Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System

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

***Abstract*—Taskflow tackles the long-standing question: How can we make it easier for developers to program parallel and heterogeneous computer-aided design (CAD) applications with high performance and simultaneous high productivity? Taskflow introduces a new powerful task graph programming model to assist developers in the implementation of parallel and heterogeneous algorithms with complex control flow. We develop an efficient system runtime to solve many of the new scheduling challenges arising out of our models and optimize the performance across latency, energy efficiency, and throughput. Taskflow has demonstrated promising performance on both micro-benchmarks and real-world applications. As an example, Taskflow solved a large-scale circuit placement problem up to 17% faster, with 1.3× fewer memory, 2.1× less power consumption, and 2.9× higher throughput than two industrial-strength systems, oneTBB and StarPU, on a machine of 40 CPUs and 4 GPUs.**

***Index Terms*—Electronic design automation, parallel programming.**

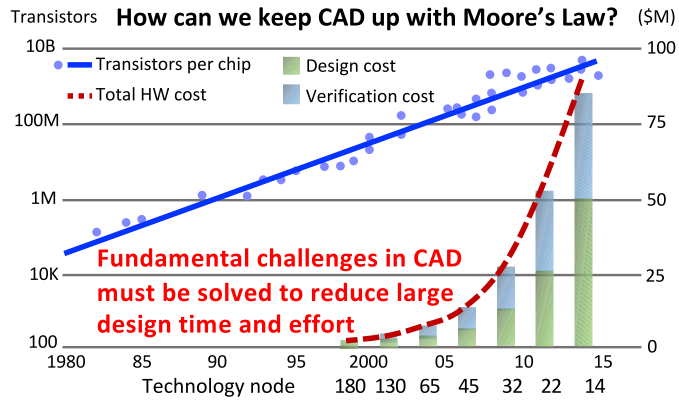

Fig. 1. Design cost versus design complexity [3].

I. Introduction

THE EVER-INCREASING design complexity in very-large-scale integration (VLSI) implementation will soon far exceed what many existing computer-aided design (CAD) tools are able to scale with reasonable design time and effort (see Fig. 1). A key fundamental challenge is that CAD must incorporate *new parallel paradigms* comprising many-core central processing units (CPUs), graphics processing units (GPUs), and custom accelerators to allow more efficient design space exploration and optimization [2]. However, this goal is impossible to reach without the aid of high-level programming models and system runtimes that target the unique parallelization challenges of CAD. This type of system innovation has significant impacts on the CAD community because it *complements* the current state-of-the-art by assisting everyone to tackle the challenges of implementing and deploying parallel CAD algorithms. Unfortunately, related system research has received very little attention in the CAD community.

Manuscript received October 6, 2020; revised March 11, 2021; accepted May 4, 2021. Date of publication May 21, 2021; date of current version April 21, 2022. Preliminary version of this paper has been presented at the IEEE/ACM International Conference on Computer-aided Design (ICCAD), Austin, TX, USA, November 2020 [1]. This article was recommended by Associate Editor C. K. Cheng. *(Corresponding author: Tsung-Wei Huang.)*

Tsung-Wei Huang and Dian-Lun Lin are with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84112 USA (e-mail: twh760812@gmail.com).

Yibo Lin is with the Department of Computer Science, Peking University, Beijing 100871, China.

Chun-Xun Lin is with MathWorks, Natick, MA, USA.

Digital Object Identifier 10.1109/TCAD.2021.3082507

Over the past years, we have invested a lot of research in existing programming systems from the scientific computing community [4]–[8]. However, almost all existing programming systems fall short of the needs by CAD, which we explain as follows.

1) *Optimization algorithms*, such as logic synthesis, placement, and routing, make essential use of *dynamic control flow* to implement various combinatorial and analytical algorithms that incorporate conditional, cyclic, and nondeterministic computational patterns. Existing task programming frameworks closely rely on directed acyclic graph (DAG) models to define tasks and dependencies. Users implement control-flow decisions *outside* the graph description via either statically unrolling the graph across fixed-length iterations or resorting to client-side decisions. This organization typically results in rather complicated procedure that lack *end-to-end* parallelism.
2) *Analysis algorithms*, such as timing and power analysis, require computations to propagate through the circuit network. Different quantities are often dependent on each other and are expensive to compute. The resulting task graph in terms of encapsulated function calls and task dependencies is typically very large (e.g., millions of tasks) However, most existing task parallel frameworks are good at small- or medium-scale tasking and they do not scale to large task graphs.

After years of research, we have arrived at a key conclusion: while designing parallel CAD algorithms is nontrivial, what makes parallelizing CAD an enormous challenge is the infrastructure work of "how efficiently expressing dependent tasks along with algorithmic control flow and scheduling them across heterogeneous computing resources?" To this end, Taskflow introduces a new task programming system to streamline the building of parallel and heterogeneous applications with complex control flow. We summarize three key contributions as follows.

1) *Heterogeneous Programming Model:* We introduce a new CPU–GPU programming model by leveraging modern C++ *closures*. Developers describe a GPU workload in a task graph called *gpuFlow* rather than a sequence of operations. Data can be captured in reference to form a stateful closure that marshals parameter exchange between CPU–GPU-dependent tasks. By abstracting GPU operations to a task graph closure, we judiciously hide implementation details and provide portable optimization to schedule GPU tasks.
2) *General Control Flow:* We develop a new conditional tasking interface to support *general control flow* beyond the DAG models. Our *condition tasks* enable developers to integrate complex control-flow decisions, such as conditional dependencies, cyclic execution, and nondeterministic flows into an *end-to-end* task graph. In case where dynamic behavior is frequent, such as optimization and branch and bound, developers can efficiently overlap tasks both inside and outside the control flow without breaking their dependency constraints.
3) *Heterogeneous Work Stealing:* We develop an efficient work-stealing algorithm to adapt worker threads 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 efficiency, and throughput.

We have evaluated Taskflow on both micro-benchmark and real-world applications to demonstrate its promising performance over existing programming systems. As an example, Taskflow solved a large-scale VLSI placement problem up to 17% faster, with 1.3× fewer memory, 2.1× less power consumption, and 2.9× higher throughput than two industrial-strength systems, oneTBB and StarPU, on a machine of 40 CPUs and 4 GPUs.

II. Taskflow Programming Model

Taskflow is built atop our prior CPU-based parallel task programming system, *Cpp-Taskflow* [9]–[11], where we generalize its idea to a heterogeneous target with control flow.

A. New Heterogeneous Task Programming Model

We introduce a new heterogeneous task programming model by leveraging modern C++ *closures*. Fig. 2 and Listing 1 show the canonical CPU–GPU hybrid saxpy (A·X plus Y) task graph and its implementation using our model. Our model lets users describe a GPU workload in a *coarse-grained task graph* called *gpuFlow* rather than a sequence of GPU operations controlled by CUDA streams [12] or OpenCL buffers [13]. A gpuFlow 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 GPU memory through `cudaMalloc`, and one `gpuFlow` task

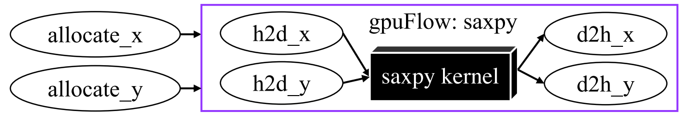

Fig. 2. Saxpy ("single-precision A·X plus Y") task graph.

```
__global__ void saxpy(int n,int a,int *x,int *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(
  [&](){ cudaMalloc(&dx, N*sizeof(float));}
  [&](){ cudaMalloc(&dy, N*sizeof(float));}
);
auto gpuflow = taskflow.emplace(
  [&](tf::gpuFlow& gf) {
    auto h2d_x = gf.copy(dx, hx.data(), N);
    auto h2d_y = gf.copy(dy, hy.data(), N);
    auto d2h_x = gf.copy(hx.data(), dx, N);
    auto d2h_y = gf.copy(hy.data(), dy, N);
    auto kernel = gf.kernel(
      GRID, BLOCK, SHM, saxpy, N, 2.0f, dx, dy
    );
    kernel.succeed(h2d_x, h2d_y)
          .precede(d2h_x, d2h_y);
  }
);
gpuflow.succeed(allocate_x, allocate_y);
```

Listing 1. Taskflow program of Fig. 2.

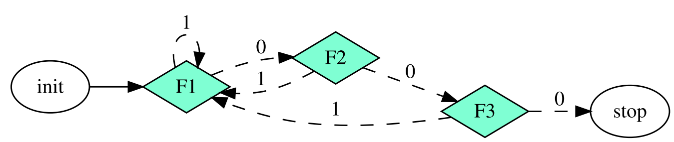

Fig. 3. Taskflow graph of three condition tasks (in diamond) to emulate three-layer nondeterministic control flow. The graph has six weak dependencies (dashed line) and one strong dependency (solid line).

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, `gpuFlow` must run after `allocate_x` and `allocate_y`.

B. New Conditional Tasking Model

We introduce a new *conditional tasking* model to overcome the limitation of existing frameworks in expressing *general control flow* beyond DAG. Fig. 3 shows an example of three-layer control flow that emulates a simplified, very common nondeterministic layout optimization in CAD. The graph consists of two regular tasks, `init` and `stop`, and three condition tasks, `F1`, `F2`, and `F3`. Each condition task forms dynamic control flow to randomly go to either the next task or 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 2 gives the implementation of Fig. 3. Creating a condition task is similar to other tasks, but it returns an integer index of which successor task to execute. The index is

```
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 2. Taskflow program of Fig. 3.

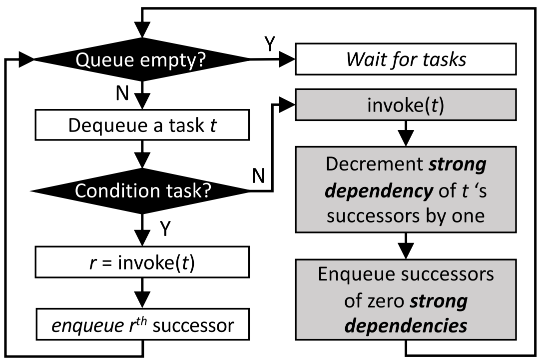

Fig. 4. Flowchart of our task scheduling.

defined with respect to the order of successors of a condition task. For instance, F1 precedes F2 and F1. With this order, if F1 returns 0, the execution proceeds to F2, or loops back to F1 otherwise.

III. Taskflow System Runtime

Taskflow enables users to express CPU–GPU-dependent tasks along with algorithmic control flow in a single heterogeneous task dependency graph (HTDG). To support our model with high performance, we design an efficient system runtime at two scheduling levels, *task level* and *worker level*.

A. Task-Level Scheduling Algorithm

Conditional tasking is powerful but challenging to schedule. Specifically, we must deal with conditional dependency and cyclic execution under the avoidance of *task race*, i.e., only one thread can touch a task at a time. To accommodate this challenge, 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 lines in Fig. 3 are weak dependencies and the solid line, init→F1, is a strong dependency. Based on these notations, we design a simple and efficient algorithm for scheduling tasks, as depicted in Fig. 4. When the scheduler receives an HTDG, it: 1) starts with tasks of zero dependencies (both strong and weak) and continues executing tasks whenever strong dependencies are met or 2) skips this rule for weak dependency and directly jumps to the task indexed by the return of a condition task. By removing the scheduling part of weak dependency, our algorithm falls back to DAG scheduling (marked in gray).

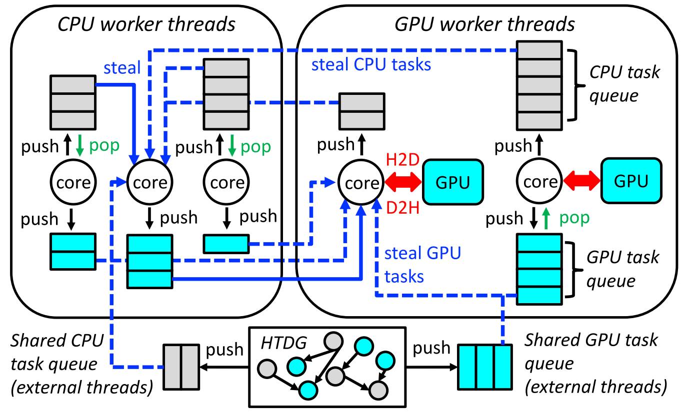

Fig. 5. Architecture of our work-stealing scheduler on two domains, CPU and GPU.

Example: Taking Fig. 3, for example, the scheduler starts with init (i.e., 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 a cyclic execution and so forth. With this concept, the scheduler will cease at stop when F1, F2, and F3 all return 0. It is user's responsibility for ensuring, based on our algorithm, a taskflow program is properly conditioned with no task race or infinite loop during the execution. For instance, adding a strong dependency from init to F2 can result in task race on F2, due to two execution paths, init→F2 and init→F1→F2.

B. Worker-Level Scheduling Algorithm

We leverage *work stealing* to execute submitted tasks with dynamic load balancing. Work stealing has been extensively studied in multicore programming [14]–[16]. A common work-stealing framework 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*. At the architecture level, our scheduler maintains a set of workers for each task domain (e.g., CPU and GPU). A worker can only steal tasks of the same domain from others. Fig. 5 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 and 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.

IV. Experimental Results

We evaluate the performance of Taskflow on two realistic workloads: 1) VLSI placement and 2) static timing analysis, that are representative of many synthesis- and analysis-driven

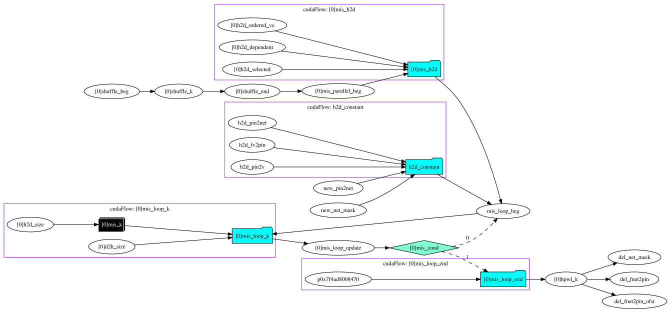

Fig. 6. Partial HTDG of 4 gpuFlows (purple boxes), 1 conditioned cycle (green diamond), and 12 static tasks (other else) for one iteration of our algorithm.

applications. 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 GNU GCC-8.3.0 with C++17 standard `-std=c++17` and optimization flag `-O2` enabled. We consider two industrial-strength libraries, oneTBB [14] and StarPU [5], as our baseline due to their excellent performance [2]. 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.

A. VLSI Placement

We applied 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. Modern placement typically incorporates hundreds of millions of cells and takes several hours to finish [17]. 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 [17], that iterates a GPU-based independent set algorithm to optimize cell layouts in a window-based manner. Each iteration contains thousands of CPU and GPU tasks with nested conditions to decide the convergence. Fig. 6 shows a partial HTDG of one iteration. A complete task graph can have up to two-million CPU–GPU dependent tasks on a million-gate design.

We implemented the hybrid CPU–GPU placement algorithm using Taskflow, oneTBB, and StarPU. The algorithm is handcrafted on one GPU and many CPUs. Since oneTBB and StarPU have no support for nested conditions, we unroll their HTDGs across fixed-length iterations found in hindsight. The overall performance is shown in Fig. 7. Using 8 CPUs and 1 GPU, Taskflow is consistently faster than others across all problem sizes (placement iterations). The gap becomes clear at large problem size; at 100 iterations, 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 the algorithm (see Fig. 6). The memory footprint (middle of Fig. 7) 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 HTDGs. On a vertical scale, increasing the number of CPUs bumps up the memory

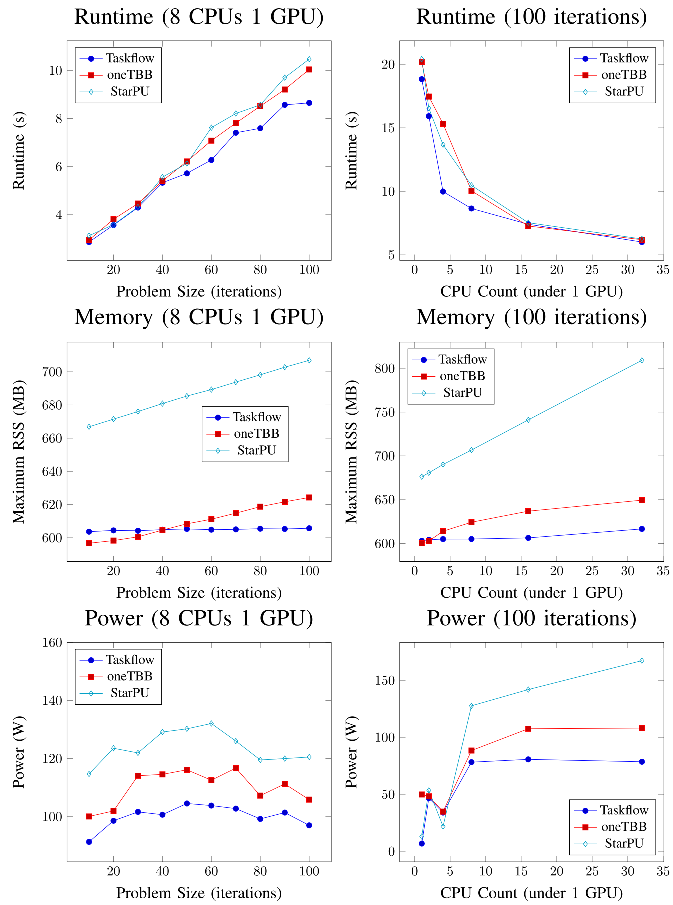

Fig. 7. Runtime, memory, and power data of the circuit adaptec1 (211K cells and 221K nets).

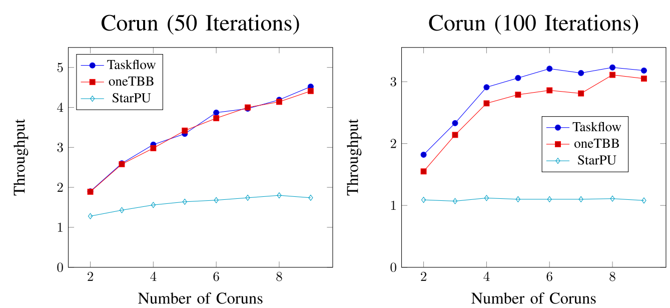

Fig. 8. Throughput of corunning placement workloads on two problem sizes using 40 CPUs and 1 GPU.

usage of all methods, but the growth rate of Taskflow is much slower than the others. In terms of energy (bottom of Fig. 7), 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 efficient worker management algorithm.

For irregular HTDGs akin to Fig. 6, resource utilization is critical to the system throughput. We corun the same program 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. Fig. 8 plots the throughput across nine coruns at two problem sizes. Both Taskflow and oneTBB achieve higher throughput than StarPU.

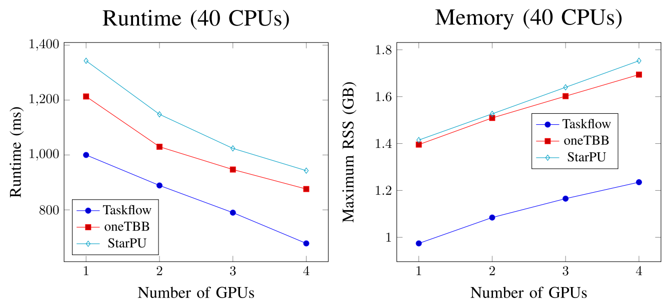

Fig. 9. Runtime and memory data of GPU-accelerated STA on a large design of 1.6M gates.

At the largest problem size, Taskflow outperforms oneTBB and StarPU across all coruns. The result again highlights the strength of our scheduler, which dynamically balances the worker count with available task parallelism.

B. Static Timing Analysis

We applied Taskflow to solve a static timing analysis (STA) problem. We implemented the GPU-accelerated graph-based timing analysis algorithm proposed by [18] using OpenTimer [19]–[22] and Taskflow. The most time-consuming task occurs in RC tree update, where we create up to four gpuFlows on four GPUs to speed up the net delay update. We perform 10 full timing iterations based on the TAU 2015 contest benchmarks [23]. We implemented oneTBB and StarPU as the baseline, both of which support explicit task constructs. Since they have no support for conditional tasking, we unroll their TDGs across these 100 iterations found in hindsight.

Fig. 9 compares the performance of Taskflow with oneTBB and StarPU on timing a million-gate design using different GPU numbers. Taskflow outperforms oneTBB and StarPU in all aspects. Both our runtime and memory scale better regardless of GPU numbers. Our memory usage is 1.7× and 1.8× less 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.

V. Conclusion

In this article, we have introduced Taskflow, a general-purpose task programming system to streamline the creation of heterogeneous programs with complex control flow. Our programming model enables developers to incorporate a broad range of computational patterns with relative ease of programming. We have developed an efficient work-stealing runtime optimized for latency, energy efficiency, and throughput. As an example, we have solved a large-scale circuit placement problem up to 17% faster, with 1.3× fewer memory, 2.1× less power consumption, and 2.9× higher throughput than two industrial-strength systems, oneTBB and StarPU, on a machine of 40 CPUs and 4 GPUs. Future work will focus on applying Taskflow to distributed CAD [24], [25] and large-scale machine learning [26].

References

[1] T.-W. Huang, "A general-purpose parallel and heterogeneous task programming system for VLSI CAD," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, 2020, pp. 1–2.
[2] Y.-S. Lu and K. Pingali, "Can parallel programming revolutionize EDA tools?" in *Advanced Logic Synthesis*. Cham, Switzerland: Springer, 2018.
[3] *DARPA Intelligent Design of Electronic Assets (IDEA) Program*. [Online]. Available: https://www.darpa.mil/program/intelligent-design-of-electronic-assets
[4] E. Ayguade *et al.*, "The design of OpenMP tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009.
[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Comput. Pract. Exp.*, vol. 23, no. 2, pp. 187–198, 2011.
[6] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitioned Global Address Space Program. Models (PGAS)*, 2014, pp. 1–11.
[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov./Dec. 2013.
[8] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.
[9] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern C++," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2019, pp. 974–983.
[10] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "An efficient and composable parallel task programming library," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2019, pp. 1–7.
[11] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A modern C++ parallel task programming library," in *Proc. ACM Multimedia Conf.*, 2019, pp. 2284–2287.
[12] *Nvidia CUDA*. [Online]. Available: https://developer.nvidia.com/cuda-zone
[13] *OpenCL*. [Online]. Available: https://opencl.org/
[14] *Intel OneTBB*. [Online]. Available: https://github.com/oneapi-src/oneTBB
[15] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Atlanta, GA, USA, 2010, pp. 1–12.
[16] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An efficient work-stealing scheduler for task dependency graph," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Hong Kong, 2020, pp. 64–71.
[17] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, Las Vegas, NV, USA, 2019, pp. 1–6.
[18] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated static timing analysis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Diego, CA, USA, 2020, pp. 1–8.
[19] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, 2015, pp. 895–902.
[20] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer v2: A new parallel incremental timing analysis engine," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 4, pp. 776–789, Apr. 2021.
[21] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A parallel incremental timing analysis engine," *IEEE Design Test*, vol. 38, no. 2, pp. 62–68, Apr. 2021.
[22] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated pash-based timing analysis," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2021.
[23] T.-W. Huang and M. D. F. Wong, "UI-Timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 11, pp. 1862–1875, Nov. 2016.
[24] T.-W. Huang and M. D. F. Wong, "Accelerated path-based timing analysis with MapReduce," in *Proc. ACM Int. Symp. Phys. Design (ISPD)*, 2015, pp. 103–110.
[25] T.-W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Austin, TX, USA, 2016, pp. 1–6.
[26] D.-L. Lin and T.-W. Huang, "A novel inference algorithm for large sparse neural network using task graph parallelism," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Waltham, MA, USA, 2020, pp. 1–7.