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 solvers 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.

I. INTRODUCTION

The ever-increasing design complexity in very-large-scale integration (VLSI) implementation will soon 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 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.

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

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 parallel and heterogeneous computer-aided design (CAD) applications work are good at small- or medium-scale tasking 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 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

Fundamental challenges in CAD must be solved to reduce large design time and effort

Transistors How can we keep CAD up with Moore’s Law?

<table>
<thead>
<tr>
<th>Transistors per chip</th>
<th>Design cost</th>
<th>Total HW cost</th>
<th>Verification cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100M</td>
<td>100M</td>
<td>100M</td>
<td>100M</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>1M</td>
<td>1M</td>
</tr>
<tr>
<td>10M</td>
<td>10M</td>
<td>10M</td>
<td>10M</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100M</td>
<td>100M</td>
<td>100M</td>
<td>100M</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>1M</td>
<td>1M</td>
</tr>
<tr>
<td>10M</td>
<td>10M</td>
<td>10M</td>
<td>10M</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100M</td>
<td>100M</td>
<td>100M</td>
<td>100M</td>
</tr>
<tr>
<td>1M</td>
<td>1M</td>
<td>1M</td>
<td>1M</td>
</tr>
<tr>
<td>10M</td>
<td>10M</td>
<td>10M</td>
<td>10M</td>
</tr>
</tbody>
</table>
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 non-deterministic flows into an end-to-end 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-

**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

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 nondeterministic control flow. The graph has six weak dependencies (dashed line) and one strong dependency (solid line).
defined with respect to the order of successors of a condition task. For instance, \( F_1 \) precedes \( F_2 \) and \( F_1 \). With this order, if \( F_1 \) returns 0, the execution proceeds to \( F_2 \), or loops back to \( F_1 \) 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, \( \text{init} \rightarrow F_1 \), 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).

#### 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.

**Example:** Taking Fig. 3, for example, the scheduler starts with \( \text{init} \) (i.e., zero weak and strong dependencies) and proceeds to \( F_1 \). Assuming \( F_1 \) returns 0, the scheduler proceeds to its first successor, \( F_2 \). Now, assuming \( F_2 \) returns 1, the scheduler proceeds to its second successor, \( F_1 \), which forms a cyclic execution and so forth. With this concept, the scheduler will cease at \( \text{stop} \) when \( F_1 \), \( F_2 \), and \( F_3 \) all return 0. It is user’s responsibility for ensuring, based on our algorithm, a taskflow program is properly conditioned without task race or infinite loop during the execution. For instance, adding a strong dependency from \( \text{init} \) to \( F_2 \) can result in task race on \( F_2 \), due to two execution paths, \( \text{init} \rightarrow F_2 \) and \( \text{init} \rightarrow F_1 \rightarrow F_2 \).

### 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
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 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 tasks, 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