Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System in Modern C++

Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT
Why Parallel Computing?

- It’s critical to advance your application performance

![Bar chart showing time to solve a machine learning workload with 1 CPU, 40 CPUs, and 1 GPU, demonstrating 10x and 100x faster performance.]
Parallel programming is crucial but very challenging ...
Taskflow offers a solution

How can we make it easier for C++ developers to quickly write parallel and heterogeneous programs with high performance scalability and simultaneous high productivity?
“Hello World” in Taskflow

#include <taskflow/taskflow.hpp> // Taskflow is header-only

int main(){
    tf::Taskflow taskflow;
    tf::Executor executor;
    auto [A, B, C, D] = taskflow.emplace(
        [] () { std::cout << "TaskA\n"; },
        [] () { std::cout << "TaskB\n"; },
        [] () { std::cout << "TaskC\n"; },
        [] () { std::cout << "TaskD\n"; }
    );
    A.precede(B, C); // A runs before B and C
    D.succeed(B, C); // D runs after B and C
    executor.run(taskflow).wait(); // submit the taskflow to the executor
    return 0;
}
Drop-in Integration

• Taskflow is header-only – *no wrangle with installation*

```bash
~$ git clone https://github.com/taskflow/taskflow.git  # clone it only once
~$ g++ -std=c++17 simple.cpp -I taskflow/taskflow -O2 -pthread -o simple
~$ ./simple
TaskA
TaskC
TaskB
TaskD
```
Built-in Profiler/Visualizer

# run the program with the environment variable TF_ENABLE_PROFILER enabled
~$ TF_ENABLE_PROFILER=simple.json ./simple
~$ cat simple.json
[
{"executor":"0","data":[{"worker":0,"level":0,"data":[{"span":[172,186],"name

# paste the profiling json data to https://taskflow.github.io/tfprof/
Agenda

• Express your parallelism in the right way
• Parallelize your applications using Taskflow
• Boost performance in real applications
Agenda

• Express your parallelism in the right way
• Parallelize your applications using Taskflow
• Boost performance in real applications
Motivation: Parallelizing VLSI CAD Tools

• Billions of tasks with diverse computational patterns

How can we write efficient C++ parallel programs for this monster computational task graph with millions of CPU-GPU dependent tasks along with algorithmic control flow"
We Invested a lot in Existing Tools …
Two Big Problems of Existing Tools

• Our problems define *complex task dependencies*
  • **Example**: analysis algorithms compute the circuit network of million of node and dependencies
  • **Problem**: existing tools are often good at loop parallelism but weak in expressing heterogeneous task graphs at this large scale

• Our problems define *complex control flow*
  • **Example**: optimization algorithms make essential use of *dynamic control flow* to implement various patterns
    • Combinatorial optimization, analytical methods
  • **Problem**: existing tools are *directed acyclic graph* (DAG)-based and do not anticipate cycles or conditional dependencies, lacking *end-to-end* parallelism
Example: An Iterative Optimizer

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

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

Millions of such tasks?  End-to-end parallelism?
Need a New C++ Parallel Programming System

While designing parallel algorithms is non-trivial …

what makes parallel programming an enormous challenge is the infrastructure work of “how to efficiently express dependent tasks along with an algorithmic control flow and schedule them across heterogeneous computing resources”
Agenda

• Express your parallelism in the right way
• Parallelize your applications using Taskflow
• Boost performance in real applications
“Hello World” in Taskflow (Revisited)

#include <taskflow/taskflow.hpp>  // Taskflow is header-only

int main()
{
    tf::Taskflow taskflow;
    tf::Executor executor;
    auto [A, B, C, D] = taskflow.emplace(
        [] () { std::cout << "TaskA\n"; },
        [] () { std::cout << "TaskB\n"; },
        [] () { std::cout << "TaskC\n"; },
        [] () { std::cout << "TaskD\n"; }
    );
    A.precede(B, C);  // A runs before B and C
    D.succeed(B, C);  // D runs after B and C
    executor.run(taskflow).wait();
    return 0;
}
Heterogeneous Tasking (cudaFlow)

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

```
const unsigned N = 1<<20;
std::vector<float> hx(N, 1.0f), hy(N, 2.0f);
float *dx{nullptr}, *dy{nullptr};
auto allocate_x = taskflow.emplace([&](){ cudaMalloc(&dx, 4*N); });
auto allocate_y = taskflow.emplace([&](){ cudaMalloc(&dy, 4*N); });

auto cudaflow = taskflow.emplace([&](tf::cudaFlow& cf) {
    auto h2d_x = cf.copy(dx, hx.data(), N); // CPU-GPU data transfer
    auto h2d_y = cf.copy(dy, hy.data(), N);
    auto d2h_x = cf.copy(hx.data(), dx, N); // GPU-CPU data transfer
    auto d2h_y = cf.copy(hy.data(), dy, N);
    auto kernel = cf.kernel(((N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy);
    kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});

cudaflow.succeed(allocate_x, allocate_y);
executor.run(taskflow).wait();
```
Three Key Motivations

• Our closure enables stateful interface
  • Users capture data in reference to marshal data exchange between CPU and GPU tasks

• Our closure hides implementation details judiciously
  • We use cudaGraph (since cuda 10) due to its excellent performance, much faster than streams in large graphs

• Our closure extend to new accelerator types (e.g., SYCL)

```cpp
auto cudaflow = taskflow::emplace([&](tf::cudaFlow& cf) {
  auto h2d_x = cf.copy(dx, hx.data(), N); // CPU-GPU data transfer
  auto h2d_y = cf.copy(dy, hy.data(), N);
  auto d2h_x = cf.copy(hx.data(), dx, N); // GPU-CPU data transfer
  auto d2h_y = cf.copy(hy.data(), dy, N);
  auto kernel = cf.kernel([(N+255)/256, 256, 0, saxpy, N, 2.0f, dx, dy];
  kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);
});
```

We do not simplify kernel programming but focus on **CPU-GPU tasking** that affects the performance to a large extent! (same for data abstraction)
Heterogeneous Tasking (syclFlow)

```cpp
auto syclflow = taskflow.emplace_on([&](tf::syclFlow& sf) {
    auto h2d_x = cf.copy(dx, hx.data(), N); // CPU-GPU data transfer
    auto h2d_y = cf.copy(dy, hy.data(), N);
    auto d2h_x = cf.copy(hx.data(), dx, N); // GPU-CPU data transfer
    auto d2h_y = cf.copy(hy.data(), dy, N);
    auto kernel = cf.parallel_for(sycl::range<1>(N), [=](sycl::id<1> id){
        dx[id] = 2.0f * dx[id] + dy[id];
    });
    kernel.succeed(h2d_x, h2d_y)
        .precede(d2h_x, d2h_y);
}, queue);
```

Create a syclFlow from a SYCL queue on a SYCL device
Conditional Tasking (Simple if-else)

```cpp
auto init = taskflow.emplace([&](){ initialize_data_structure(); } )
    .name("init");
auto optimizer = taskflow.emplace([&](){ matrix_solver(); } )
    .name("optimizer");
auto converged = taskflow.emplace([&](){ return converged() ? 1 : 0 } )
    .name("converged");
auto output = taskflow.emplace([&](){ std::cout << "done!\n"; });
    .name("output");

init.precede(optimizer);
optimizer.precede(converged);
converged.precede(optimizer, output);  // return 0 to the optimizer again
```

Condition task integrates control flow into a task graph to form **end-to-end** parallelism.
Conditional Tasking (While/For Loop)

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

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

Existing frameworks on expressing conditional tasking or dynamic control flow suffer from exponential growth of code complexity
Everything is Unified in Taskflow

• Use “emplace” to create a task
• Use “precede” to add a task dependency
• No need to learn different sets of API
• You can create a really complex graph
  • Subflow(ConditionTask(cudaFlow))
  • ConditionTask(StaticTask(cudaFlow))
  • Composition(Subflow(ConditionTask))
  • Subflow(ConditionTask(cudaFlow))
  • …
• Scheduler performs end-to-end optimization
  • Runtime, energy efficiency, and throughput
Agenda

• Express your parallelism in the right way
• Parallelize your applications using Taskflow
• Boost performance in real applications
Application 1: VLSI Placement

• Optimize cell locations on a chip

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

VLSI optimization makes essential use of dynamic control flow
Application 1: VLSI Placement (cont’d)

- Runtime, memory, power, and throughput

Performance improvement comes from the *end-to-end* expression of CPU-GPU dependent tasks using condition tasks.
Application 2: Machine Learning

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

Each cudaFlow contains >1000 of GPU tasks

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

Champions of HPEC 2020 Graph Challenge: [https://graphchallenge.mit.edu/champions](https://graphchallenge.mit.edu/champions)
Comparison with TBB and StarPU

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

Different models give different implementations. The parallel code/algorithm may run fast, yet the parallel computing infrastructure to support that algorithm may dominate the entire performance.

Taskflow enables *end-to-end* expression of CPU-GPU dependent tasks along with algorithmic control flow.
Conclusion

• Taskflow is a lightweight parallel task programming system
  • Simple, efficient, and transparent tasking models
  • Efficient heterogeneous work-stealing executor
  • Promising performance in large-scale ML and VLSI CAD

• Taskflow is not to replace anyone but to
  • Complement the current state-of-the-art
  • Leverage modern C++ to express task graph parallelism

• Taskflow is very open to collaboration
  • We want to provide more higher-level algorithms
  • We want to broaden real use cases
  • We want to enhance the core functionalities (e.g., pipeline)
Thank You All Using Taskflow!
Use the right tool for the right job

Taskflow: https://taskflow.github.io

Dr. Tsung-Wei Huang
tzung-wei.huang@utah.edu

Thank You