Cpp-Taskflow v2: A General-purpose Parallel and Heterogeneous Task Programming System at Scale

Tsung-Wei Huang[1], Dian-Lun Lin[1], Yibo Lin[2], and Chun-Xun Lin[3]

[1]Department of Electrical and Computer Engineering, University of Utah
[2]Center for Energy-Efficient Computing and Applications, Peking University
[3]Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign

Abstract

The Cpp-Taskflow project addresses the long-standing question: *How can we make it easier for developers to write parallel and heterogeneous programs with high performance and simultaneous high productivity?* Cpp-Taskflow develops a simple and powerful task programming model to enable efficient implementations of heterogeneous decomposition strategies. Our programming model empowers users with both static and dynamic task graph constructions to incorporate a broad range of computational patterns including hybrid CPU-GPU computing, dynamic control flow, and irregularity. We develop an efficient heterogeneous work-stealing strategy that adapts worker threads to available task parallelism at any time during the graph execution. We have demonstrated promising performance of Cpp-Taskflow on both micro-benchmark and real-world applications. As an example, we solved a large machine learning workload by up to 1.5× faster, 1.6× less memory, and 1.7× fewer lines of code than two industrial-strength systems, oneTBB and StarPU, on a machine of 40 CPUs and 4 GPUs.

1 Introduction

Modern scientific computing relies on a heterogeneous mix of computational patterns, domain algorithms, and specialized hardware to achieve key scientific milestones that go beyond traditional capabilities. However, programming these applications often requires complex expert-level tools and a deep understanding of software methodologies. Specifically, the lack of a suitable software environment that can overcome the complexity of programming large parallel and heterogeneous systems has posed a significant barrier for many organizations to facilitate transformational discoveries [50]. Decades of research in high productivity computing has yielded methods and languages that offer either programmer productivity or performance scalability, but rarely both simultaneously [3]. Neither programming models nor runtimes, despite some improvements in domain-specific problems such as machine learning, are mature enough to allow us to migrate generic applications to heterogeneous targets in a timely manner.

Cpp-Taskflow v2 is a general-purpose task programming system to streamline the creation of parallel and heterogeneous applications comprising CPUs, GPU, and custom accelerators. We have based Cpp-Taskflow v2 on the state-of-the-art manycore programming system, Cpp-Taskflow v1 [27], which has extensive user experience in the circuit design industry [28, 29, 41, 42, 44, 51], and generalized its idea to heterogeneous computing with relative ease of programming. Hereafter, Cpp-Taskflow refers to v2 unless otherwise stated. [1] Cpp-Taskflow explores effective tradeoff between *programming productivity*, *solution generality*, and *performance scalability*. We summarize three major contributions of Cpp-Taskflow as follows:

- **Expressive, powerful programming model.** We enable developers to efficiently implement parallel and heterogeneous decomposition strategies using task graph models. Our model empowers users with both static and dynamic task graph constructions to incorporate a broad range of computational patterns using an expressive, unified application programming interface (API). The power of our expressiveness lets developers perform rather a lot of work without writing a lot of code. 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.
- **General control flow.** We support general control flow that goes beyond the capability of conventional directed acyclic graph (DAG) models. Developers benefit from the ability to make rapid control-flow decisions using our conditional tasking interface. Applications can describe loops, nested cycles, branches, or non-deterministic decision points together with other task constructs in a uniform graph entity. In case where dynamic behavior is frequent, such as optimization and branch and bound, developers can efficiently overlap tasks both inside and outside the path of decision making.
- **Heterogeneous work stealing.** We develop an efficient work-stealing strategy to adapt worker threads to dynamically generated task parallelism at any time during the

[1]Cpp-Taskflow: https://https://github.com/cpp-taskflow/.

graph execution. Our strategy prevents the graph execution from underutilized threads that is harmful for performance, while avoiding excessive waste of thread resources when available tasks are scarce. The result largely improves the overall system performance, including latency, energy efficiency, and throughput. Our strategy is generalizable to arbitrary heterogeneous domains.

We have evaluated Cpp-Taskflow on both micro-benchmark and real-world applications and demonstrated its promising performance over existing programming systems. As an example, Cpp-Taskflow solved a large-scale circuit placement problem by up to 17% faster, 1.3× fewer memory, 2.1× less power consumption, and 2.9× higher throughput using 1.9× fewer lines of code than two industrial-strength systems, oneTBB and StarPU, on a machine of 40 CPUs and 4 GPUs. We believe Cpp-Taskflow stands out as a unique system given the ensemble of software tradeoffs and architecture decisions we have made.

We organize the paper as follows: Section 2 describes the motivation of Cpp-Taskflow. Sections 3 and 4 introduce the programming model of Cpp-Taskflow and its system runtime. Section 5 demonstrates the experimental results on micro-benchmark and realistic applications. Section 6 describes the related work. Finally, we draw acknowledgment and conclusion in Sections 7 and 8.

2 Motivation: Parallelizing VLSI CAD

The key motivation behind Cpp-Taskflow is our ("author's research team and industrial partners") research and development (R&D) experience in parallelizing computer-aided design (CAD) tools for very large-scale integration (VLSI) systems. Our area of expertise yields promising and realistic insights for the pros and cons of existing toolchains, and why we need a new programming system to advance the state of the art. In fact, CAD has solved many of the most difficult computational problems in the world. The computational challenges we address are representative of a wide range of scientific computing applications.

The semiconductor industry never stops seeking to reduce the design time and effort in integrated circuit (IC) implementation that incorporates billions of transistors [19, 31]. The recent DARPA Intelligent Design of Electronic Assets (IDEA) program directly called out the need for a *no human in the loop*, 24-hour design framework for IC implementation [2]. A central theme is *the core CAD algorithms must incorporate new parallel paradigms* to allow more efficient design space exploration and optimization. This is an extremely challenging job, particularly, for R&D. Figure 1 shows a canonical CAD flow and highlights computational problems in the layout generation stage. CAD spans various computing disciplines and makes essential use of dynamic control flow and irregular patterns. Each problem has

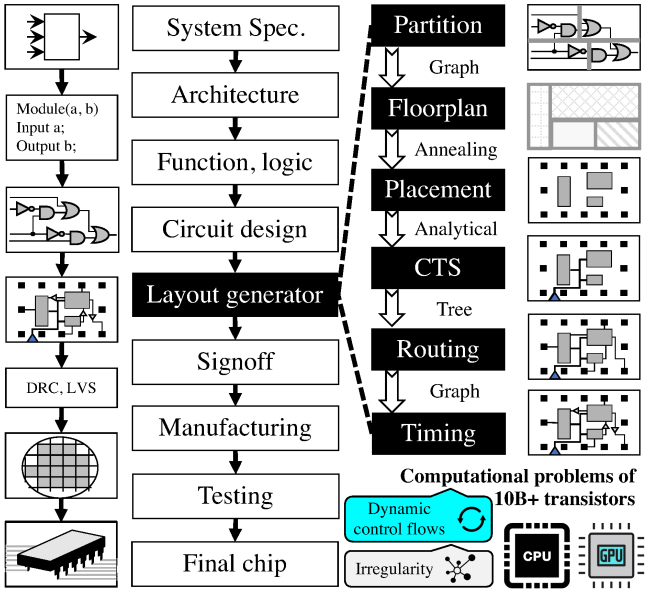


Figure 1: Cpp-Taskflow is motivated to address the computational challenges in parallelizing VLSI CAD. CAD spans various computing disciplines and makes essential use of dynamic control flow and irregular computational patterns, with all the difficulties it entails.

unique computational patterns and performance characteristics, both of which require very strategic decompositions to benefit from parallelism. The resulting task graph in terms of encapsulated function calls and task dependencies are extremely large and complex. Programming these tasks in a scalable manner has been one greatest hurdle to overcome for reasonable turnaround time and performance [22, 46, 48].

2.1 Issues of Existing Programming Systems

Over the past five years, we have invested a lot of R&D effort in existing task programming systems [6,10,11,18,21,24,32]. Each of these systems has its own pros and cons and deserves a reason to exist. In our use case, however, few of them can effectively answer the question: *how can we program large heterogeneous CAD workloads with high performance and simultaneous high programming productivity?* We highlight three major issues. First, existing frameworks are disadvantageous from an ease-of-use standpoint. Users often need to sort out many distinct notations and library details before implementing a heterogeneous algorithm. Second, existing models mostly rely on DAG models to define tasks and dependencies. Very few of them support general control flow in task parallelism. Third, existing scheduling algorithms are good at either CPU- or GPU-intensive workloads, but rarely both simultaneously. Neither are they generalizable to arbitrary heterogeneous domains.

2.2 Need for a New Programming System

Based on many years of research, we conclude that a new programming solution is needed. While being inspired by parallelizing CAD, we aim for a *general-purpose C++ task graph programming system* to streamline parallel and heterogeneous programming and benefit a wide variety of scientific computing. We target on a single heterogeneous node comprising manycore CPUs, GPUs, and custom accelerators. We are interested in irregular workloads of intensive CPU-GPU dependent tasks and dynamic control flow, where existing frameworks fail to handle efficiently. We do not devote effort to simplifying kernel programming but focus on heterogeneous tasking that affects the overall system performance to a large extent. Our model is compatible with existing frameworks and together we can enable proliferation of new algorithms and methodologies in face of future heterogeneity.

3 Unified Programming Model

We discuss the programming model of Cpp-Taskflow in five types, *static task*, *dynamic task*, *composable task*, *condition task*, and *cudaFlow task*. These tasks are associated with each other to represent a generic *task dependency graph* (TDG). The API used for one task type is nearly applicable for the other task types. Developers need not to learn a different set of API and can pick up the knowledge at a fast pace.

> *We help developers quickly write large parallel and heterogeneous programs with high performance scalability and simultaneous high productivity.*
>
> — Project Mantra

3.1 Static Tasking

Static tasking captures the static parallel structure of a decomposition strategy and is defined only by the program itself. It has a *flat* task hierarchy and cannot spawn new tasks from a running task graph. Listing 1 demonstrates an example Cpp-Taskflow program. The code *explains itself*. The program creates a TDG of four tasks, A, B, C, and D. The dependency constraints state that task A runs before task B and task C, and task D runs after task B and task C. There is neither explicit thread managements nor complex lock controls in the code.

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

Listing 1: A static TDG in Cpp-Taskflow.

Cpp-Taskflow is *object-oriented*. A task in Cpp-Taskflow is defined as a *callable* object for which the operation `std::invoke` is applicable. A taskflow object is the gateway to create a TDG and submit it to an *executor* that manages a set of worker threads to run tasks. Each time users create a task, the taskflow adds a node to the present TDG and returns a *task handle*. A task handle is a lightweight class object that wraps up a particular node in a graph and provides an extensible layer to modify task attributes. Each node has a general-purpose polymorphic function wrapper to store and invoke any callable target (task) given by users. Hereafter, we use "task A" to represent the task stored in node A.

3.2 Dynamic Tasking

Dynamic tasking refers to the creation of a TDG during the execution context of a task. Dynamic tasks are created from a running TDG. These tasks are spawned from a parent task and are grouped together to form a TDG called *subflow*. The same methods defined for static tasking are all applicable for dynamic tasking. Figure 2 shows an example of dynamic tasking. The TDG has four static tasks, A, C, D, and B. The precedence constraints force task A to run before tasks B and C, and task D to run after tasks B and C. During the execution of task B, it spawns another TDG of three tasks B1, B2, and B3 (marked as cyan), where task B1 and task B2 run before task B3. In Cpp-Taskflow, tasks B1, B2, and B3 are grouped to a subflow parented at task B.

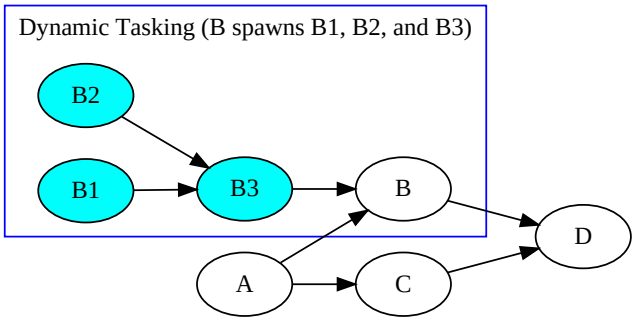


Figure 2: A TDG of four static tasks (A, B, C, and D) and three dynamic tasks (B1, B2, and B3).

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

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

Listing 2: Cpp-Taskflow code of Figure 2.

Listing 2 shows the Cpp-Taskflow code in Figure 2. A dynamic task accepts an argument of type `tf::Subflow` that is created by the executor and passed to the execution of task B. A subflow inherits all graph building blocks from static tasking. By default, a spawned subflow joins its parent task, 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 the taskflow.

3.3 Composable Tasking

Composable tasking enables developers to define task hierarchies and create large TDGs from composition of modular and reusable blocks that are easier to optimization. Graph decomposition is a key element to improve productivity and scalability of programming large parallel workloads. Figure 3 gives an example of a TDG using these three constructs. 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 composed of a taskflow of two dependent tasks A and B.

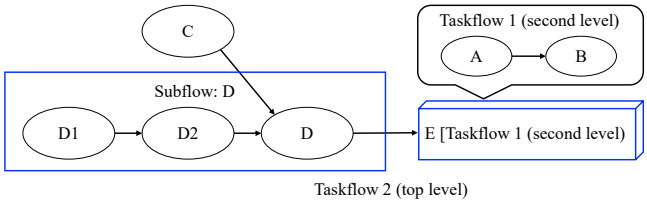


Figure 3: An example of taskflow composition.

```
tf::Taskflow taskflow1, taskflow2;
auto [A, B] = taskflow1.emplace(
  [] () { std::cout << "TaskA"; },
  [] () { std::cout << "TaskB"; }
);
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);
A.precede(B);
C.precede(D);
D.precede(E);
```

Listing 3: Cpp-Taskflow code of Figure 3.

Listing 3 shows the Cpp-Taskflow code of Figure 3. It declares two taskflows, `taskflow1` and `taskflow2`. The second taskflow defines a `module` task that is composed of the first taskflow, preceded by task D. A module task does not own the taskflow but maintains a soft mapping to the taskflow. Users can create multiple module tasks from the same taskflow but they must not run concurrently. Figure 4 shows an invalid taskflow composition, since the two module tasks may race. Composition can be nested or recursive. Our runtime is able to run each layer of taskflow hierarchies, regardless of static or dynamic tasking.

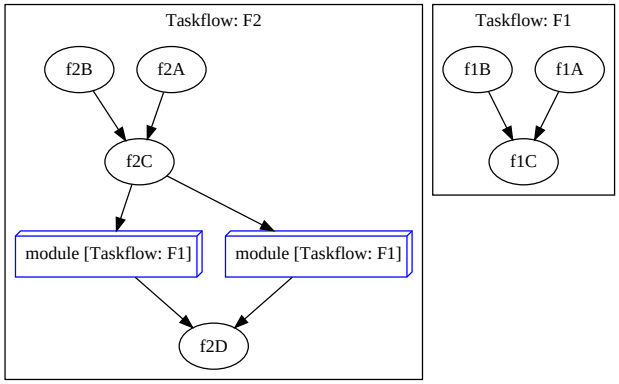


Figure 4: An invalid taskflow composition. The two module tasks composed of F1 may race.

3.4 Conditional Tasking

One major limitation of existing tasking frameworks is *static* control flow. Most of their models are DAG-based. For tasking over sequences, we unroll fixed-length iterations statically. In case of nested or non-deterministic conditionals, we resort to client-side control-flow decisions. To overcome this limitation, we develop a powerful interface of conditional tasking to support general control flow in task parallelism. A *condition task* evaluates a set of instructions and returns the next immediate successor to execute. Developers use condition tasks to implement branches or cycles, both static and dynamic, to skip or iterate the execution of a subgraph. Our conditional tasking is different from a dataflow graph, in the sense that control-flow decisions are encoded as task dependency uniformly with other tasks. Figure 5 gives an example of two static tasks and three condition tasks (drawn in

diamond shape). Each condition task returns a random binary value to indicate the subsequent execution path. The expected number of condition tasks to execute before stop is eight (a probability of 1/8).

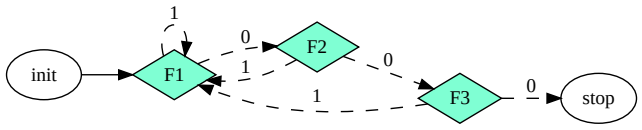


Figure 5: A dynamic control flow graph of three condition tasks each flipping a binary coin to decide the next path. The graph has 6 weak dependencies and 1 strong dependency.

```
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 4: Cpp-Taskflow code of Figure 5.

Listing 4 shows the Cpp-Taskflow code of Figure 5. Creating a condition task is similar to creating a static task, but returns an integer index of which successor task to execute. The index is defined with respect to the order of successors defined in a condition task. For instance, condition task `F3` precedes task `stop` and task `F1`. If the return value of `F3` is one, it loops back to task `F1`, or proceeds to task `stop` otherwise.

3.4.1 Strong Dependency versus Weak Dependency

Condition tasks are powerful for making rapid control-flow decisions across dependent tasks, but they are mistake-prone. The preceding link coming out of a condition task is defined as *weak dependency* (dashed lines in Figure 5), while other links are *strong dependency* (solid lines in Figure 5). When the executor receives a taskflow, the scheduler starts with tasks of zero dependencies (both weak and strong dependencies) and continues to execute successive tasks whenever strong dependencies are met. However, the scheduler skips this rule for a condition task and jumps directly to its successor indexed by the return value. It is users' responsibility to condition a control-flow graph correctly. Users must inspect graphs using our execution logic to infer if *task race* is possible, or ensure application algorithms properly disjoint conditionals. For instance, the control flow of Figure 5 can expand to a tree of tasks based on our execution logic. Each path from the root to a leaf represents a possible execution sequence, but none of them can overlap at the same time.

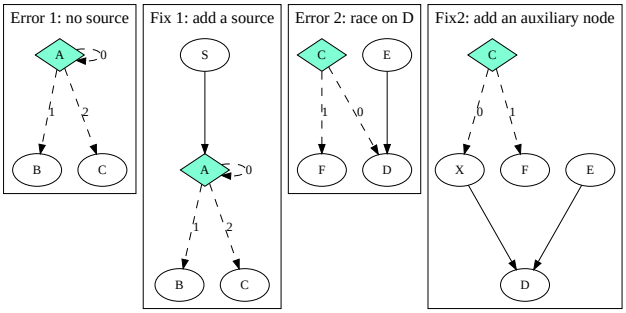


Figure 6: Common pitfalls of conditional tasking.

Figure 6 shows two common pitfalls of conditional tasking. The first example of three tasks has no source for the scheduler to start with. A simple fix is to add a task of zero dependencies. The second example may race on task D, if the conditional of C returns zero at the same time task 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. The second example may be feasible if E implies F. However, we do not recommend such a construct as it complicates debugging.

3.5 Heterogeneous Tasking

Cpp-Taskflow supports heterogeneous tasking for users to accelerate a wide range of computing programs by harnessing the power of accelerators Each task class of Cpp-Taskflow has a domain identifier indicating the target device (e.g., host, CUDA) to run a task. Different domain tasks are stored uniformly using a *variant* construct. Our runtime is able to distinguish task domains and allocate scheduling resources accordingly.

3.5.1 Concurrent CPU-GPU Tasking

We enable CPU-GPU collaborative computing using graph-based models. Developers describe a GPU workload as a task graph rather than a sequence of single operations. Our GPU tasking interface is referred to as *cudaFlow*. A cudaFlow is a task in the GPU domain. It defines a set of methods to construct a TDG of GPU operations such as data transfers and kernel offloading. A cudaFlow spawns a GPU task graph at its execution context for *stateful* parameter capture and offloads GPU operations to one or many GPUs. Figure 7 gives an example of the canonical saxpy workload using cudaFlow. The taskflow defines two static tasks, allocate_x and allocate_y, to allocate GPU memory, and one cudaFlow task to spawn a GPU TDG consisting of two host-to-device (H2D) transfers, one saxpy kernel, and two device-to-host (D2H) transfers.

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

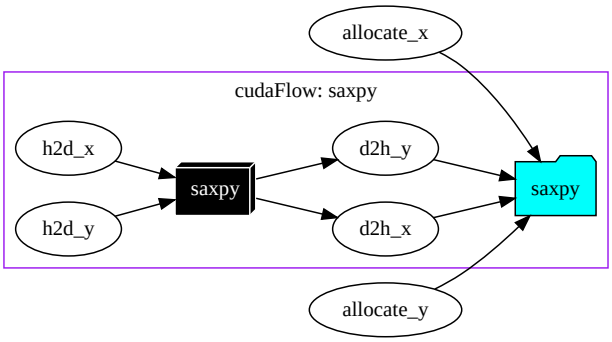


Figure 7: A saxpy ("single-precision A·X plus Y") task graph using two CPU tasks and one cudaFlow task consisting of one saxpy kernel and four data transfer tasks.

```
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(
  [&](){ cudaMalloc(&dx, N*sizeof(float));}
  [&](){ cudaMalloc(&dy, N*sizeof(float));}
);
auto cudaflow = taskflow.emplace(
  [&](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(
      (N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy
    );
    kernel.succeed(h2d_x, h2d_y)
          .precede(d2h_x, d2h_y);
  }
);
cudaflow.succeed(allocate_x, allocate_y);
```

Listing 5: Cpp-Taskflow code of Figure 7.

Listing 5 shows the Cpp-Taskflow code of Figure 7. The code is self-explanatory. The cudaFlow task takes an argument of type `tf::cudaFlow` created by the scheduler passing to the lambda that defines the GPU task graph. Users can capture parameters by reference to facilitate the decomposition of CPU-GPU dependent tasks. For example, `allocate_x` and `allocate_y` may overlap but they both precede the cudaFlow. Their changes on `dx` and `dy` are visible to the cudaFlow. Parameters to create a kernel task consist of the execution configuration (grid, block, and shared memory) and the kernel arguments. By default, all kernels are placed on the same device of the cudaFlow. Users decide which kernel goes to which device. An example is shown in Listing 6, with two kernels on GPU 0 and 1, respectively. We do not handle automatic GPU placement or live migration, which requires another layer of memory abstraction, but focus on distributing GPU tasks based on the given layout.

```
taskflow.emplace([&](tf::cudaFlow& cf) {
  cf.device(0);
  cf.kernel(grid, dim, 0, kernel0, data0);
  cf.kernel_on(1, grid, dim, 0, kernel1, data1);
});
```

Listing 6: Multi-GPU programming with cudaFlow.

3.5.2 Abstraction for Heterogeneous Accelerators

Three advantages inspire the design of our cudaFlow interface. The first advantage is *abstraction*. Users work at a suitable level of granularity for writing GPU operations that is commensurate with their domain knowledge. Our runtime sees the entire graph and performs platform-specific optimization. For example, we leverage CUDA graph to launch multiple dependent GPU operations using a single CPU call to reduce overheads [8]. The second advantage is *stateful execution*. Users describe GPU work with captured data to form a *stateful closure*. Other tasks can alter *referenced* data and make the change visible to the closure. Stateful execution improves not only the flexibility of our model, but also facilitates overlap of CPU and GPU tasks. The third advantage is *extensibility*. Users can extend the closure interface to each unique heterogeneous domain and define, at a minimum, methods for (1) offloading a kernel to a device, (2) allocating memory for buffers, and (3) transferring buffers to and from host memory. Our runtime is able to arrange proper scheduling resources for each domain task.

4 System Runtime and Scheduler

We leverage work stealing to design an efficient task executor that is generalizable to arbitrary heterogeneous domains.

4.1 Work Stealing Basics

Work stealing is a dynamic scheduling strategy for multi-threaded computer programs. It has been widely adopted in both commercial and open-source software [6, 38, 39]. A common practice spawns multiple worker threads where each worker iteratively drains out the tasks from its local queue and transitions to a *thief* to steal a task from a randomly selected peer called *victim* [17]. When a task completes, it submits new tasks from its immediate successors whenever dependencies are met [14]. The scheduler loops this procedure until the program terminates or no tasks are available. Algorithm 1 shows the canonical work-stealing loop proposed by Arora, Blumofe, and Plaxton (ABP for short) [17]. Each worker switches back and forth between an *active* worker that is executing a task and a *thief* that is attempting to steal a task. When multiple steals happen at the same task queue, only one may proceed. To mitigate the

Algorithm 1: ABP_worker_loop(w)

Input: w: current worker

```
1  t ← NIL;
2  while stop ≠ true do
3      if t ≠ NIL then
4          do
5              execute_task(w, t);
6              t ← w.task_queue.pop();
7          while t ≠ NIL;
8      end
9      yield();
10     v ← randomly_select_a_worker();
11     t ← v.task_queue.steal();
12 end
```

tension between aggressive thieves and wasted resources incurred by failed steals, ABP implements a *yielding* mechanism.

For accelerator tasks, a common solution is to encapsulate, for instance, a GPU operation in a CPU function and schedule it as a normal CPU task or dedicate it to a set of GPU workers [24, 25, 40]. It remains unclear which scheduler architecture performs best under which worker management strategy, and vice versa. Neither are they generalizable to arbitrary heterogeneous domains.

4.2 Heterogeneous Work Stealing Challenges

The biggest challenge for achieving efficient heterogeneous work stealing is worker management–*with an aim of optimal thread usage for executing a TDG*. During the execution of TDGs, a CPU task may submit both new CPU and GPU tasks and vice versa whenever dependencies are met. The available task parallelism changes dynamically, and there is no way to predict next coming tasks especially under dynamic control flow. The scheduler must effectively balance the number of working threads with dynamically generated tasks. Busy waiting on tasks with a *yielding* mechanism is a common framework to decrease the rate of oversubscribed steal attempts [17]. However, this approach is not cost-efficient, because it relies on the operating system (OS) to blindly decide which threads to relinquish the control over processors. Sleep-based mechanism is another way to suspend the workers frequently failing at 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 threads for running useful jobs. Reducing wasteful steals can improve the overall system performance, including latency, throughput, and energy efficiency to a large degree [23]. However, 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 difficult to program correctly. We need to handle various challenges arising out of concurrency controls, notification of workers, and heterogeneous coordination.

4.3 Architecture

At the architecture level, our scheduler maintains a separate execution domain for each task. We keep a set of workers per domain. A worker can only steal tasks of the same domain from others. We develop an efficient algorithm to *adapt* the number of workers to dynamically generated tasks. Our algorithm can effectively prevent threads from being underutilized and oversubscribed, thereby improving the overall system performance to a large degree. Figure 8 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 TDGs. 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 the corresponding slots without additional synchronization with other workers.

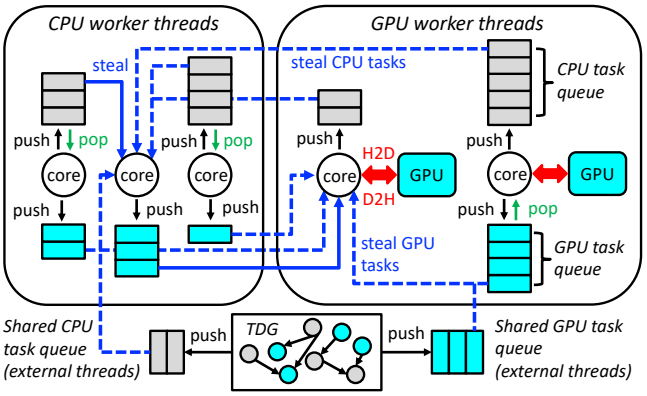


Figure 8: Architecture of our work-stealing scheduler on two domains, CPU and GPU.

We leverage two state-of-the-art 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 [35]. 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 [5]. 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. We develop one event notifier for each domain, based on Dekker's algorithm packaged in the Eigen library [4]. Details can be referred to [5].

4.4 Algorithm

At a high level, our algorithm keeps the per-domain invariant, *one worker is making steal attempts while an active worker exists, unless all workers are active*. We reach the goal through an adaptive strategy–*the last thief to become active will wake up a worker 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. Notice that our invariant is different from watchdog that always keeps one thread busy in looking for tasks to avoid false parallelism in sleep-based design [23]. In case of no task parallelism, the watchdog becomes waste. Our scheduler design is object-oriented. The scheduler lives in an *executor* object that manages a set of workers per domain and other tasking details.

Algorithm 2: worker_loop(w)

```
   Input: w: a worker
1  t ← NIL;
2  while true do
3  |  exploit_task(w, t);
4  |  if wait_for_task(w, t) == false then
5  |  |  break;
6  |  end
7  end
```

Our scheduling algorithm is symmetric by domain. Upon spawned, each worker enters the loop in Algorithm 2. 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 exploiting tasks and the number of workers that are making steal attempts, respectively. 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. Line 2:4 implements our adaptive strategy using two lightweight atomic operations. Notice that the order of these two comparisons matters, as they are used to synchronize with other workers in the later algorithms. Line 5:8

Algorithm 3: exploit_task(w, t)

```
   Input: w: a worker (domain d_w)
   Input: 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
```

drains 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 10).

Algorithm 4: execute_task(w, t)

```
   Input: w: a worker
   Input: t: a task
   Input: v: a visitor to the task variant
1  r ← task_callable_visitor(t, v);
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`. The key idea is to apply a visitor to the task variant (line 1). By default, the visitor defines separate algorithms for five task types, static task, dynamic task, module task, condition task, and cudaFlow task, and it can be extended to custom accelerator tasks. If the visitor returns a value, that is, a condition task, we directly submit the task of the indexed successor (line 2:5). Otherwise, we remove the task dependency from all immediate successors and submit new tasks of zero strong dependency (line 6:10). There are many framework-specific details we do not cover due to space limit. One important component is the detection of when a TDG completes, as condition tasks may trigger nested and non-deterministic cycles. We keep a local variable per worker to record the number of executed tasks and an atomic variable per TDG (including nested subflows) to track the number of submitted tasks. The TDG completes when the total task count executed balances

Algorithm 5: submit_task(w, t)

```
   Input: w: a worker (domain d_w)
   Input: 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
```

off the submission number. 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 is no active workers or thieves (line 3:5). The function `submit_task` is internal to the workers of an executor. External threads never touch this call.

When a worker completes all tasks in its local queue, it proceeds to `wait_for_task` (line 4 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 (line 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 (line 9:35). The predicate in 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 this worker*. Then, we inspect our predicate by examining the shared task queue again (line 10:21), 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 (line 11:12); if the steal succeeds and it is the last thief, the worker goes active and notifies a worker (line 13:17), or otherwise enters the steal loop again (line 19). Now, the worker is almost to sleep except if it is the last thief and: (1) an active worker in its domain exists (line 23:26) or (2) at least one task queue of the same domain from other workers is nonempty (line 27:32). 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 line 2:4 in Algorithm 3 and line 3:5 in Algorithm 5, and vice versa, preventing the problem of undetected task parallelism. Passing all the above conditions, the worker commits the request to wait on our predicate (line 34).

Algorithm 6: wait_for_task(w, t)

```
   Input: w: a worker (domain d_w)
   Input: 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 |  else
19 |  |  goto Line 2;
20 |  end
21 end
22 if AtomDec(thieves[d_w]) == 0 then
23 |  if actives[d_w] > 0 then
24 |  |  notifier[d_w].cancel_wait(w);
25 |  |  goto Line 1;
26 |  end
27 |  foreach worker x ∈ W do
28 |  |  if x.task_queue[d_w].empty() ≠ true then
29 |  |  |  notifier[d_w].cancel_wait(w);
30 |  |  |  goto Line 1;
31 |  |  end
32 |  end
33 end
34 notifier[d_w].commit_wait(w);
35 return true;
```

Algorithm 7 implements `explore_task`. At each iteration, the worker (thief) randomly selects a victim from *all* workers (line 4). If the victim is the worker itself, it steals a task of the same domain from the shared task queue (line 5), or from the victim (line 7). A steal may fail, when multiple workers contend for the same queue. We use two parameters, `MAX_STEALS` and `MAX_YIELDS`, to control the tension between how aggressively a thief steals a task, and when it *yields* system resources to others (line 12:17). The motiva-

Algorithm 7: explore_task(w, t)

Input: w: a worker (a thief in domain d_w)
Input: t: an empty task

```
1  steals ← 0;
2  yields ← 0;
3  while true do
4      if v ← random_worker(); v == w then
5          t ← task_queue[d_w].steal();
6      else
7          t ← v.task_queue[d_w].steal();
8      end
9      if t ≠ NIL then
10         break;
11     else
12         if ++steals ≥ MAX_STEALS then
13             yield();
14             if ++yields == MAX_YIELDS then
15                 break;
16             end
17         end
18     end
19 end
```

tion is to steal aggressively in the first few iterations avoiding losing cores to other concurrent programs. In our experiments, setting `MAX_STEALS` to twice the number of workers and `MAX_YIELDS` to 100 produces decent and stable performance.

Algorithm 8: submit_graph(g)

Input: g: a TDG 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
```

Up to this time, we have discussed the core work-stealing algorithm. To submit a TDG for execution, we call `submit_graph`, shown in Algorithm 8. The caller thread inserts each task of zero dependencies to the shared task queues and notifies a worker of the corresponding domain (line 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 line 4:5 synchronizes with line 10:21 of Algorithm 6 and vice versa, preventing undetected parallelism in which all workers are sleeping.

Theorem 1. *Our work-stealing algorithm can correctly complete the execution of a TDG.*

Proof. Proving the correctness of our algorithm is equivalent to showing that undetected task parallelism or leaky tasks are not possible in our work stealing. There are two places 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 be visible to a worker in the same domain of the task for two situations: (1) if a worker has prepared or committed to wait (line 9:34 in Algorithm 6), it will be notified; (2) otherwise, at least one worker will eventually go through line 9:21 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 22:33 of Algorithm 6 to steal this task. Similarly, the latter case is not possible under our 2PC, as it contradicts the guarding scan in line 9:35 of Algorithm 6. □

5 Experimental Results

We evaluate the performance of Cpp-Taskflow on two fronts: micro-benchmark and realistic workloads, including machine learning and VLSI design automation. The former is to study tasking performance without much bias of application algorithms, while the latter demonstrates the strength of our system in real use cases. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz, 4 GeForce RTX 2080 GPUs, and 256 GB RAM. We compiled all programs using Nvidia CUDA nvcc 10.1 on a host compiler of GNU GCC-8.3.0 with C++14 standards `-std=c++14` and optimization flags `-O2` enabled. Each run of N CPU cores and M GPUs corresponds to N CPU and M GPU worker threads. All data is an average of ten runs.

5.1 Baseline

We consider oneTBB, StarPU, HPX, and OpenMP for the baseline because of their extensive experience with users. More importantly, their comprehensive documentations allow us to conduct fair performance study without making mistakes due to undocumented pitfalls. Each baseline represents a specific programming paradigm. oneTBB (ver-

sion 2020 U2) is an industrial-strength parallel programming system under Intel oneAPI [6]. 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 TDGs. StarPU (version 1.3) is a CPU-GPU task programming system widely used in the high-performance computing (HPC) community [18]. It provides a C-based syntax for writing TDGs 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 [32]. It supports implicit task graph programming through aggregating *future* objects in a dataflow API. OpenMP (version 4.5 in GCC toolchains) is a directive-based programming framework for handling loop parallelism [11]. It supports only *static* graph encoding using task dependency clauses. A rather common approach is instead to levelize the graph and propagate computations level by level.

5.2 Micro-benchmark

We randomly generate a set of DAGs with equal distribution of CPU and GPU tasks. Each task performs a simple vector addition of 1K elements. For fair purposes, we use CUDA Graph for all baselines. Table 1 summarizes the programming effort of each method, measured by SLOCCount [13] and SCC [12]. Cpp-Taskflow requires the least amount of lines of code (LOC) and written tokens. The cyclomatic complexity measured at a single function and the whole program is also the smallest. It is important to investigate the overhead of a task graph for finding the best granularity of an algorithm. As shown in Table 2, the static size of a task, compiled on our platform, is 272, 136, and 1472 bytes for Cpp-Taskflow, oneTBB, and StarPU, respectively. We do not report the data of HPX and OpenMP because they do not have explicit task constructs at the functional level. The time it takes for Cpp-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 TDGs. The overhead always occupies 5-10% of the total execution time regardless of the TDG size.

Figure 9 shows the overall performance comparison between Cpp-Taskflow and the baseline at different TDG sizes. In terms of runtime, Cpp-Taskflow outperforms others across most data points. We complete the largest TDG by 1.61×, 1.44×, 1,53×, and 1.40× faster than oneTBB, StarPU, HPX, and OpenMP, respectively. The memory footprint of Cpp-Taskflow is higher than other methods, about 106–174 MB more at the largest TDG, because our scheduler associates each worker with a separate task queue per domain. The benefit of such cost is significant speed gain and better worker management that also improves other performance aspects.

Table 1: Programming Effort on Micro-benchmark

Method	LOC	#Tokens	CC	WCC	#People	Cost
Cpp-Taskflow	69	650	6	8	0.14	$1630
oneTBB	182	1854	8	15	0.27	$4515
StarPU	253	2216	8	21	0.34	$6354
HPX	255	2264	10	24	0.34	$6433
OpenMP	182	1896	13	19	0.27	$4515

CC: maximum cyclomatic complexity in a single function
WCC: weighted cyclomatic complexity of the program
People: estimated number of developers required
Cost: estimated cost to develop

Table 2: Overhead of Task Graph Creation

Method	S_{task}	T_{task}	T_{edge}	$\rho_{<10}$	$\rho_{<5}$	$\rho_{<1}$
Cpp-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%

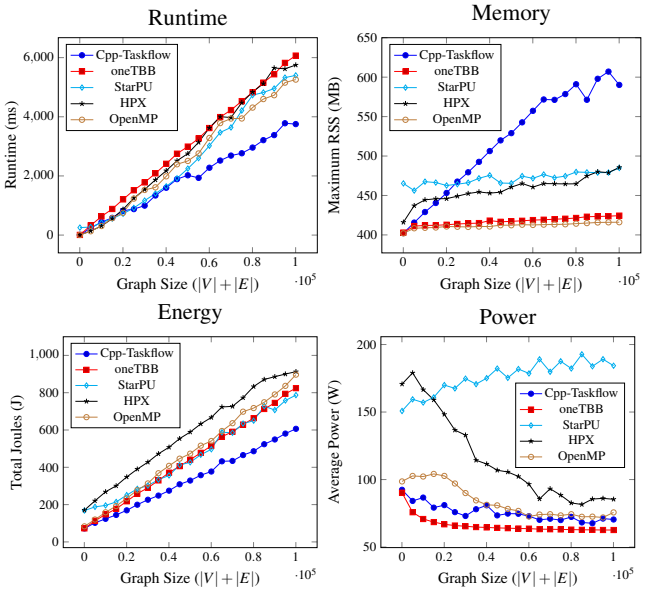


Figure 9: Overall system performance at different problem sizes using 40 CPUs and 4 GPUs.

We use the Linux `perf` tool to measure the power consumption of all cores plus LLC (`power/energy-pkg/`). The total joules consumed by Cpp-Taskflow is consistently smaller than the others, due to our adaptive worker control. Cpp-Taskflow, oneTBB, and OpenMP are more power-efficient than HPX and StarPU. The margin between Cpp-Taskflow and StarPU continues to increase as we enlarge the TDG size.

Figure 10 displays the runtime distribution of each method over a hundred runs of two TDGs. The boxplot shows the

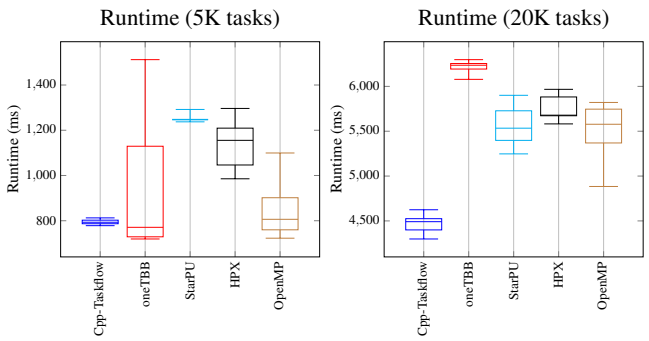


Figure 10: Runtime distribution of two task graphs.

runtime of Cpp-Taskflow is more consistent than others. We observed oneTBB has a wide variation in the small TDG. We attribute this to the lack of a dedicated scheduling strategy for GPU tasks, which leads to unbalanced parallelism and unpredictable execution delay. Similar problems exist in OpenMP as well.

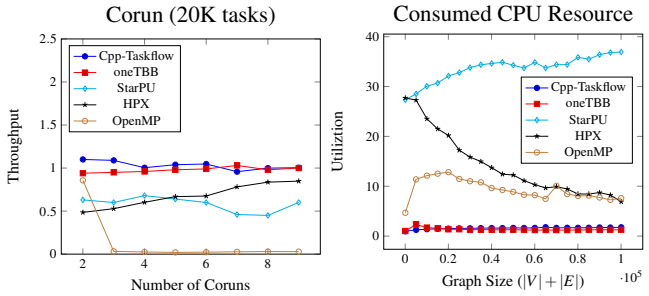


Figure 11: Throughput of corunning task graphs and CPU utilization at different problem sizes under 40 CPUs and 4 GPUs.

We next compare the throughput of each method on corunning TDGs. This experiment emulates a server-like environment where multiple client programs run concurrently on the same machine to compete for resources. The effect of worker management propagates to all simultaneous parallel processes. We consider up to nine corun processes each executing the same TDG of 20K tasks. 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 11 compares the throughput of each method and relates the result to the CPU utilization. Both Cpp-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 clearly see both Cpp-Taskflow and oneTBB make effective use of CPU resources to schedule tasks. However, StarPU keeps threads busy most of the time and has little adaptive control to balance thread resources with dynamic task parallelism.

5.3 Large Sparse Neural Network Inference

We applied Cpp-Taskflow to solve the Large Sparse Deep Neural Network (LSDNN) Inference Challenge, a recent effort aimed at driving progress in sparse AI analytics [34]. Each dataset is comprised of a sparse matrix Y, containing the input data for the network, 1920 layers of neurons stored in sparse matrices Ws, truth categories, and the bias values used for the inference. Preloading the network to the GPU is impossible, and thus we implemented a task-based decomposition algorithm inspired by [20,30]. A partial TDG is shown in Figure 12. We created up to 4 cudaFlows on 4 GPUs. Each cudaFlow contains thousands of GPU operations to run partitioned matrices in a data dispatching loop. Each condition task is CPU-heavy in partitioning Y and scoring results. We consider oneTBB and StarPU for the baseline, both of which support explicit task constructs. Since they have no support for conditional tasking, we unroll their TDGs across fixed-length iterations found in hindsight.

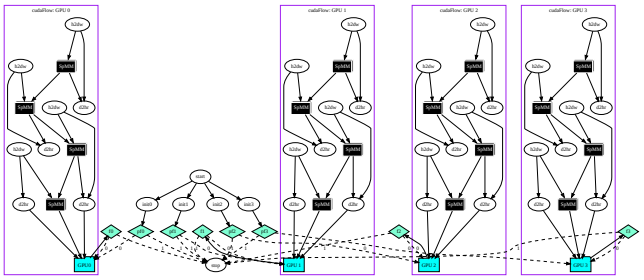


Figure 12: A partial TDG of 4 cudaFlows, 6 static tasks, and 8 conditioned cycles for the inference workload.

Table 3: Programming Effort on LSDNN Inference

Method	LOC	#Tokens	CC	WCC	#People	Cost
Cpp-Taskflow	281	1663	5	17	0.36	$7150
oneTBB	433	2200	10	22	0.41	$8917
StarPU	467	2845	12	25	0.50	$12171

CC: maximum cyclomatic complexity in a single function
WCC: weighted cyclomatic complexity of the program
People: estimated number of developers required
Cost: estimated cost to develop

Table 3 compares the programming effort between the three implementations. Cpp-Taskflow has the least amount of coding effort and program complexity, due to our cudaFlow interface and condition tasks. Figure 13 compares the performance of solving a LSDNN of 1920 layers each of 4096 neurons under different CPU and GPU numbers. Cpp-Taskflow outperforms oneTBB and StarPU in all aspects.

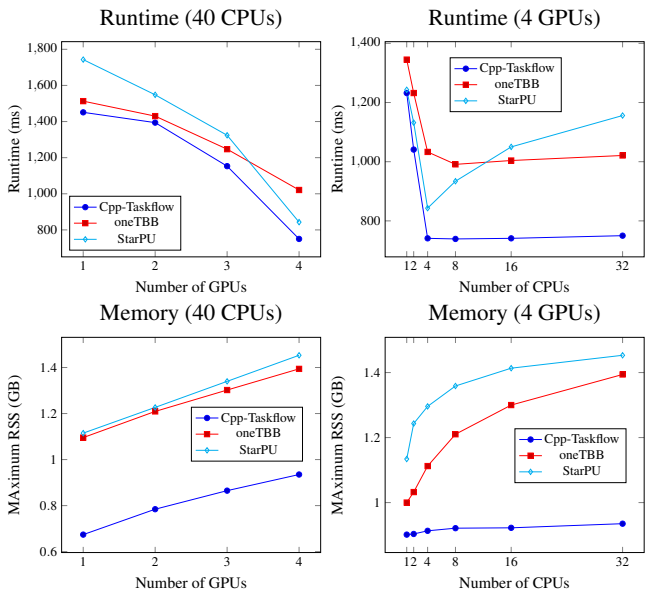


Figure 13: Runtime and memory data of the LSDNN (1920 layers, 4096 neurons per layer) under different CPU and GPU numbers

Both our runtime and memory scale better regardless of CPU and GPU numbers. Using 4 GPUs, when performance saturates at 4 CPUs, we do not suffer from further runtime growth as StarPU, due to our adaptive work stealing. Our memory usage is 1.5× and 1.6× fewer than oneTBB and StarPU, respectively. This highlights the benefit of our condition task, which encodes control-flow decisions directly in a cyclic TDG rather than unrolling it statically across iterations.

5.4 VLSI Placement

We applied Cpp-Taskflow to solve a VLSI placement problem. The goal is to determine the physical locations of cells (logic gates) in a fixed layout region using minimal interconnect wirelength. Figure 14 shows a placement layout of an industrial design [45]. Modern placement typically incorporates hundreds of millions of cells and takes several hours to finish [43]. To reduce the long runtime, recent work started investigating new CPU-GPU algorithms. We consider a matching-based hybrid CPU-GPU placement refinement algorithm in DREAMPlace [43, 44], that iterates the following (see Figure 15): (1) a GPU-based maximal independent set algorithm to identify cell candidates, (2) a CPU-based partition algorithm to cluster adjacent cells, and (3) a CPU-based bipartite matching algorithm to find the best permutation of cell locations. Each iteration contains overlapped CPU and GPU tasks with nested conditions to decide the convergence. Figure 16 shows a partial TDG of one iteration.

We implemented the hybrid CPU-GPU placement algorithm using Cpp-Taskflow, oneTBB, and StarPU. The algorithm is crafted on one GPU and many CPUs. Since oneTBB and StarPU have no support for nested conditions, we un-

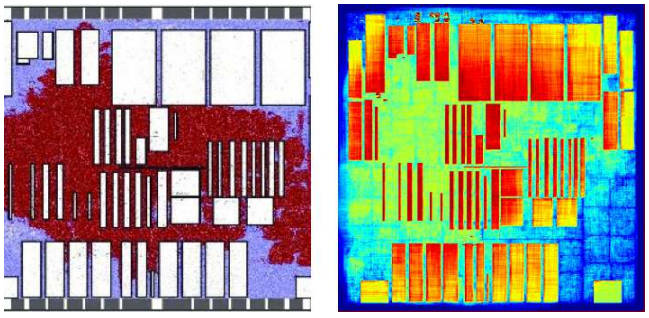


Figure 14: A placement layout and congestion map of the industrial circuit, adaptec1 (211K cells and 221K nets) [45].

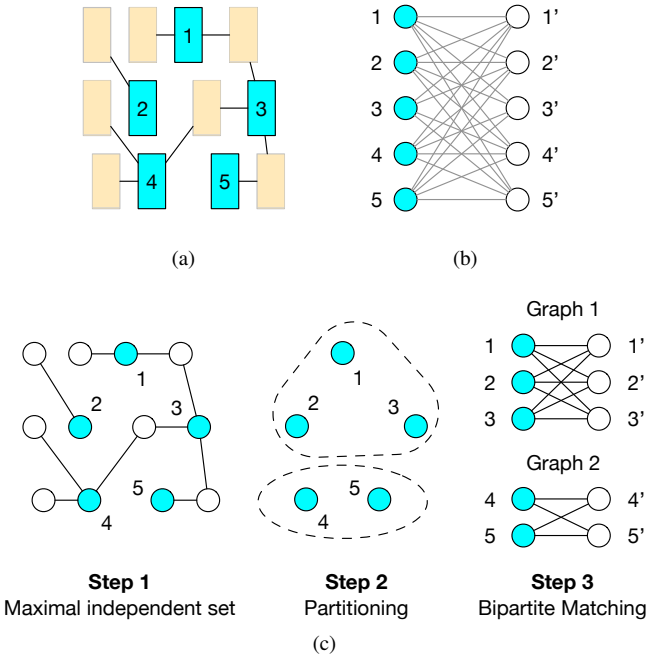


Figure 15: An iterative matching-based placement algorithm [43].

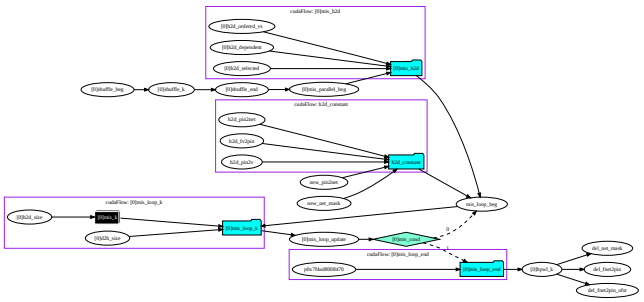


Figure 16: A partial TDG of 4 cudaFlows, 1 conditioned cycle, and 12 static tasks for one iteration of Figure 15.

Table 4: Programming Effort on VLSI Placement

Method	LOC	#Tokens	CC	WCC	#People	Cost
Cpp-Taskflow	677	4180	20	53	0.57	$15054
oneTBB	1000	6415	33	78	0.72	$21736
StarPU	1279	8136	41	90	0.87	$29686

CC: maximum cyclomatic complexity in a single function
WCC: weighted cyclomatic complexity of the program
People: estimated number of developers required
Cost: estimated cost to develop

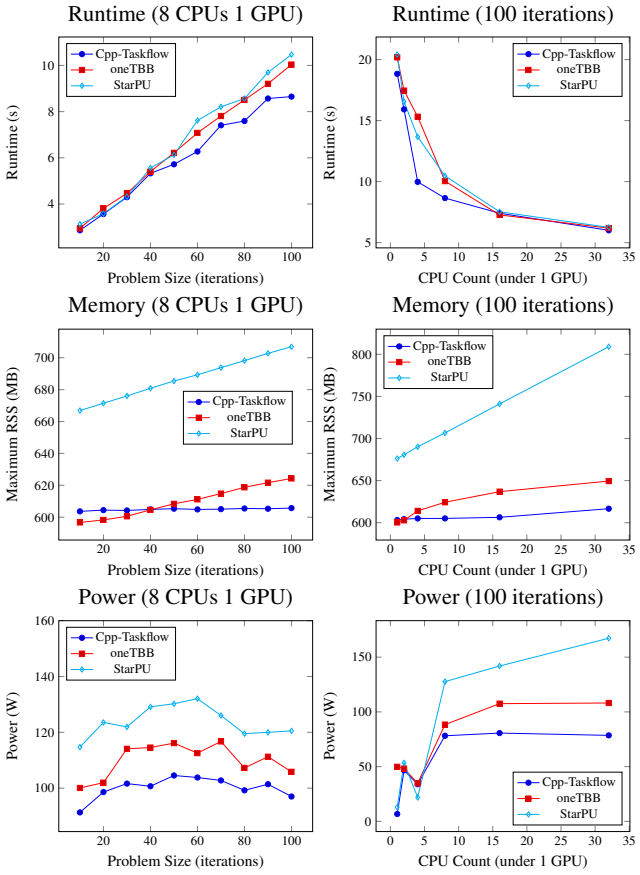


Figure 17: Runtime, memory, and power data of the circuit adaptec1 (211K cells and 221K nets).

roll their TDGs across fixed-length iterations found in hindsight. Table 4 lists the programming effort of each method. Cpp-Taskflow outperforms oneTBB and StarPU in all aspects. The whole program is 1.5× and 1.7× less complex than that of oneTBB and StarPU, respectively. The overall performance is shown in Figure 17. Using 8 CPUs and 1 GPU, Cpp-Taskflow is consistently faster than others across all problem sizes (placement iterations). The gap becomes clear at large problem size; at 100 iterations, Cpp-Taskflow is 17% faster than oneTBB and StarPU. We observed similar results using other CPU numbers. Performance saturates at about 16 CPUs, primarily due to the inherent irregularity of

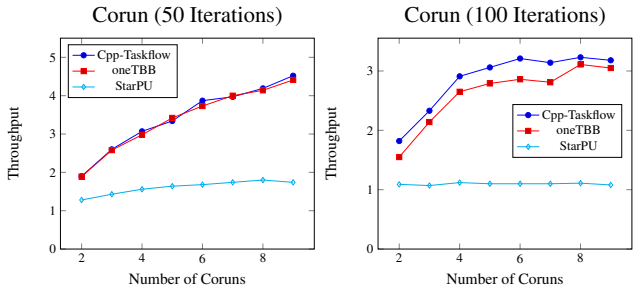


Figure 18: Throughput of corunning placement workloads on two problem sizes using 40 CPUs and 1 GPU.

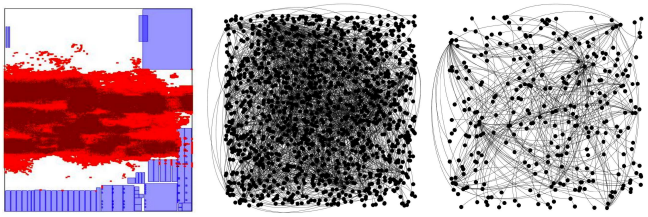
the algorithm (see Figure 16). The memory footprint shows the benefit of our conditional tasking. We keep nearly no growth of memory when the problem size increases, whereas StarPU and oneTBB grow linearly due to unrolled TDGs. On a vertical scale, increasing the number of CPUs bumps up the memory usage of all methods, but the growth rate of Cpp-Taskflow is much slower than the others. In terms of energy, our scheduler is very power-efficient in completing the placement workload, regardless of problem sizes and CPU numbers. Beyond 16 CPUs where performance saturates, our system does not suffer from increasing power as StarPU, due to our adaptive work stealing.

For irregular TDGs akin to Figure 16, resource utilization is critical to the system throughput. We corun the same program by up to nine processes that compete for 40 CPUs and 1 GPU. Corunning a CAD program is very common for searching the best parameters for an algorithm. Figure 18 plots the throughput across nine coruns at two problem sizes. Both Cpp-Taskflow and oneTBB achieve higher throughput than StarPU. At the largest problem size, Cpp-Taskflow outperforms oneTBB and StarPU across all coruns. The result again highlights the strength of our scheduler, which always adapts the workers to available task parallelism.

5.5 VLSI Timing Analysis

We demonstrate the performance of Cpp-Taskflow in a real-world VLSI timing analyzer. Efficient parallel timing analysis is extremely challenging to design and implement, due to large irregularity and graph-oriented computing. Figure 19 shows a timing analysis graph on an industrial design of 2M gates [26]. We consider our research project *OpenTimer*, an open-source static timing analyzer that has been used in many industrial and academic projects [28]. The first release v1 in 2015 implemented the *pipeline-based levelization* algorithm using the OpenMP 4.5 task dependency clause [28]. To overcome the performance bottleneck, we rewrote the core incremental timing engine using Cpp-Taskflow in the recent release v2. Since OpenTimer is a large project of more than 50K lines of code, it is difficult to rewrite the code

with other programming frameworks. We focus on comparing with OpenMP which had been available in v1.



(a) Circuit (1.01mm^2) (b) Graph (3M gates) (c) A signal path

Figure 19: The timing graph of an industrial design [26].

Table 5: Software Cost of OpenTimer v1 and v2

Tool	Task Model	LOC	CC	Effort	Dev	Cost
v1	OpenMP 4.5	9,123	58	2.04	2.90	$275,287
v2	Cpp-Taskflow	4,482	20	0.97	1.83	$130,523

CC: maximum cyclomatic complexity in a single function
Effort: development effort estimate, person-years
Dev: estimated average number of developers
Cost: total estimated cost to develop

Table 5 measures the software costs of two OpenTimer versions using the Linux tool SLOCCount under the organic mode [13]. In OpenTimer v2, a large amount of exhaustive OpenMP dependency clauses that were used to carry out task dependencies are now replaced with only a few lines of flexible Cpp-Taskflow code (9123 vs 4482). The maximum cyclomatic complexity in a single function is reduced from 58 to 20. We attribute this to Cpp-Taskflow's programmability, which can affect the way developers design efficient algorithms and parallel decomposition strategies. For example, OpenTimer v1 relied on a bucket-list data structure to model the task dependency in a pipeline fashion using OpenMP. We found it very difficult to go beyond this paradigm because of the insufficient support for dynamic dependencies in OpenMP. With Cpp-Taskflow in place, we can break this bottleneck and easily model both static and dynamic task dependencies at programming time and runtime. The TDG flows computations naturally and asynchronously with the timing graph, producing faster runtime performance. Figure 20 shows a TDG used to carry out one iteration of incremental timing on a sample circuit.

Figure 21 compares the performance between OpenTimer v1 and v2. We evaluated the runtime versus incremental iterations under 16 CPUs on two industrial circuit designs tv80 (5.3K gates and 5.3K nets) and vga_lcd (139.5K gates and 139.6K nets) with 45nm NanGate cell library [26]. Each incremental iteration refers a design modification followed by a timing query to trigger a timing update. In v1, this includes the time to reconstruct the data structure required by OpenMP to alter the task dependencies. In v2, this includes

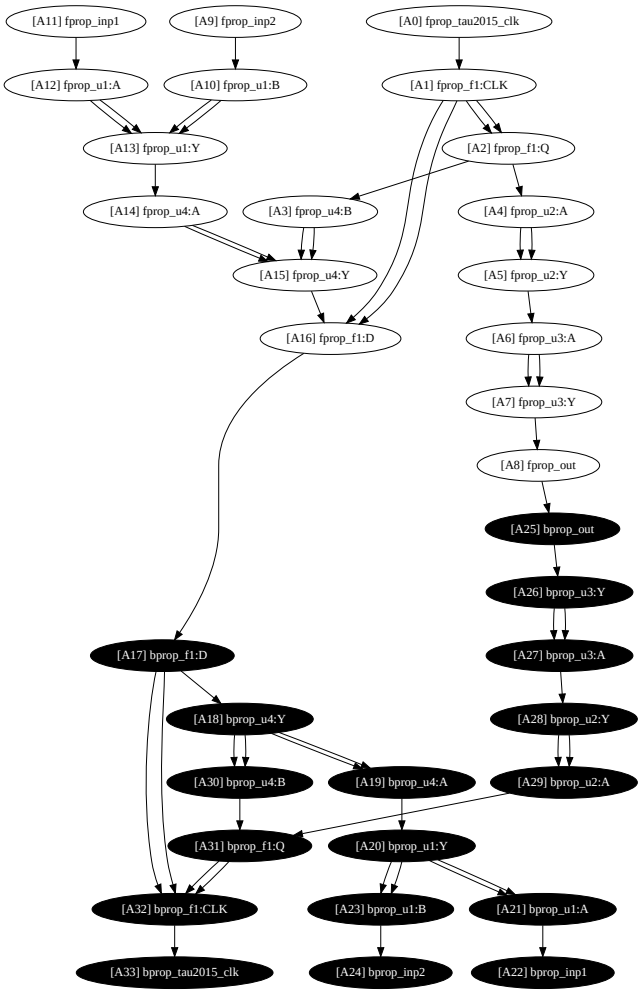


Figure 20: A TDG to carry out an iteration of timing update. The graph consists of forward timing propagation tasks (in white) and backward timing propagation tasks (in black).

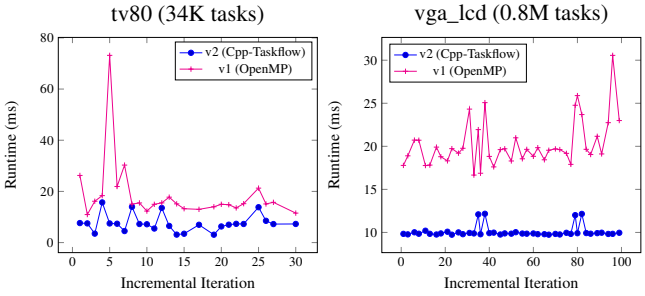


Figure 21: Runtime comparisons of the incremental timing between OpenTimer v1 (OpenMP) and v2 (Cpp-Taskflow) for two circuits, tv80 (34K tasks) and vga_lcd (0.8M tasks), under 16 CPUs.

the time to create and launch a new TDG to perform a parallel timing update. As shown in Figure 21, v2 is consis-

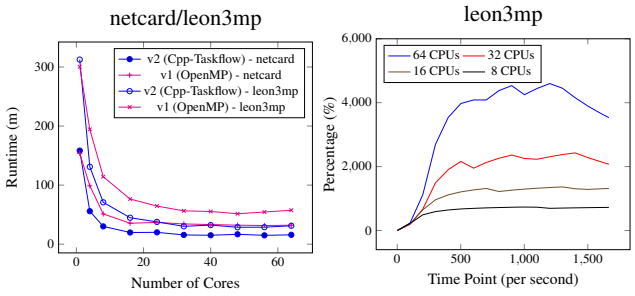


Figure 22: Scalability and CPU profile of Cpp-Taskflow on two large circuits, netcard (8M tasks) and leon3mp (6.7M tasks).

tently faster than v1. The maximum speed-up is 9.8× on tv80 and 3.1× on vga_lcd. This also demonstrates the performance of Cpp-Taskflow on batch jobs each consisting of a different task pattern (average speed-up is 2.9× on tv80 and 2.0× on vga_lcd). The fluctuation of the curve is caused by design modifiers; some are local changes and others affect the entire timing landscape giving rise to large TDGs. The scalability of Cpp-Taskflow is shown in Figure 22. We used two million-scale designs, netcard (1.4M gates) and leon3mp (1.2M gates) from the OpenCores [26], to evaluate the runtime of v1 and v2 across different number of CPUs. There are two important observations. First, v2 is slightly slower than v1 at one CPU (3-4%), where all OpenMP's constructs are disabled. This shows the graph overhead of Cpp-Taskflow; yet it is negligible. Second, v2 is consistently faster than v1 regardless of CPU counts except for one. This highlights that Cpp-Taskflow's programming model largely improves the design of a parallel VLSI timing engine that would otherwise not be possible with OpenMP.

6 Related Work

Heterogeneous programming system is one of the main driving force to advance scientific computing [50]. Directive-based programming models such as OpenMP [11], OmpSs [9], OpenMPC [37], OpenACC [10], and X-Kaapi [25], 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 compute patterns [36]. Functional approaches such as StarPU [18], PaRSEC [21], oneTBB [6], HPX [32], QUARK [52], XKAAPI++ [40], Fastflow [16], Charm++ [33], and Kokkos [24] 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 own pros and cons and dominates certain application domains.

CPU-GPU co-scheduling plays a key role in heterogeneous programming systems. Work stealing is a popular strategy to reduce the complexity of load balancing [17,40]. It has inspired many systems such as Cilk++ [39], X10 [49], Nabbit [14], oneTBB [6], TPL [38], and Java runtime [7]. Most progress were made for manycore CPU architectures. An efficient counterpart for hybrid CPU-GPU or more general heterogeneous systems remains demanding. A key challenge is the worker management. Instead of keeping all workers busy most of the time such as ABP [17] and StarPU [18], both oneTBB [6] 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. Other approaches such as A-Steal [15] targeting at a space-sharing environment and HERMES [47] tuning hardware frequency scaling have improved certain performance aspects of work stealing in the CPU domain. However, how to migrate the above approaches to a heterogeneous target remains unknown.

7 Acknowledgment

We gratefully acknowledge contributions from users of Cpp-Taskflow and members of the wider scientific computing community.

8 Conclusion

In this paper, we have introduced Cpp-Taskflow, a general-purpose task programming system to streamline the creation of large and complex heterogeneous programs. Our programming model enables developers to incorporate a broad range of computational patterns with relative ease of programming. We have developed an efficient runtime of heterogeneous work stealing that is generalizable to arbitrary domains. Experimental results have demonstrated promising performance of Cpp-Taskflow over existing systems. As an example, we have solved a large-scale VLSI placement problem by up to 17% faster, 1.3× fewer memory, 2.1× less power consumption, and 2.9× higher throughput using 1.9× fewer lines of code than two industrial-strength systems, oneTBB and StarPU, on a machine of 40 CPUs and 4 GPUs.

Cpp-Taskflow is a work in progress, and we are committed to support trustworthy development for both academic and industrial research projects using parallel computing. Our experience with Cpp-Taskflow is encouraging. Many scientific software developers are using Cpp-Taskflow in both prototype and production, and Cpp-Taskflow is helping our research colleagues to make new advances in scientific computing, including VLSI design and machine learning. Cpp-Taskflow is open in GitHub [1].

References

[1] Cpp-Taskflow. URL: http://github.com/cpp-taskflow.

[2] DARPA Intelligent Design of Electronic Assets (IDEA). URL: https://www.darpa.mil/program/intelligent-design-of-electronic-assets.

[3] DARPA Microsystems Exploration: Performant Automation of Parallel Program Assembly (PAPPA) Program.

[4] Eigen. URL: http://eigen.tuxfamily.org/index.php.

[5] Event Count: Two-phase Commit Protocol. URL: http://www.1024cores.net/home/lock-free-algorithms/eventcounts.

[6] Intel oneTBB. URL: https://github.com/oneapi-src/oneTBB.

[7] Java Fork/Join Framework. URL: https://www.java.com/en/.

[8] Nvidia CUDA Graph. URL: https://devblogs.nvidia.com/cuda-10-features-revealed/.

[9] OmpSs. URL: https://pm.bsc.es/ompss.

[10] OpenACC. URL: http://www.openacc-standard.org.

[11] OpenMP. URL: https://www.openmp.org/.

[12] SCC. URL: https://github.com/boyter/scc.

[13] SLOCCount. URL: https://dwheeler.com/sloccount/.

[14] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IEEE IPDPS*, pages 1–12, 2010. URL: https://ieeexplore.ieee.org/document/5470403.

[15] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *PPoPP*, pages 112–120, 2007. URL: 10.1145/1229428.1229448.

[16] 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. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13.

[17] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *ACM SPAA*, pages 119–129, 1998. URL: https://doi.org/10.1145/277651.277678.

[18] 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. URL: https://dl.acm.org/doi/10.1002/cpe.1631.

[19] R. Iris Bahar, Alex K. Jones, Srinivas Katkoori, Patrick H. Madden, Diana Marculescu, and Igor L. Markov. Workshops on Extreme Scale Design Automation (ESDA) Challenges and Opportunities for 2025 and Beyond. 2014. URL: https://cra.org/ccc/visioning/visioning-activities/2014-activities/extreme-scale-design-automation/.

[20] M. Bisson and M. Fatica. A GPU Implementation of the Sparse Deep Neural Network Graph Challenge. In *IEEE HPEC*, pages 1–8, 2019. URL: https://ieeexplore.ieee.org/document/8916223.

[21] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6):36–45, 2013. URL: https://doi.org/10.1109/MCSE.2013.98.

[22] B. Catanzaro, K. Keutzer, and Bor-Yiing Su. Parallelizing CAD: A timely research agenda for EDA. In *ACM/IEEE DAC*, pages 12–17, 2008. URL: https://ieeexplore.ieee.org/document/4555773.

[23] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. BWS: Balanced Work Stealing for Time-sharing Multicores. In *ACM EuroSys*, pages 365–378, 2012. URL: https://dl.acm.org/doi/pdf/10.1145/2168836.2168873.

[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. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[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. URL: https://ieeexplore.ieee.org/document/6569905.

[26] J. Hu, G. Schaeffer, and V. Garg. TAU 2015 Contest on Incremental Timing Analysis. In *IEEE/ACM ICCAD*, pages 895–902, 2015. URL: https://ieeexplore.ieee.org/document/7372664.

[27] 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. URL: https://ieeexplore.ieee.org/document/8821011/.

[28] Tsung-Wei Huang and Martin Wong. OpenTimer: A high-performance timing analysis tool. In *IEEE/ACM ICCAD*, pages 895–902, 2015. URL: https://ieeexplore.ieee.org/document/7372666.

[29] Tsung-Wei Huang and Martin Wong. UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR. *IEEE TCAD*, 35(11):1862–1875, 2016.

[30] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NIPS*, 2018. URL: https://arxiv.org/abs/1811.06965.

[31] Andrew B. Kahng. Reducing Time and Effort in IC Implementation: A Roadmap of Challenges and Solutions. In *IEEE/ACM DAC*, 2018. URL: https://doi.org/10.1145/3195970.3199854.

[32] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. PGAS, pages 6:1–6:11, 2014. URL: https://dl.acm.org/doi/10.1145/2676870.2676883.

[33] 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. URL: https://doi.org/10.1145/165854.165874.

[34] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. Sparse deep neural network graph challenge. *IEEE HPEC*, 2019. URL: http://dx.doi.org/10.1109/HPEC.2019.8916336.

[35] 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. URL: https://dl.acm.org/doi/10.1145/2442516.2442524.

[36] 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. URL: https://ieeexplore.ieee.org/document/6468490.

[37] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *IEEE/ACM SC*, pages 1–11, 2010. URL: https://ieeexplore.ieee.org/abstract/document/5644879.

[38] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. In *ACM OOPSLA*, pages 227–241, 2009. URL: https://dl.acm.org/doi/10.1145/1639949.1640106.

[39] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010. URL: https://doi.org/10.1007/s11227-010-0405-3.

[40] João V. F. Lima, Thierry Gautier, Vincent Danjean, Bruno Raffin, and Nicolas Maillard. Design and Analysis of Scheduling Strategies for multi-CPU and multi-GPU Architectures. *Parallel Comput.*, 44:37–52, 2015. URL: https://doi.org/10.1016/j.parco.2015.03.001.

[41] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. A Modern C++ Parallel Task Programming Library. In *ACM MM*, page 2284–2287, 2019. URL: https://doi.org/10.1145/3343031.3350537.

[42] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. An Efficient and Composable Parallel Task Programming Library. In *IEEE HPEC*, pages 1–7, 2019. URL: https://ieeexplore.ieee.org/document/8916447.

[43] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement. In *ACM/IEEE DAC*, pages 117:1–117:6, 2019. URL: https://doi.org/10.1145/3316781.3317803.

[44] Yibo Lin, Wuxi Li, Jiaqi Gu, Haoxing Ren, Brucek Khailany, and David Z. Pan. ABCDPlace: Accelerated Batch-based Concurrent Detailed Placement on Multi-threaded CPUs and GPUs. *IEEE TCAD*, 2020. URL: https://doi.org/10.1109/TCAD.2020.2971531.

[45] J. Lu, H. Zhuang, P. Chen, H. Chang, C. Chang, Y. Wong, L. Sha, D. Huang, Y. Luo, C. Teng, and

C. Cheng. ePlace-MS: Electrostatics-Based Placement for Mixed-Size Circuits. *IEEE TCAD*, 34(5):685–698, 2015. URL: https://ieeexplore.ieee.org/document/7008518.

[46] Yi-Shan Lu and Keshav Pingali. Can Parallel Programming Revolutionize EDA Tools? *Advanced Logic Synthesis*, 2018. URL: https://link.springer.com/chapter/10.1007/978-3-319-67295-3_2.

[47] Haris Ribic and Yu David Liu. Energy-efficient Work-stealing Language Runtimes. In *ACM ASPLOS*, pages 513–528, 2014. URL: https://dl.acm.org/doi/10.1145/2541940.2541971.

[48] Leon Stok. Developing Parallel EDA Tools [The Last Byte]. *IEEE Design Test*, 30(1):65–66, 2013. URL: https://ieeexplore.ieee.org/document/6524125.

[49] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. *SIGPLAN*, 47(8):267–276, 2012. URL: https://doi.org/10.1145/2145816.2145850.

[50] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. 2018. URL: https://doi.org/10.2172/1473756.

[51] 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. URL: https://ieeexplore.ieee.org/document/8942060.

[52] Asim Yarkhan. Dynamic Task Execution on Shared and Distributed Memory Architectures. *PhD thesis*, 2012. URL: https://trace.tennessee.edu/utk_graddiss/1575/.