2023

# An Introduction to C++ Coroutines Through a Thread Scheduling Demonstration

Dian-Lun Lin

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples
Bio: https://dian-lun-lin.github.io/

C++ now

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

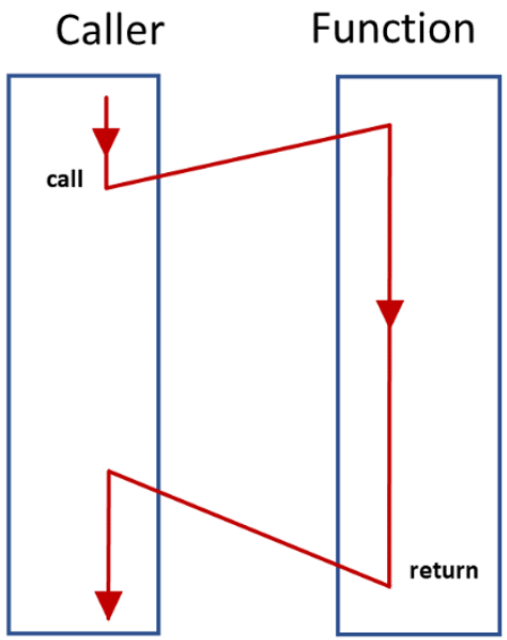
# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# What is Coroutine

suspend and resume!

- A coroutine is a function that can suspend itself and resume by caller
- A “typical” function is a subset of coroutine



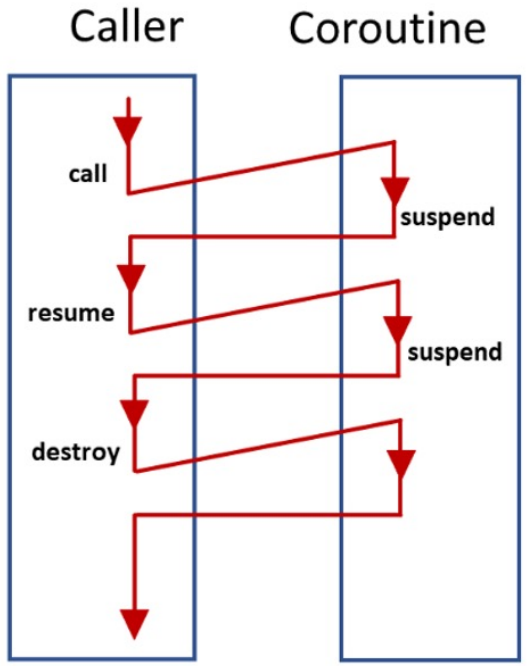




Image credit: Moderns C++

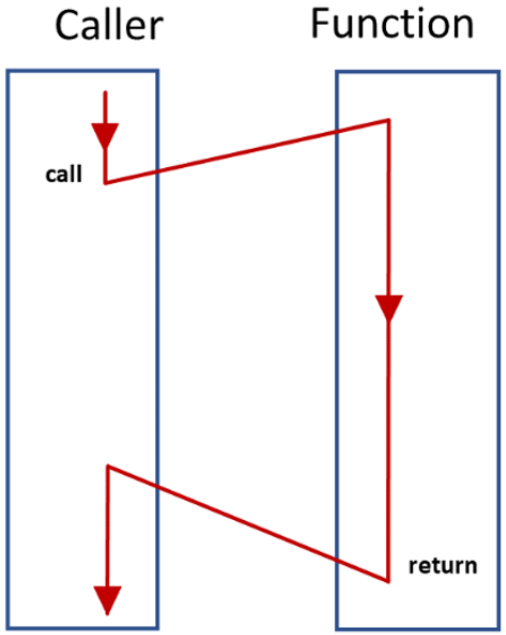
# Why Coroutine

- Imaging you want to do two things when you go home…

1. Boil the water

2. Take a shower

Suppose each thing is a function
Suppose you are single




wait, wait, and wait…

# Why Coroutine

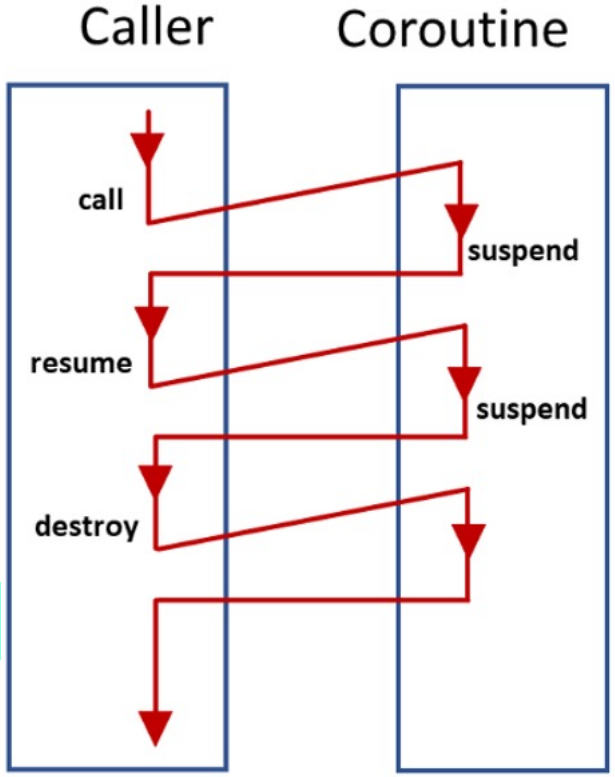
- Imaging you want to do two things when you go home…

1. Boil the water

2. Take a shower

take a shower

enjoy your hot water




turn on the stove

turn off the stove

There is an overlap!

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~ other computing resource! GPU, TPU, async I/O, …

Without coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 void gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~ other computing resource! GPU, TPU, async I/O, …

Without coroutine

```
2  void cpu_work() {
3    cpu_matmul(matA, matB, ...);
4  }
5
6  void gpu_work() {
7    cudaStream_t stream;
8    cudaStreamCreate(stream);
9    gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~

other computing resource!
GPU, TPU, async I/O, …

Without coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 void gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~ other computing resource! GPU, TPU, async I/O, …

Without coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 void gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~ other computing resource! GPU, TPU, async I/O, …

Without coroutine

```
2  void cpu_work() {
3    cpu_matmul(matA, matB, ...);
4  }
5
6  void gpu_work() {
7    cudaStream_t stream;
8    cudaStreamCreate(stream);
9    gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

With coroutine

```
2  void cpu_work() {
3    cpu_matmul(matA, matB, ...);
4  }
5
6  Coro gpu_work() {
7    cudaStream_t stream;
8    cudaStreamCreate(stream);
9    gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
15
16 // cpu_work and gpu_work are independent
17 // assume we only have one CPU thread
18 int main() {
19   auto coro = gpu_work();
20   cpu_work();
21
22   while(!coro.done()) { coro.resume(); }
23 }
```

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~ other computing resource! GPU, TPU, async I/O, …

Without coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 void gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

With coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
15
16 // cpu_work and gpu_work are independent
17 // assume we only have one CPU thread
18 int main() {
19   auto coro = gpu_work();
20   cpu_work();
21
22   while(!coro.done()) { coro.resume(); }
23 }
```

# Why Coroutine

- Coroutine is very useful if you have a ~~stove~~ other computing resource! GPU, TPU, async I/O, …

## Without coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 void gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB, ...);
10   cudaStreamSynchronize(stream);
11   cudaStreamDestroy(stream);
12 }
13
14 // cpu_work and gpu_work are independent
15 // assume we only have one CPU thread
16 int main() {
17   cpu_work();
18   gpu_work();
19
20   // alternatively
21   gpu_work();
22   cpu_work();
23 }
```

## With coroutine

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
15
16 // cpu_work and gpu_work are independent
17 // assume we only have one CPU thread
18 int main() {
19   auto coro = gpu_work();
20   cpu_work();
21
22   while(!coro.done()) { coro.resume(); }
23 }
```

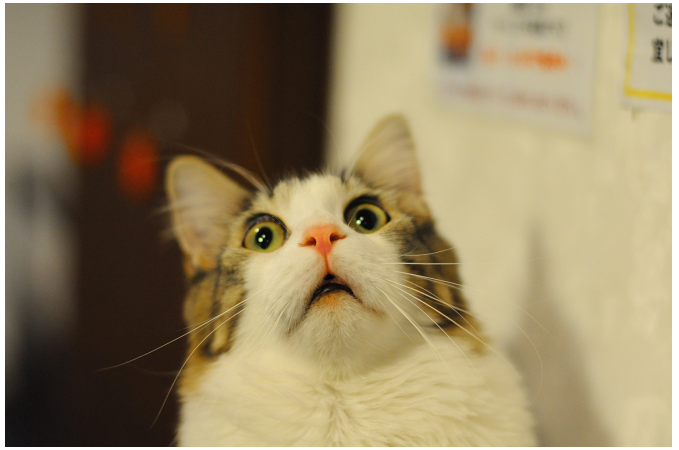
# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# C++ Coroutine

- Introduced in C++ 20

- While coroutine concept is simple, C++ coroutine is not easy…
  - Lots of customization points
  - Not that straightforward
  - Lack of examples



- Implementing a C++ coroutine requires:
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle

# How to Define a Coroutine

Use
co_await, co_yield, and/or co_return

Return a
coroutine object specifying a promise

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
15
16 // cpu_work and gpu_work are independent to each other
17 // assume we only have one CPU thread
18 int main() {
19   auto coro = gpu_work();
20   cpu_work();
21
22   while(!coro.done()) { coro.resume(); }
23 }
```

# How to Define a Coroutine

Use
co_await, co_yield, and/or co_return

Return a
coroutine object specifying a promise

```
 2 void cpu_work() {
 3   cpu_matmul(matA, matB, ...);
 4 }
 5
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
15
16 // cpu_work and gpu_work are independent to each other
17 // assume we only have one CPU thread
18 int main() {
19   auto coro = gpu_work();
20   cpu_work();
21
22   while(!coro.done()) { coro.resume(); }
23 }
```

# How to Define a Coroutine

The coroutine class must define promise_type

```
 2 struct Coro {
 3
 4   struct promise_type {
 5     std::suspend_always initial_suspend() noexcept { return {}; }
 6     std::suspend_always final_suspend() noexcept { return {}; }
 7
 8     Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
 9     void return_void() {}
10     void unhandled_exception() {}
11   };
12
13   Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}
14
15   std::coroutine_handle<promise_type> handle;
16 };
```

# How to Define a Coroutine

The coroutine class must define promise_type

```
struct Coro {

  struct promise_type {
    std::suspend_always initial_suspend() noexcept { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }

    Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
    void return_void() {}
    void unhandled_exception() {}
  };

  Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}

  std::coroutine_handle<promise_type> handle;
};
```

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Promise

- Promise controls **coroutine's behavior**
  - The suspension of beginning and end of a coroutine
  - The creation of return object
  - Exception handling
  - …

# How to Define a Promise

The coroutine class must define promise_type

```
 2 struct Coro {
 3
 4   struct promise_type {
 5     std::suspend_always initial_suspend() noexcept { return {}; }
 6     std::suspend_always final_suspend() noexcept { return {}; }
 7
 8     Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
 9     void return_void() {}
10     void unhandled_exception() {}
11   };
12
13   Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}
14
15   std::coroutine_handle<promise_type> handle;
16 };
```

# Promise

```
2  struct Coro {
3
4    struct promise_type {
5      std::suspend_always initial_suspend() noexcept { return {}; }
6      std::suspend_always final_suspend() noexcept { return {}; }
7
8      Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
9      void return_void() {}
10     void unhandled_exception() {}
11   };
12
13   Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}
14
15   std::coroutine_handle<promise_type> handle;
16 };
```

The suspension of beginning of a coroutine

# Promise

```
 2 struct Coro {
 3
 4   struct promise_type {
 5     std::suspend_always initial_suspend() noexcept { return {}; }
 6     std::suspend_always final_suspend() noexcept { return {}; }
 7
 8     Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
 9     void return_void() {}
10     void unhandled_exception() {}
11   };
12
13   Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}
14
15   std::coroutine_handle<promise_type> handle;
16 };
```

The suspension of end of a coroutine

# Promise

```
2  struct Coro {
3
4    struct promise_type {
5      std::suspend_always initial_suspend() noexcept { return {}; }
6      std::suspend_always final_suspend() noexcept { return {}; }
7
8      Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
9      void return_void() {}
10     void unhandled_exception() {}
11   };
12
13   Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}
14
15   std::coroutine_handle<promise_type> handle;
16 };
```

The creation of return object

# Promise

```
2  struct Coro {
3
4    struct promise_type {
5      std::suspend_always initial_suspend() noexcept { return {}; }
6      std::suspend_always final_suspend() noexcept { return {}; }
7
8      Coro get_return_object() { return std::coroutine_handle<promise_type>::from_promise(*this); }
9      void return_void() {}
10     void unhandled_exception() {}
11   };
12
13   Coro(std::coroutine_handle<promise_type> handle): handle{handle} {}
14
15   std::coroutine_handle<promise_type> handle;
16 };
```

Exception handling

# Why do We Need to Define Promise

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```

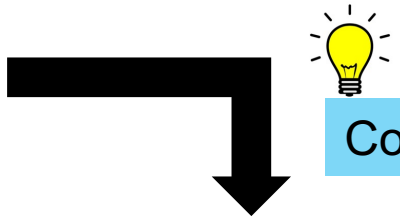
# Why do We Need to Define Promise

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```

That's see what compiler sees

# Coroutine/Promise – Compiler's View

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```



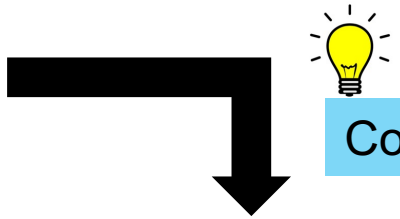


Compiler's view (simplified)

```
 2 Coro gpu_work() {
 3   Coro::promise_type p();
 4   Coro coro_obj = p.get_return_object();
 5
 6   try {
 7     co_await p.initial_suspend();
 8     cudaStream_t stream;
 9     cudaStreamCreate(stream);
10     gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
11     while(cudaStreamQuery(stream) != cudaSuccess) {
12       co_await std::suspend_always{};
13     }
14     cudaStreamDestory(stream);
15   } catch(...) {
16     p.unhandled_exception();
17   }
18   co_await p.final_suspend();
19 }
```

# Coroutine/Promise – Compiler's View

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```



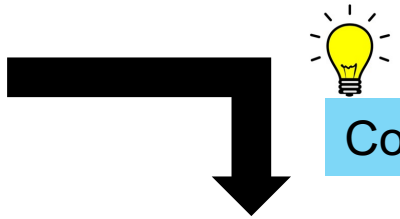


Compiler's view (simplified)

```
 2 Coro gpu_work() {
 3   Coro::promise_type p();
 4   Coro coro_obj = p.get_return_object();
 5
 6   try {
 7     co_await p.initial_suspend();
 8     cudaStream_t stream;
 9     cudaStreamCreate(stream);
10     gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
11     while(cudaStreamQuery(stream) != cudaSuccess) {
12       co_await std::suspend_always{};
13     }
14     cudaStreamDestory(stream);
15   } catch(...) {
16     p.unhandled_exception();
17   }
18   co_await p.final_suspend();
19 }
```

# Coroutine/Promise – Compiler's View

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```



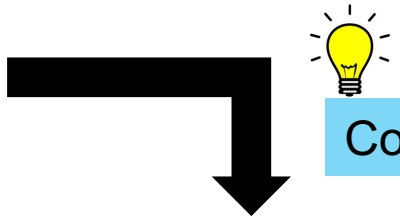


Compiler's view (simplified)

```
 2 Coro gpu_work() {
 3   Coro::promise_type p();
 4   Coro coro_obj = p.get_return_object();
 5
 6   try {
 7     co_await p.initial_suspend();
 8     cudaStream_t stream;
 9     cudaStreamCreate(stream);
10     gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
11     while(cudaStreamQuery(stream) != cudaSuccess) {
12       co_await std::suspend_always{};
13     }
14     cudaStreamDestory(stream);
15   } catch(...) {
16     p.unhandled_exception();
17   }
18   co_await p.final_suspend();
19 }
```

# Coroutine/Promise – Compiler's View

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```



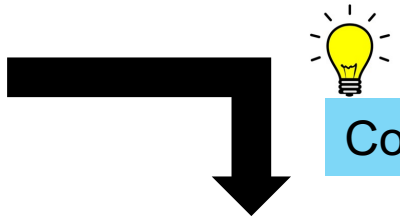


Compiler's view (simplified)

```
 2 Coro gpu_work() {
 3   Coro::promise_type p();
 4   Coro coro_obj = p.get_return_object();
 5
 6   try {
 7     co_await p.initial_suspend();
 8     cudaStream_t stream;
 9     cudaStreamCreate(stream);
10     gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
11     while(cudaStreamQuery(stream) != cudaSuccess) {
12       co_await std::suspend_always{};
13     }
14     cudaStreamDestory(stream);
15   } catch(...) {
16     p.unhandled_exception();
17   }
18   co_await p.final_suspend();
19 }
```

# Coroutine/Promise – Compiler's View

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```



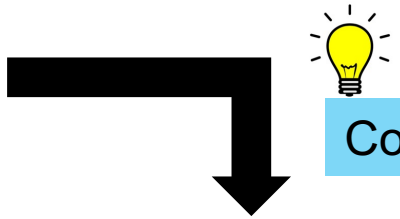


Compiler's view (simplified)

```
 2 Coro gpu_work() {
 3   Coro::promise_type p();
 4   Coro coro_obj = p.get_return_object();
 5
 6   try {
 7     co_await p.initial_suspend();
 8     cudaStream_t stream;
 9     cudaStreamCreate(stream);
10     gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
11     while(cudaStreamQuery(stream) != cudaSuccess) {
12       co_await std::suspend_always{};
13     }
14     cudaStreamDestory(stream);
15   } catch(...) {
16     p.unhandled_exception();
17   }
18   co_await p.final_suspend();
19 }
```

# That's Why We Need to Define Promise

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```



Compiler's view (simplified)

```
 2 Coro gpu_work() {
 3   Coro::promise_type p();
 4   Coro coro_obj = p.get_return_object();
 5
 6   try {
 7     co_await p.initial_suspend();
 8     cudaStream_t stream;
 9     cudaStreamCreate(stream);
10     gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
11     while(cudaStreamQuery(stream) != cudaSuccess) {
12       co_await std::suspend_always{};
13     }
14     cudaStreamDestory(stream);
15   } catch(...) {
16     p.unhandled_exception();
17   }
18   co_await p.final_suspend();
19 }
```

Promise controls behavior of coroutine!

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Awaitable

- Awaitable controls a specific **suspension point behavior**

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```

# Awaitable

- Awaitable controls a specific **suspension point behavior**

```
 6 Coro gpu_work() {
 7   cudaStream_t stream;
 8   cudaStreamCreate(stream);
 9   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
10   while(cudaStreamQuery(stream) != cudaSuccess) {
11     co_await std::suspend_always{};
12   }
13   cudaStreamDestory(stream);
14 }
```

Should I always suspend?
Or maybe I can conditionally resume?

# How to Define an Awaitable

To define an Awaitable (actually Awaiter):

- await_ready()
- await_suspend()
- await_resume()

```
2  Coro gpu_work() {
3    cudaStream_t stream;
4    cudaStreamCreate(stream);
5    gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
6    while(cudaStreamQuery(stream) != cudaSuccess) {
7      co_await std::suspend_always{};
8    }
9   cudaStreamDestory(stream);
10 }
11
12 // compiler transform
13 auto&& awaiter = std::suspend_always{};
14 if(!awaiter.await_ready()) {
15   awaiter.await_suspend(std::coroutine_handle<>...);
16   //<suspend/resume>
17 }
19 awaiter.await_resume();
```

# Compiler’s View (Simplified)

To define an Awaitable (actually Awaiter):

- await_ready()
- await_suspend()
- await_resume()

```
Coro gpu_work() {
  cudaStream_t stream;
  cudaStreamCreate(stream);
  gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
  while(cudaStreamQuery(stream) != cudaSuccess) {
    co_await std::suspend_always{};
  }
 cudaStreamDestory(stream
}

// compiler transform
auto&& awaiter = std::suspend_always{};
if(!awaiter.await_ready()) {
  awaiter.await_suspend(std::coroutine_handle<>...);
  //<suspend/resume>
}
awaiter.await_resume();
```

Compiler’s view

# Built-in Awaitable

```
namespace std {

   struct suspend_never {
       constexpr bool await_ready() const noexcept { return true; }
       constexpr void await_suspend(coroutine_handle<>) const noexcept {}
       constexpr void await_resume() const noexcept {}
   };

   struct suspend_always {
       constexpr bool await_ready() const noexcept { return false; }
       constexpr void await_suspend(coroutine_handle<>) const noexcept {}
       constexpr void await_resume() const noexcept {}
   };

}
```

```
 2 Coro gpu_work() {
 3   cudaStream_t stream;
 4   cudaStreamCreate(stream);
 5   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
 6   while(cudaStreamQuery(stream) != cudaSuccess) {
 7     co_await std::suspend_always{};
 8   }
 9  cudaStreamDestory(stream
10 }
11
12 // compiler transform
13 auto&& awaiter = std::suspend_always{};
14 if(!awaiter.await_ready()) {
15   awaiter.await_suspend(std::coroutine_handle<>...);
16   //<suspend/resume>
17 }
19 awaiter.await_resume();
```

Compiler's view

# Variants of await_suspend()

```
namespace std {

   struct suspend_never {
       constexpr bool await_ready() const noexcept { return true; }
       constexpr void await_suspend(coroutine_handle<>) const noexcept {}
       constexpr void await_resume() const noexcept {}
   };

   struct suspend_always {
       constexpr bool await_ready() const noexcept { return false; }
       constexpr void await_suspend(coroutine_handle<>) const noexcept {}
       constexpr void await_resume() const noexcept {}
   };

}
```

```
 2 Coro gpu_work() {
 3   cudaStream_t stream;
 4   cudaStreamCreate(stream);
 5   gpu_matmul<<<8, 256, 0, stream>>>(matA, matB);
 6   while(cudaStreamQuery(stream) != cudaSuccess) {
 7     co_await std::suspend_always{};
 8   }
 9  cudaStreamDestory(stream
10 }
11
12 // compiler transform
13 auto&& awaiter = std::suspend_always{};
14 if(!awaiter.await_ready()) {
15   awaiter.await_suspend(std::coroutine_handle<>...);
16   //<suspend/resume>
17 }
19 awaiter.await_resume();
```

Compiler's view

void await_suspend(…) -> always suspend
bool await_suspend(…) -> suspend if return true
std::coroutine_handle<> await_suspend(…) -> resume returned coroutine

Symmetric coroutine transfer

# Promise v.s. Awaitable

- Promise controls coroutine behavior
  - initial_suspend(), final_suspend(), exception handling, …
- Awaitable controls suspension point behavior
  - co_await std::suspend_always()

Understand how compiler reads your coroutine

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Coroutine Handle

- Like a pointer to the coroutine
- You can access promise and coroutine via coroutine handle

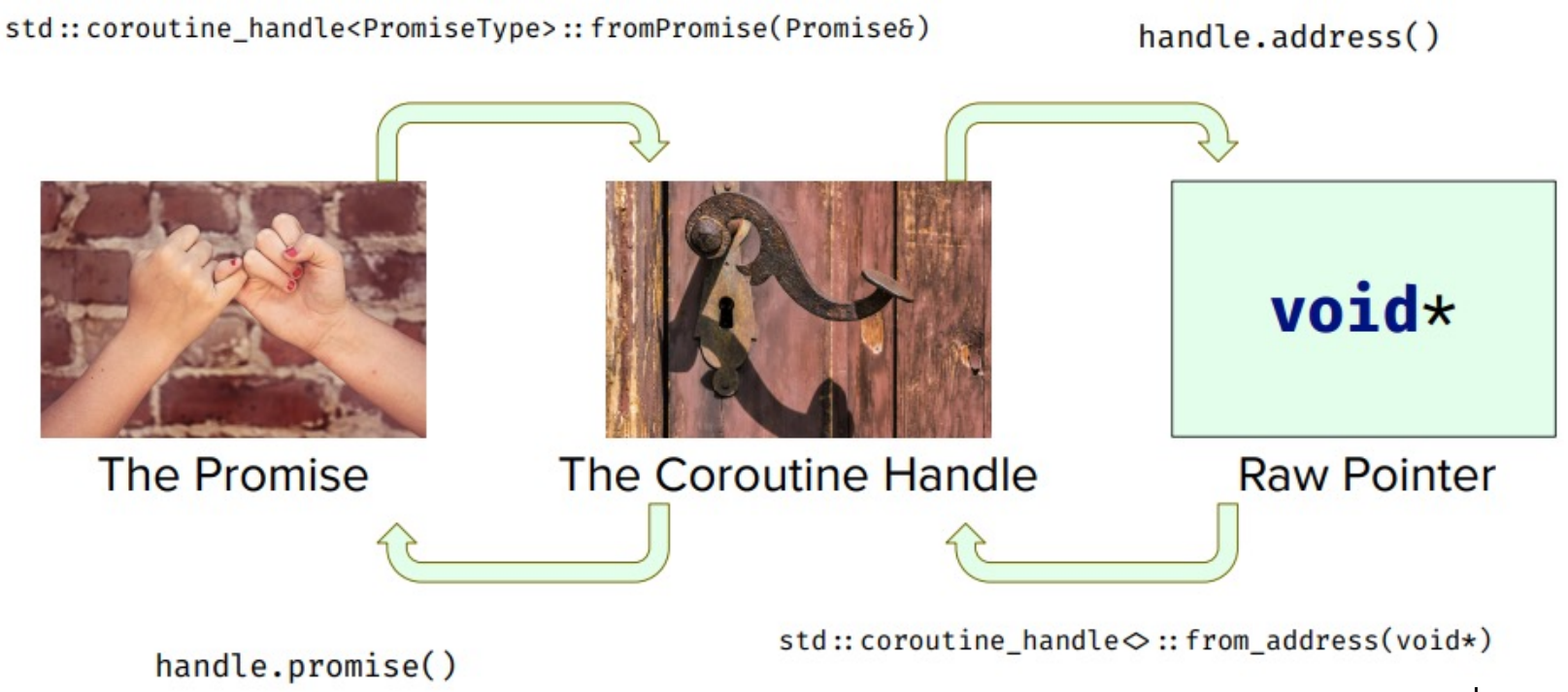



Image credit: Andreas Buhr

# Coroutine Handle Takes Promise as Template

- coroutine_handle<> can represent any coroutine
- However, you lost the ability to use handle.promise()

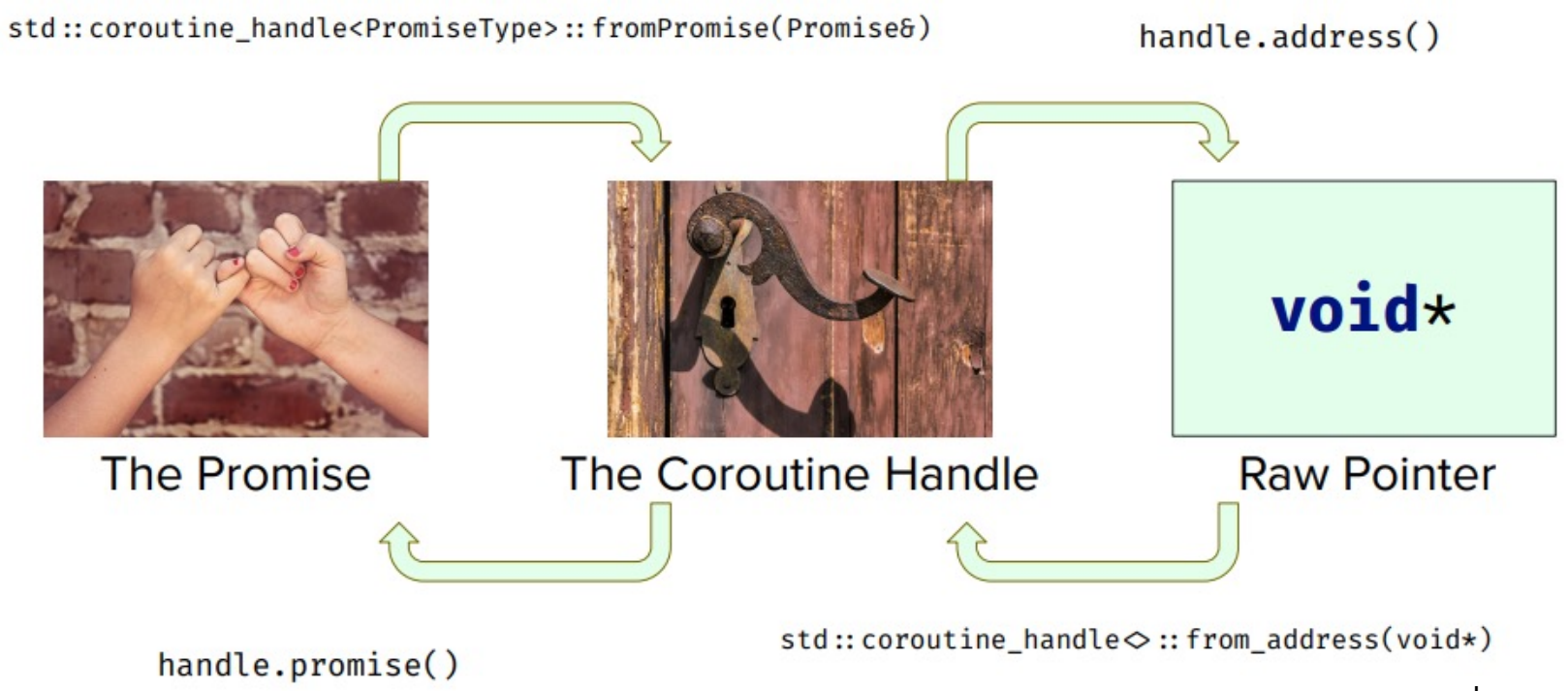



Image credit: Andreas Buhr

# Coroutine Handle

- handle.resume() -> resume the coroutine
- handle.done()  -> check if the coroutine has completed
- handle.destroy() -> destroy the coroutine

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Coroutine Tasks and Scheduler APIs

Single-threaded scheduler

Coroutine tasks

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Coroutine Tasks and Scheduler APIs

Single-threaded scheduler

Coroutine tasks

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle() : get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Definition of coroutine/promise_type

Single-threaded scheduler

```
 5 struct Task {
 6
 7   struct promise_type {
 8     std::suspend_always initial_suspend() noexcept { return {}; }
 9     std::suspend_always final_suspend() noexcept { return {}; }
10
11     Task get_return_object() {
         return std::coroutine_handle<promise_type>::from_promise(*this);
       }
12     void return_void() {}
13     void unhandled_exception() {}
14   };
15
16   Task(std::coroutine_handle<promise_type> handle): handle{handle} {}
17
18   auto get_handle() { return handle; }
19
20   std::coroutine_handle<promise_type> handle;
21 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Definition of coroutine/promise_type

Single-threaded scheduler

```
 5 struct Task {
 6
 7   struct promise_type {
 8     std::suspend_always initial_suspend() noexcept { return {}; }
 9     std::suspend_always final_suspend() noexcept { return {}; }
10
11     Task get_return_object() {
         return std::coroutine_handle<promise_type>::from_promise(*this);
       }
12     void return_void() {}
13     void unhandled_exception() {}
14   };
15
16   Task(std::coroutine_handle<promise_type> handle): handle{handle} {}
17
18   auto get_handle() { return handle; }
19
20   std::coroutine_handle<promise_type> handle;
21 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Definition of coroutine/promise_type

Single-threaded scheduler

```
 5 struct Task {
 6
 7   struct promise_type {
 8     std::suspend_always initial_suspend() noexcept { return {}; }
 9     std::suspend_always final_suspend() noexcept { return {}; }
10
11     Task get_return_object() {
         return std::coroutine_handle<promise_type>::from_promise(*this);
       }
12     void return_void() {}
13     void unhandled_exception() {}
14   };
15
16   Task(std::coroutine_handle<promise_type> handle): handle{handle} {}
17
18   auto get_handle() { return handle; }
19
20   std::coroutine_handle<promise_type> handle;
21 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Scheduler Implementation

```
24 class Scheduler {
25
26   std::queue<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.front();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
```

Single-threaded scheduler

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle(): get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

# Scheduler Implementation

```
24 class Scheduler {
25
26   std::queue<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.front();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
```

Single-threaded scheduler

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle(): get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

# Scheduler Implementation

```
24 class Scheduler {
25
26   std::queue<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.front();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
```

Single-threaded scheduler

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle(): get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

# Scheduler Implementation

```
24 class Scheduler {
25
26   std::queue<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.front();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
```

Single-threaded scheduler

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle(): get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

# Scheduler Implementation

```
24 class Scheduler {
25
26   std::queue<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.front();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
```

Single-threaded scheduler

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle(): get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

# Results of Using Task Queue

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskA
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Queue

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskA
Hello from TaskB
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Queue

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskA
Hello from TaskB
Executing the TaskA
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Queue

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskA
Hello from TaskB
Executing the TaskA
Executing the TaskB
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Queue

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskA
Hello from TaskB
Executing the TaskA
Executing the TaskB
TaskA is finished
TaskB is finished
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Change from Queue to Stack

```
24 class Scheduler {
25
26   std::queue<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.front();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
49 };
```

Single-threaded scheduler

# Change from Queue to Stack

```
24 class Scheduler {
25
     // std::queue<std::coroutine_handle<>> _tasks;
26   std::stack<std::coroutine_handle<>> _tasks;
27
28   public:
29
30     void emplace(std::coroutine_handle<> task) {
31       _tasks.push(task);
32     }
33
34     void schedule() {
35       while(!_tasks.empty()) {
36         auto task = _tasks.top();
37         _tasks.pop();
38         task.resume();
39
40         if(!task.done()) {
41           _tasks.push(task);
42         }
43         else { task.destroy(); }
44       }
45     }
46     auto suspend() {
47       return std::suspend_always{};
48     }
49 };
```

Replace task queue with task stack

Single-threaded scheduler

# Results of Using Task Stack

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Stack

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskB
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Stack

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskB
Executing the TaskB
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Stack

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskB
Executing the TaskB
TaskB is finished
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Stack

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskB
Executing the TaskB
TaskB is finished
Hello from TaskA
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Results of Using Task Stack

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31 }
```

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

Single-threaded scheduler

Results

```
Start scheduling...
Hello from TaskB
Executing the TaskB
TaskB is finished
Hello from TaskA
Executing the TaskA
TaskA is finished
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/single-threaded

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Coroutine Tasks and Scheduler APIs

Multi-threaded scheduler

Coroutine tasks

```
 4 Task TaskA(Scheduler& sch) {
 5   std::cout << "Hello from TaskA\n";
 6   co_await sch.suspend();
 7   std::cout << "Executing the TaskA\n";
 8   co_await sch.suspend();
 9   std::cout << "TaskA is finished\n";
10 }
11
12 Task TaskB(Scheduler& sch) {
13   std::cout << "Hello from TaskB\n";
14   co_await sch.suspend();
15   std::cout << "Executing the TaskB\n";
16   co_await sch.suspend();
17   std::cout << "TaskB is finished\n";
18 }
```

emplace(…) : emplace a coroutine handle (task)
schedule(…) : schedule all emplaced tasks
get_handle(): get coroutine handle

```
21 int main() {
22
23   Scheduler sch;
24
25   sch.emplace(TaskA(sch).get_handle());
26   sch.emplace(TaskB(sch).get_handle());
27
28   std::cout << "Start scheduling...\n";
29
30   sch.schedule();
31   sch.wait();
32 }
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Definition of coroutine/promise_type

Multi-threaded scheduler

```
 5 struct Task {
 6
 7   struct promise_type {
 8     std::suspend_always initial_suspend() noexcept { return {}; }
 9     std::suspend_always final_suspend() noexcept { return {}; }
10
11     Task get_return_object() {
         return std::coroutine_handle<promise_type>::from_promise(*this);
       }
12     void return_void() {}
13     void unhandled_exception() {}
14   };
15
16   Task(std::coroutine_handle<promise_type> handle): handle{handle} {}
17
18   auto get_handle() { return handle; }
19
20   std::coroutine_handle<promise_type> handle;
21 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

_tasks : store all emplaced tasks
_pending_tasks : store tasks ready to resume()
_workers : store all threads
_mtx and _cv : block/unblock threads
_stop : signal threads to return
_finished : count finished tasks

_enqueue() : insert a task for execution
_process() : resume a task

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

_tasks : store all emplaced tasks
_pending_tasks : store tasks ready to resume()
_workers : store all threads
_mtx and _cv : block/unblock threads
_stop : signal threads to return
_finished : count finished tasks

_enqueue() : insert a task for execution
_process() : resume a task

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

_tasks : store all emplaced tasks
_pending_tasks : store tasks ready to resume()
_workers : store all threads
_mtx and _cv : block/unblock threads
_stop : signal threads to return
_finished : count finished tasks

_enqueue() : insert a task for execution
_process() : resume a task

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

_tasks : store all emplaced tasks
_pending_tasks : store tasks ready to resume()
_workers : store all threads
_mtx and _cv : block/unblock threads
_stop : signal threads to return
_finished : count finished tasks

_enqueue() : insert a task for execution
_process() : resume a task

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

_tasks : store all emplaced tasks
_pending_tasks : store tasks ready to resume()
_workers : store all threads
_mtx and _cv : block/unblock threads
_stop : signal threads to return
_finished : count finished tasks

_enqueue() : insert a task for execution
_process() : resume a task

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Member Variables/functions in Scheduler

_tasks : store all emplaced tasks
_pending_tasks : store tasks ready to resume()
_workers : store all threads
_mtx and _cv : block/unblock threads
_stop : signal threads to return
_finished : count finished tasks

_enqueue() : insert a task for execution
_process() : resume a task

Multi-threaded scheduler

```
30 class Scheduler {
31
32   public:
33
34     Scheduler(size_t num_threads);
35
36     void emplace(std::coroutine_handle<> task);
37     auto suspend();
38     void schedule();
39     void wait();
40
41   private:
42
43     void _enqueue(std::coroutine_handle<> task);
44     void _process(std::coroutine_handle<> task);
45
46     std::vector<std::coroutine_handle<>> _tasks;
47     std::queue<std::coroutine_handle<>> _pending_tasks;
48     std::vector<std::thread> _workers;
49
50     std::mutex _mtx;
51     std::condition_variable _cv;
52     bool _stop{false};
53     std::atomic<size_t> _finished{0};
54 };
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                       return _stop || (!_pending_tasks.empty());
                    });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                       return _stop || (!_pending_tasks.empty());
                    });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                       return _stop || (!_pending_tasks.empty());
                    });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                     return _stop || (!_pending_tasks.empty());
                  });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                       return _stop || (!_pending_tasks.empty());
                    });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                       return _stop || (!_pending_tasks.empty());
                    });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# Multi-threaded Scheduler Using Centralized Queue

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                       return _stop || (!_pending_tasks.empty());
                    });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

# The difference

```
56 Scheduler::Scheduler(size_t num_threads) {
57   _workers.reserve(num_threads);
58
59   for(size_t t = 0; t < num_threads; ++t) {
60     _workers.emplace_back([this]() {
61         while(true) {
62           std::coroutine_handle<> task;
63           {
64             std::unique_lock<std::mutex> lock(_mtx);
65             _cv.wait(lock, [this]{
                     return _stop || (!_pending_tasks.empty());
                  });
66             if(_stop) {
67               return;
68             }
69
70             task = _pending_tasks.front();
71             _pending_tasks.pop();
72           }
73           if(task) {
74             _process(task);
75           }
76         }
77       }
78     );
79   }
80 }
```

Multi-threaded scheduler

The difference between coroutine and function: task type

std::function<void()>

# Definition of _process() and _enqueue()

```
102 void Scheduler::_process(std::coroutine_handle<> task) {
103   task.resume();
104
105   if(!task.done()) {
106     _enqueue(task);
107   }
108   else {
109      task.destroy();
110     if(_finished.fetch_add(1) + 1 == _tasks.size()) {
111       {
112         std::unique_lock<std::mutex> lock(_mtx);
113         _stop = true;
114       }
115       _cv.notify_all();
116     }
117   }
118 }
119 void Scheduler::_enqueue(std::coroutine_handle<> task) {
120   {
121     std::unique_lock<std::mutex> lock(_mtx);
122     _pending_tasks.push(task);
123   }
124   _cv.notify_one();
125 }
```

Multi-threaded scheduler

- Resume a task
- If the task is not done, enqueue the task back to _pending_tasks
- If the task is done, increase _finished by one
  - Check if all tasks are finished

# Definition of _process() and _enqueue()

```
102 void Scheduler::_process(std::coroutine_handle<> task) {
103   task.resume();
104
105   if(!task.done()) {
106     _enqueue(task);
107   }
108   else {
109      task.destroy();
110     if(_finished.fetch_add(1) + 1 == _tasks.size()) {
111       {
112         std::unique_lock<std::mutex> lock(_mtx);
113         _stop = true;
114       }
115       _cv.notify_all();
116     }
117   }
118 }
119 void Scheduler::_enqueue(std::coroutine_handle<> task) {
120   {
121     std::unique_lock<std::mutex> lock(_mtx);
122     _pending_tasks.push(task);
123   }
124   _cv.notify_one();
125 }
```

Multi-threaded scheduler

- Enqueue a task and notify one of workers
- The queue is protected by a lock

# Definition of Scheduler APIs

Multi-threaded scheduler

```
82 void Scheduler::emplace(std::coroutine_handle<> task) {
83   _tasks.emplace_back(task);
84 }
85
86 void Scheduler::schedule() {
87   for(auto task: _tasks) {
88     _enqueue(task);
89   }
90 }
91
92 void Scheduler::wait() {
93   for(auto& w: _workers) {
94     w.join();
95   }
96 }
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/multi-threaded

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Definition of Coroutine Tasks and Scheduler API

CPU-GPU scheduler

```cpp
template <typename T>
__global__
void gpu_work() {
  // do some GPU work...
}

Task TaskA(Scheduler& sch) {

  std::cout << "Start TaskA\n";
  cudaStream_t stream;

  cudaStreamCreate(&stream);

  gpu_work<<<8, 256, 0, stream>>>();

  while(cudaStreamQuery(stream) != cudaSuccess) {
    co_await sch.suspend();
  }

  std::cout << "TaskA is finished\n";
  cudaStreamDestroy(stream);
}
```

```cpp
int main() {

  Scheduler sch;

  sch.emplace(TaskA(sch).get_handle());
  sch.emplace(TaskB(sch).get_handle());

  std::cout << "Start scheduling...\n";

  sch.schedule();
  sch.wait();

}
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/cpu-gpu

# Definition of Coroutine Tasks and Scheduler API

CPU-GPU scheduler

```cpp
template <typename T>
__global__
void gpu_work() {
  // do some GPU work...
}

Task TaskA(Scheduler& sch) {

  std::cout << "Start TaskA\n";

  cudaStream_t stream;

  cudaStreamCreate(&stream);

  gpu_work<<<8, 256, 0, stream>>>();

  while(cudaStreamQuery(stream) != cudaSuccess) {
    co_await sch.suspend();
  }

  std::cout << "TaskA is finished\n";

  cudaStreamDestroy(stream);
}
```

```cpp
int main() {

  Scheduler sch;

  sch.emplace(TaskA(sch).get_handle());
  sch.emplace(TaskB(sch).get_handle());

  std::cout << "Start scheduling...\n";

  sch.schedule();
  sch.wait();

}
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/cpu-gpu

# Definition of Coroutine Tasks and Scheduler API

CPU-GPU scheduler

```
 6 template <typename T>
 7 __global__
 8 void gpu_work() {
 9   // do some GPU work...
10 }
11
12 Task TaskA(Scheduler& sch) {
13
14   std::cout << "Start TaskA\n";
15
16   cudaStream_t stream;
17
18   cudaStreamCreate(&stream);
19
20   gpu_work<<<8, 256, 0, stream>>>();
21
22   while(cudaStreamQuery(stream) != cudaSuccess) {
23     co_await sch.suspend();
24   }
25
26   std::cout << "TaskA is finished\n";
27
28   cudaStreamDestroy(stream);
29 }
```

```
52 int main() {
53
54   Scheduler sch;
55
56   sch.emplace(TaskA(sch).get_handle());
57   sch.emplace(TaskB(sch).get_handle());
58
59   std::cout << "Start scheduling...\n";
60
61   sch.schedule();
62   sch.wait();
63
64 }
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/cpu-gpu

# Definition of Coroutine Tasks and Scheduler API

CPU-GPU scheduler

```cpp
template <typename T>
__global__
void gpu_work() {
  // do some GPU work...
}

Task TaskA(Scheduler& sch) {

  std::cout << "Start TaskA\n";

  cudaStream_t stream;

  cudaStreamCreate(&stream);

  gpu_work<<<8, 256, 0, stream>>>();

  while(cudaStreamQuery(stream) != cudaSuccess) {
    co_await sch.suspend();
  }

  std::cout << "TaskA is finished\n";

  cudaStreamDestroy(stream);
}
```

No need of another scheduler implementation!

```cpp
int main() {

  Scheduler sch;

  sch.emplace(TaskA(sch).get_handle());
  sch.emplace(TaskB(sch).get_handle());

  std::cout << "Start scheduling...\n";

  sch.schedule();
  sch.wait();

}
```

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/cpu-gpu

# Agenda

- **Learn what is coroutine**
- **Understand the motivation behind coroutine**
- **Dive into C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **Demonstrate scheduler implementation**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **Evaluate the scheduler on a CPU-GPU microbenchmark**

# Definition of Task

Task for without coroutine scheduler

- Both cpu_loop() and cuda_loop() will loop for certain time
- All tasks are independent

```
58 void wo_coro_work(
59   dim3 dim_grid, dim3 dim_block,
60   size_t BLOCK_SIZE, int cpu_ms, int gpu_ms
61 ) {
62   cpu_loop(cpu_ms);
63   cudaStream_t stream;
64   cudaStreamCreate(&stream);
65   cuda_loop<<<dim_grid, dim_block, 0, stream>>>(gpu_ms);
66   cudaStreamSynchronize(stream);
67   cudaStreamDestroy(stream);
68 }
```

# Definition of Task

Task for without coroutine scheduler

```
58 void wo_coro_work(
59   dim3 dim_grid, dim3 dim_block,
60   size_t BLOCK_SIZE, int cpu_ms, int gpu_ms
61 ) {
62   cpu_loop(cpu_ms);
63   cudaStream_t stream;
64   cudaStreamCreate(&stream);
65   cuda_loop<<<dim_grid, dim_block, 0, stream>>>(gpu_ms);
66   cudaStreamSynchronize(stream);
67   cudaStreamDestroy(stream);
68 }
```

- Both cpu_loop() and cuda_loop() will loop for certain time
- All tasks are independent

Task for with coroutine scheduler

```
44 cudaCoro::Task work(
45   cudaCoro::Scheduler& sch, dim3 dim_grid, dim3 dim_block,
46   size_t BLOCK_SIZE,  int cpu_ms, int gpu_ms
47 ) {
48   cpu_loop(cpu_ms);
49   cudaStream_t stream;
50   cudaStreamCreate(&stream);
51   cuda_loop<<<dim_grid, dim_block, 0, stream>>>(gpu_ms);
52   while(cudaStreamQuery(stream) != cudaSuccess) {
53     co_await sch.suspend();
54   }
55   cudaStreamDestroy(stream);
56 }
```

# Hardware Platform

- 4 CPU threads
  - 3.6 GHz
  - 32 GB memory
- 1 NVIDIA 2080 ti GPU
- Compiler
  - CUDA v12.0
  - g++ 12.2.1
  - -O3 enabled
- Comparision between scheduler with/without using coroutines

Code: https://github.com/dian-lun-lin/cpp_coroutine_examples/benchmark

# Experimental Results - Different #Tasks

cpu_loop(): 10ms, cuda_loop(): ~10ms

| | Without coroutine | With coroutine | Speedup |
|---|---|---|---|
| #Tasks=4 | 79ms | 80ms | 0x |
| | | | |

# Experimental Results - Different #Tasks

cpu_loop(): 10ms, cuda_loop(): ~10ms

| | Without coroutine | With coroutine | Speedup |
|---|---|---|---|
| #Tasks=4 | 79ms | 80ms | 0x |
| | | | |

Take-home message:
- There is certain cost of using coroutines!

# Experimental Results - Different #Tasks

cpu_loop(): 10ms, cuda_loop(): ~10ms

| | Without coroutine | With coroutine | Speedup |
|---|---|---|---|
| #Tasks=4 | 79ms | 80ms | 0x |
| #Tasks=16 | 139ms | 114ms | 1.2x |

Take-home message:
- There is certain cost of using coroutines!

# Experimental Results - Different #Tasks

cpu_loop(): 10ms, cuda_loop(): ~10ms

| | Without coroutine | With coroutine | Speedup |
|---|---|---|---|
| #Tasks=4 | 79ms | 80ms | 0x |
| #Tasks=16 | 139ms | 114ms | 1.2x |
| #Tasks=64 | 382ms | 238ms | 1.6x |

Take-home message:
- There is certain cost of using coroutines!

# Experimental Results - Different #Tasks

cpu_loop(): 10ms, cuda_loop(): ~10ms

| | Without coroutine | With coroutine | Speedup |
|---|---|---|---|
| #Tasks=4 | 79ms | 80ms | 0x |
| #Tasks=16 | 139ms | 114ms | 1.2x |
| #Tasks=64 | 382ms | 238ms | 1.6x |
| #Tasks=256 | 1355ms | 725ms | 1.9x |

Take-home message:
- There is certain cost of using coroutines!

# Experimental Results - Different #Tasks

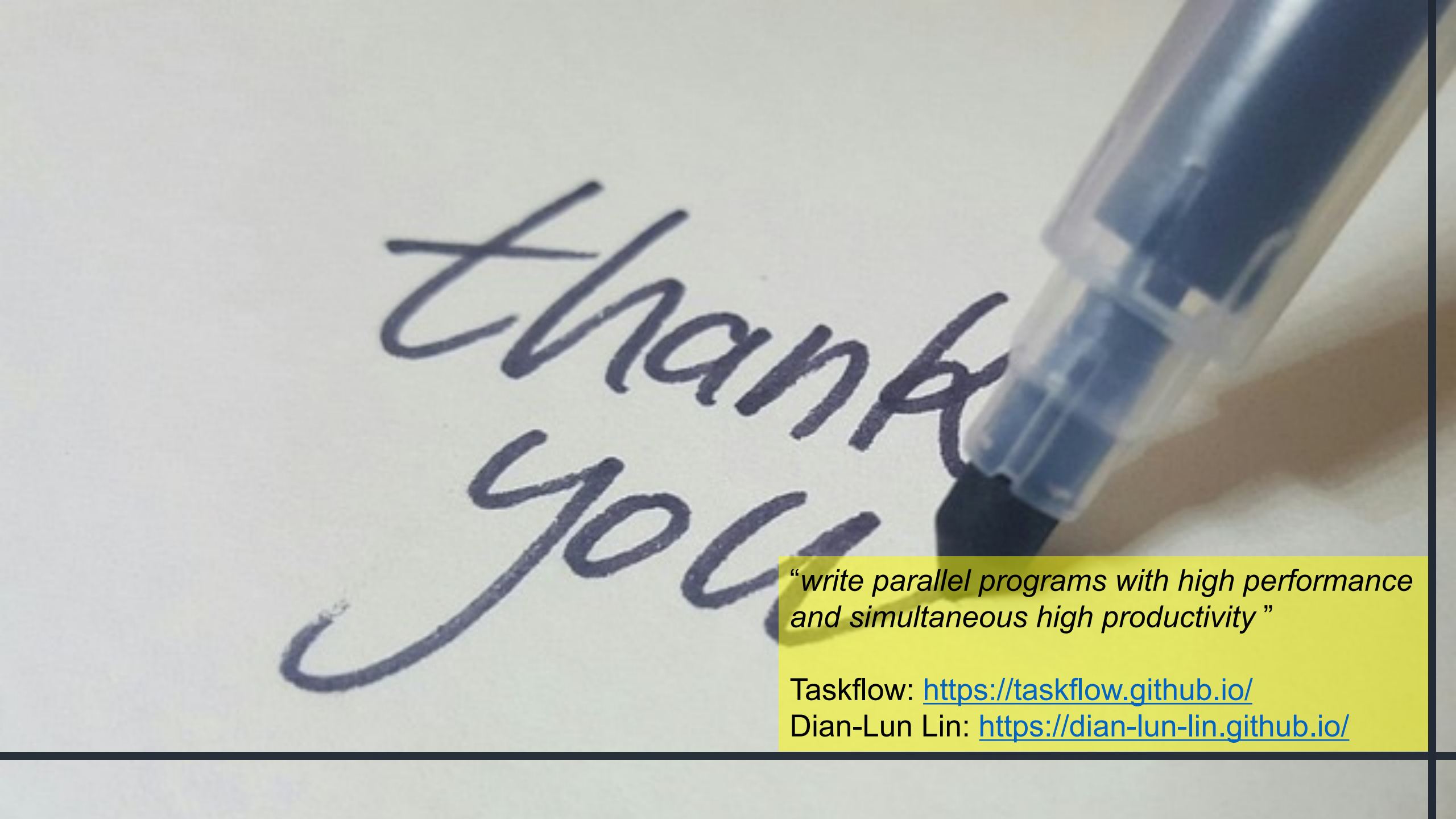
cpu_loop(): 10ms, cuda_loop(): ~10ms

| | Without coroutine | With coroutine | Speedup |
|---|---|---|---|
| #Tasks=4 | 79ms | 80ms | 0x |
| #Tasks=16 | 139ms | 114ms | 1.2x |
| #Tasks=64 | 382ms | 238ms | 1.6x |
| #Tasks=256 | 1355ms | 725ms | 1.9x |

Take-home message:
- There is certain cost of using coroutines!
- More tasks = more overlaps

# Closing

- **We have presented what is coroutine**
- **We have presented the motivation behind coroutine**
  - **Coroutine is very useful for heterogenous computing**
- **We have presented four components of C++ coroutine**
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle
- **We have presented three scheduler implementations**
  - Single-threaded
  - Multi-threaded
  - CPU-GPU
- **We have presented performance of scheduler using C++ coroutine**




*"write parallel programs with high performance and simultaneous high productivity"*

Taskflow: https://taskflow.github.io/
Dian-Lun Lin: https://dian-lun-lin.github.io/