A Case For and an Implementation Of Composable CUDA Graph Algorithms

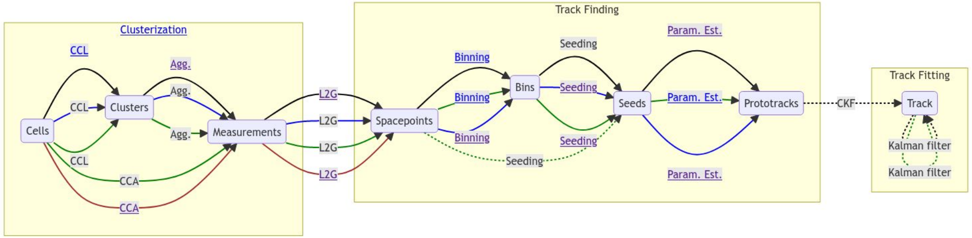
Stephen Nicholas Swatman, ACTS Parallelization Meeting, January 27th, 2023

Context

- Track reconstruction in traccc is a dataflow programming problem
- Data moves between different algorithms
- Arbitrary sub-chains must be independently executable
- The `traccc::algorithm` class was designed to model algorithms to be
  - Composable
  - Asynchronous
- Did not end up really taking off

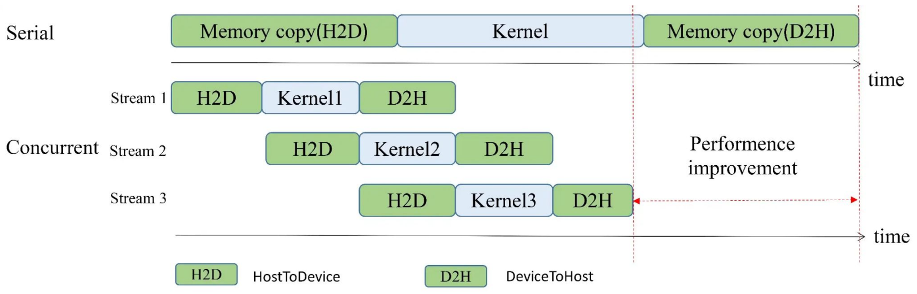
Asynchronicity

- Asynchronous execution is key in dataflow programming
- Allows hiding of latency from kernel launches, data movement, allocations, etc.
- SYCL programming model mostly enforces this implicitly
- CUDA programming model requires explicit asynchronicity



Asynchronous CUDA Programming

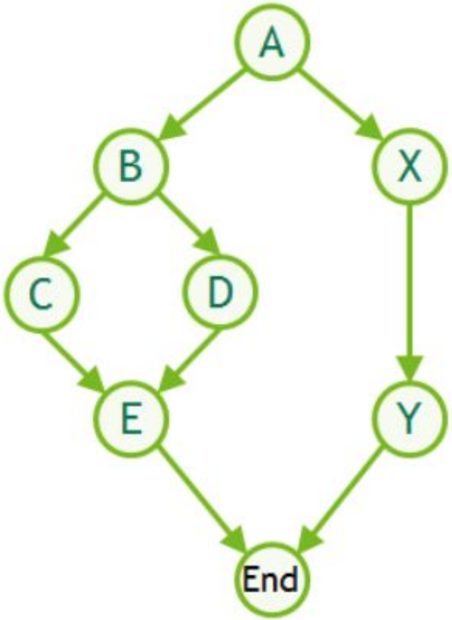
- CUDA models asynchronicity using ordered *streams*
- Streams allow asynchronous kernel launches, memory copies, etc.
- But streams enforce execution in order of enqueuement



Zhang et at. (2021)

CUDA Graphs

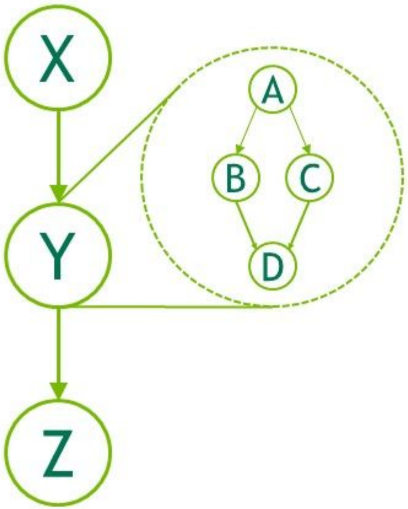
- CUDA graphs provide an abstraction over streams
- Graphs allow dependency-bound re-ordering of processes!
- Potential for more efficient execution of dataflow code



NVIDIA

CUDA Graphs in traccc

- To use CUDA Graphs in traccc, need to represent every algorithm as a stand-alone graph
    - Must be individually executable
- CUDA supports embedding graphs inside other graphs!
- Problem solved!



NVIDIA

Sadly not: child graph nodes cannot contain allocations

No allocations in child graphs

- Most of our algorithms require intermediate allocations of scratch space
- This makes it impossible to use embedded CUDA graphs
- Need a different solution!

Description

Creates a new node which executes an embedded graph, and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

If `hGraph` contains allocation or free nodes, this call will return an error.

The node executes an embedded child graph. The child graph is cloned in this call.

Proposal: traccc graph algorithm descriptor

- If CUDA will not let us use its mechanism of composition, we will design our own
- Proposal: classes describing how to build (sub-)graphs to perform a given algorithm!
- Described by config type *C*, argument type *A*, and return type *R*
- Model computation $(C, A) \to R$

```cpp
class alg1 {
    public:
    using result_type = std::tuple<int *, std::size_t>;
    using config_type = std::size_t;
    using argument_type = std::monostate;

    static std::tuple<cudaGraph_t, cudaGraphNode_t, result_type> create_graph(
        config_type c, argument_type) {
        cudaGraph_t g;

        CUDA_ERROR_CHECK(cudaGraphCreate(&g, 0));

        cudaGraphNode_t allocation_node;

        cudaMemAllocNodeParams alloc_params;
        memset(&alloc_params, 0, sizeof(alloc_params));
        alloc_params.bytesize = c * sizeof(int);
        alloc_params.poolProps.allocType = cudaMemAllocationTypePinned;
        alloc_params.poolProps.location.id = 0;
        alloc_params.poolProps.location.type = cudaMemLocationTypeDevice;

        CUDA_ERROR_CHECK(cudaGraphAddMemAllocNode(&allocation_node, g, nullptr,
                                                  0, &alloc_params));

        return {g, allocation_node,
                result_type{reinterpret_cast<int *>(alloc_params.dptr), c}};
    }
};
```

Two classes of nodes

Initial algorithm nodes P_0

- Exclusively usable as the first computation in a chain
- Required to have a static method:

```
static std::tuple<cudaGraph_t,
cudaGraphNode_t, R>
create_graph(C, A)
```

Non-initial algorithm nodes P_+

- Exclusively usable in composition *after* an initial node
- Required to have a static method:

```
static std::tuple<cudaGraphNode_t,
R> append_graph(cudaGraph_t,
cudaGraphNode_t, C, A)
```

Concepts

- If available, requirements are verified using C++20 concepts!

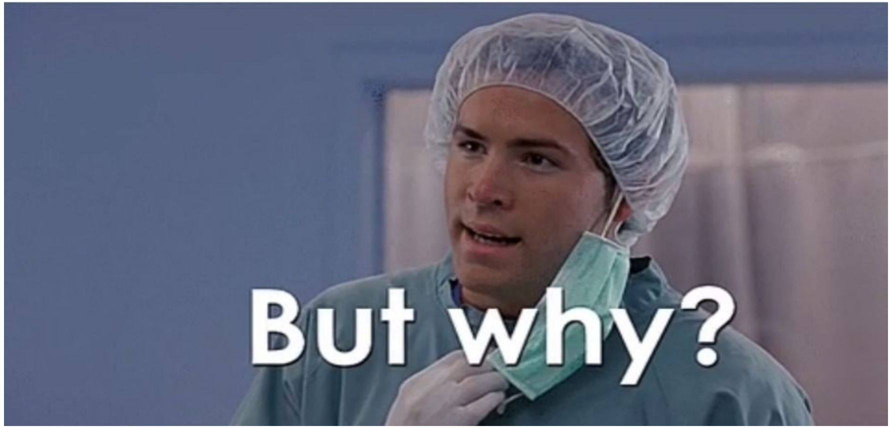
```
template <typename T>
concept graph_descriptor_c = requires {
    typename T::result_type;
    typename T::config_type;
    typename T::argument_type;
};

template <typename T>
concept noninitial_graph_descriptor_c = graph_descriptor_c<T>and requires {
    requires requires(cudaGraph_t g, cudaGraphNode_t n,
                      typename T::config_type c, typename T::argument_type a) {
        { T::append_graph(g, n, c, a) }
        ->std::same_as<std::tuple<cudaGraphNode_t, typename T::result_type>>;
    };
};

template <typename T>
concept initial_graph_descriptor_c = graph_descriptor_c<T>and requires {
    requires requires(typename T::config_type c, typename T::argument_type a) {
        { T::create_graph(c, a) }
        ->std::same_as<
            std::tuple<cudaGraph_t, cudaGraphNode_t, typename T::result_type>>;
    };
};
```

Composition of graphs

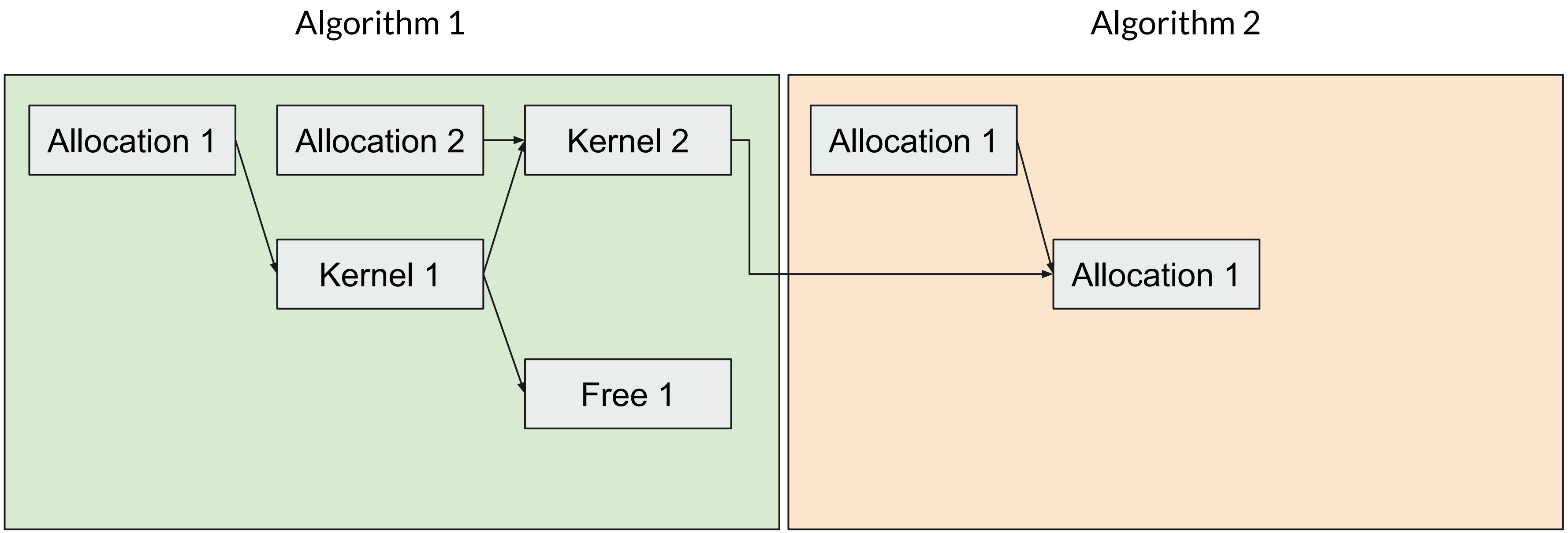
- Graph descriptors can be composed using two rules:
  - $P_0(C, A, R) \circ P_+(C', A', R') = P_0(C \times C', A, R')$
  - $P_+(C, A, R) \circ P_+(C', A', R') = P_+(C \times C', A, R')$
- Implementation included in current pull request!

What does this buy us

- Over straight CUDA graphs
  - Ability to compose algorithms arbitrarily
- Over simple CUDA streams
  - Inter-algorithm re-ordering of operations
  - Intra-algorithm re-ordering of operations



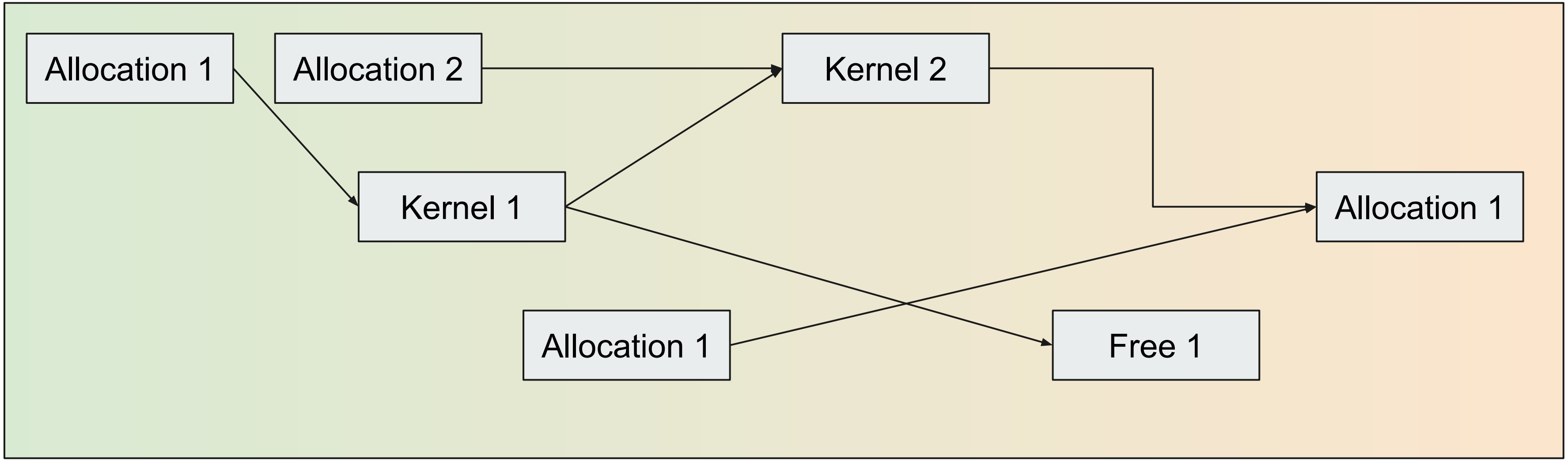
Example



Algorithm 1
Algorithm 2
Allocation 1
Allocation 2
Kernel 2
Kernel 1
Free 1
Allocation 1
Allocation 1

Example

Algorithm 2 ○ Algorithm 1



Status

- Initial implementation of this in #307
- Includes practical example of programming with these graph descriptors
- Converting clusterization + spacepoint formation + seeding to this model
- Drop-in compatibility with `traccc::algorithm` through `traccc::graph_algorithm`
- Feedback and comments **very** welcome!