# Taro:

## Task graph-based Asynchronous Programming Using C++ Coroutine

DIAN-LUN LIN


# Agenda

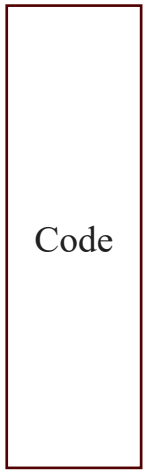
- Understand the motivation behind Taro
- Learn to use the Taro C++ programming model
- Dive into the Taro’s coroutine-aware scheduling algorithm
- Evaluate Taro on microbenchmarks and a real-world application
- Conclusion

# Agenda

- **Understand the motivation behind Taro**
- Learn to use the Taro C++ programming model
- Dive into the Taro's coroutine-aware scheduling algorithm
- Evaluate Taro on microbenchmarks and a real-world application
- Conclusion

# What is Task Graph-based Programming System (TGPS)

- TGPS encapsulates function calls and their dependencies in a top-down task graph

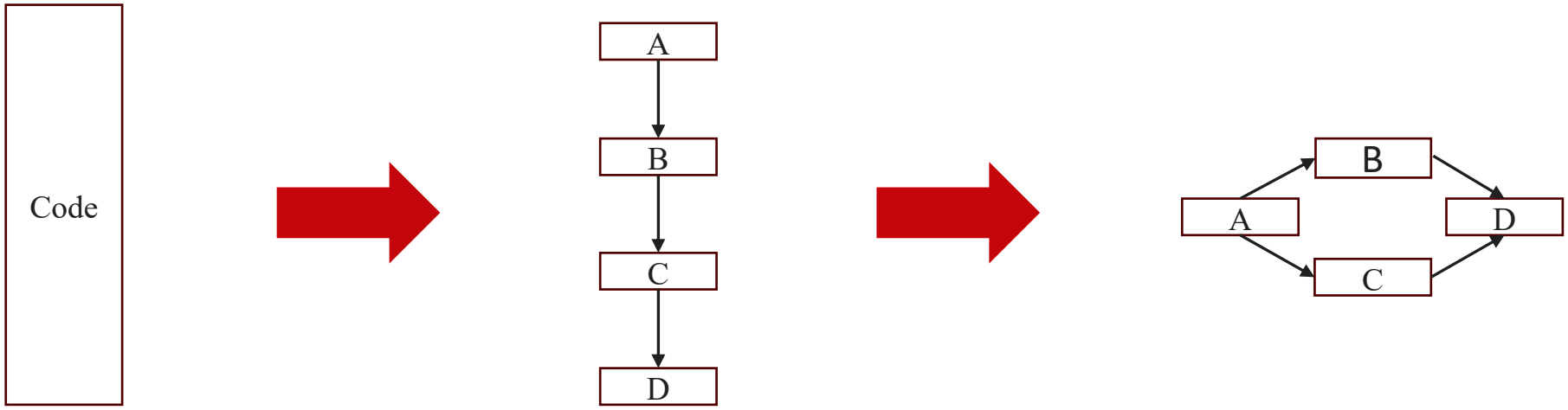


# What is Task Graph-based Programming System (TGPS)

- TGPS encapsulates function calls and their dependencies in a top-down task graph

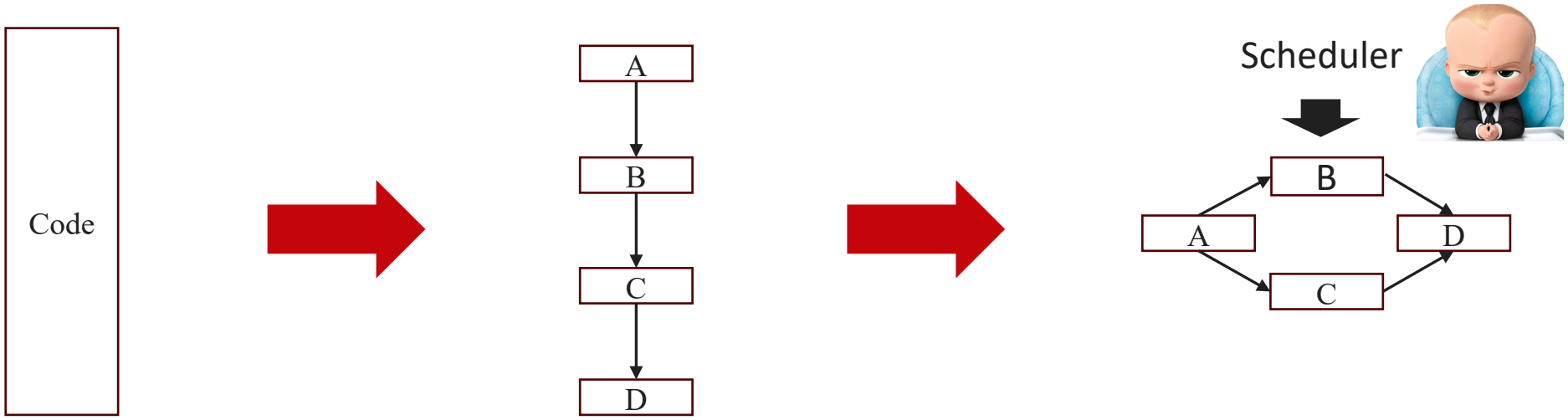
# What is Task Graph-based Programming System (TGPS)

- TGPS encapsulates function calls and their dependencies in a top-down task graph

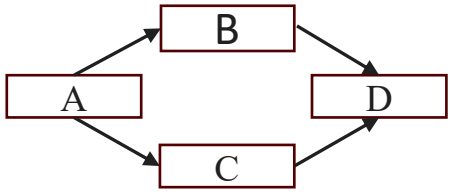


# What is Task Graph-based Programming System (TGPS)

- TGPS encapsulates function calls and their dependencies in a top-down task graph



# Why TGPS

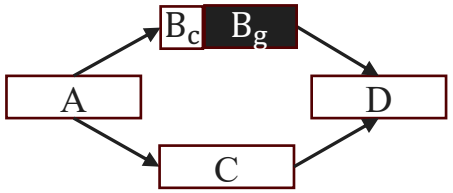



1. Easy to write and express a task graph
2. Allow to implement irregular parallel  decomposition strategies

```
 1 Scheduler sched;
 2 task_a = sched.emplace([](&){
 3   // Code block A;
 4 });
 5 task_b = sched.emplace([](&){
 6   // Code block B;
 7 });
 8 task_c = sched.emplace([](&){
 9   // Code block C;
10 });
11 task_d = sched.emplace([](&){
12   // Code block D;
13 });
14
```

```
15 task_a.precede(task_b);
16 task_a.precede(task_c);
17 task_b.precede(task_d);
18 task_c.precede(task_d);
19
20 sched.schedule();
21 sched.wait();
```

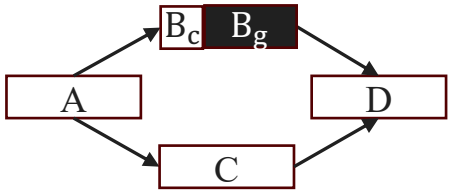
# Existing TGPSs on Heterogenous Computing - Challenge




$B_c$ : CPU operation
$B_g$ : GPU operation

```
 1 Scheduler sched;
 2 task_a = sched.emplace([](&){
 3   // Code block A;
 4 });
 5 task_b = sched.emplace([](&){
 6   // Code block B;
 7 });
 8 task_c = sched.emplace([](&){
 9   // Code block C;
10 });
11 task_d = sched.emplace([](&){
12   // Code block D;
13 });
14
```

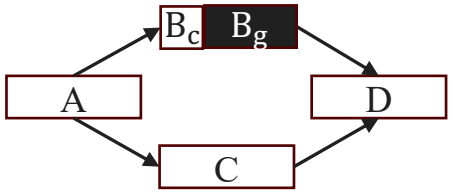
# Existing TGPSs on Heterogenous Computing - Challenge




$B_c$ : CPU operation
$B_g$ : GPU operation

```
1 Scheduler sched;
2 task_a = sched.emplace([](&){
3   // Code block A;
4 });
5 task_b = sched.emplace([](&){
6   // CPU code;
    // GPU code;
7 }); // CPU thread blocks until GPU finishes
8 task_c = sched.emplace([](&){
9   // Code block C;
10 });
11 task_d = sched.emplace([](&){
12   // Code block D;
13 });
14
```

# Existing TGPSs on Heterogenous Computing - Challenge



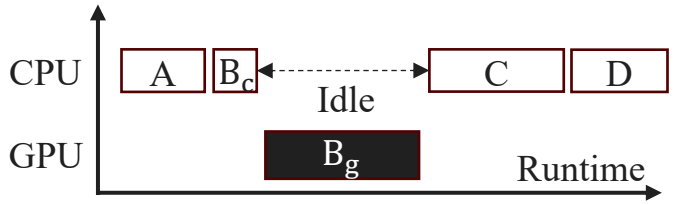

$B_c$ : CPU operation
$B_g$ : GPU operation

```
1 Scheduler sched;
2 task_a = sched.emplace([](&){
3   // Code block A;
4 });
5 task_b = sched.emplace([](&){
6   // CPU code;
    // GPU code;
7 }); // CPU thread blocks until GPU finishes
8 task_c = sched.emplace([](&){
9   // Code block C;
10 });
11 task_d = sched.emplace([](&){
12   // Code block D;
13 });
14
```

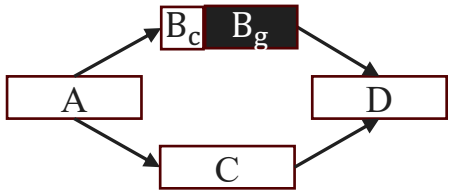
Atomic execution per task

# Existing TGPSs on Heterogenous Computing - Challenge

Assume one CPU and one GPU

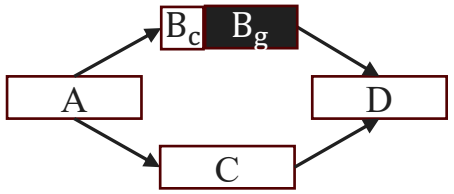



$B_c$ : CPU operation
$B_g$ : GPU operation

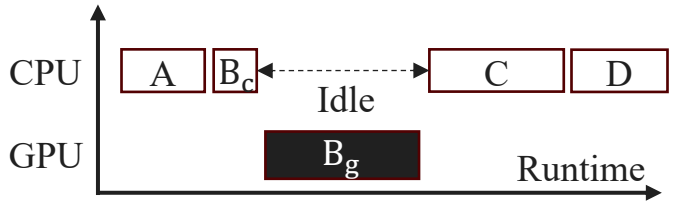


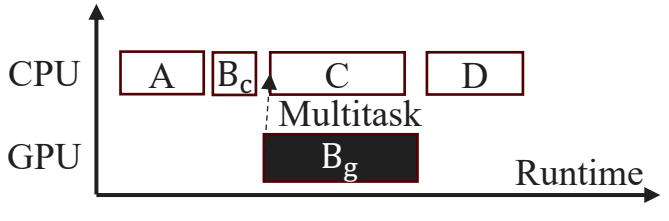
# Existing TGPSs on Heterogenous Computing - Challenge

$B_c$ : CPU operation
$B_g$ : GPU operation

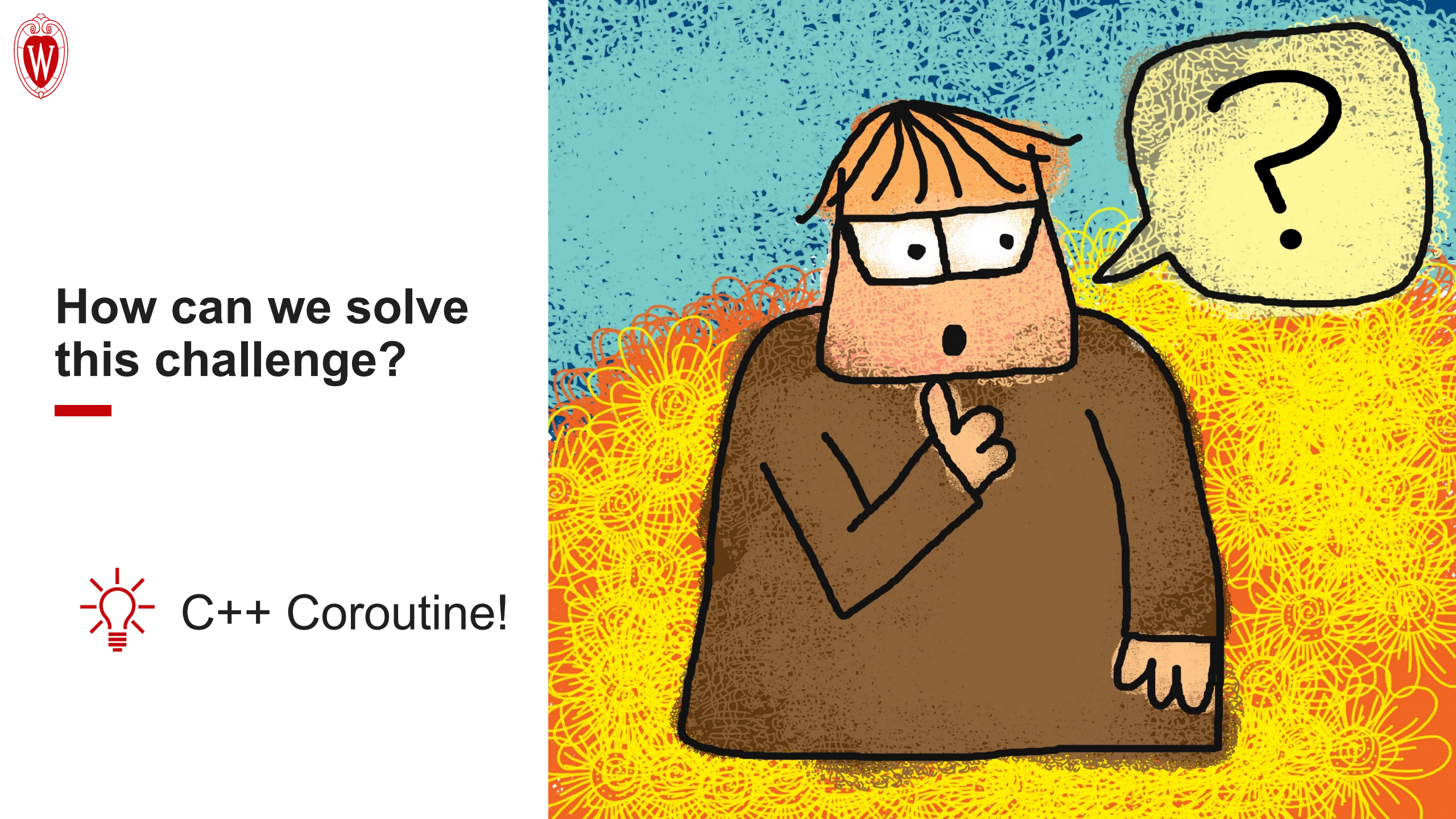



Assume one CPU and one GPU




# How can we solve this challenge?



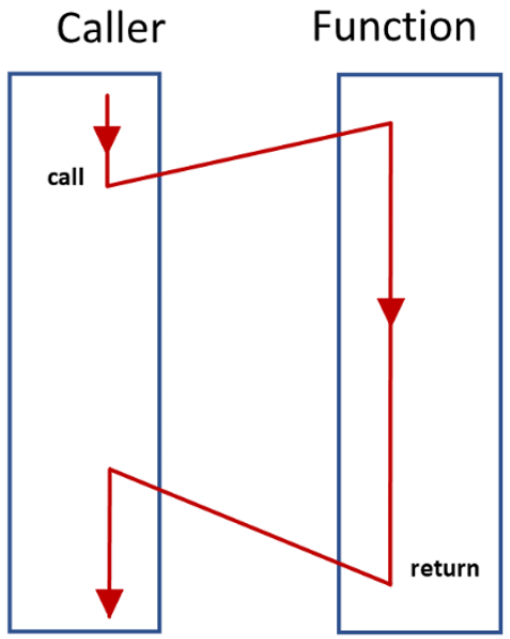
# How can we solve this challenge?

C++ Coroutine!

# 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



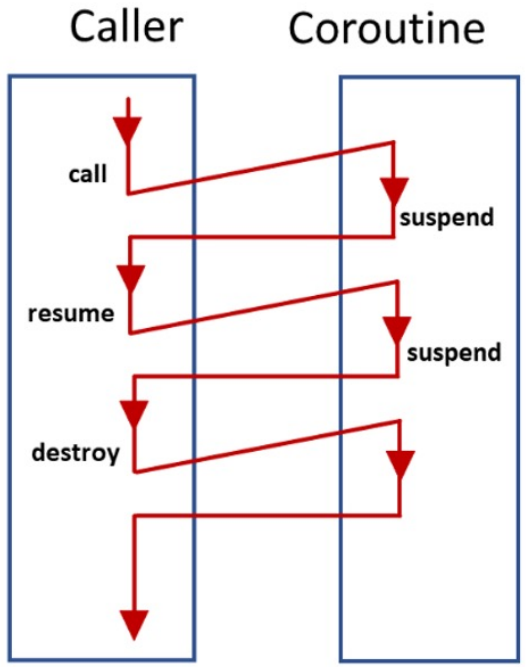




Image credit: Modernes C++, Rainer Grimm

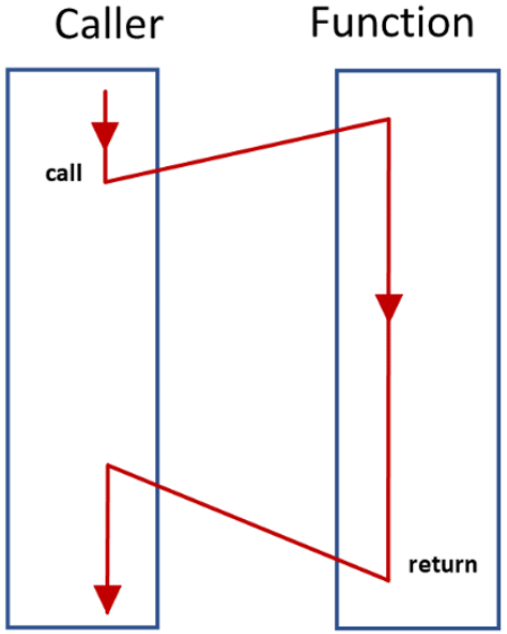
# Why Coroutine

- 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…

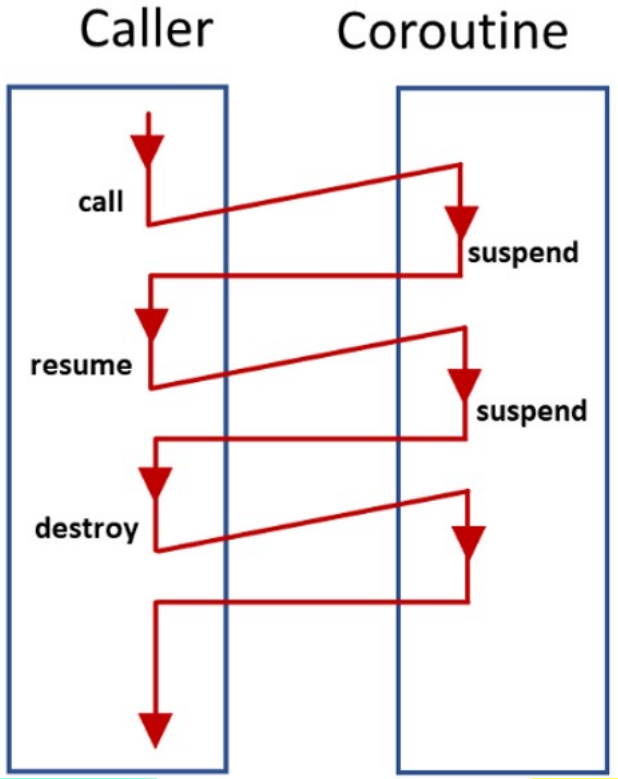
# Why Coroutine

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

1. Boil the water

2. Take a shower

Multitask to take a shower




turn on the stove

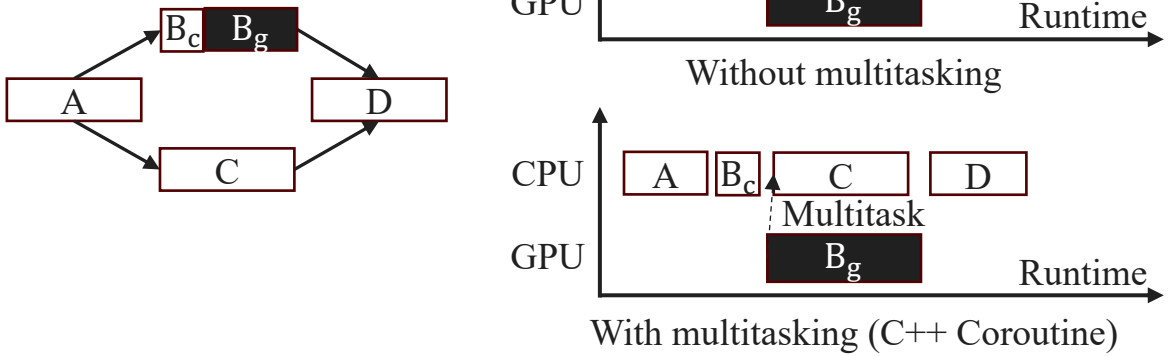
turn off the stove

enjoy your hot water

There is an overlap!

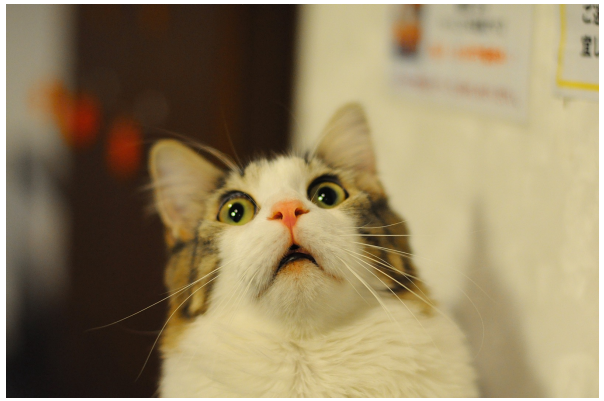
# Why Coroutine

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



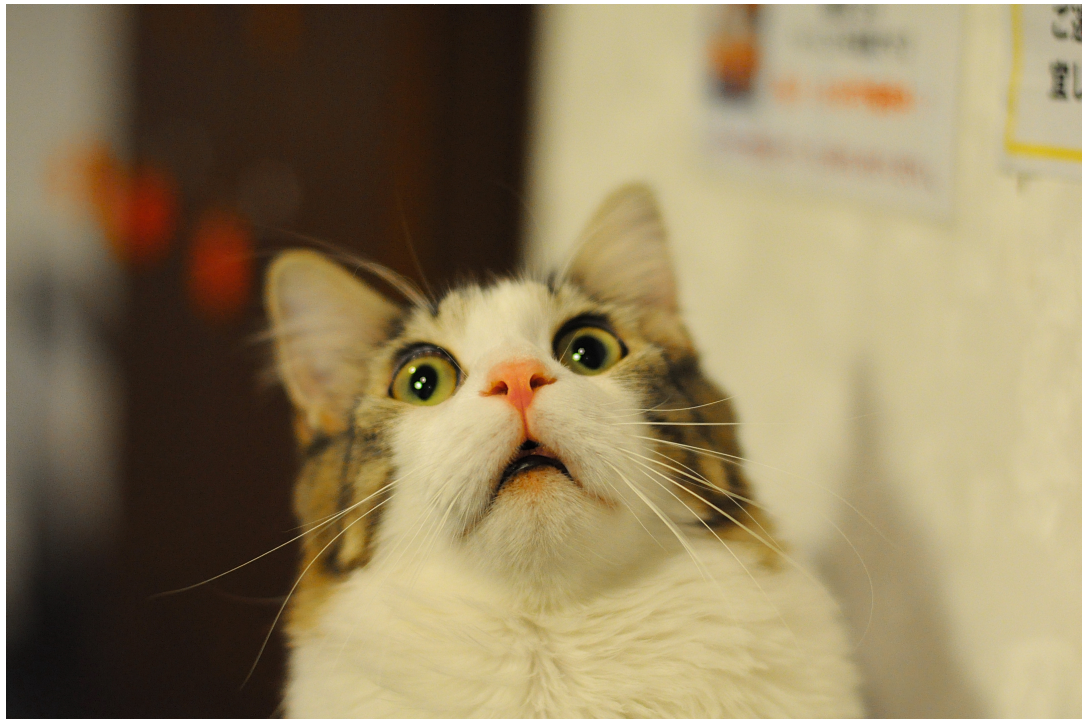
# Why Not Coroutine…

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



# Why Not Coroutine…

- Implementing a C++ function requires function itself
- But implementing a C++ coroutine requires:
  - Coroutine
  - Promise
  - Awaitable
  - Coroutine handle



# To I

void awai
bool awai
std::corou

# A new heterogeneous TGPS using C++ Coroutine?

Abstract away C++ Coroutine details

# A new heterogeneous TGPS using C++ Coroutine?

Abstract away C++ Coroutine details

*Taro!*

# Agenda

- Understand the motivation behind Taro
- **Learn to use the Taro C++ programming model**
- Dive into the Taro's coroutine-aware scheduling algorithm
- Evaluate Taro on microbenchmarks and a real-world application
- Conclusion

# Synchronous/Asynchronous Mechanisms

Synchronous

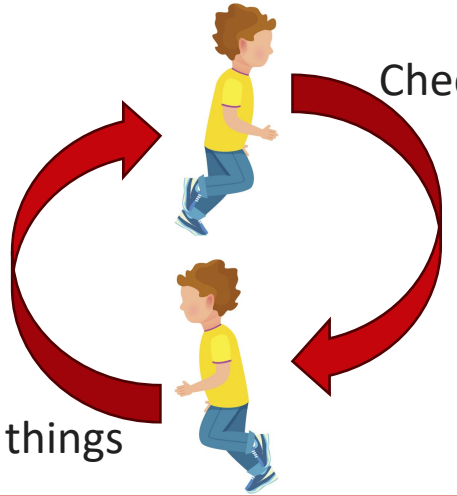
Wait

Boil the water

Multitasking!

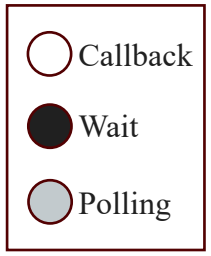
Asynchronous

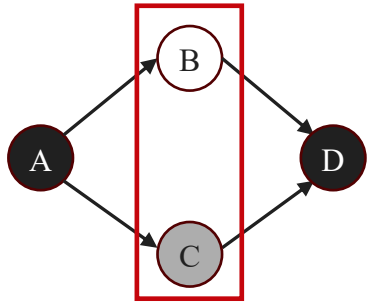
Polling




Callback

# Taro's Programming Model – Example 1

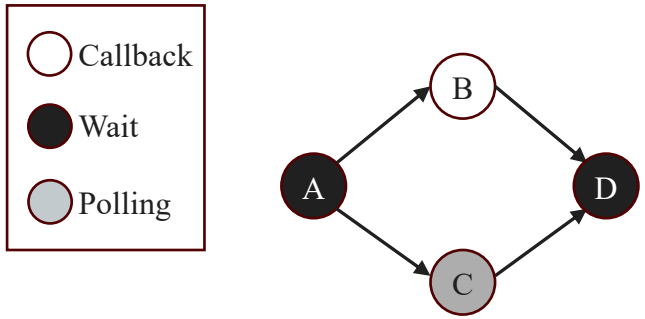



C++ Coroutine to enable multitasking

Taro: https://github.com/dian-lun-lin/taro

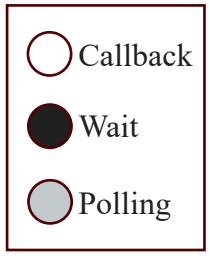
# Taro's Programming Model – Example 1

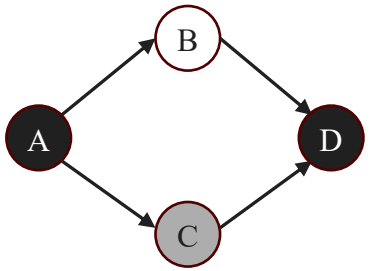



```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1

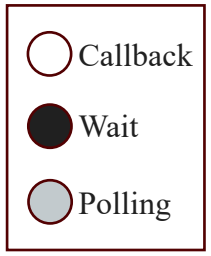


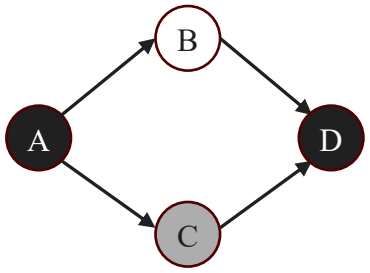




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1

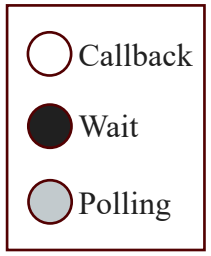


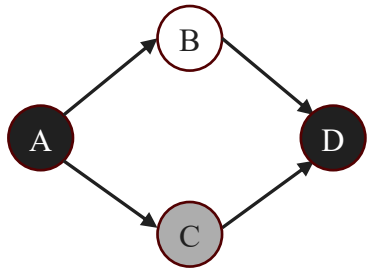




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1



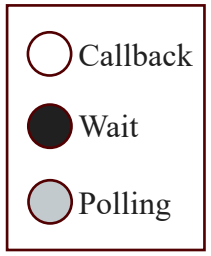



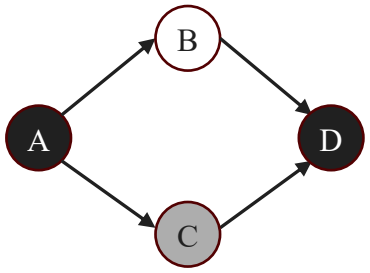

```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

CUDA stream for offloading GPU kernels

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model



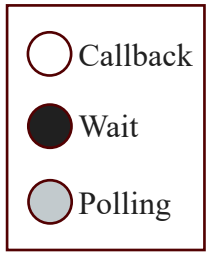



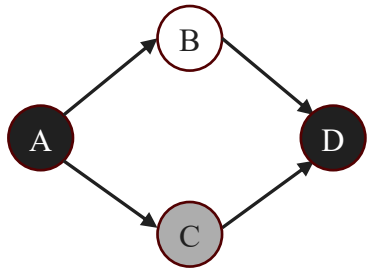

```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Offloaded GPU kernel

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1

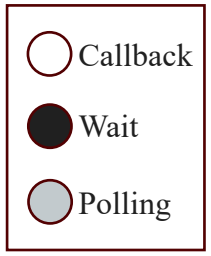


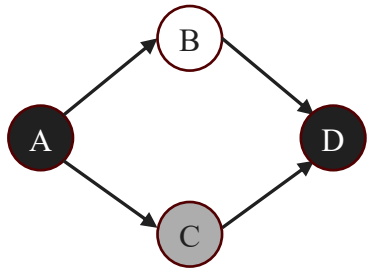




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1



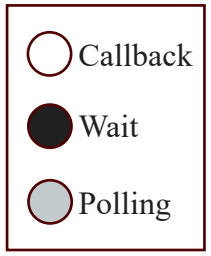



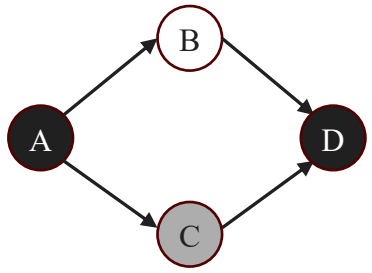

```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

C++ Coroutine

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1

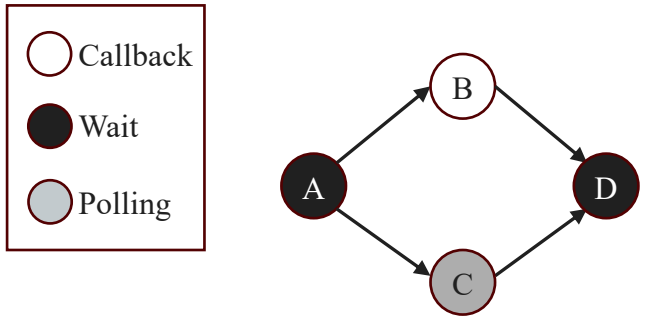



```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1

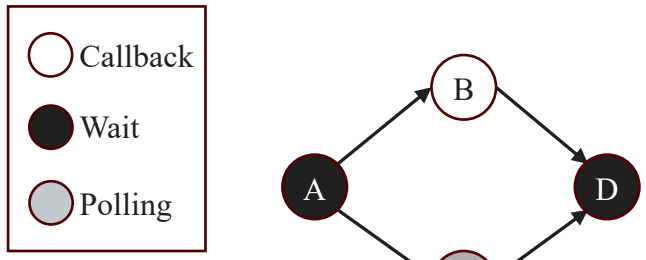



```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

CUDA stream for offloading GPU kernels

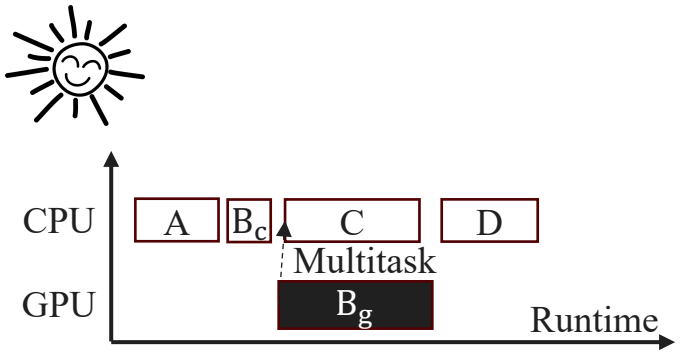
Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1





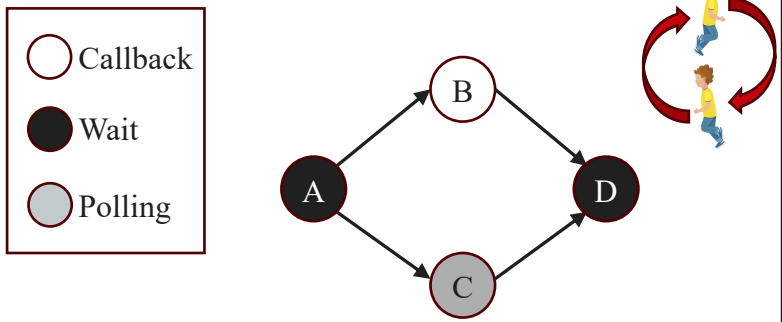





```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() {
  cuda.wait([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  cpu_work_b1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
```

Taro: https://github.com/dian-lun-lin/taro

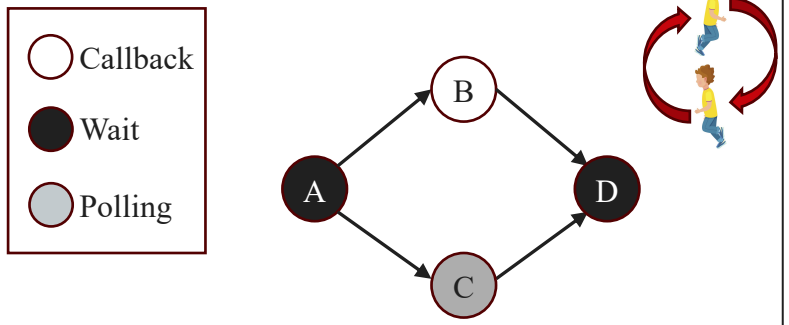
# Taro's Programming Model – Example 1




```
19 auto task_c = taro.emplace([&]() -> taro::Coro  {
20   co_await cuda.suspend_polling([&](cudaStream_t stream) {
21     kernel_c1<<<32, 256, 0, stream>>>();
22   }); // suspend and multitask
23   cpu_work_c1(); // execute cpu_work_c1 after kernel_c1 finish
24 });
25 auto task_d = taro.emplace([&]() {
26   cuda.wait([&](cudaStream_t stream) {
27     kernel_d1<<<32, 256, 0, stream>>>();
28   }); // synchronize
29 });
30
31 task_a.precede(task_b);
32 task_a.precede(task_c);
33 task_b.precede(task_d);
34 task_c.precede(task_d);
35
36 taro.schedule();
37 taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

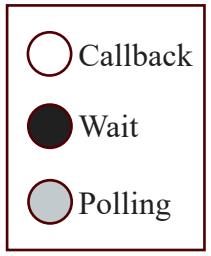
# Taro's Programming Model – Example 1

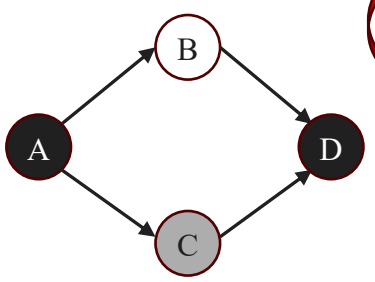


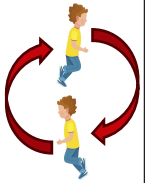

```
19 auto task_c = taro.emplace([&]() -> taro::Coro  {
20   co_await cuda.suspend_polling([&](cudaStream_t stream) {
21     kernel_c1<<<32, 256, 0, stream>>>();
22   }); // suspend and multitask
23   cpu_work_c1(); // execute cpu_work_c1 after kernel_c1 finish
24 });
25 auto task_d = taro.emplace([&]() {
26   cuda.wait([&](cudaStream_t stream) {
27     kernel_d1<<<32, 256, 0, stream>>>();
28   }); // synchronize
29 });
30
31 task_a.precede(task_b);
32 task_a.precede(task_c);
33 task_b.precede(task_d);
34 task_c.precede(task_d);
35
36 taro.schedule();
37 taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1



```
19 auto task_c = taro.emplace([&]() -> taro::Coro  {
20   co_await cuda.suspend_polling([&](cudaStream_t stream) {
21     kernel_c1<<<32, 256, 0, stream>>>();
22   }); // suspend and multitask
23   cpu_work_c1(); // execute cpu_work_c1 after kernel_c1 finish
24 });
25 auto task_d = taro.emplace([&]() {
26   cuda.wait([&](cudaStream_t stream) {
27     kernel_d1<<<32, 256, 0, stream>>>();
28   }); // synchronize
29 });
30
31 task_a.precede(task_b);
32 task_a.precede(task_c);
33 task_b.precede(task_d);
34 task_c.precede(task_d);
35
36 taro.schedule();
37 taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 1

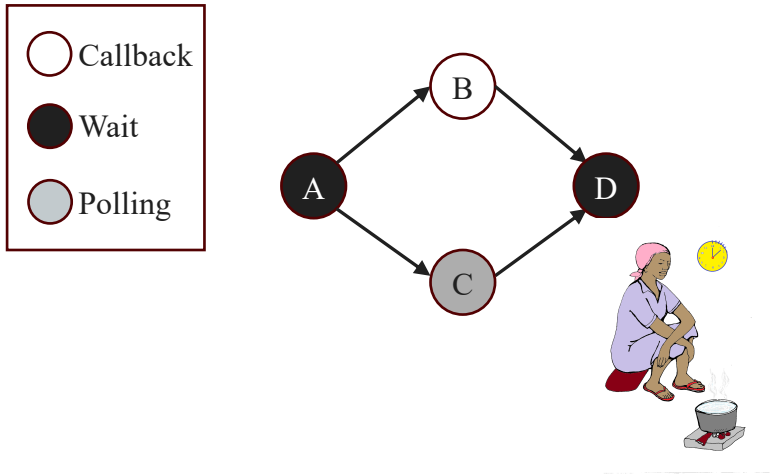
```
1 auto task_c = taro.emplace([&]() {
2   kernel_c1<<<32, 256, 0, stream>>>();
3   cudaStreamSynchronize(stream);
3   cpu_work_c1();
4 });
```

Guaranteed same results but no need of synchronization

```
19 auto task_c = taro.emplace([&]() -> taro::Coro  {
20   co_await cuda.suspend_polling([&](cudaStream_t stream) {
21     kernel_c1<<<32, 256, 0, stream>>>();
22   }); // suspend and multitask
23   cpu_work_c1(); // execute cpu_work_c1 after kernel_c1 finish
24 });
25 auto task_d = taro.emplace([&]() {
26   cuda.wait([&](cudaStream_t stream) {
27     kernel_d1<<<32, 256, 0, stream>>>();
28   }); // synchronize
29 });
30
31 task_a.precede(task_b);
32 task_a.precede(task_c);
33 task_b.precede(task_d);
34 task_c.precede(task_d);
35
36 taro.schedule();
37 taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

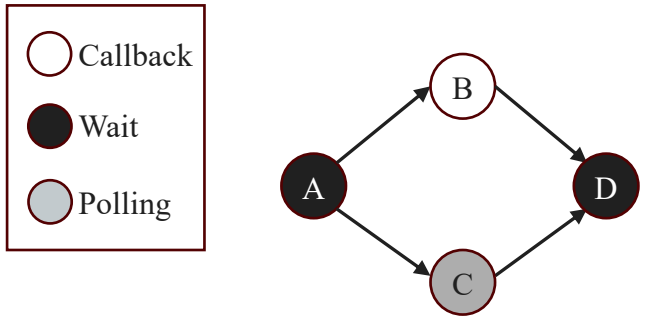
# Taro's Programming Model – Example 1




```
19 auto task_c = taro.emplace([&]() -> taro::Coro  {
20   co_await cuda.suspend_polling([&](cudaStream_t stream) {
21     kernel_c1<<<32, 256, 0, stream>>>();
22   }); // suspend and multitask
23   cpu_work_c1(); // execute cpu_work_c1 after kernel_c1 finish
24 });
25 auto task_d = taro.emplace([&]() {
26   cuda.wait([&](cudaStream_t stream) {
27     kernel_d1<<<32, 256, 0, stream>>>();
28   }); // synchronize
29 });
30
31 task_a.precede(task_b);
32 task_a.precede(task_c);
33 task_b.precede(task_d);
34 task_c.precede(task_d);
35
36 taro.schedule();
37 taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

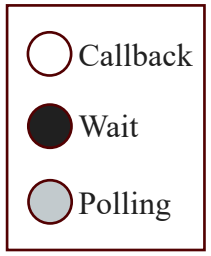
# Taro's Programming Model – Example 1

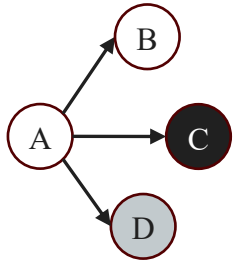



```
19 auto task_c = taro.emplace([&]() -> taro::Coro  {
20   co_await cuda.suspend_polling([&](cudaStream_t stream) {
21     kernel_c1<<<32, 256, 0, stream>>>();
22   }); // suspend and multitask
23   cpu_work_c1(); // execute cpu_work_c1 after kernel_c1 finish
24 });
25 auto task_d = taro.emplace([&]() {
26   cuda.wait([&](cudaStream_t stream) {
27     kernel_d1<<<32, 256, 0, stream>>>();
28   }); // synchronize
29 });
30
31 task_a.precede(task_b);
32 task_a.precede(task_c);
33 task_b.precede(task_d);
34 task_c.precede(task_d);
35
36 taro.schedule();
37 taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2

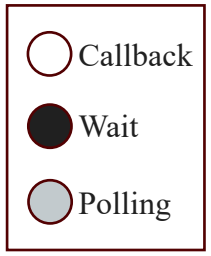


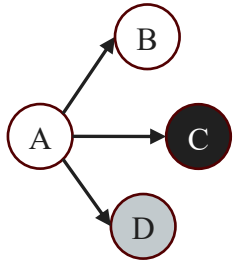




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
  cpu_work_b1(); // execute cpu_work_b1 after kernel_b1, b2 finish
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2

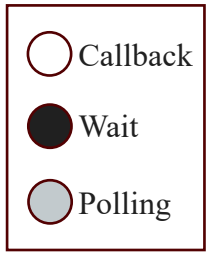


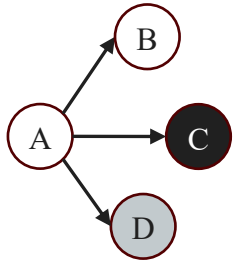




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
  cpu_work_b1(); // execute cpu_work_b1 after kernel_b1, b2 finish
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
  cpu_work_b1(); // execute cpu_work_b1 after kernel_b1, b2 finish
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2

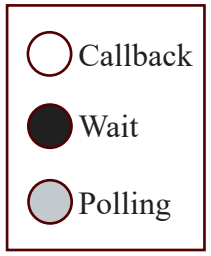
Guaranteed same results but no need of synchronization

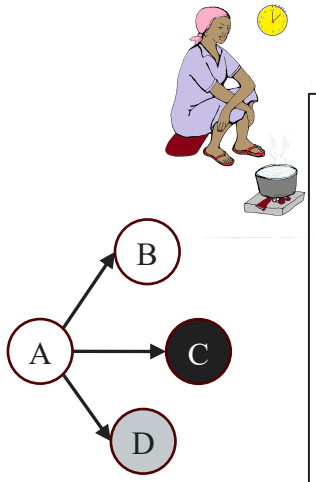
```
auto task_b = taro.emplace([&]() {
  kernel_b1<<<32, 256, 0, stream>>>();
  kernel_b2<<<32, 256, 0, stream>>>();
  cudaStreamSynchronize(stream);
  cpu_work_b1();
});
```

```
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
  cpu_work_b1(); // execute cpu_work_b1 after kernel_b1, b2 finish
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2

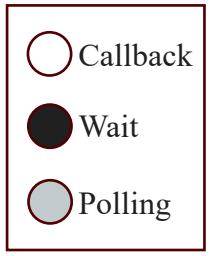


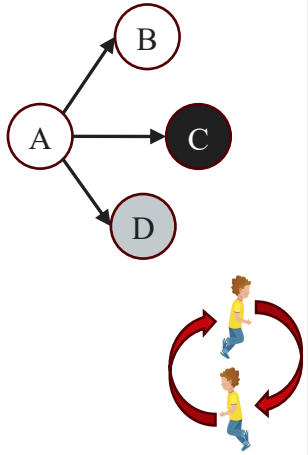




```cpp
auto task_c = taro.emplace([&]() {
  cpu_work_c1();
  cuda.wait([&](cudaStream_t stream) {
    kernel_c1<<<32, 256, 0, stream>>>();
  }); // synchronize
});
auto task_d = taro.emplace([&]() -> taro::Coro {
  cuda.suspend_polling([&](cudaStream_t stream) {
    kernel_d1<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
  cpu_work_d1(); // execute cpu_work_d1 after kernel_d1 finish
});

task_a.precede(task_b);
task_a.precede(task_c);
task_a.precede(task_d);

taro.schedule();
taro.wait();

```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2

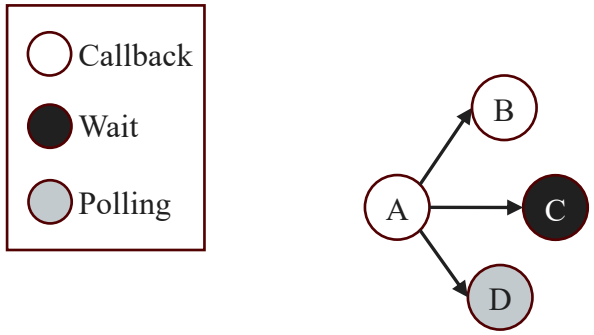



```
21 auto task_c = taro.emplace([&]() {
22   cpu_work_c1();
23   cuda.wait([&](cudaStream_t stream) {
24     kernel_c1<<<32, 256, 0, stream>>>();
25   }); // synchronize
26 });
27 auto task_d = taro.emplace([&]() -> taro::Coro {
28   cuda.suspend_polling([&](cudaStream_t stream) {
29     kernel_d1<<<32, 256, 0, stream>>>();
30   }); // suspend and multitask
31   cpu_work_d1(); // execute cpu_work_d1 after kernel_d1 finishes
32 });
33
34 task_a.precede(task_b);
35 task_a.precede(task_c);
36 task_a.precede(task_d);
37
38 taro.schedule();
39 taro.wait();
40
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model – Example 2







```
21 auto task_c = taro.emplace([&]() {
22   cpu_work_c1();
23   cuda.wait([&](cudaStream_t stream) {
24     kernel_c1<<<32, 256, 0, stream>>>();
25   }); // synchronize
26 });
27 auto task_d = taro.emplace([&]() -> taro::Coro {
28   cuda.suspend_polling([&](cudaStream_t stream) {
29     kernel_d1<<<32, 256, 0, stream>>>();
30   }); // suspend and multitask
31   cpu_work_d1(); // execute cpu_work_d1 after kernel_d1 finish
32 });
33
34 task_a.precede(task_b);
35 task_a.precede(task_c);
36 task_a.precede(task_d);
37
38 taro.schedule();
39 taro.wait();
40
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model - Extendibility

```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  }); // suspend and multitask
  cpu_work_b1(); // execute cpu_work_b1 after kernel_b1, b2 finish
});
```

Taro: https://github.com/dian-lun-lin/taro

# Taro’s Programming Model - Extendibility

```
1 #include <taro/taro.hpp>
2
3 #include <taro/scheduler/custom.hpp> // support custom accelerator
4
5 taro::Taro taro{NUM_THREADS};
6 auto custom = taro.custom_scheduler(…);
```

Taro: https://github.com/dian-lun-lin/taro

# Taro’s Programming Model - Extendibility

```
1  #include <taro/taro.hpp>
2
3  #include <taro/scheduler/custom.hpp> // support custom accelerator
4
5  taro::Taro taro{NUM_THREADS};
6  auto custom = taro.custom_scheduler(…);
7
8
9  auto task_a = taro.emplace([&]() -> taro::Coro {
10   custom.suspend_callback([&](…) {
11      // custom accelerator operations
12   });
13 });
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Programming Model - Extendibility

```
1  #include <taro/taro.hpp>
2  #include <taro/scheduler/cuda.hpp>
3  #include <taro/scheduler/custom.hpp> // support custom accelerator
4
5  taro::Taro taro{NUM_THREADS};
6  auto custom = taro.custom_scheduler(…);
7  auto cuda = taro.cuda_scheduler(NUM_STREAMS);
8
9  auto task_a = taro.emplace([&]() -> taro::Coro {
10   custom.suspend_callback([&](…) {
11      // custom accelerator operations
12   });
13 });
14 auto task_b = taro.emplace([&]() {
15   cuda.wait([&](…) {
16      // GPU operations
17   });
18 });
19 task_a.precede(task_b);
```

Taro: https://github.com/dian-lun-lin/taro

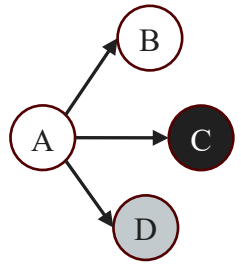
# Taro’s goal

- Simplicity
- Efficiency
- Extendibility

# Agenda

- Understand the motivation behind Taro
- Learn to use the Taro C++ programming model
- **Dive into the Taro's coroutine-aware scheduling algorithm**
- Evaluate Taro on microbenchmarks and a real-world application
- Conclusion

# Taro's Scheduler – Example 2

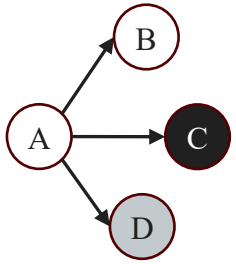


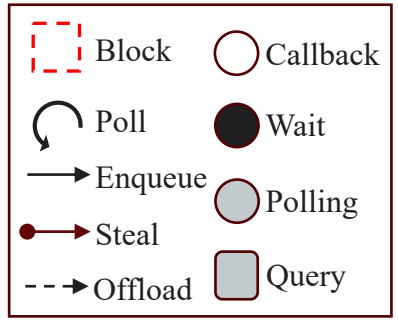

```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  });
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  });
  cpu_work_b1();
});
```

```cpp
auto task_c = taro.emplace([&](cudaStream_t stream) {
  cpu_work_c1();
  cuda.wait([&](){
    kernel_c1<<<32, 256, 0, stream>>>();
  });
});
auto task_d = taro.emplace([&]() {
  cuda.suspend_polling([&](cudaStream_t stream) {
    kernel_d1<<<32, 256, 0, stream>>>();
  });
  cpu_work_d1();
});

task_a.precede(task_b);
task_a.precede(task_c);
task_a.precede(task_d);

taro.schedule();
taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler



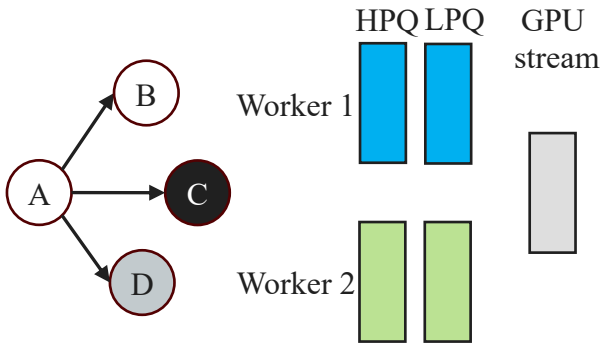



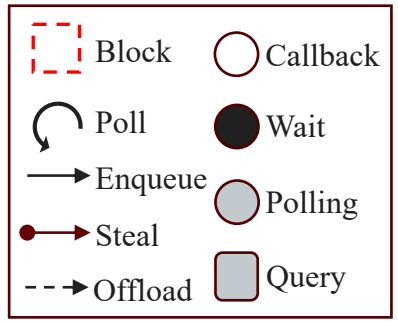

Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks

We want to resume a suspended task by the same worker as soon as possible to avoid another worker stealing the task

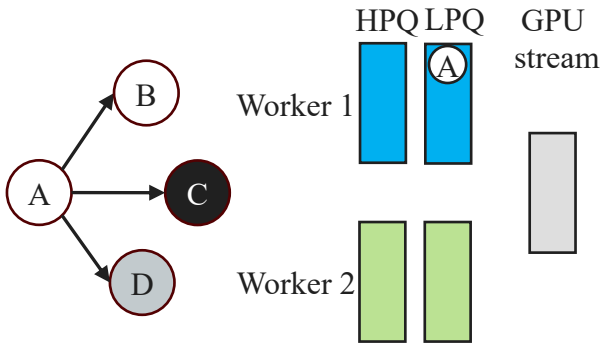


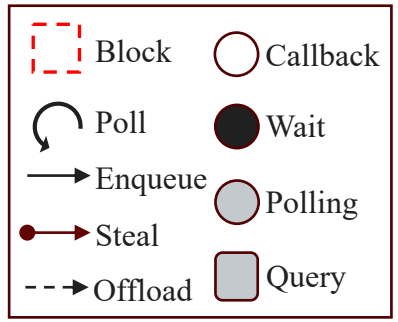




Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks

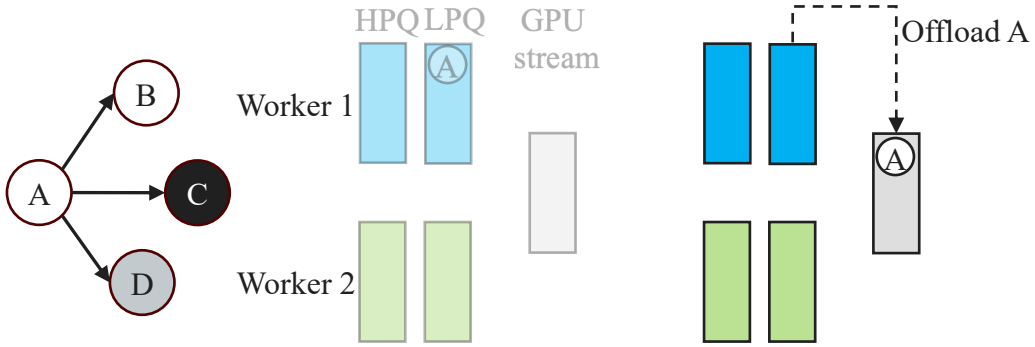


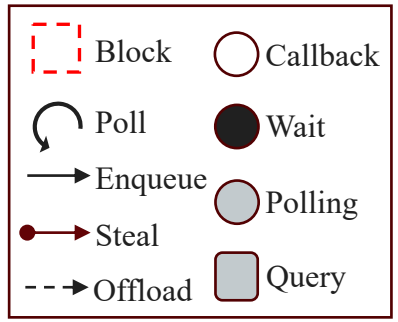

Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks



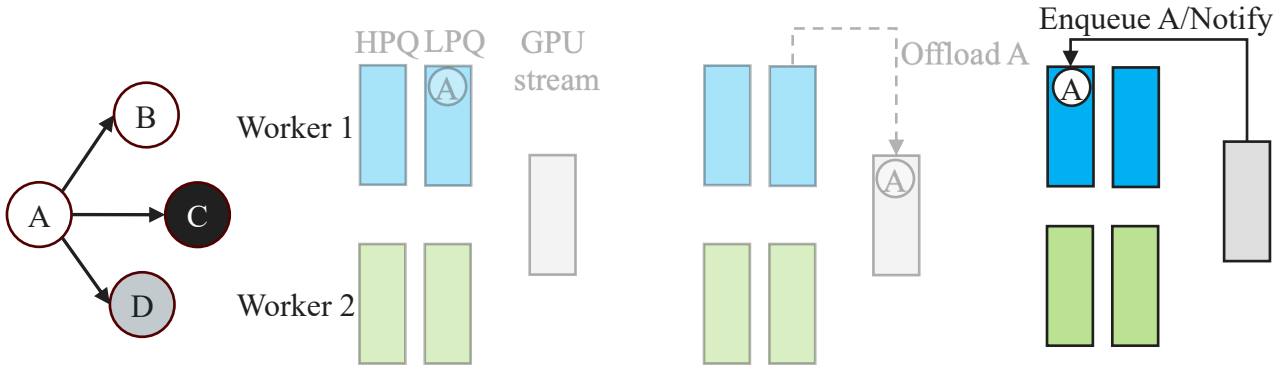

Worker 1
1. Offload GPU kernels in task A
2. Suspend task A
3. Go to sleep




Taro: https://github.com/dian-lun-lin/taro

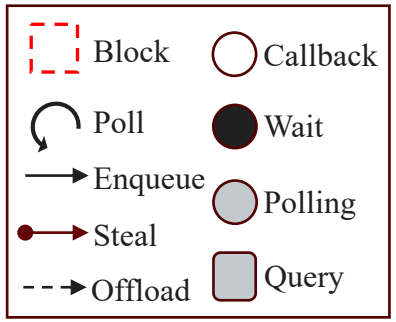
# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks




CUDA runtime

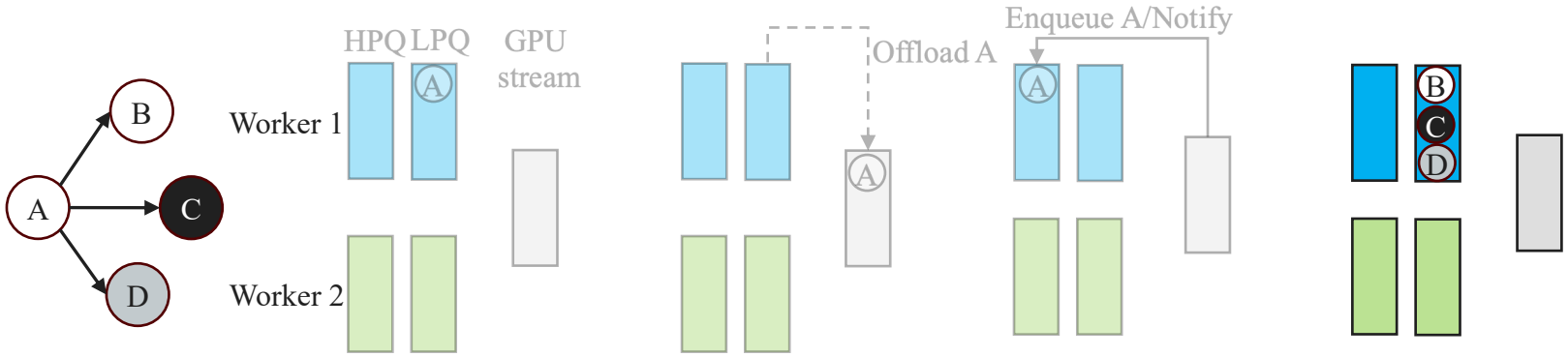
1. Invoke the callback
2. Insert task A back to Worker 1's HPQ
3. Notify Worker 1

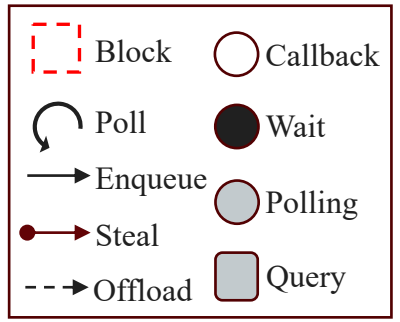



Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks

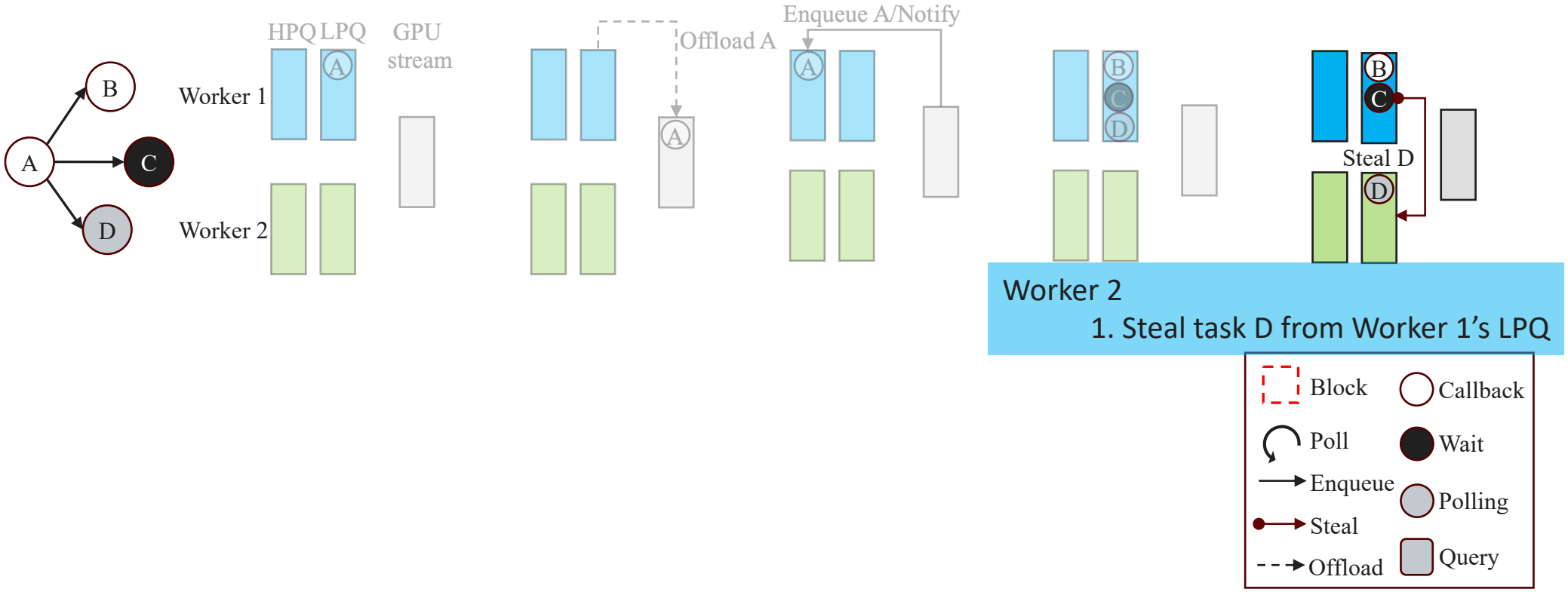



Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

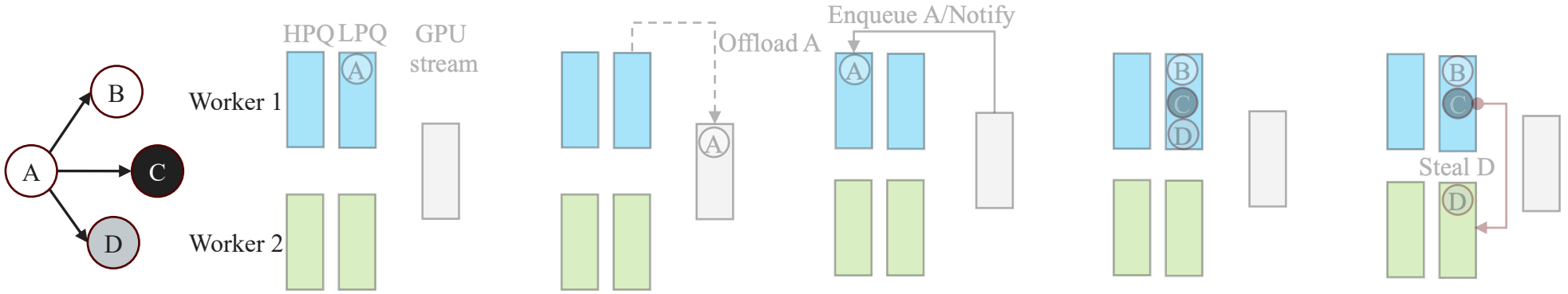
- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks

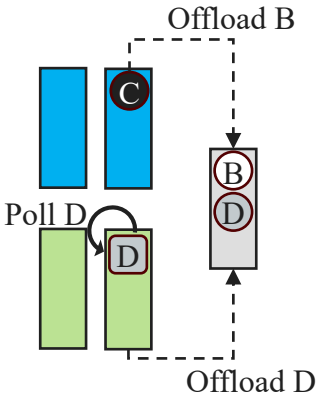



Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

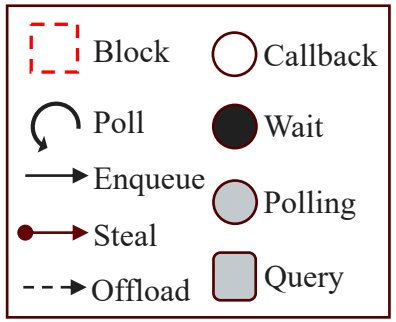
- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks




Worker 1
1. Offload GPU kernels in task B
2. Suspend task B
3. Multitask to task C

Worker 2
1. Offload GPU kernels in task D
2. Suspend task D
3. Create a query task to poll Task D's status

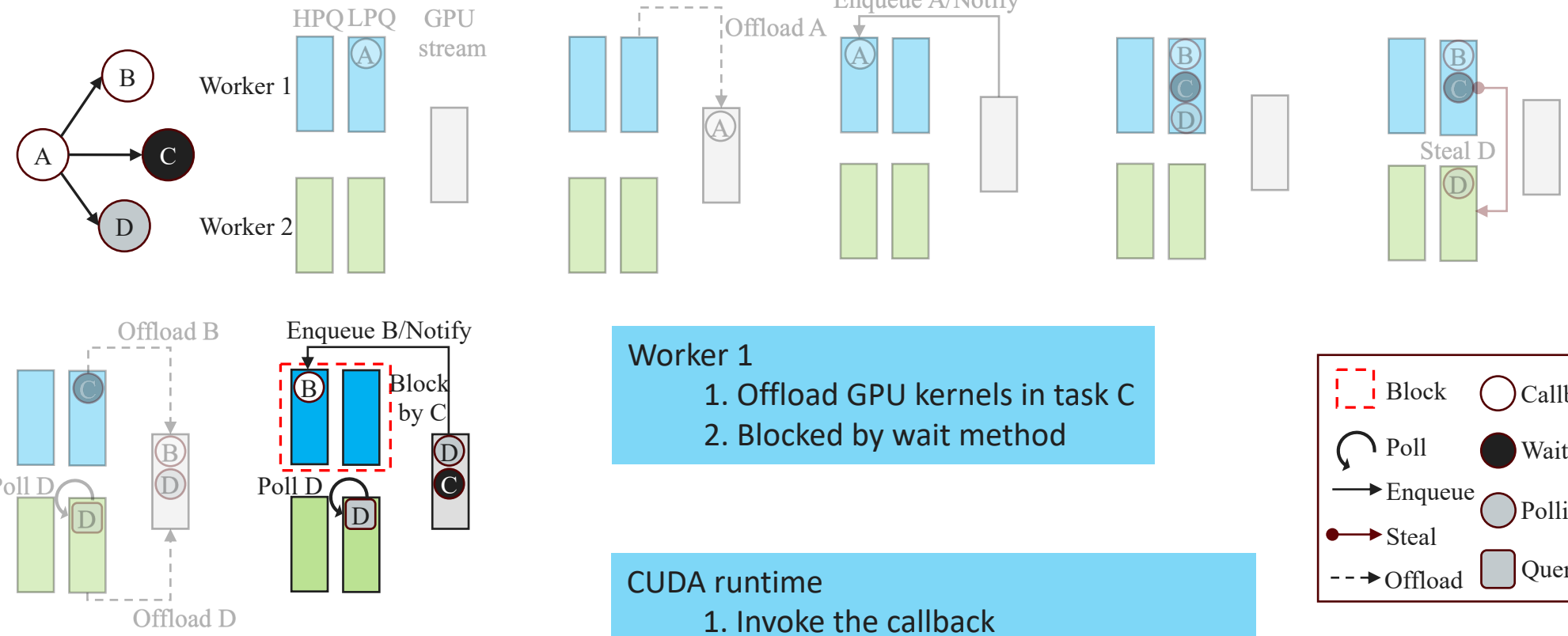



A query task is to poll a task's status

Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks




Worker 1
1. Offload GPU kernels in task C
2. Blocked by wait method

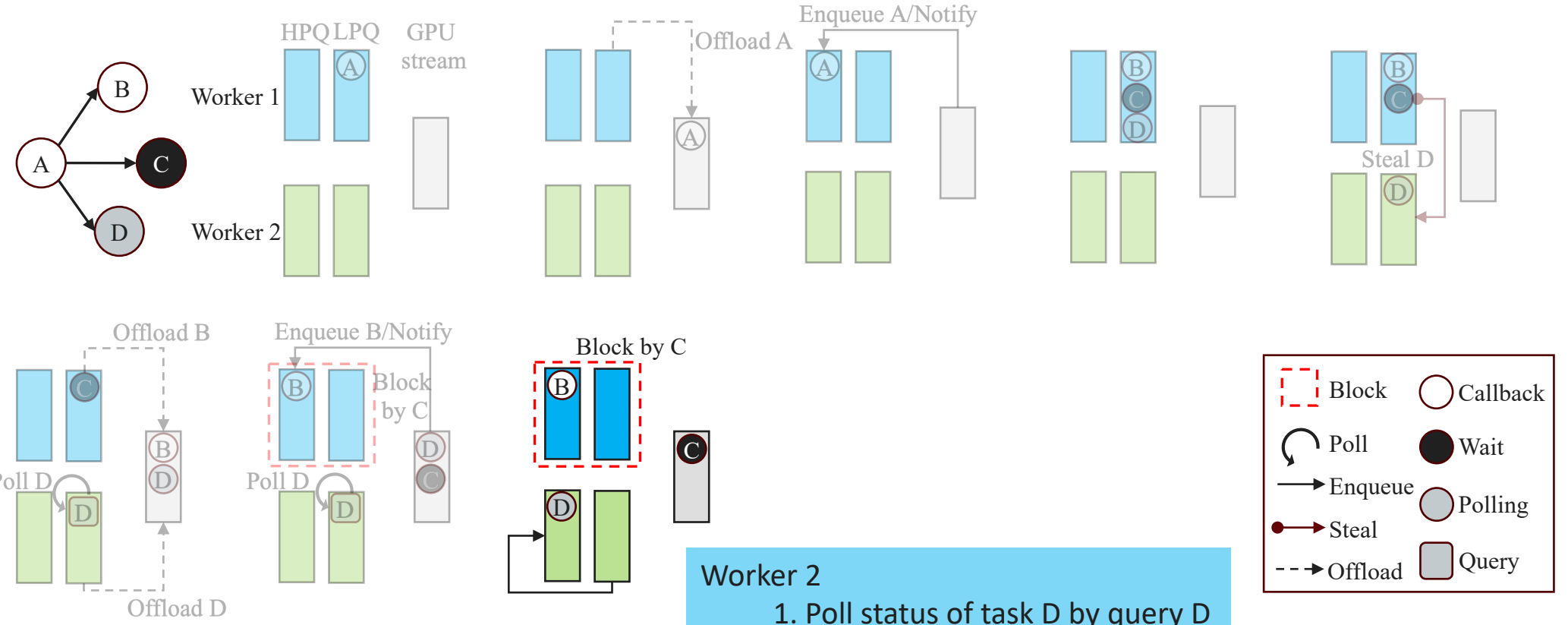
CUDA runtime
1. Invoke the callback
2. Insert task B back to Worker 1's HPQ
3. Notify Worker 1




Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks



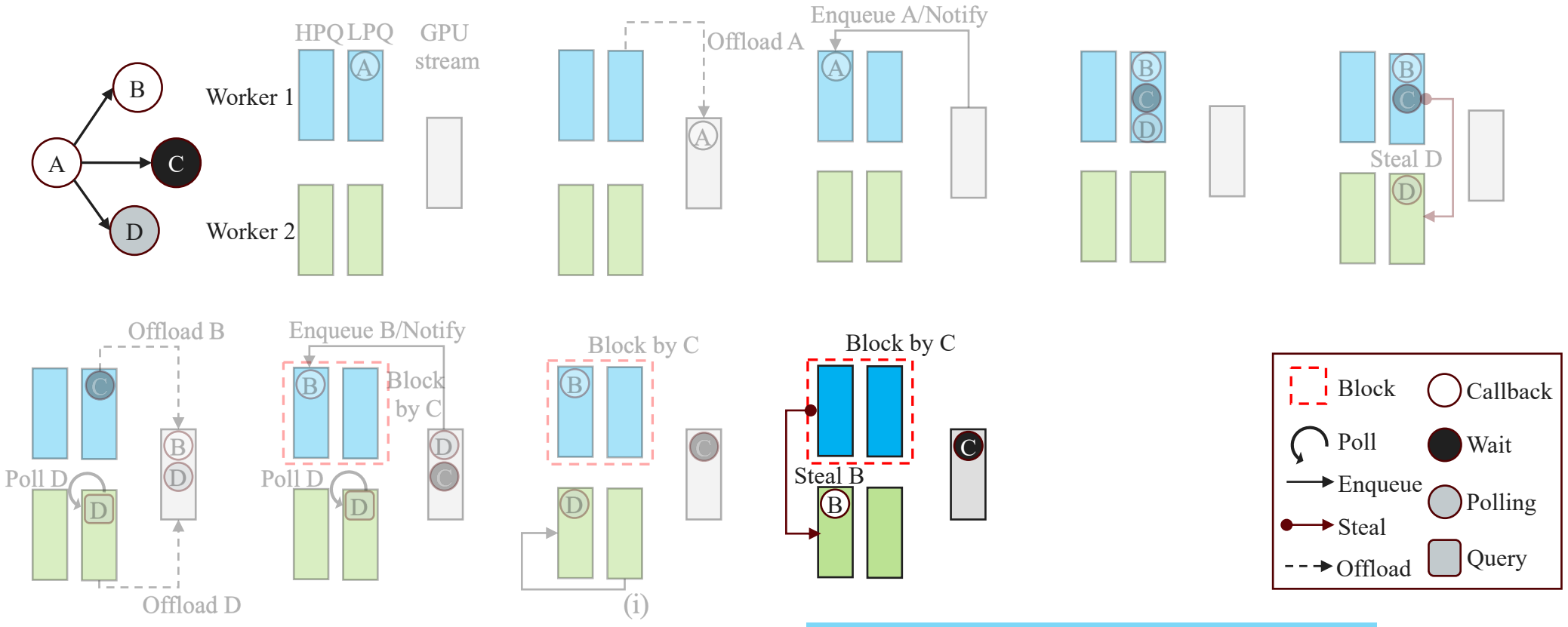

Worker 2
1. Poll status of task D by query D
2. Find task D is finished
3. Push task D back to its HPQ
4. Resume task D

Worker 1 cannot resume task B since it is blocked by task C (atomic execution)

Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks



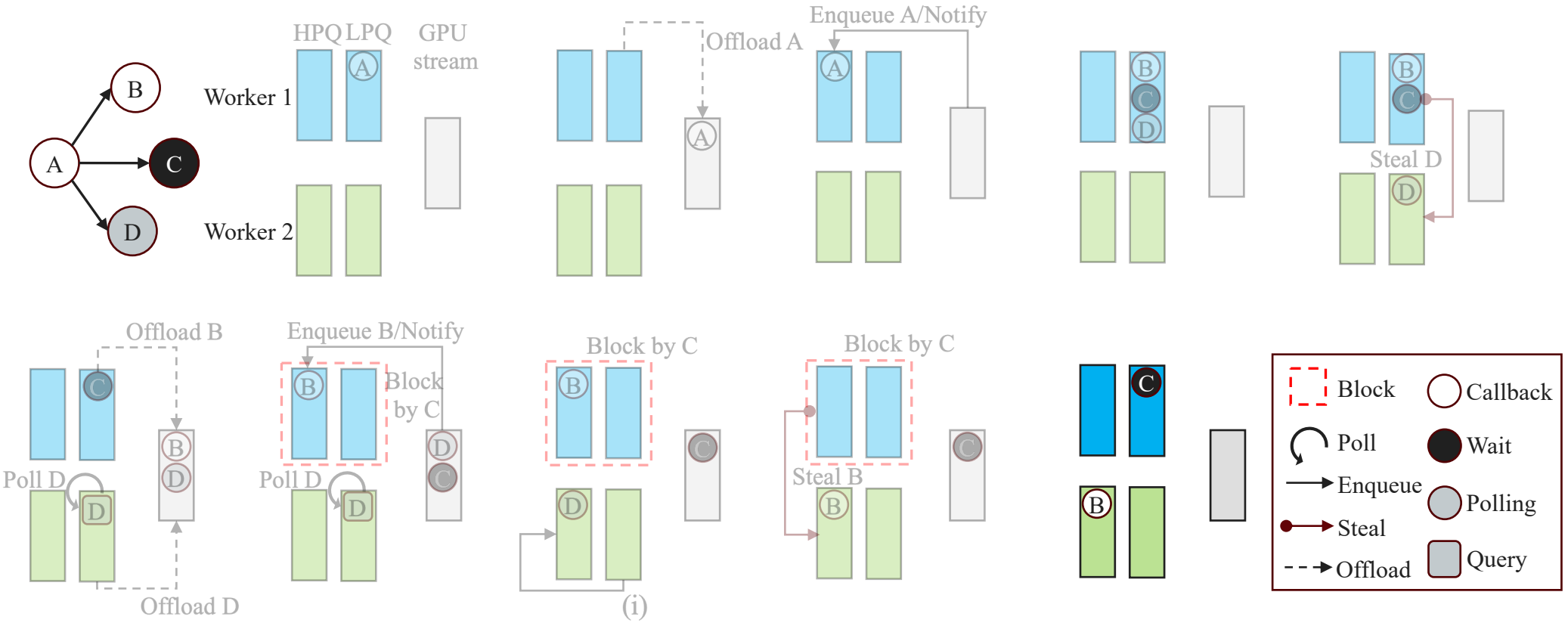

Worker 1 cannot resume task B since it is blocked by task C (atomic execution)

Worker 2
1. Steal task B from Worker 1's HPQ

Taro: https://github.com/dian-lun-lin/taro

# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks

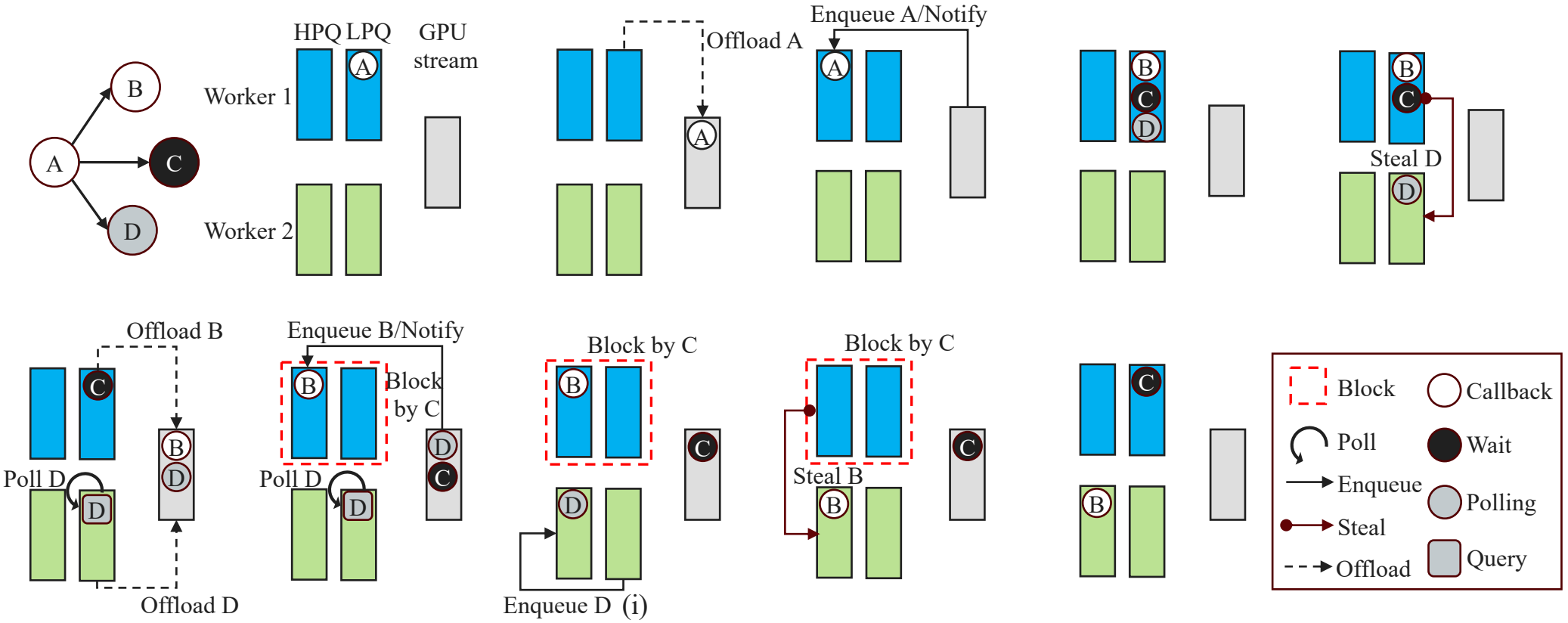



Finish the remaining tasks

Taro: https://github.com/dian-lun-lin/taro

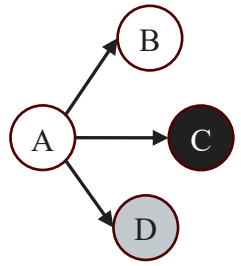
# Taro's Scheduler

- Assume two CPU threads (workers) and one GPU stream
- Each worker has:
  - High-priority queue (HPQ): store suspended tasks
  - Low-priority queue (LPQ): store new tasks




Taro: https://github.com/dian-lun-lin/taro

# You Write Code, Taro Handles Others




```cpp
#include <taro/taro.hpp>
#include <taro/scheduler/cuda.hpp>

taro::Taro taro{NUM_THREADS};
auto cuda = taro.cuda_scheduler(NUM_STREAMS);

auto task_a = taro.emplace([&]() -> taro::Coro {
  cpu_work_a1();
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_a1<<<32, 256, 0, stream>>>();
    kernel_a2<<<32, 256, 0, stream>>>();
  });
});
auto task_b = taro.emplace([&]() -> taro::Coro {
  co_await cuda.suspend_callback([&](cudaStream_t stream) {
    kernel_b1<<<32, 256, 0, stream>>>();
    kernel_b2<<<32, 256, 0, stream>>>();
  });
  cpu_work_b1();
});
```

```cpp
auto task_c = taro.emplace([&](cudaStream_t stream) {
  cpu_work_c1();
  cuda.wait([&](){
    kernel_c1<<<32, 256, 0, stream>>>();
  });
});
auto task_d = taro.emplace([&]() {
  cuda.suspend_polling([&](cudaStream_t stream) {
    kernel_d1<<<32, 256, 0, stream>>>();
  });
  cpu_work_d1();
});

task_a.precede(task_b);
task_a.precede(task_c);
task_a.precede(task_d);

taro.schedule();
taro.wait();
```

Taro: https://github.com/dian-lun-lin/taro

# What's More – C++20 Synchronization Primitives

Taro's scheduler is based on C++20 atomic notify

std::atomic<T>::notify_one

```
void notify_one() noexcept;          (1)  (since C++20)
void notify_one() volatile noexcept; (2)  (since C++20)
```

~1.6x speed-up

| Data Type | Condition Variables | Two Atomic Flags | One Atomic Flag | Atomic Bool |
|---|---|---|---|---|
| Time | 0.52 | 0.32 | 0.31 | 0.38 |

Table credit: Modernes C++, Rainer Grimm

Taro: https://github.com/dian-lun-lin/taro

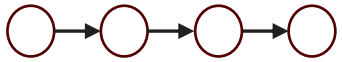
# Agenda

- Understand the motivation behind Taro
- Learn to use the Taro C++ programming model
- Dive into the Taro's coroutine-aware scheduling algorithm
- **Evaluate Taro on microbenchmarks and a real-world application**
- Conclusion

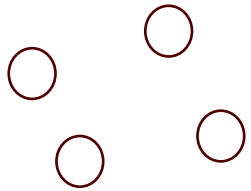
# Hardware Platform & Baselines

- Eight CPU threads (3.2 GHz)
- One NVIDIA 3080 ti GPU
- Baselines
  - Microbenchmarks
    - **Taskflow**
    - **Taskflow$^{coro}$**
      1. Utilize executor::async (similar to std::async) to wrap up coroutine.resume()
      2. When a suspended coroutine is ready, create an async task to schedule
    - **Boost Fiber**
      1. Wrap Boost Fiber within our task graph framework to allow for the creation and management of task graphs
  - Large-scale circuit simulation workload
    - State-of-the-art CPU-GPU circuit simulator, **RTLflow**

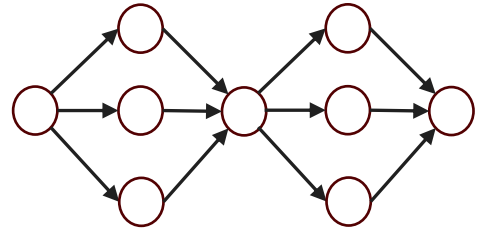
# Experimental Results – Microbenchmarks



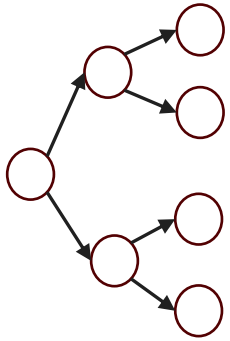
(a) Linear Chain



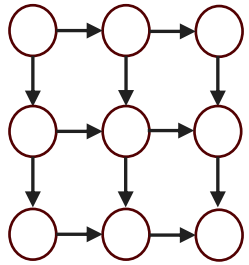
(b) Embarrassing Parallelism



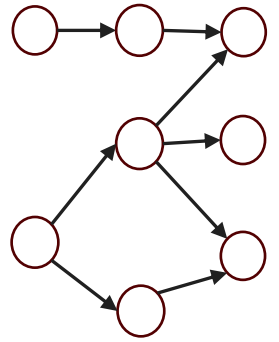
(c) Map Reduce



(d) Divide and Conquer



(e) Wavefront Parallelism



(f) Random DAG

# Experimental Results – Microbenchmarks

- Data-driven task

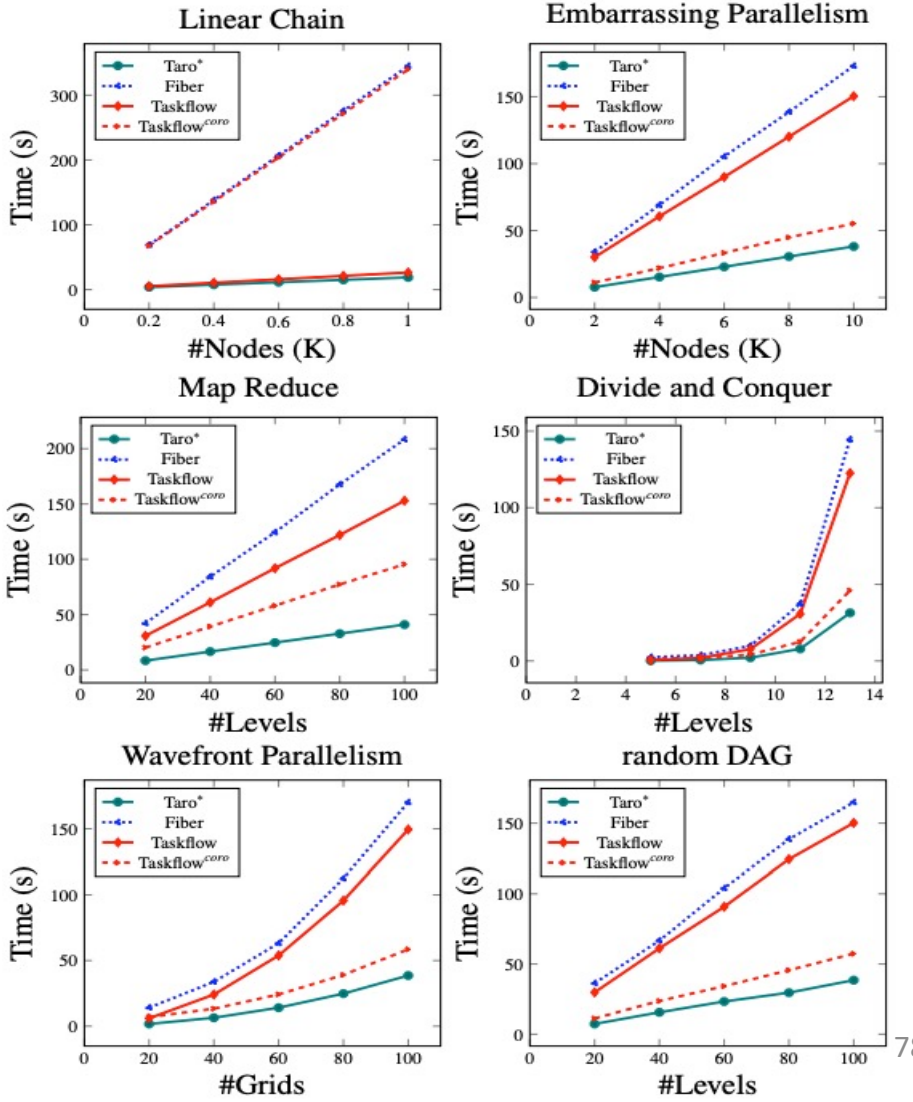
Assume each task is a data-driven task

```
1  taro::Coro data_driven(auto& cuda) {
2    int k = 0;
3
4    while(k++ < 300) {
5      // CPU operations
6      co_await cuda.suspend_polling([=](cudaStream_t st) {
7        // GPU operations...
8      });
9    }
10 }
```

# Experimental Results – Microbenchmarks

- Data-driven task

Taro achieves up to 18.3× and 18.0× speed-up when compared to Fiber and Taskflow$^{coro}$



# Experimental Results – Microbenchmarks

- Fixed-runtime task

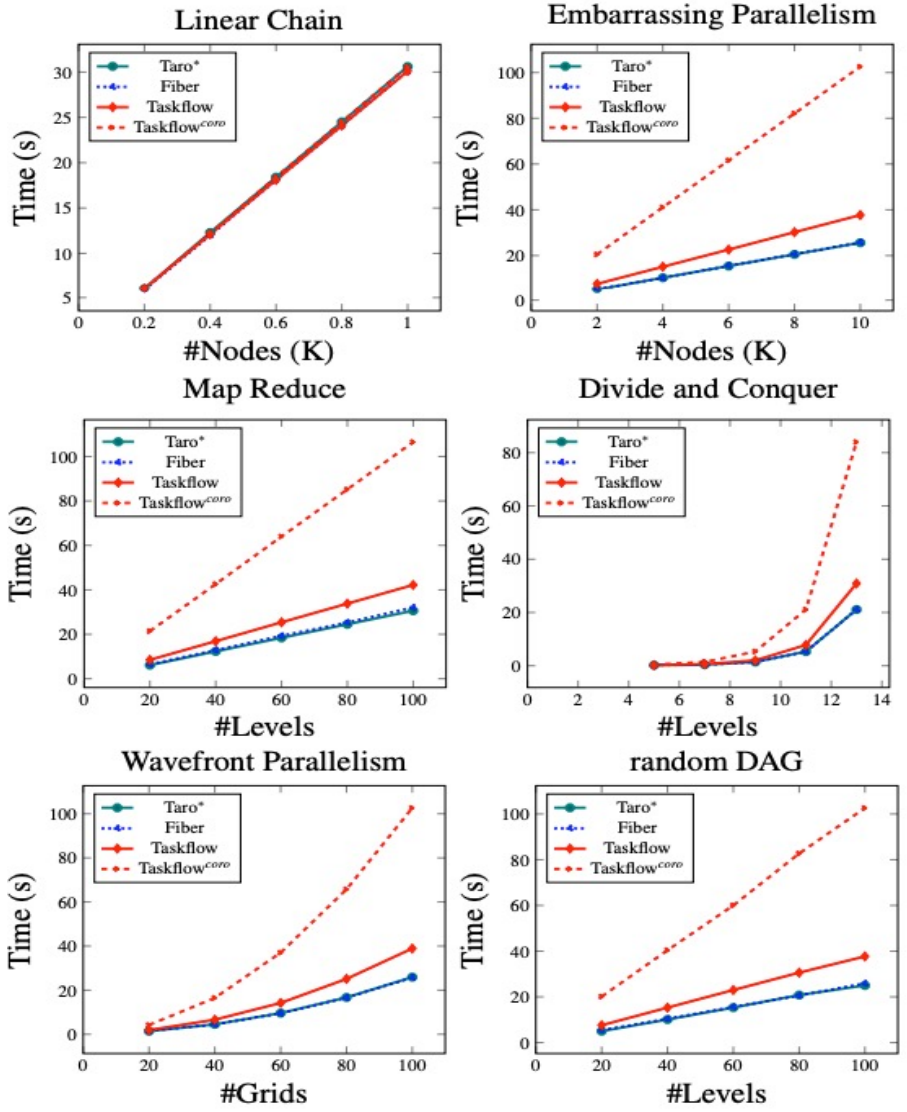
Assume each task is a fixed-runtime task

```
1 taro::Coro fixed_runtime(auto& cuda, size_t ctime, size_t gtime) {
2
3   cpu_busy_loop(ctime);
4   co_await _cuda.suspend_callback([=](cudaStream_t st) {
5     cuda_busy_loop<<<8, 256, 0, st>>>(gtime); // large overhead
6   })
7 }
```

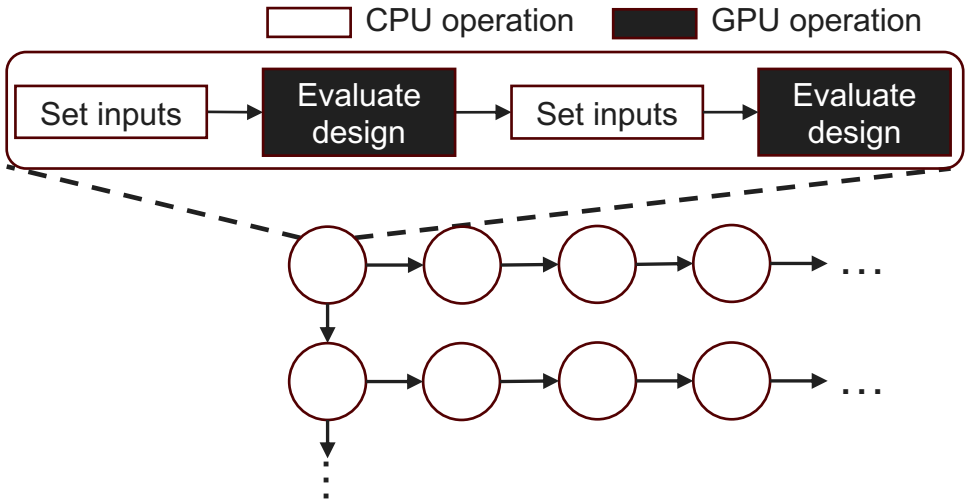
# Experimental Results – Microbenchmarks

- Fixed-runtime task

Both Taro and Boost::Fiber achieve the best results



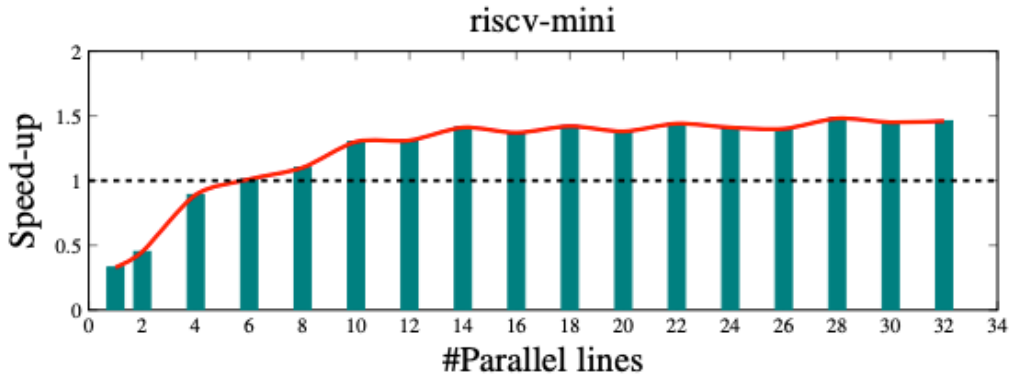
# Experimental Results – Circuit Simulation Workload

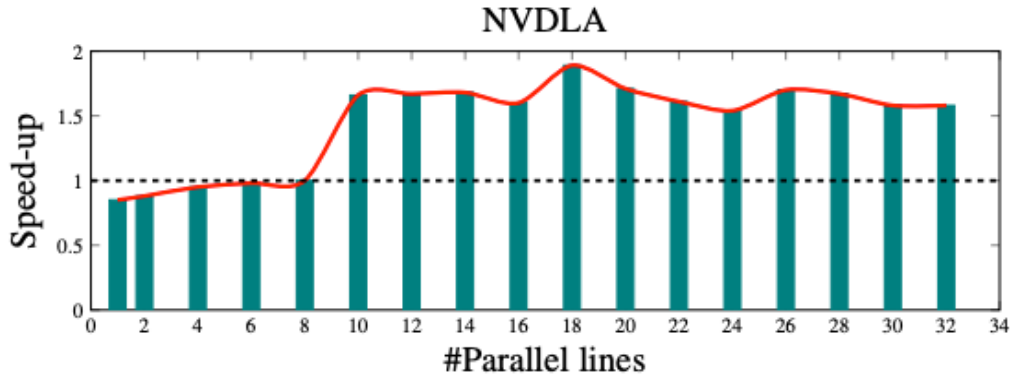



A task graph that consists of multiple parallel lines

# Experimental Results – Circuit Simulation Workload

Achieved speed-up by Taro over RTLflow at different task graph sizes on riscv-mini and NVDLA circuit

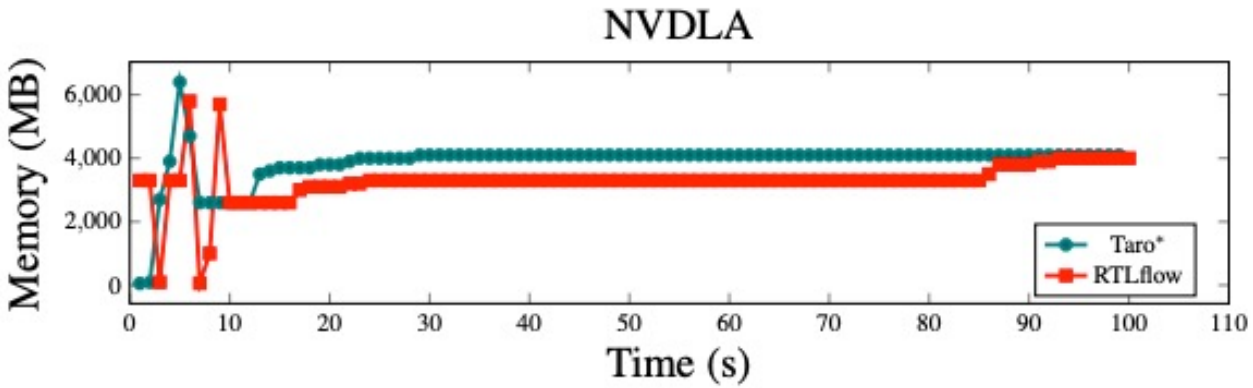



Taro achieve 1.7x speed-up over the state-of-the-art simulator RTLflow

# Experimental Results – Circuit Simulation Workload

Memory usage at each second for RTLflow and Taro




Taro shows comparable memory usage to RTLflow

# Conclusion

- We have presented the **motivation** behind Taro
- We have presented the Taro **C++ programming model**
  - Synchronous and Asynchronous mechanisms
- We have presented the Taro **coroutine-aware scheduling algorithm**
- We have presented the performance of Taro
- Future work
  - **Document**
  - **Extend**

Get involved!

Different accelerators

Computation patterns (e.g., Pipeline using symmetric coroutine transfer)

Async IO


Taro: https://github.com/dian-lun-lin/taro
Dian-Lun Lin: https://dian-lun-lin.github.io/