COSMIC

A Graph-Based, Extensible Framework for the Future of Adaptive Optics RTC development

Julien Bernard

Introduction

- MAVIS is an instrument being built for the ESO's VLT AOF.

- Come and watch François Rigaut presentation: Entering the final design phase for the MAVIS RTC at 14:30 !

Summary

- The technical stack
- COSMIC evolution
- From prototype to RTC

The technical stack

- Language: C++23, python 3.13, CUDA 12
- We opted to rely as much as possible to exiting tools and libraries
  - CMake, pip and py-build-cmake (PEP 517 compliant build backend)
  - Boost, pybind11, microsoft-gsl, Taskflow, matx, gtest, benchmark
- Use modern language and standard library features
  - C++ is an excessively complex language but nothing impossible for a trained team and good practices

```cpp
int sum(int* s, int n) {
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += s[i];

    return sum;
}

int main() {

    std::vector<int> v = {1, 2, 3, 4, 5};
    std::array<int, 5> a = {1, 2, 3, 4, 5};
    int arr[] = {1, 2, 3, 4, 5};

    sum(v.data(), v.size());
    sum(a.data(), a.size());
    sum(arr, 5);

    sum(nullptr, 0); // What happens here?

}
```

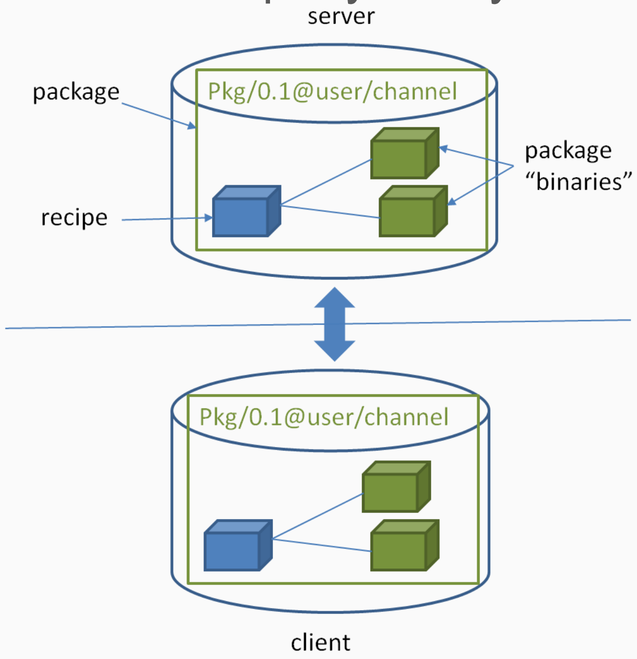
The technical stack

- Language: C++23, python 3.13, CUDA 12
- We opted to rely as much as possible to exiting tools and libraries
  - CMake, pip and py-build-cmake (PEP 517 compliant build backend)
  - Boost, pybind11, microsoft-gsl, Taskflow, matx, gtest, benchmark
- Use modern language and standard library features
  - C++ is an excessively complex language but nothing impossible for a trained team and good practices

```cpp
int sum(std::span<int> s) {
    int sum = 0;
    for (int i : s)
        sum += i;

    return sum;
}

int main() {

    std::vector<int> v = {1, 2, 3, 4, 5};
    std::array<int, 5> a = {1, 2, 3, 4, 5};
    int arr[] = {1, 2, 3, 4, 5};

    sum(v);
    sum(a);
    sum(arr);

    // sum(nullptr); It's not working anymore !

}
```

CONAN: A C++ package manager

- Decentralized package manager
- build helper
  - manage configurations [Release, Config], [Static, Shared], and more.
- Allow source or binary only package
- development mode à la `pip install -e`
- Allows to test new third-party library in minutes



CONAN 1

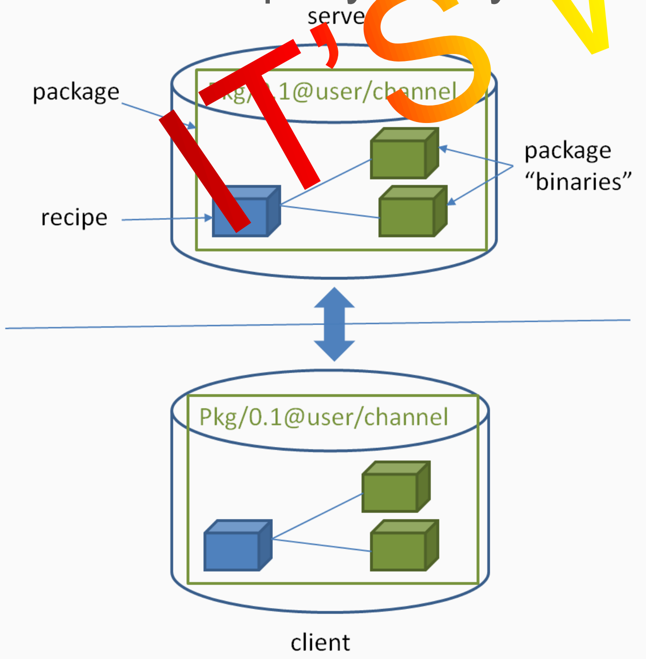
CONAN: A C++ package manager

CONAN 2

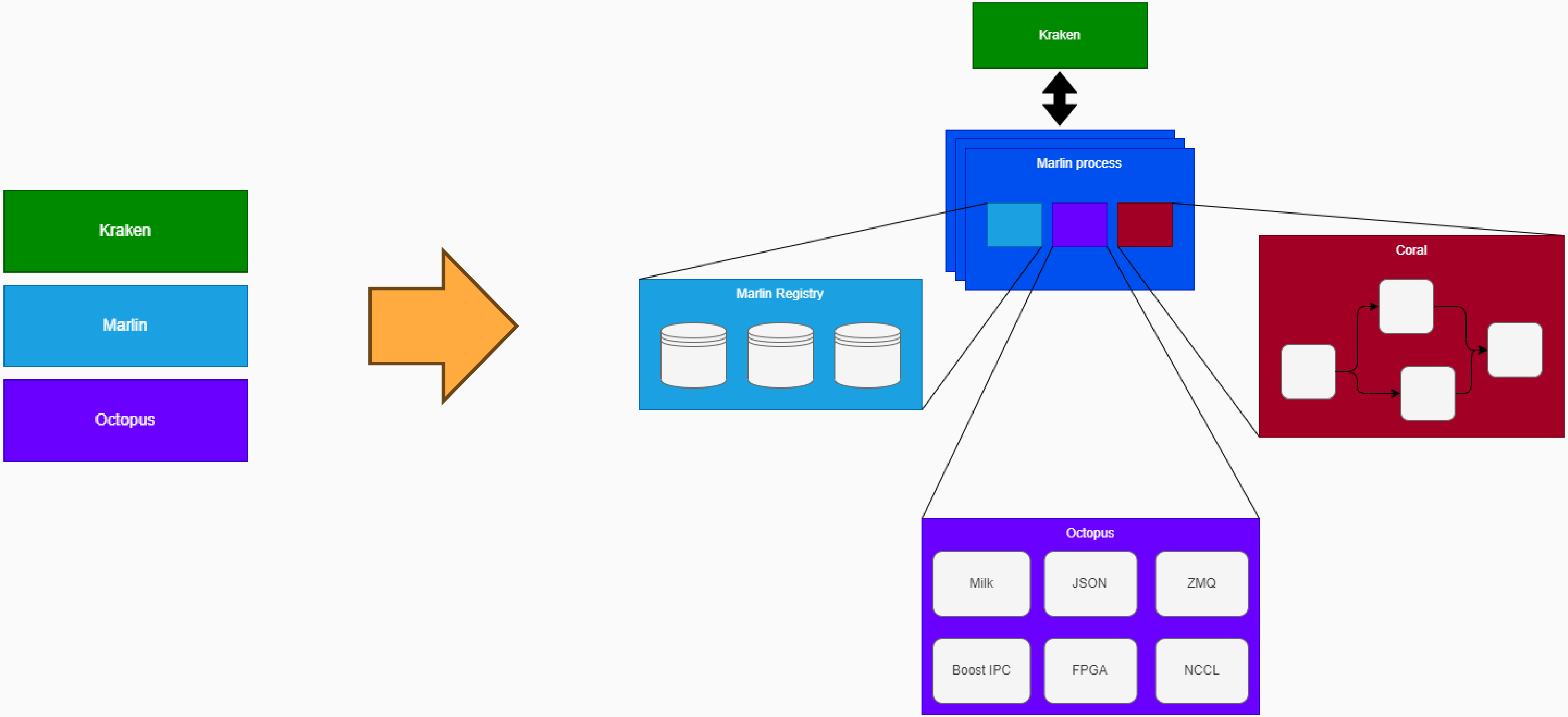
- Decentralized package manager
- build helper
  - manage configurations [Release, Config], [Static, Shared], and more.
- Allow source or binary only package
- development mode à la `pip install -e`
- Allows to test new third-party library in minutes

A bit hard to use !

IT'S WORTH IT !



The COSMIC evolution



Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !

```
# reconstruct pseudo-open-loop slopes:
pol_slopes[:] = slopes - D_mat @ command_eff

# project POL slopes to mode space, and filter them with IIR "gain"
modes[:] = (1 - gain) * modes - gain * (R_mat @ pol_slopes)

# project modes to actuator space:
cmds = P_mat @ modes
```

Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !
    - Instantiate nodes locally

```
D_mvm = marlin.registry.create("cuda:la:mvm", D_mat, ...)
R_mvm = marlin.registry.create("cuda:la:mvm", R_mat, ...)
P_mvm = marlin.registry.create("cuda:la:mvm", P_mat, ...)

# ...

D_mvm.compute(stream, command_eff, result_slopes)

pol_slopes[:] = slopes - result_slopes

R_mvm.compute(stream, pol_slopes, result_modes)

modes[:] = (1 - gain) * modes - gain * result_modes

P_mvm.compute(stream, modes, cmds)
```

Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !
  - Instantiate nodes locally
  - Port to C++/CUDA (using MatX)

```
// reconstruct pseudo-open-loop slopes:
matvec(slopes_pol, D_mat, cmds_eff, stream);
(slopes_pol = slopes - slopes_pol).run(stream);

// project POL slopes to mode space, and filter them with IIR "gain"
matvec(modes_tmp, R_mat, slopes_pol, stream);
(modes = (1 - gain) * modes - gain * modes_tmp).run(stream);

// project modes to actuator space:
matvec(cmds, P_mat, modes, stream);
```

Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !
  - Instantiate nodes locally
  - Port to C++/CUDA (using MatX)
  - Again, use locale node instances

```
auto D_mvm = cuda::la::mvm<float>(...);
auto R_mvm = cuda::la::mvm<float>(...);
auto P_mvm = cuda::la::mvm<float>(...);

// reconstruct pseudo-open-loop slopes:
D_mvm.compute(stream, slopes_pol, cmds_eff);
(slopes_pol = slopes - slopes_pol).run(stream);

// project POL slopes to mode space, and filter them with IIR "gain"
R_mvm.compute(stream, modes_tmp, R_mat, slopes_pol);
(modes = (1 - gain) * modes - gain * modes_tmp).run(stream);

// project modes to actuator space:
P_mvm.compute(stream, cmds, P_mat, modes);
```

Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !
  - Instantiate nodes locally
  - Port to C++/CUDA (using MatX)
  - Again, use locale node instances
  - Them you put it in a graph !

```
ccg::children children; children.reserve(3);

// reconstruct pseudo-open-loop slopes:
children.emplace_back([&](cudaStream_t stream){
    D_mvm.compute(stream, slopes_pol, cmds_eff);
    (slopes_pol = slopes - slopes_pol).run(stream);
}, ...);

// project POL slopes to mode space, and filter them with IIR "gain"
children.emplace_back([&](cudaStream_t stream){
    R_mvm.compute(stream, modes_tmp, R_mat, slopes_pol);
    (modes = (1 - gain) * modes - gain * modes_tmp).run(stream);
}, ...);

// project modes to actuator space:
children.emplace_back([&](cudaStream_t stream){
    P_mvm.compute(stream, cmds, P_mat, modes);
}, ...);

coral::edges edges{ {0, 1}, {1, 2} };

auto pipeline = ccg::pipeline(std::move(children),
std::move(edges));
```

Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !
    - Instantiate nodes locally
    - Port to C++/CUDA (using MatX)
    - Again, use locale node instances
    - Them you put it in a graph !
    - And finally put it into a node and register it

```
struct Polc {
  void compute(context ctx, span<float> slopes, span<float> cmds) {
    // reconstruct pseudo-open-loop slopes:
    D_mvm.compute(ctx, slopes_pol, cmds_eff);
    (slopes_pol = slopes - slopes_pol).run(ctx.stream);

    // project POL slopes to mode space, and filter them with IIR...
    R_mvm.compute(ctx, modes_tmp, R_mat, slopes_pol);
    (modes = (1 - gain) * modes - gain * modes_tmp).run(ctx.stream);

    // project modes to actuator space:
    P_mvm.compute(ctx, cmds, P_mat, modes);
  }
};

MARLIN_REGISTER(m) {
  class_<Pocl>(“cuda:mavis:Polc", m)
    .def("compute", &Polc::compute)
    .def_poperty("D", &Polc::D_mvm)
    .def_poperty("R", &Polc::R_mvm)
    .def_poperty("P", &Polc::P_mvm);
}
```

Fast prototyping

- More hassle-free steps available between a simple python script and a fully working RTC
- Work as much as possible locally
- Use interactive language and debugger

- Let's consider a simple POLC example !
  - Instantiate nodes locally
  - Port to C++/CUDA (using MatX)
  - Again, use locale node instances
  - Them you put it in a graph !
  - And finally put it into a node and register it

```python
polc = marlin.registry.create("cuda:mavis:Polc", ...)

polc.compute(stream, slopes, cmds)
```

Coral

- A direct acyclic graph library with support for hardware accelerators and complex control-flows

- Fixed specification
- Control flow utilities
- Adaptor utilities

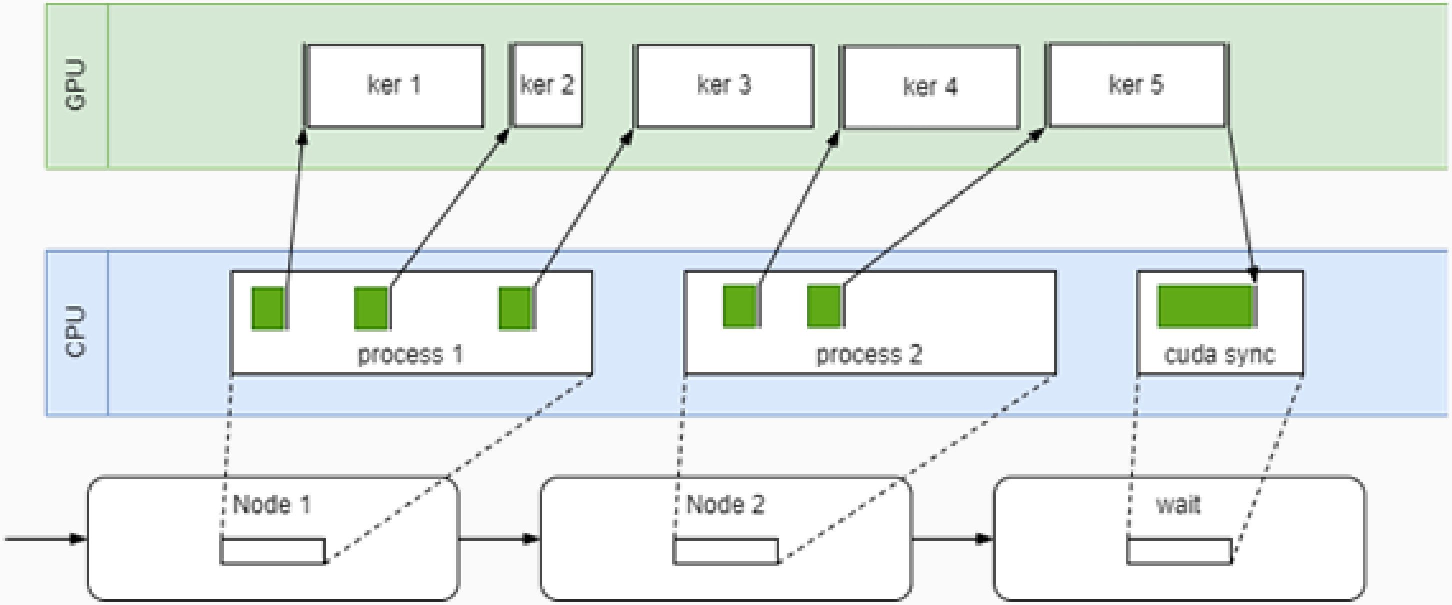
- Support for:
  - Host pipeline execution using Taskflow
  - Asynchronous host execution using C++ coroutines (experimental)
  - CUDA pipeline execution

```
node:
  pipeline
  logic
  adaptor

pipeline:
  node* + dep*

dep:
  index + index

logic:
  conditional
  switch
  while

conditional | switch | while :
  node + condition

adaptor:
  node (other)

launcher:
  node & contex
```

CORAL CUDA pipeline model

- GPU execution perform better with asynchronous execution
- We only focus on scheduling operations in the right order on the host. CUDA runtime takes care of the rest

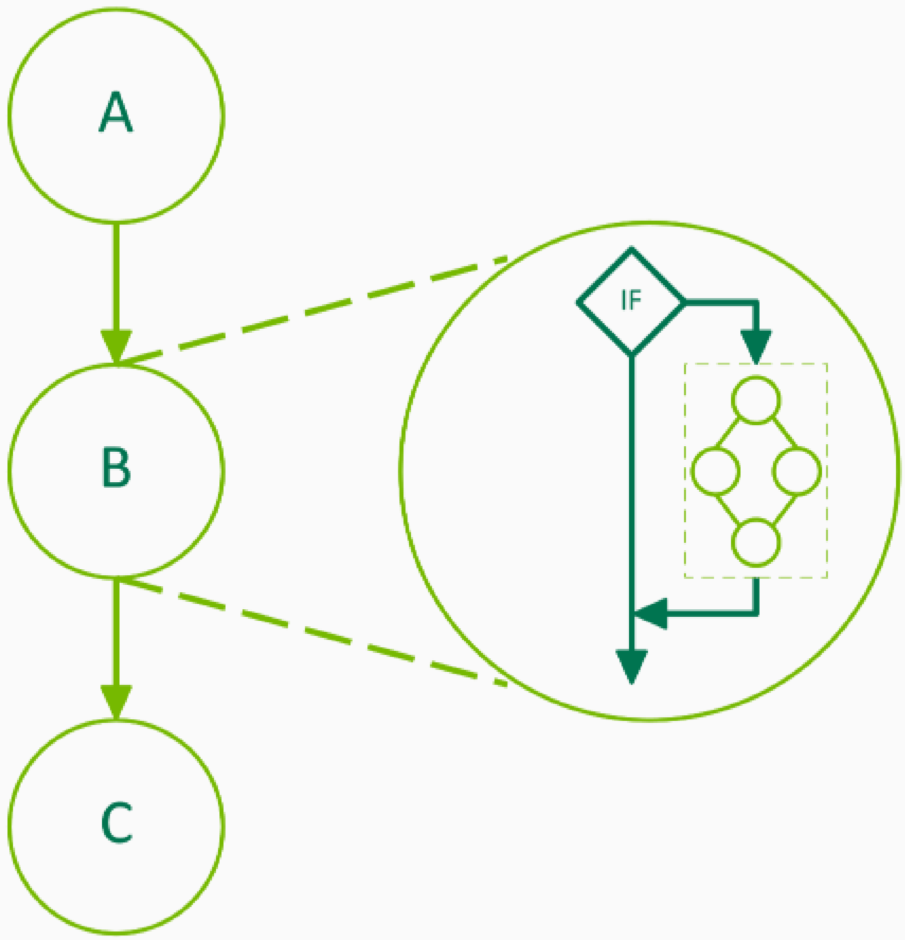
- Until CUDA 12, we were limited using simple DAG without control flow



CORAL CUDA pipeline model

- Thanks to CUDA 12 it is now possible to implement complex control flow on device using device cuda graph.
- For now, we have 3 types of control flow: condition, switch and while

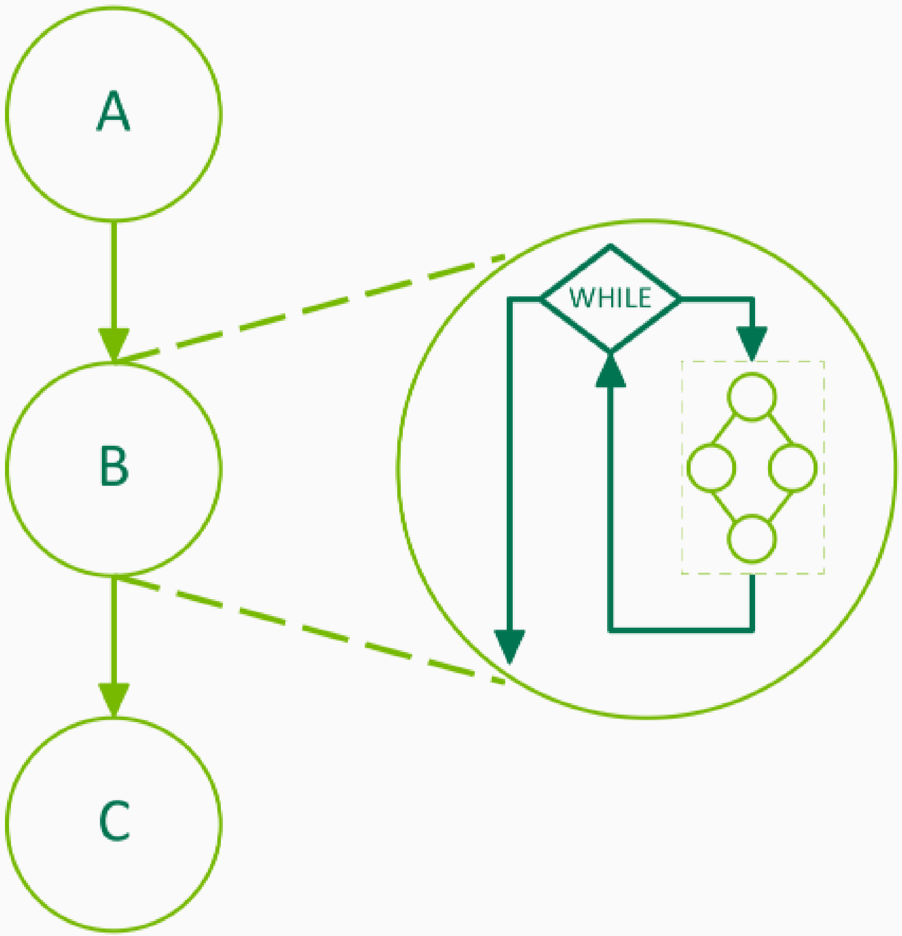
```
auto condition =
coral::cuda::logic::predicate_launcher(conditional_op{});

auto graph = ccg::logic::conditional_graph(node, condition);
```



CORAL CUDA pipeline model

- Thanks to CUDA 12 it is now possible to implement complex control flow on device using device cuda graph.
- For now, we have 3 types of control flow: condition, switch and while

```
auto condition =
coral::cuda::logic::while_launcher(conditional_op{});

auto graph = ccg::logic::while_graph(node, condition);
```



Thanks for you attention !

Questions ?