run after `allocate_x` and `allocate_y`. We emplace this cudaFlow on GPU 1 (`emplace_on`). When defining cudaFlows on specific GPUs, users are responsible for ensuring all involved memory operations stay in valid GPU contexts.

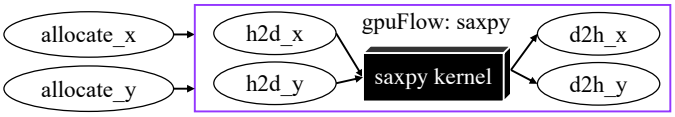


Fig. 7: A saxpy ("single-precision A·X plus Y") task graph using two CPU tasks and one cudaFlow task.

```
__global__ void saxpy(int n,int a,float *x,float *y);

const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};

auto [allocate_x, allocate_y] = taskflow.emplace(
  [&](){ cudaMallocManaged(&dx, N*sizeof(float));}
  [&](){ cudaMallocManaged(&dy, N*sizeof(float));}
);
auto cudaFlow = taskflow.emplace_on(
  [&](tf::cudaFlow& cf) {
    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.kernel(
      GRID, BLOCK, SHM, saxpy, N, 2.0f, dx, dy
    );
    kernel.succeed(h2d_x, h2d_y)
          .precede(d2h_x, d2h_y);
  }, 1
);
cudaFlow.succeed(allocate_x, allocate_y);
```

Listing 7: Taskflow program of Figure 7.

Our cudaFlow has the three key motivations. First, users focus on the graph-level expression of dependent GPU operations without wrangling with low-level streams. They can easily visualize the graph by Taskflow to reduce turnaround time. Second, our closure *forces* users to express their intention on what data storage mechanism should be used for each captured variable. For example, Listing 7 captures all data (e.g., `hx`, `dx`) in *reference* to form a *stateful closure*. When `allocate_x` and `allocate_y` finish, the cudaFlow closure can access the correct state of `dx` and `dy`. This property is very important for heterogeneous graph parallelism because CPU and GPU tasks need to share states of data to collaborate with each other. Our model makes it easy and efficient to capture data regardless of its scope. Third, by abstracting GPU operations to a task graph closure, we judiciously hide implementation details for portable optimization. By default, a cudaFlow maps to a CUDA graph that can be executed using a single CPU call. On a platform that does not support CUDA Graph, we fall back to a stream-based execution.

Taskflow does not dynamically choose whether to execute tasks on CPU or GPU, and does not manage GPU data with another abstraction. This is a software decision we have made when designing cudaFlow based on our experience in parallelizing CAD using existing TGCSs. While it is always interesting to see what abstraction is best suited for which application, in our field, developing high-performance CAD algorithms requires many custom efforts on optimizing the memory and data layouts [28], [26]. Developers tend to do this statically in their own hands, such as direct control over raw pointers and explicit memory placement on a GPU, while leaving tedious details of runtime load balancing to a dynamic scheduler. After years of research, we have concluded to not abstract memory or data because they are application-dependent. This decision allows Taskflow to be *framework-neutral* while enabling application code to take full advantage of native or low-level GPU programming toolkits.

```
taskflow.emplace_on([&](tf::cudaFlowCapturer& cfc) {
  auto h2d_x = cfc.copy(dx, hx.data(), N);
  auto h2d_y = cfc.copy(dy, hy.data(), N);
  auto d2h_x = cfc.copy(hx.data(), dx, N);
  auto d2h_y = cfc.copy(hy.data(), dy, N);
  auto kernel = cfc.on([&](cudaStream_t s){
    invoke_3rdparty_saxpy_kernel(s);
  });
  kernel.succeed(h2d_x, h2d_y)
        .precede(d2h_x, d2h_y);
}, 1);
```

Listing 8: Taskflow program of Figure 7 using a capturer.

Constructing a GPU task graph using cudaFlow requires all kernel parameters are known in advance. However, third-party applications, such as cuDNN and cuBLAS, do not open these details but provide an API for users to invoke hidden kernels through custom streams. The burden is on users to decide a stream layout and witness its concurrency across dependent GPU tasks. To deal with this problem, we design a cudaFlow *capturer* to capture GPU tasks from existing stream-based APIs. Listing 8 outlines an implementation of the same saxpy task graph in Figure 7 using a cudaFlow capturer, assuming the saxpy kernel is only invocable through a stream-based API.

Both cudaFlow and cudaFlow capturer can work seamlessly with condition tasks. Control-flow decisions frequently happen at the boundary between CPU and GPU tasks. For example, a heterogeneous k-means algorithm iteratively uses GPU to accelerate the finding of k centroids and then uses CPU to check if the newly found centroids converge to application rules. Taskflow enables an end-to-end expression of such a workload in a single graph entity, as shown in Figure 8 and Listing 9. This capability largely improves the efficiency of modeling complex CPU-GPU

workloads, and our scheduler can dynamically overlap CPU and GPU tasks across different control-flow blocks.

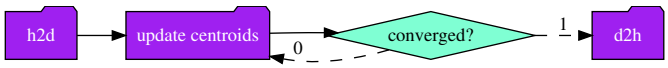


Fig. 8: A cyclic task graph using three cudaFlow tasks and one condition task to model an iterative k-means algorithm.

```
auto [h2d, update, cond, d2h] = taskflow.emplace(
  [&](tf::cudaFlow& cf){ /* copy input to GPU */ },
  [&](tf::cudaFlow& cf){ /* update kernel */ },
  [&](){ return converged() ? 1 : 0; },
  [&](tf::cudaFlow& cf){ /* copy result to CPU */ }
);
h2d.precede(update);
update.precede(cond);
cond.precede(update, d2h);
```

Listing 9: Taskflow program of Figure 8.

5 Taskflow System Runtime

Taskflow enables users to express CPU-GPU dependent tasks that integrate control flow into an HTDG. To support our model with high performance, we design the system runtime at two scheduling levels, *task level* and *worker level*. The goal of task-level scheduling is to (1) devise a feasible, efficient execution for in-graph control flow and (2) transform each GPU task into a runnable instance on a GPU. The goal of worker-level scheduling is to optimize the execution performance by dynamically balancing the worker count with task parallelism.

5.1 Task-level Scheduling Algorithm

5.1.1 Scheduling Condition Tasks

Conditional tasking is powerful but challenging to schedule. Specifically, we must deal with conditional dependency and cyclic execution without encountering *task race*, i.e., only one thread can touch a task at a time. More importantly, we need to let users easily understand our task scheduling flow such that they can infer if a written task graph is properly conditioned and schedulable. To accommodate these challenges, we separate the execution logic between condition tasks and other tasks using two dependency notations, *weak dependency* (out of condition tasks) and *strong dependency* (other else). For example, the six dashed arrows in Figure 5 are weak dependencies and the solid arrow `init`→`F1` is a strong dependency. Based on these notations, we design a simple and efficient algorithm for scheduling tasks, as depicted in Figure 9. When the scheduler receives an HTDG, it (1) starts with tasks of *zero* dependencies (both strong and weak) and continues executing tasks whenever *strong* remaining dependencies are met, or (2) skips this rule for weak dependency and directly jumps to the task indexed by the return of that condition task.

Taking Figure 5 for example, the scheduler starts with `init` (zero weak and strong dependencies) and proceeds to `F1`. Assuming `F1` returns `0`, the scheduler proceeds to its first successor, `F2`. Now, assuming `F2` returns `1`, the scheduler proceeds to its second successor, `F1`, which forms

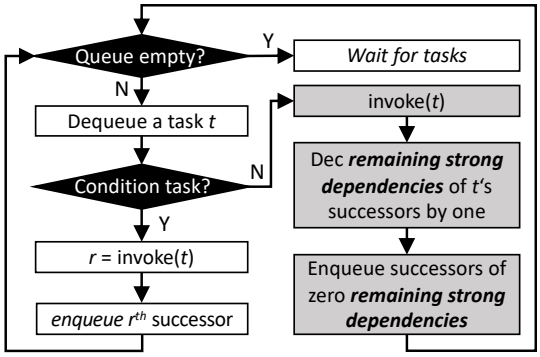


Fig. 9: Flowchart of our task scheduling.

a cyclic execution and so forth. With this concept, the scheduler will cease at `stop` when `F1`, `F2`, and `F3` all return `0`. Based on this scheduling algorithm, users can quickly infer whether their task graph defines correct control flow. For instance, adding a strong dependency from `init` to `F2` may cause task race on `F2`, due to two execution paths, `init`→`F2` and `init`→`F1`→`F2`.

Figure 10 shows two common pitfalls of conditional tasking, based on our task-level scheduling logic. The first example has no source for the scheduler to start with. A simple fix is to add a task `S` of zero dependencies. The second example may race on `D`, if `C` returns 0 at the same time `E` finishes. A fix is to partition the control flow at `C` and `D` with an auxiliary node `X` such that `D` is strongly conditioned by `E` and `X`.

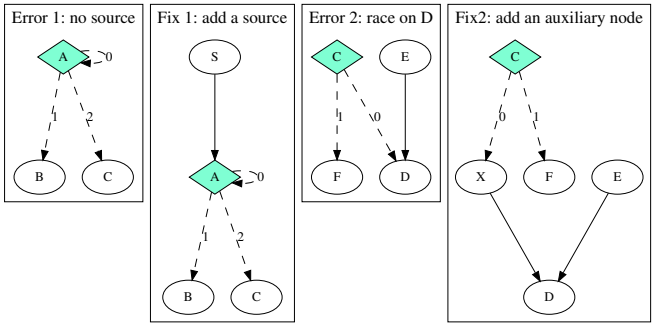


Fig. 10: Common pitfalls of conditional tasking.

5.1.2 Scheduling GPU Tasks

We leverage modern *CUDA Graph* [4] to schedule GPU tasks. CUDA graph is a new asynchronous task graph programming model introduced in CUDA 10 to enable more efficient launch and execution of GPU work than streams. There are two types of GPU tasks, cudaFlow and cudaFlow capturer. For each scheduled cudaFlow task, since we know all the operation parameters, we construct a CUDA graph that maps each task in the cudaFlow, such as copy and kernel, and each dependency to a node and an edge in the CUDA graph. Then, we submit it to the CUDA runtime for execution. This organization is simple and efficient, especially under modern GPU architectures (e.g., Nvidia Ampere) that support hardware-level acceleration for graph parallelism.

On the other hand, for each scheduled cudaFlow capturer task, our runtime transforms the captured GPU tasks and dependencies into a CUDA graph using *stream capture* [4]. The objective is to decide a stream layout optimized for kernel concurrency without breaking task dependencies. We design a greedy round-robin algorithm to transform a cudaFlow capturer to a CUDA graph, as shown in Algorithm 1. Our algorithm starts by *levelizing* the capturer graph into a two-level array of tasks in their topological orders. Tasks at the same level can run simultaneously. However, assigning each independent task here a unique stream does not produce decent performance, because GPU has a limit on the maximum kernel concurrency (e.g., 32 for RTX 2080). We give this constraint to users as a tunable parameter, $max\_streams$. We assign each levelized task an id equal to its index in the array at its level. Then, we can quickly assign each task a stream using the round-robin arithmetic (line 6). Since tasks at different levels have dependencies, we need to record an event (lines 13:17) and wait on the event (lines 7:11) from both sides of a dependency, saved for those issued in the same stream (line 8 and line 14).

Algorithm 1: make_graph(G)

Input: a cudaFlow capturer C
Output: a transformed CUDA graph G

```
1  S ← get_capture_mode_streams(max_streams);
2  L ← levelize(C);
3  l ← L.min_level;
4  while l <= L.max_level do
5      foreach t ∈ L.get_tasks(l) do
6          s ← (t.id mod max_streams);
7          foreach p ∈ t.predecessors do
8              if s ≠ (p.id mod max_streams) then
9                  stream_wait_event(S[s], p.event);
10             end
11         end
12         stream_capture(t, S[s]);
13         foreach n ∈ t.successors do
14             if s ≠ (n.id mod max_streams) then
15                 stream_record_event(S[s], p.event);
16             end
17         end
18     end
19 end
20 G ← end_capture_mode_streams(S);
21 return G;
```

Figure 11 gives an example of transforming a user-given cudaFlow capturer graph into a native CUDA graph using two streams (i.e., $max\_stream = 2$) for execution. The algorithm first levelizes the graph by performing a topological traversal and assign each node an id equal to its index at the level. For example, `A` and `B` are assigned 0 and 1, `C`, `D`, and `E` are assigned 0, 1, and 2, and so on. These ids are used to quickly determine the mapping between a stream and a node in our round-robin loop, because CUDA stream only allows inserting events from the latest node in the queue. For instance, when `A` and `B` are assigned to stream 0 (upper row) and stream 1 (lower row) during the level-by-level traversal (line 4 of Algorithm 1), we can determine ahead of the stream numbers of their successors and find out the two cross-stream dependencies, `A`→`D` and `B`→`E`, that need recording events. Similarly, we can wait on recorded events by scanning the predecessors of each node to find out cross-stream event dependencies.

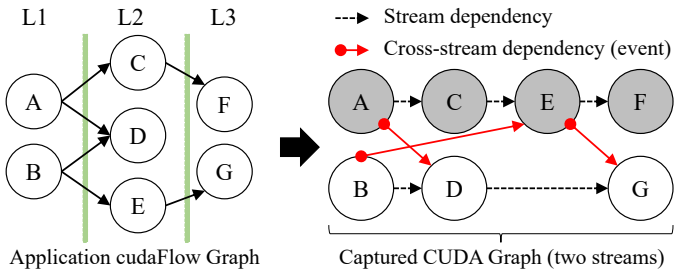


Fig. 11: Illustration of Algorithm 1 on transforming an application cudaFlow capturer graph into a native CUDA graph using two streams.

5.2 Worker-level Scheduling Algorithm

At the worker level, we leverage *work stealing* to execute submitted tasks with dynamic load balancing. Work stealing has been extensively studied in multicore programming [2], [12], [13], [16], [23], [41], [40], [48], [53], but an efficient counterpart for hybrid CPU-GPU or more general heterogeneous systems remains demanding. This is a challenging research topic, especially under Taskflow's HTDG model. When executing an HTDG, a CPU task can submit both CPU and GPU tasks and vice versa whenever dependencies are met. The available task parallelism changes dynamically, and there are no ways to predict the next coming tasks under dynamic control flow. To achieve good system performance, the scheduler must balance the number of worker threads with dynamically generated tasks to control the number of *wasteful steals* because the wasted resources should have been used by useful workers or other concurrent programs [13], [23].

Keeping workers busy in awaiting tasks with a *yielding* mechanism is a commonly used work-stealing framework [16], [17], [25]. However, this approach is not cost-efficient, because it can easily over-subscribe resources when tasks become scarce, especially around the decision-making points of control flow. The sleep-based mechanism is another way to suspend the workers frequently failing in steal attempts. A worker is put into sleep by waiting for a condition variable to become true. When the worker sleeps, OS can grant resources to other workers for running useful jobs. Also, reducing wasteful steals can improve both the inter-operability of a concurrent program and the overall system performance, including latency, throughput, and energy efficiency to a large extent [23]. Nevertheless, deciding *when and how to put workers to sleep, wake up workers to run, and balance the numbers of workers with dynamic task parallelism* is notoriously challenging to design correctly and implement efficiently.

Our previous work [43] has introduced an adaptive work-stealing algorithm to address a similar line of the challenge yet in a CPU-only environment by maintaining

a loop invariant between active and idle workers. However, extending this algorithm to a heterogeneous target is not easy, because we need to consider the adaptiveness in different heterogeneous domains and bound the total number of wasteful steals across all domains at any time of the execution. To overcome this challenge, we introduce a new scheduler architecture and an adaptive worker management algorithm that are both generalizable to arbitrary heterogeneous domains. We shall prove the proposed work-stealing algorithm can deliver a strong upper bound on the number of wasteful steals at any time during the execution.

5.2.1 Heterogeneous Work-stealing Architecture

At the architecture level, our scheduler maintains a set of workers for each task domain (e.g., CPU, GPU). A worker can only steal tasks of the same domain from others. Figure 12 shows the architecture of our work-stealing scheduler on two domains, CPU and GPU. By default, the number of domain workers equals the number of domain devices (e.g., CPU cores, GPUs). We associate each worker with two separate task queues, a CPU task queue (CTQ) and a GPU task queue (GTQ), and declare a pair of CTQ and GTQ shared by all workers. The shared CTQ and GTQ pertain to the scheduler and are primarily used for external threads to submit HTDGs. A CPU worker can push and pop a new task into and from its local CTQ, and can steal tasks from all the other CTQs; the structure is symmetric to GPU workers. This separation allows a worker to quickly insert dynamically generated tasks to their corresponding queues without contending with other workers.

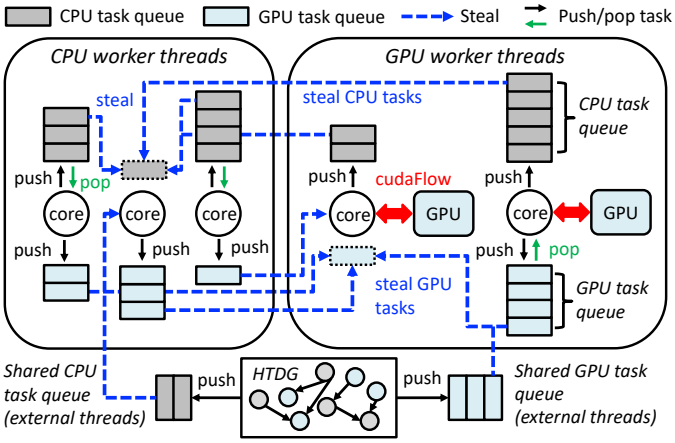


Fig. 12: Architecture of our work-stealing scheduler on two domains, CPU and GPU.

We leverage two existing concurrent data structures, *work-stealing queue* and *event notifier*, to support our scheduling architecture. We implemented the task queue based on the lock-free algorithm proposed by [37]. Only the queue owner can pop/push a task from/into one end of the queue, while multiple threads can steal a task from the other end at the same time. Event notifier is a two-phase commit protocol (2PC) that allows a worker to wait on a binary predicate in a *non-blocking* fashion [11]. The idea is similar to the 2PC in distributed systems and computer networking. The waiting worker first checks the predicate and calls `prepare_wait` if it evaluates to false. The waiting worker then checks the predicate again and calls `commit_wait` to wait, if the outcome remains false, or `cancel_wait` to cancel the request. Reversely, the notifying worker changes the predicate to true and call `notify_one` or `notify_all` to wake up one or all waiting workers. Event notifier is particularly useful for our scheduler architecture because we can keep notification between workers non-blocking. We develop one event notifier for each domain, based on Dekker's algorithm by [11].

5.2.2 Heterogeneous Work-stealing Algorithm

Atop this architecture, we devise an efficient algorithm to *adapt* the number of active workers to dynamically generated tasks such that threads are not underutilized when tasks are abundant nor overly subscribed when tasks are scarce. Our adaptiveness is different from existing frameworks, such as constant wake-ups [2], [23], data locality [21], [50], and watchdogs [23]. Instead, we extend our previous work [43] to keep a *per-domain invariant* to control the numbers of thieves and, consequently, wasteful steals based on the active worker count: *When an active worker exists, we keep at least one worker making steal attempts unless all workers are active.*

Unlike the CPU-only scheduling environment in [43], the challenge to keep this invariant in a heterogeneous target comes from the heterogeneously dependent tasks and cross-domain worker notifications, as a CPU task can spawn a GPU task and vice versa. Our scheduler architecture is particularly designed to tackle this challenge by separating decision controls to a per-domain basis. This design allows us to realize the invariant via an adaptive strategy–*the last thief to become active will wake up a worker in the same domain to take over its thief role, and so forth*. External threads (non-workers) submit tasks through the shared task queues and wake up workers to run tasks.

Algorithm 2: worker_loop(w)

```
Input: w: a worker
Per-worker global: t: a task (initialized to NIL)
1 while true do
2 |   exploit_task(w, t);
3 |   if wait_for_task(w, t) == false then
4 |   |   break;
5 |   end
6 end
```

Our scheduling algorithm is symmetric by domain. Upon spawned, each worker enters the loop in Algorithm 2. Each worker has a per-worker global pointer t to a task that is either stolen from others or popped out from the worker's local task queue after initialization; the notation will be used in the rest of algorithms. The loop iterates two functions, `exploit_task` and `wait_for_task`. Algorithm 3 implements the function `exploit_task`. We use two scheduler-level arrays of atomic variables, `actives` and `thieves`, to record for each domain the number of workers that are actively running tasks and the number of workers that are

Algorithm 3: exploit_task(w, t)

```
   Input: w: a worker (domain d_w)
   Per-worker global: t: a task
1  if t ≠ NIL then
2      if AtomInc(actives[d_w]) == 1 and thieves[d_w] == 0
         then
3          notifier[d_w].notify_one();
4      end
5      do
6          execute_task(w, t);
7          t ← w.task_queue[d_w].pop();
8      while t ≠ NIL;
9      AtomDec(actives[d_w]);
10 end
```

making steal attempts, respectively.[1] Our algorithm relies on these atomic variables to decide when to put a worker to sleep for reducing resource waste and when to bring back a worker for running new tasks. Lines 2:4 implement our adaptive strategy using two lightweight atomic operations. In our pseudocodes, the two atomic operations, *AtomInc* and *AtomDec*, return the results after incrementing and decrementing the values by one, respectively. Notice that the order of these two comparisons matters (i.e., active workers and then thieves), as they are used to synchronize with other workers in the later algorithms. Lines 5:8 drain out the local task queue and executes all the tasks using `execute_task` in Algorithm 4. Before leaving the function, the worker decrements `actives` by one (line 9).

Algorithm 4: execute_task(w, t)

```
   Input: w: a worker
   Per-worker global: t: a task
1  r ← invoke_task_callable(t);
2  if r.has_value() then
3      submit_task(w, t.successors[r]);
4      return;
5  end
6  foreach s ∈ t.successors do
7      if AtomDec(s.strong_dependents) == 0 then
8          submit_task(w, s);
9      end
10 end
```

Algorithm 4 implements the function `execute_task`. We invoke the callable of the task (line 1). If the task returns a value (i.e., a condition task), we directly submit the task of the indexed successor (lines 2:5). Otherwise, we remove the task dependency from all immediate successors and submit new tasks of zero remaining strong dependencies (lines 6:10). The detail of submitting a task is shown in Algorithm 5. The worker inserts the task into the queue of the corresponding domain (line 1). If the task does not belong to the worker's domain (line 2), the worker wakes up one worker from that domain if there are no active workers or thieves (lines 3:5). The function `submit_task` is internal to the workers of a scheduler. External threads never touch this call.

1. While our pseudocodes use array notations of atomic variables for the sake of brevity, the actual implementation considers padding to avoid false-sharing effects.

Algorithm 5: submit_task(w, t)

```
   Input: w: a worker (domain d_w)
   Per-worker global: t: a task (domain d_t)
1  w.task_queue[d_t].push(t);
2  if d_w! = d_t then
3      if actives[d_t] == 0 and thieves[d_t] == 0 then
4          notifier[d_t].notify_one();
5      end
6  end
```

When a worker completes all tasks in its local queue, it proceeds to `wait_for_task` (line 3 in Algorithm 2), as shown in Algorithm 6. At first, the worker enters `explore_task` to make steal attempts (line 2). When the worker steals a task and it is the last thief, it notifies a worker of the same domain to take over its thief role and returns to an active worker (lines 3:8). Otherwise, the worker becomes a *sleep candidate*. However, we must avoid underutilized parallelism, since new tasks may come at the time we put a worker to sleep. We use 2PC to adapt the number of active workers to available task parallelism (lines 9:41). The predicate of our 2PC is *at least one task queue, both local and shared, in the worker's domain is nonempty*. At line 8, the worker has drained out its local queue and devoted much effort to stealing tasks. Other task queues in the same domain are most likely to be empty. We put this worker to a sleep candidate by submitting a wait request (line 9). From now on, *all the notifications from other workers will be visible to at least one worker, including this worker*. That is, if another worker call `notify` at this moment, the 2PC guarantees one worker within the scope of lines 9:41 will be notified (i.e., line 42). Then, we inspect our predicate by examining the shared task queue again (lines 10:20), since external threads might have inserted tasks at the same time we call `prepare_wait`. If the shared queue is nonempty (line 10), the worker cancels the wait request and makes an immediate steal attempt at the queue (lines 11:12); if the steal succeeds and it is the last thief, the worker goes active and notifies a worker (lines 13:18), or otherwise enters the steal loop again (line 19). If the shared queue is empty (line 20), the worker checks whether the scheduler received a stop signal from the executor due to exception or task cancellation, and notifies all workers to leave (lines 21:28). Now, the worker is almost ready to sleep except if it is the last thief and: (1) an active worker in its domain exists (lines 30:33) or (2) at least one task queue of the same domain from other workers is nonempty (lines 34:39). The two conditions may happen because a task can spawn tasks of different domains and trigger the scheduler to notify the corresponding domain workers. Our 2PC guarantees the two conditions synchronize with lines 2:4 in Algorithm 3 and lines 3:5 in Algorithm 5, and vice versa, preventing the problem of undetected task parallelism. Passing all the above conditions, the worker commits to wait on our predicate (line 41).

Algorithm 7 implements `explore_task`, which resem-

Algorithm 6: wait_for_task(w, t)

Input: w: a worker (domain d_w)
Per-worker global: t: a task
Output: a boolean signal of stop

```
1  AtomInc(thieves[d_w]);
2  explore_task(w, t);
3  if t ≠ NIL then
4  |  if AtomDec(thieves[d_w]) == 0 then
5  |  |  notifier[d_w].notify_one();
6  |  end
7  |  return true;
8  end
9  notifier[d_w].prepare_wait(w);
10 if task_queue[d_w].empty() ≠ true then
11 |  notifier[d_w].cancel_wait(w);
12 |  t ← task_queue[d_w].steal();
13 |  if t ≠ NIL then
14 |  |  if AtomDec(thieves[d_w]) == 0 then
15 |  |  |  notifier[d_w].notify_one();
16 |  |  end
17 |  |  return true;
18 |  end
19 |  goto Line 2;
20 end
21 if stop == true then
22 |  notifier[d_w].cancel_wait(w);
23 |  foreach domain d ∈ D do
24 |  |  notifier[d].notify_all();
25 |  end
26 |  AtomDec(thieves[d_w]);
27 |  return false;
28 end
29 if AtomDec(thieves[d_w]) == 0 then
30 |  if actives[d_w] > 0 then
31 |  |  notifier[d_w].cancel_wait(w);
32 |  |  goto Line 1;
33 |  end
34 |  foreach worker x ∈ W do
35 |  |  if x.task_queue[d_w].empty() ≠ true then
36 |  |  |  notifier[d_w].cancel_wait(w);
37 |  |  |  goto Line 1;
38 |  |  end
39 |  end
40 end
41 notifier[d_w].commit_wait(w);
42 return true;
```

Algorithm 7: explore_task(w, t)

Input: w: a worker (a thief in domain d_w)
Per-worker global: t: a task (initialized to **NIL**)

```
1 steals ← 0;
2 while t != NIL and ++steals ≤ MAX_STEAL do
3 |  yield();
4 |  t ← steal_task_from_random_victim(d_w);
5 end
```

Algorithm 8: submit_graph(g)

Input: g: an HTDG to execute

```
1 foreach t ∈ g.source_tasks do
2 |  scoped_lock lock(queue_mutex);
3 |  d_t ← t.domain;
4 |  task_queue[d_t].push(t);
5 |  notifier[d_t].notify_one();
6 end
```

bles the normal work-stealing loop [16]. At each iteration, the worker (thief) tries to steal a task from a randomly selected victim, including the shared task queue, in the same domain. We use a parameter MAX_STEALS to control the number of iterations. In our experiments, setting MAX_STEAL to ten times the number of all workers is sufficient enough for most applications. Up to this time, we have discussed the core work-stealing algorithm. To submit an HTDG for execution, we call `submit_graph`, shown in Algorithm 8. The caller thread inserts all tasks of zero dependencies (both strong and weak dependencies) to the shared task queues and notifies a worker of the corresponding domain (lines 4:5). Shared task queues may be accessed by multiple callers and are thus protected under a lock pertaining to the scheduler. Our 2PC guarantees lines 4:5 synchronizes with lines 10:20 of Algorithm 6 and vice versa, preventing undetected parallelism in which all workers are sleeping.

6 Analysis

To justify the efficiency of our scheduling algorithm, we draw the following theorems and give their proof sketches.

Lemma 1. *For each domain, when an active worker (i.e., running a task) exists, at least one another worker is making steal attempts unless all workers are active.*

Proof. We prove Lemma 1 by contradiction. Assuming there are no workers making steal attempts when an active worker exists, this means an active worker (line 2 in Algorithm 3) fails to notify one worker if no thieves exist. There are only two scenarios for this to happen: (1) all workers are active; (2) a non-active worker misses the notification before entering the 2PC guard (line 9 in Algorithm 6). The first scenario is not possible as it has been excluded by the lemma. If the second scenario is true, the non-active worker must not be the last thief (contradiction) or it will notify another worker through line 3 in Algorithm 6. The proof holds for other domains as our scheduler design is symmetric. □

Theorem 1. *Our work-stealing algorithm can correctly complete the execution of an HTDG.*

Proof. There are two places where a new task is submitted, line 4 in Algorithm 8 and line 1 in Algorithm 5. In the first place, where a task is pushed to the shared task queue by an external thread, the notification (line 5 in Algorithm 8) is visible to a worker in the same domain of the task for two situations: (1) if a worker has prepared or committed to wait (lines 9:41 in Algorithm 6), it will be notified; (2) otherwise, at least one worker will eventually go through lines 9:20 in Algorithm 6 to steal the task. In the second

place, where the task is pushed to the corresponding local task queue of that worker, at least one worker will execute it in either situation: (1) if the task is in the same domain of the worker, the work itself may execute the task in the subsequent `exploit_task`, or a thief steals the task through `explore_task`; (2) if the worker has a different domain from the task (line 2 in Algorithm 5), the correctness can be proved by contradiction. Assuming this task is undetected, which means either the worker did not notify a corresponding domain worker to run the task (false at the condition of line 3 in Algorithm 5) or notified one worker (line 4 in Algorithm 5) but none have come back. In the former case, we know at least one worker is active or stealing, which will eventually go through line 29:40 of Algorithm 6 to steal this task. Similarly, the latter case is not possible under our 2PC, as it contradicts the guarding scan in lines 9:41 of Algorithm 6. □

Theorem 2. *Our work-stealing algorithm does not under-subscribe thread resources during the execution of an HTDG.*

Proof. Theorem 2 is a byproduct of Lemma 1 and Theorem 1. Theorem 1 proves that our scheduler never has task leak (i.e., undetected task parallelism). During the execution of an HTDG, whenever the number of tasks is larger than the present number of workers, Lemma 1 guarantees one worker is making steal attempts, unless all workers are active. The 2PC guard (lines 34:39 in Algorithm 6) ensures that worker will successfully steal a task and become an active worker (unless no more tasks), which in turn wakes up another worker if that worker is the last thief. As a consequence, the number of workers will catch up on the number of tasks one after one to avoid under-subscribed thread resources. □

Theorem 3. *At any moment during the execution of an HTDG, the number of wasteful steals is bounded by* $\mathcal{O}(MAX\_STEALS \times (|W| + |D| \times (E/e_s)))$*, where* W *is the worker set,* D *is the domain set,* E *is the maximum execution time of any task, and* e_s *is the execution time of Algorithm 7.*

Proof. We give a direct proof for Theorem 3 using the following notations: D denotes the domain set, d denotes a domain (e.g., CPU, GPU), W denotes the entire worker set, W_d denotes the worker set in domain d, w_d denotes a worker in domain d (i.e., $w_d \in W_d$), e_s denotes the time to complete one round of steal attempts (i.e., Algorithm 7), e_d denotes the maximum execution time of any task in domain d, and E denotes the maximum execution time of any task in the given HTDG.

At any time point, the worst case happens at the following scenario: *for each domain* d *only one worker* w_d *is actively running one task while all the other workers are making unsuccessful steal attempts*. Due to Lemma 1 and lines 29:40 in Algorithm 6, only one thief w'_d will eventually remain in the loop, and the other $|W_d| - 2$ thieves will go sleep after one round of unsuccessful steal attempts (line 2 in Algorithm 6) which ends up with $MAX\_STEALS \times (|W_d| - 2)$ wasteful steals. For the only one thief w'_d, it keeps failing in steal attempts until the task running by the only active worker w_d finishes, and then both go sleep. This results in another $MAX\_STEALS \times (e_d/e_s) + MAX\_STEALS$ wasteful steals; the second terms comes from the active worker because it needs another round of steal attempts (line 2 in Algorithm 6) before going to sleep. Consequently, the number of wasteful steals across all domains is bounded as follows:

$$\begin{aligned}
&\sum_{d \in D} MAX_STEALS \times (|W_d| - 2 + (e_d/e_s) + 1) \\
&\leq MAX_STEALS \times \sum_{d \in D} (|W_d| + e_d/e_s) \\
&\leq MAX_STEALS \times \sum_{d \in D} (|W_d| + E/e_s) \\
&= \mathcal{O}(MAX_STEALS \times (|W| + |D| \times (E/e_s)))
\end{aligned} \tag{1}$$

We do not derive the bound over the execution of an HTDG but the worst-case number of wasteful steals at any time point, because the presence of control flow can lead to non-deterministic execution time that requires a further assumption of task distribution. □

7 Experimental Results

We evaluate the performance of Taskflow on two fronts: micro-benchmarks and two realistic workloads, VLSI incremental timing analysis and machine learning. We use micro-benchmarks to analyze the tasking performance of Taskflow without much bias of application algorithms. We will show that the performance benefits of Taskflow observed in micro-benchmarks become significant in real workloads. We will study the performance across runtime, energy efficiency, and throughput. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon CPU cores at 2.00 GHz, 4 GeForce RTX 2080 GPUs, and 256 GB RAM. We compiled all programs using Nvidia CUDA v11 on a host compiler of clang++ v10 with C++17 standard `-std=c++17` and optimization flag `-O2` enabled. We do not observe significant difference between `-O2` and `-O3` in our experiments. Each run of N CPU cores and M GPUs corresponds to N CPU and M GPU worker threads. All data is an average of 20 runs.

7.1 Baseline

Give a large number of TGCSs, it is impossible to compare Taskflow with all of them. Each of the existing systems has its pros and cons and dominates certain applications. We consider oneTBB [2], StarPU [17], HPX [34], and OpenMP [7] each representing a particular paradigm that has gained some successful user experiences in CAD due to performance [45]. oneTBB (2021.1 release) is an industrial-strength parallel programming system under Intel oneAPI [2]. We consider its FlowGraph library and encapsulate each GPU task in a CPU function. At the time of this writing, FlowGraph does not have dedicated work stealing for HTDGs. StarPU (version 1.3) is a CPU-GPU task programming system widely used in the scientific computing community [17]. It provides a C-based syntax for writing HTDGs on top of a work-stealing runtime highly optimized for CPUs and GPUs. HPX (version 1.4) is a C++ standard library for concurrency and parallelism [34]. It supports implicit task graph programming through aggregating *future* objects in a dataflow API. OpenMP (version

4.5 in clang toolchains) is a directive-based programming framework for handling loop parallelism [7]. It supports static graph encoding using task dependency clauses.

To measure the expressiveness and programmability of Taskflow, we hire five PhD-level C++ programmers outside our research group to implement our experiments. We educate them the essential knowledge about Taskflow and baseline TGCSs and provide them all algorithm blocks such that they can focus on programming HTDGs. For each implementation, we record the lines of code (LOC), the number of tokens, cyclomatic complexity (measured by [8]), time to finish, and the percentage of time spent on debugging. We average these quantities over five programmers until they obtain the correct result. This measurement may be subjective but it highlights the programming productivity and turnaround time of each TGCSs from a real user's perspective.

7.2 Micro-benchmarks

We randomly generate a set of DAGs (i.e., HTDGs) with equal distribution of CPU and GPU tasks. Each task performs a SAXPY operation over 1K elements. For fair purpose, we implemented CUDA Graph [4] for all baselines; each GPU task is a CUDA graph of three GPU operations, H2D copy, kernel, and H2D copy, in this order of dependencies. Table 1 summarizes the programming effort of each method. Taskflow requires the least amount of lines of code (LOC) and written tokens. The cyclomatic complexity of Taskflow measured at a single function and across the whole program is also the smallest. The development time of Taskflow-based implementation is much more productive than the others. For this simple graph, Taskflow and oneTBB are very easy for our programmers to implement, whereas we found they spent a large amount of time on debugging task graph parallelism with StarPU, HPX, and OpenMP.

TABLE 1: Programming Effort on Micro-benchmark

Method	LOC	#Tokens	CC	WCC	Dev	Bug
Taskflow	69	650	6	8	14	1%
oneTBB	182	1854	8	15	25	6%
StarPU	253	2216	8	21	47	19%
HPX	255	2264	10	24	41	33%
OpenMP	182	1896	13	19	57	49%

CC: maximum cyclomatic complexity in a single function
WCC: weighted cyclomatic complexity of the program
Dev: minutes to complete the implementation
Bug: time spent on debugging as opposed to coding task graphs

Next, we study the overhead of task graph parallelism among Taskflow, oneTBB, and StarPU. As shown in Table 2, the static size of a task, compiled on our platform, is 272, 136, and 1472 bytes for Taskflow, oneTBB, and StarPU, respectively. We do not report the data of HPX and OpenMP because they do not support explicit task graph construction at the functional level. The time it takes for Taskflow to create a task and add a dependency is also faster than oneTBB and StarPU. We amortize the time across 1M operations because all systems support pooled memory to recycle tasks. We found StarPU has significant overhead in creating HTDGs. The overhead always occupies 5-10% of the total execution time regardless of the HTDG size.

TABLE 2: Overhead of Task Graph Creation

Method	S_{task}	T_{task}	T_{edge}	$\rho_{<10}$	$\rho_{<5}$	$\rho_{<1}$
Taskflow	272	61 ns	14 ns	550	2550	35050
oneTBB	136	99 ns	54 ns	1225	2750	40050
StarPU	1472	259 ns	384 ns	7550	-	-

S_{task}: static size per task in bytes
T_{task}/T_{edge}: amortized time to create a task/dependency
ρ_v: graph size where its creation overhead is below v%

Figure 13 shows the overall performance comparison between Taskflow and the baseline at different HTDG sizes. In terms of runtime (top left of Figure 13), Taskflow outperforms others across most data points. We complete the largest HTDG by 1.37×, 1.44×, 1,53×, and 1.40× faster than oneTBB, StarPU, HPX, and OpenMP, respectively. The memory footprint (top right of Figure 13) of Taskflow is close to oneTBB and OpenMP. HPX has higher memory because it relies on aggregated futures to describe task dependencies at the cost of shared states. Likewise, StarPU does not offer a closure-based interface and thus requires a flat layout (i.e., codelet) to describe tasks. We use the Linux `perf` tool to measure the power consumption of all cores plus LLC [3]. The total joules (bottom left of Figure 13) consumed by Taskflow is consistently smaller than the others, due to our adaptive worker management. In terms of power (bottom right of Figure 13), Taskflow, oneTBB, and OpenMP are more power-efficient than HPX and StarPU. The difference between Taskflow and StarPU continues to increase as we enlarge the HTDG size.

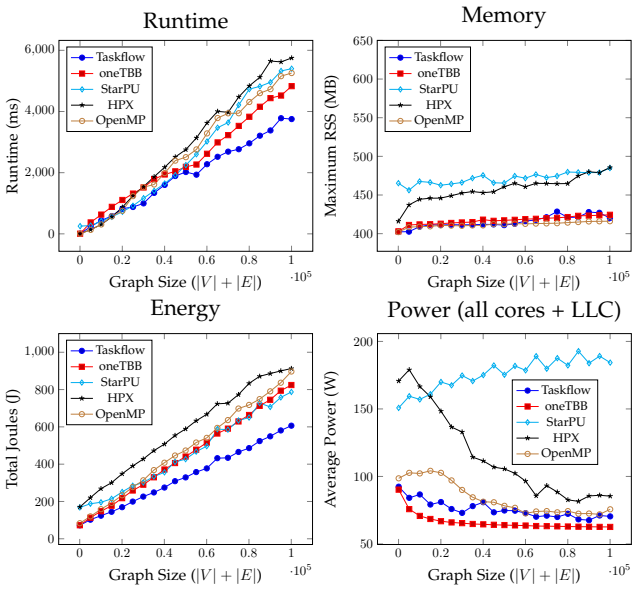


Fig. 13: Overall system performance at different problem sizes using 40 CPUs and 4 GPUs.

Figure 14 displays the runtime distribution of each method over a hundred runs of two HTDGs, 5K and 20K tasks. The boxplot shows that the runtime of Taskflow is more consistent than others and has the smallest variation. We attribute this result to the design of our scheduler, which effectively separates task execution into CPU and GPU workers and dynamically balances cross-domain wasteful steals with task parallelism.

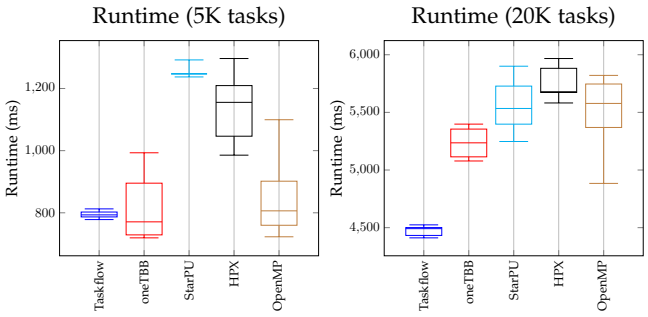


Fig. 14: Runtime distribution of two task graphs.

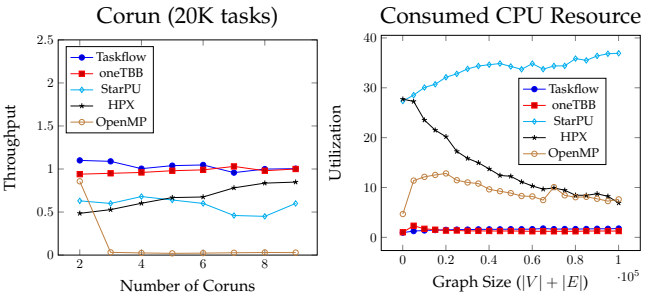


Fig. 15: Throughput of corunning task graphs and CPU utilization at different problem sizes under 40 CPUs and 4 GPUs.

Finally, we compare the throughput of each method on corunning HTDGs. This experiment emulates a server-like environment where multiple programs run simultaneously on the same machine to compete for the same resources. The effect of worker management propagates to all parallel processes. We consider up to nine corun processes each executing the same HTDG of 20K tasks. We use the weighted speedup [23] to measure the system throughput. Figure 15 compares the throughput of each method and relates the result to the CPU utilization. Both Taskflow and oneTBB produce significantly higher throughput than others. Our throughput is slightly better than oneTBB by 1–15% except for seven coruns. The result can be interpreted by the CPU utilization plot, reported by `perf stat`. We can see both Taskflow and oneTBB make effective use of CPU resources to schedule tasks. However, StarPU keeps workers busy most of the time and has no mechanism to dynamically control thread resources with task parallelism.

Since both oneTBB and StarPU provides explicit task graph programming models and work-stealing for dynamic load balancing, we will focus on comparing Taskflow with oneTBB and StarPU for the next two real workloads.

7.3 VLSI Incremental Timing Analysis

As part of our DARPA project, we applied Taskflow to solve a VLSI incremental static timing analysis (STA) problem in an optimization loop. The goal is to optimize the timing landscape of a circuit design by iteratively applying *design transforms* (e.g., gate sizing, buffer insertion) and evaluating the timing improvement until all data paths are passing, aka *timing closure*. Achieving timing closure is one of the most time-consuming steps in the VLSI design closure flow process because optimization algorithms can call a timer millions or even billions of times to incrementally analyze the timing improvement of a design transform. We consider the GPU-accelerated critical path analysis algorithm [26] and run it across one thousand incremental iterations based on the design transforms given by TAU 2015 Contest [29]. The data is generated by an industrial tool to evaluate the performance of an incremental timing algorithm. Each incremental iteration corresponds to at least one design modifier followed by a timing report operation to trigger incremental timing update of the timer.

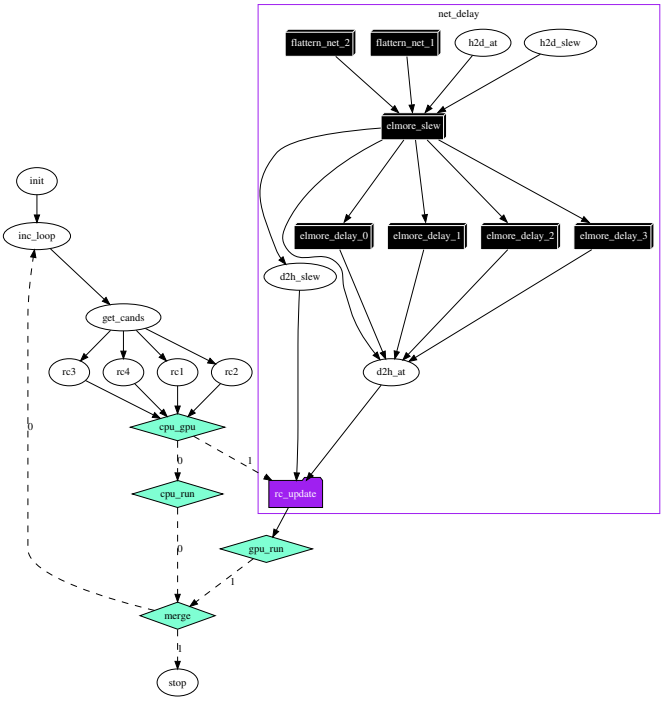


Fig. 16: A partial HTDG of 1 cudaFlow task (purple box), 4 condition tasks (green diamond), and 8 static tasks (other else) for one iteration of timing-driven optimization.

Figure 16 shows a partial Taskflow graph of our implementation. One condition task forms a loop to implement iterative timing updates and the other three condition tasks branch the execution to either CPU-based timing update (over 10K tasks) or GPU-based timing update (cudaFlow tasks). The motivation here is to adapt the timing update to different incrementalities. For example, if a design transform introduces only a few hundreds of nodes to update, there is no need to offload the computation to GPUs due to insufficient amount of data parallelism. The cudaFlow task composes over 1K operations to compute large interconnect delays, which often involves several gigabytes of parasitic data. Since oneTBB FlowGraph and StarPU do not support control flow, we unroll their task graphs across fixed-length iterations found in hindsight to avoid expensive synchronization at each iteration; the number of concatenated graphs is equal to the number of iterations.

Table 3 compares the programming effort between Taskflow, oneTBB, and StarPU. In a rough view, the implementation complexity using Taskflow is much less than that of oneTBB and StarPU. The amount of time spent on implementing the algorithm is about 3.9 hours for Taskflow, 6.1 hours for oneTBB, and 4.3 hours for StarPU. It takes 3–

TABLE 3: 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

4× more time to debug oneTBB and StarPU than Taskflow, mostly on control flow. Interestingly, while StarPU involves more LOC and higher cyclomatic complexity than oneTBB, our programmers found StarPU easier to write due to its C-styled interface. Although there is no standard way to conclude the programmability of a library, we believe our measurement highlights the expressiveness of Taskflow and its ease of use from a real user's perspective.

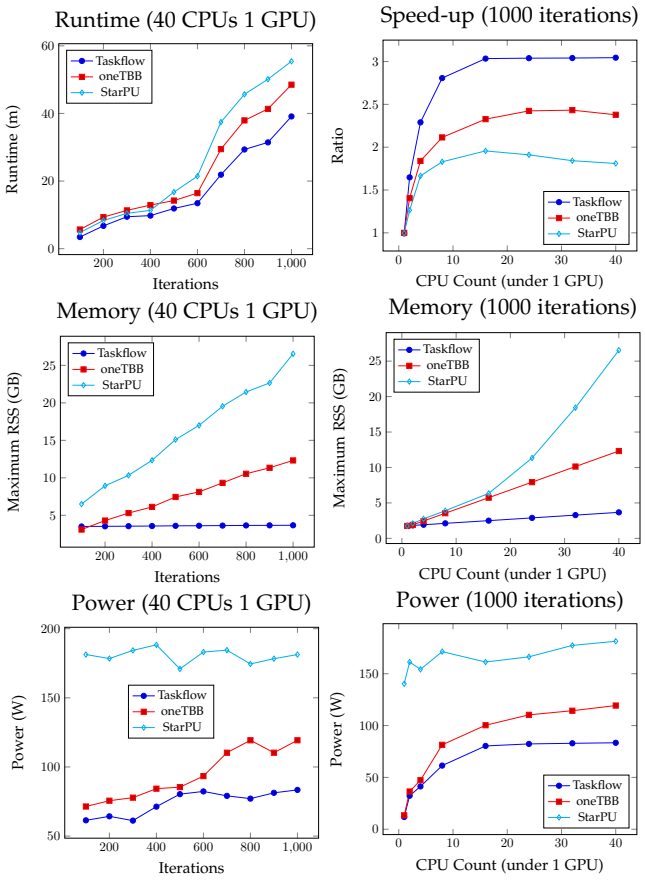


Fig. 17: Runtime, memory, and power data of 1000 incremental timing iterations (up to 11K tasks and 17K dependencies per iteration) on a large design of 1.6M gates.

The overall performance is shown in Figure 17. Using 40 CPUs and 1 GPU, Taskflow is consistently faster than oneTBB and StarPU across all incremental timing iterations. The gap continues to enlarge as increasing iteration numbers; at 100 and 1000 iterations, Taskflow reaches the goal in 3.45 and 39.11 minutes, whereas oneTBB requires 5.67 and 4.76 minutes and StarPU requires 48.51 and 55.43 minutes, respectively. Note that the gain is significant because a typical timing closure algorithm can invoke millions to billions of iterations that take several hours to finish [30]. We observed similar results at other CPU numbers; in terms of the runtime speed-up over 1 CPU (all finish in 113 minutes), Taskflow is always faster than oneTBB and StarPU, regardless of the CPU count. Speed-up of Taskflow saturates at about 16 CPUs (3×), primarily due to the inherent irregularity of the algorithm (see Figure 16). The memory footprint (middle of Figure 17) shows the benefit of our conditional tasking. By reusing condition tasks in the incremental timing loop, we do not suffer significant memory growth as oneTBB and StarPU. On a vertical scale, increasing the number of CPUs bumps up the memory usage of both methods, but Taskflow consumes much less because we use only simple atomic operations to control wasteful steals. In terms of energy efficiency (bottom of Figure 17, measured on all cores plus LLC using `power/energy-pkg` [3]), our scheduler is very power-efficient in completing the timing analysis workload, regardless of iterations and CPU numbers. Beyond 16 CPUs where performance saturates, Taskflow does not suffer from increasing power as oneTBB and StarPU, because our scheduler efficiently balances the number of workers with dynamic task parallelism.

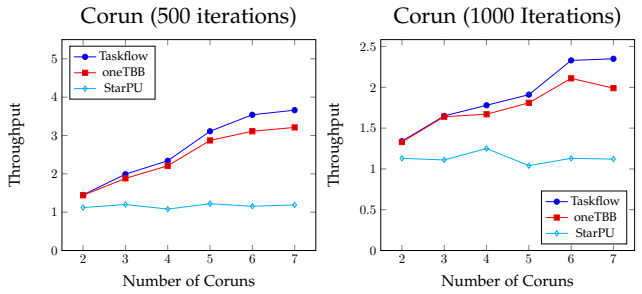


Fig. 18: Throughput of corunning timing analysis workloads on two iteration numbers using 40 CPUs and 1 GPU.

We next compare the throughput of each implementation by corunning the same program. Corunning programs is a common strategy for optimization tools to search for the best parameters. The effect of worker management propagates to all simultaneous processes. Thus, the throughput can be a good measurement for the inter-operability of a scheduling algorithm. We corun the same timing analysis program up to seven processes that compete for 40 CPUs and 1 GPU. We use the *weighted speedup* to measure the system throughput, which is the sum of the individual speedup of each process over a baseline execution time [23]. A throughput of one implies that the corun's throughput is the same as if the processes were run consecutively. Figure 18 plots the throughput across nine coruns at two iteration numbers. Both Taskflow and oneTBB achieve decent throughput greater than one and are significantly better than StarPU. We found StarPU keep workers busy most of the time and has no mechanism to balance the number of workers with dynamically generated task parallelism. For irregular HTDGs akin to Figure 16, worker management is critical for corunning processes. When task parallelism becomes sparse, especially around the decision-making point of an iterative control flow, our scheduler can adaptively reduce the wasteful steals based on the active worker count, and we offer a stronger bound than oneTBB (Theorem

3). Saved wasteful resources can thus be used by other concurrent programs to increase the throughput.

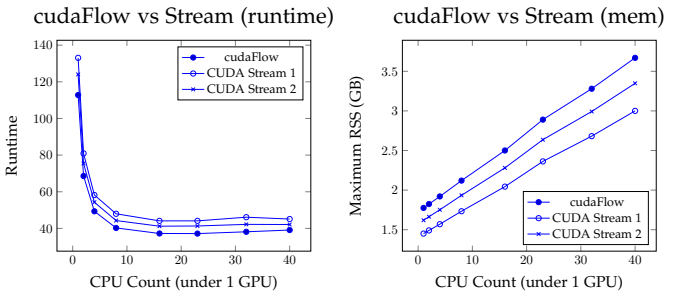


Fig. 19: Comparison of runtime and memory between cudaFlow (CUDA Graph) and stream-based execution in the VLSI incremental timing analysis workload.

Figure 19 shows the performance advantage of CUDA Graph and its cost in handling this large GPU-accelerated timing analysis workloads. The line *cudaFlow* represents our default implementation using explicit CUDA graph construction. The other two lines represent the implementation of the same GPU task graph but using stream and event insertions (i.e., non-CUDA Graph). As partially shown in Figure 16, our cudaFlow composes over 1K dependent GPU operations to compute the interconnect delays. For large GPU workloads like this, the benefit of CUDA Graph is clear; we observed 9–17% runtime speed-up over stream-based implementations. The performance improvement mostly comes from reduced kernel call overheads and graph-level scheduling optimizations by CUDA runtime. Despite the improved performance, cudaFlow incurs higher memory costs because CUDA Graph stores all kernel parameters in advance for optimization. For instance, creating a node in CUDA Graph can take over 300 bytes of opaque data structures.

7.4 Large Sparse Neural Network Inference

We applied Taskflow to solve the MIT/Amazon Large Sparse Deep Neural Network (LSDNN) Inference Challenge, a recent effort aimed at new computing methods for sparse AI analytics [36]. Each dataset comprises a sparse matrix of the input data for the network, 1920 layers of neurons stored in sparse matrices, truth categories, and the bias values used for the inference. Preloading the network to the GPU is impossible. Thus, we implement a model decomposition-based kernel algorithm inspired by [19] and construct an end-to-end HTDG for the entire inference workload. Unlike VLSI incremental timing analysis, this workload is both CPU- and GPU-heavy. Figure 20 illustrates a partial HTDG. We create up to 4 cudaFlows on 4 GPUs. Each cudaFlow contains more than 2K GPU operations to run partitioned matrices in an iterative data dispatching loop formed by a condition task. Other CPU tasks evaluate the results with a golden reference. Since oneTBB FlowGraph and StarPU do not support in-graph control flow, we unroll their task graph across fixed-length iterations found offline.

Figure 21 compares the performance of solving a 1920-layered LSDNN each of 4096 neurons under different CPU and GPU numbers. Taskflow outperforms oneTBB and

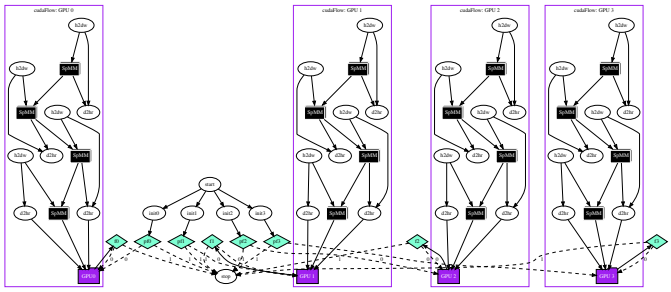


Fig. 20: A partial HTDG of 4 cudaFlows (purple boxes), 8 conditioned cycles (green diamonds), and 6 static tasks (other else) for the inference workload.

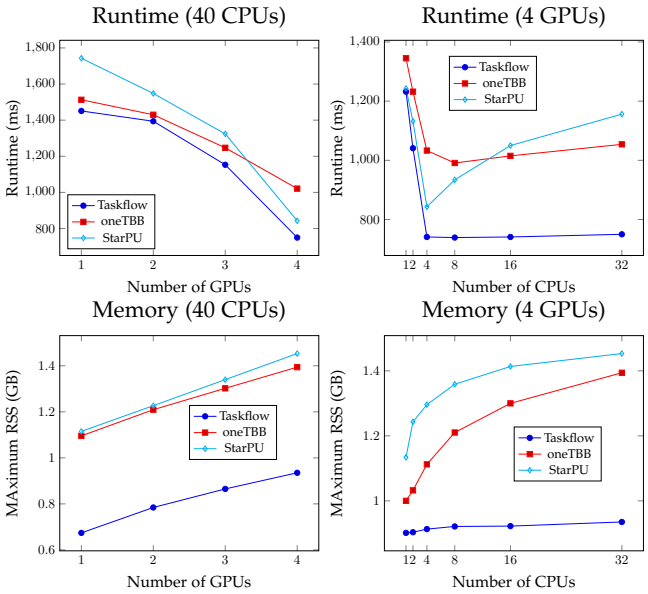


Fig. 21: Runtime and memory data of the LSDNN (1920 layers, 4096 neurons per layer) under different CPU and GPU numbers

StarPU in all aspects. Both our runtime and memory scale better regardless of the CPU and GPU numbers. Using 4 GPUs, when performance saturates at 4 CPUs, we do not suffer from further runtime growth as oneTBB and StarPU. This is because our work-stealing algorithm more efficiently control wasteful steals upon available task parallelism. On the other hand, our memory usage is 1.5-1.7× less than oneTBB and StarPU. This result highlights the benefit of our condition task, which integrates iterative control flow into a cyclic HTDG, rather than unrolling it statically across iterations.

We next compare the throughput of each implementation by corunning the same inference program to study the inter-operability of an implementation. We corun the same inference program up to nine processes that compete for 40 CPUs and 4 GPUs. We use weighted speedup to measure the throughput. Figure 22 plots the throughput of corunning inference programs on two different sparse neural networks. Taskflow outperforms oneTBB and StarPU across all coruns. oneTBB is slightly better than StarPU because StarPU tends to keep all workers busy all the time and results in large numbers of wasteful steals. The largest difference is observed at five coruns of inferencing

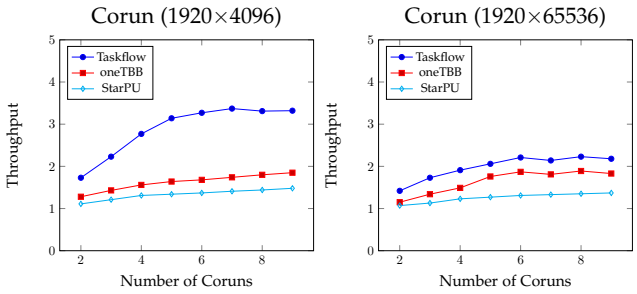


Fig. 22: Throughput of corunning inference workloads on two 1920-layered neural networks, one with 4096 neurons per layer and another with 65536 neurons per layer.

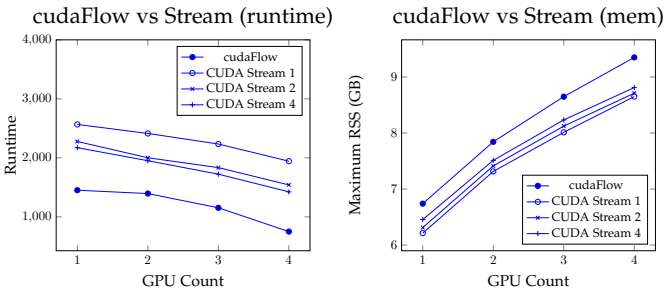


Fig. 24: Comparison of runtime and memory between cudaFlow (CUDA Graph) and stream-based execution.

the 1920×4096 neural network, where our throughput is 1.9× higher than oneTBB and 2.1× higher than StarPU. These CPU- and GPU-intensive workloads highlight the effectiveness of our heterogeneous work stealing. By keeping a per-domain invariant, we can control cross-domain wasteful steals to a bounded value at any time during the execution.

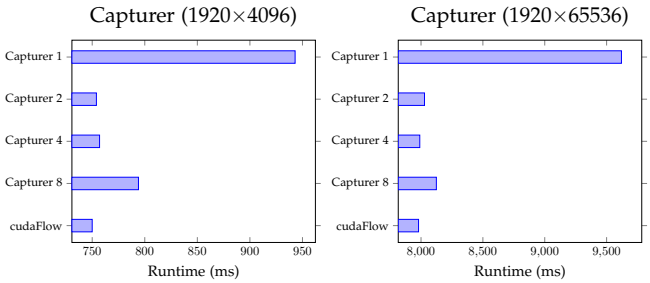


Fig. 23: Performance of our cudaFlow capturer using 1, 2, 4, and 8 streams to complete the inference of two neural networks.

We study the performance of our cudaFlow capturer using different numbers of streams (i.e., $max\_streams$). For complex GPU workloads like Figure 20, stream concurrency is crucial to GPU performance. As shown in Figure 23, explicit construction of a CUDA graph using cudaFlow achieves the best performance, because the CUDA runtime can dynamically decide the stream concurrency with internal optimization. For applications that must use existing stream-based APIs, our cudaFlow capturer achieves comparable performance as cudaFlow by using two or four streams. Taking the 1920×65536 neural network for example, the difference between our capturer of four streams and cudaFlow is only 10 ms. For this particular workload, we do not observe any performance benefit beyond four streams. Application developers can fine-tune this number.

We finally compare the performance of cudaFlow with stream-based execution. As shown in Figure 24, the line cudaFlow represents our default implementation using explicit CUDA graph construction, and the other lines represent stream-based implementations for the same task graph using one, two, and four streams. The advantage of CUDA Graph is clearly demonstrated in this large machine learning workload of over 2K dependent GPU operations per cudaFlow. Under four streams that deliver the best performance for the baseline, cudaFlow is 1.5× (1451 vs 2172) faster at one GPU and is 1.9× (750 vs 1423) faster at four GPUs. The cost of this performance improvement is increased memory usage because CUDA Graph needs to store all the operating parameters in the graph. For instance, under four streams, cudaFlow has 4% and 6% higher memory usage than stream-based execution at one and four GPUs, respectively.

8 Related Work

8.1 Heterogeneous Programming Systems

Heterogeneous programming systems are the main driving force to advance scientific computing. Directive-based programming models [5], [6], [7], [25], [39] allow users to augment program information of loop mapping onto CPUs/GPUs and data sharing rules to designated compilers for automatic parallel code generation. These models are good at loop-based parallelism but cannot handle irregular task graph patterns efficiently [38]. Functional approaches [2], [15], [17], [18], [20], [24], [31], [34], [35], [42] offer either implicit or explicit task graph constructs that are more flexible in runtime control and on-demand tasking. Each of these systems has its pros and cons. However, few of them enable end-to-end expressions of heterogeneously dependent tasks with general control flow.

8.2 Heterogeneous Scheduling Algorithms

Among various heterogeneous runtimes, work stealing is a popular strategy to reduce the complexity of load balancing [16], [42] and has inspired the designs of many parallel runtimes [2], [12], [40], [41], [53]. A key challenge in work-stealing designs is worker management. Instead of keeping all workers busy most of the time [16], [17], [31], both oneTBB [2] and BWS [23] have developed sleep-based strategies. oneTBB employs a mixed strategy of fixed-number worker notification, exponential backoff, and noop assembly. BWS modifies OS kernel to alter the yield behavior. [43] takes inspiration from BWS and oneTBB to develop an adaptive work-stealing algorithm to minimize the number of wasteful steals. Other approaches, such as [13] that targets a space-sharing environment, [48] that tunes hardware frequency scaling, [22], [49] that balance load on distributed memory, [21], [27], [52], [58] that deal with data locality, and [50] that focuses on memory-bound applications have improved work stealing in certain performance aspects, but their results are limited to the CPU domain. How to migrate the above approaches to a heterogeneous target remains an open question.

In terms of GPU-based task schedulers, Whippletree [51] design a fine-grained resource scheduling algorithm for sparse and scattered parallelism atop a custom program model. [46] leverages reinforcement learning to place machine learning workloads onto GPUs. Hipacc [47] introduces a pipeline-based optimization for CUDA graphs to speed up image processing workloads. [56] develops a compiler to transforms OpenMP directives to a CUDA graph. These works have primarily focused on scheduling GPU tasks in various applications, which are orthogonal to our generic heterogeneous scheduling approaches.

9 Acknowledgements

The project is supported by the DARPA contract FA 8650-18-2-7843 and the NSF grant CCF-2126672. We appreciate all Taskflow contributors and reviewers' comments for improving this paper.

10 Conclusion

In this paper, we have introduced Taskflow, a lightweight task graph computing system to streamline the creation of heterogeneous programs with control flow. Taskflow has introduced a new programming model that enables an end-to-end expression of heterogeneously dependent tasks with general control flow. We have developed an efficient work-stealing runtime optimized for latency, energy efficiency, and throughput, and derived theory results to justify its efficiency. We have evaluated the performance of Taskflow on both micro-benchmarks and real applications. As an example, Taskflow solved a large-scale machine learning problem up to 29% faster, 1.5× less memory, and 1.9× higher throughput than the industrial system, oneTBB, on a machine of 40 CPUs and 4 GPUs.

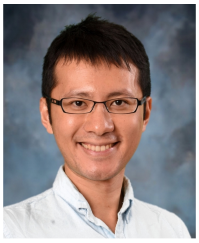
Taskflow is an on-going project under active development. We are currently exploring three directions: First, we are designing a distributed tasking model based on partitioned taskflow containers with each container running on a remote machine. Second, we are extending our model to incorporate SYCL [9] to provide a single-source heterogeneous task graph programming environment. The author Dr. Huang is a member of SYCL Advisory Panel and is collaborating with the working group to design a new SYCL Graph abstraction. Third, we are researching automatic translation methods between different task graph programming models using Taskflow as an intermediate representation. Programming-model translation has emerged as an important research area in today's diverse computing environments because no one programming model is optimal across all applications. The recent 2021 DOE X-Stack program directly calls for novel translation methods to facilitate performance optimizations on different computing environments.

One important future direction is to collaborate with Nvidia CUDA teams to design a conditional tasking interface within the CUDA Graph itself. This design will enable efficient control-flow decisions to be made completely in CUDA runtime, thereby largely reducing the control-flow cost between CPU and GPU.

References

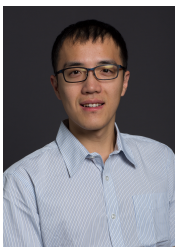
[1] DARPA Intelligent Design of Electronic Assets (IDEA) Program. https://www.darpa.mil/program/intelligent-design-of-electronic-assets.
[2] Intel oneTBB. https://github.com/oneapi-src/oneTBB.
[3] Linux Kernel Profiler. https://man7.org/linux/man-pages/man1/perf-stat.1.html.
[4] Nvidia CUDA Graph. https://devblogs.nvidia.com/cuda-10-features-revealed/.
[5] OmpSs. https://pm.bsc.es/ompss.
[6] OpenACC. http://www.openacc-standard.org.
[7] OpenMP. https://www.openmp.org/.
[8] SLOCCount. https://dwheeler.com/sloccount/.
[9] SYCL. https://www.khronos.org/sycl/.
[10] Taskflow github. https://taskflow.github.io/.
[11] Two-phase Commit Protocol. http://www.1024cores.net/home/lock-free-algorithms/eventcounts.
[12] K. Agrawal, C. E. Leiserson, and J. Sukha. Nabbit: Executing task graphs using work-stealing. In *IEEE IPDPS*, pages 1–12, 2010.
[13] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *PPoPP*, pages 112–120, 2007.
[14] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. In *ACM/IEEE DAC*, 2019.
[15] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley and Sons, Ltd, 2017.
[16] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *ACM SPAA*, pages 119–129, 1998.
[17] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, 2011.
[18] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *IEEE/ACM SC*, pages 1–11, 2012.
[19] M. Bisson and M. Fatica. A GPU Implementation of the Sparse Deep Neural Network Graph Challenge. In *IEEE HPEC*, pages 1–8, 2019.
[20] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45, 2013.
[21] Quan Chen, Minyi Guo, and Haibing Guan. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures. In *ACM ICS*, page 3–12, 2014.
[22] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *ACM SC*, SC '09, 2009.
[23] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. BWS: Balanced Work Stealing for Time-sharing Multicores. In *ACM EuroSys*, pages 365–378, 2012.
[24] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014.
[25] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *IEEE IPDPS*, pages 1299–1308, 2013.
[26] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Pash-based Timing Analysis. In *IEEE/ACM DAC*, 2021.
[27] Yi Guo. A Scalable Locality-Aware Adaptive Work-Stealing Scheduler for Multi-Core Task Parallelism. 2010.
[28] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. GPU-accelerated Static Timing Analysis. In *IEEE/ACM ICCAD*, pages 1–8, 2020.
[29] J. Hu, G. Schaeffer, and V. Garg. TAU 2015 Contest on Incremental Timing Analysis. In *IEEE/ACM ICCAD*, pages 895–902, 2015.

[30] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD*, (4):776–789, 2021.
[31] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS*, pages 974–983, 2019.
[32] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE TCAD*, 2021 (to appear).
[33] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. Cpp-taskflow: A general-purpose parallel task programming system at scale. *IEEE TCAD*, 40(8):1687–1700, 2021.
[34] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *PGAS*, pages 6:1–6:11, 2014.
[35] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A Portable Concurrent Object Oriented System Based on C++. In *ACM ASPLOS*, page 91–108, New York, NY, USA, 1993.
[36] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. Sparse deep neural network graph challenge. *IEEE HPEC*, 2019.
[37] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and Efficient Work-stealing for Weak Memory Models. In *ACM PPoPP*, pages 69–80, 2013.
[38] S. Lee and J. S. Vetter. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In *IEEE/ACM SC*, pages 1–11, 2012.
[39] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *IEEE/ACM SC*, pages 1–11, 2010.
[40] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *ACM OOPSLA*, pages 227–241, 2009.
[41] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.
[42] João V.F. Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin, and Nicolas Ma illard. Design and Analysis of Scheduling Strategies for multi-CPU and multi-GPU Architectures. *Parallel Comput.*, 44:37–52, 2015.
[43] Chun-Xun Lin, Tsung-Wei Huang, and Martin D. F. Wong. An efficient work-stealing scheduler for task dependency graph. In *2020 IEEE ICPADS*, pages 64–71, 2020.
[44] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan. ABCD-Place: Accelerated Batch-based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs. *IEEE TCAD*, 2020.
[45] Yi-Shan Lu and Keshav Pingali. Can Parallel Programming Revolutionize EDA Tools? *Advanced Logic Synthesis*, 2018.
[46] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *ACM ICML*, page 2430–2439, 2017.
[47] Bo Qiao, M. Akif Özkan, Jürgen Teich, and Frank Hannig. The best of both worlds: Combining cuda graph with an image processing dsl. In *IEEE/ACM DAC*, pages 1–6, 2020.
[48] Haris Ribic and Yu David Liu. Energy-efficient Work-stealing Language Runtimes. In *ACM ASPLOS*, pages 513–528, 2014.
[49] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-Based Global Load Balancing. In *PPoPP*, page 201–212, 2011.
[50] Shumpei Shiina and Kenjiro Taura. Almost Deterministic Work Stealing. In *ACM SC*, 2019.
[51] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6), November 2014.
[52] Warut Suksompong, Charles E. Leiserson, and Tao B. Schardl. On the efficiency of localized work stealing. *Information Processing Letters*, 116(2):100–106, Feb 2016.
[53] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. *SIGPLAN*, 47(8):267–276, 2012.
[54] D. F. Wong, H. W. Leong, and C. L. Liu. *Simulated Annealing for VLSI Design*. Kluwer Academic Publishers, USA, 1988.
[55] B. Xu, K. Zhu, M. Liu, Y. Lin, S. Li, X. Tang, N. Sun, and D. Z. Pan. MAGICAL: Toward Fully Automated Analog IC Layout Leveraging Human and Machine Intelligence: Invited Paper. In *IEEE/ACM ICCAD*, pages 1–8, 2019.
[56] Chenle Yu, Sara Royuela, and Eduardo Quiñones. Openmp to cuda graphs: A compiler-based transformation to enhance the programmability of nvidia devices. In *International Workshop on Software and Compilers for Embedded Systems*, page 42–47, 2020.
[57] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic Control Flow in Large-Scale Machine Learning. In *IEEE EuroSys*, 2018.
[58] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. Bandwidth and Locality Aware Task-Stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. *ACM Trans. Archit. Code Optim.*, 15(4), 2018.



Tsung-Wei Huang received the B.S. and M.S. degrees from the Department of Computer Science, National Cheng Kung University (NCKU), Tainan, Taiwan, in 2010 and 2011, respectively. He obtained his Ph.D. degree in the Electrical and Computer Engineering (ECE) Department at the University of Illinois at Urbana-Champaign (UIUC). He is currently an assistant professor in the ECE department at the University of Utah. Dr. Huang has been building software systems for parallel computing and timing analysis. His PhD thesis won the prestigious 2019 ACM SIGDA Outstanding PhD Dissertation Award for his contributions to distributed and parallel VLSI timing analysis.

Dian-Lun Lin received the B.S. degree from the Department of Electrical Engineering at Taiwan's Cheng Kung University and M.S. degree from the Department of Computer Science at National Taiwan University. He is current a Ph.D. student at the Department of Electrical and Computer Engineering at the University of Utah. His research interests are in parallel and heterogeneous computing with a specific focus on CAD applications.



Chun-Xun Lin received the B.S. degree in Electrical Engineering from the National Cheng Kung University, Tainan, Taiwan, and the M.S. degree in Electronics Engineering from the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan, in 2009 and 2011, respectively. He received his Ph.D. degree from the department of Electrical and Computer Engineering (ECE) at the University of Illinois at Urbana-Champaign (UIUC) in 2020. His research interest is in parallel processing.

Yibo Lin (S'16–M'19) received the B.S. degree in microelectronics from Shanghai Jiaotong University in 2013, and his Ph.D. degree from the Electrical and Computer Engineering Department of the University of Texas at Austin in 2018. He is current an assistant professor in the Computer Science Department associated with the Center for Energy-Efficient Computing and Applications at Peking University, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.