Using Alpaka in CMSSW framework

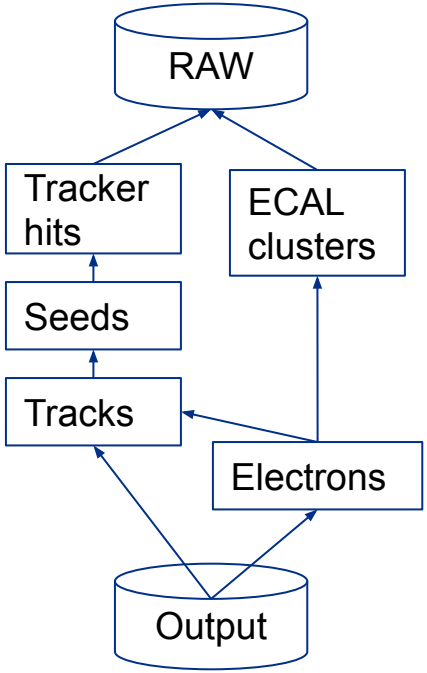
Matti Kortelainen

HSF Frameworks WG meeting

28 June 2023

CMSSW

- CMSSW is the data processing software (framework) of CMS
- Implements multithreading using oneTBB utilizing tasks as concurrent units of work
- Event data is processed through a DAG of algorithms (“framework modules”)
  - DAG is defined by data dependencies between the modules
    - Declared by the module constructors
  - No explicit notion of the DAG though, scheduling decisions are local
- Model for offloading: modules talk directly to the offloading API
  - Run kernels concurrently, overlap with data transfers
  - Offload chains or DAGs of modules
  - Minimize data transfers between CPU and GPU

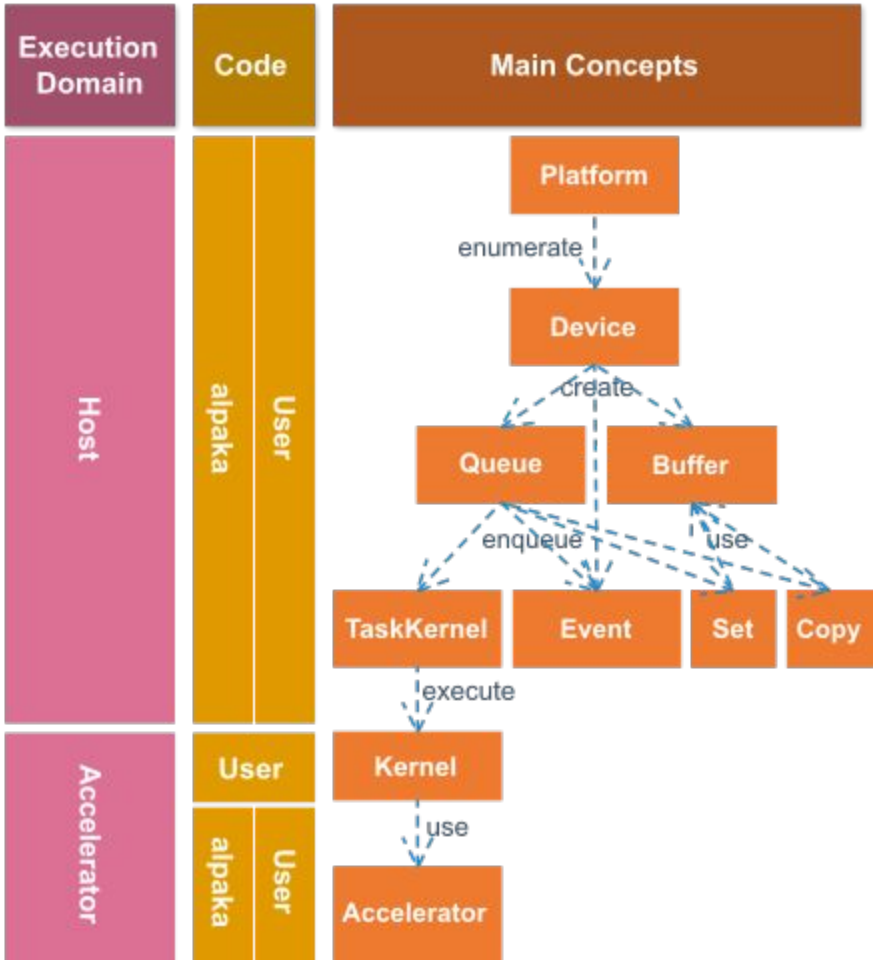


Alpaka: https://github.com/alpaka-group/alpaka

- Templated, header-only C++ library for compute accelerator development
- Aims to provide performance portability through the abstraction of the underlying levels of parallelism
  - Abstraction level is similar to CUDA
- Backends include CPU serial, OpenMP 2, TBB, CUDA, HIP, SYCL (experimental)
- Code for each backend is compiled with the “native compiler” of the platform
- Backend(s) are enabled at compilation time with macros
  - E.g. `-DALPAKA_ACC_GPU_CUDA_ENABLED` to enable CUDA backend
  - With `-DALPAKA_HOST_ONLY` can restrict the code to a subset that can be compiled with the host compiler
- Has CMake integration, but we have not tried it out

Alpaka concepts

- Platform: e.g. CPU, CUDA, HIP
- Device: e.g. CPU, CUDA, HIP
  - >= 1 per platform
- Queue (“CUDA stream”)
  - Work queue, blocking or non-blocking
  - Arbitrary number of Queues per Device
- Event (“CUDA event”)
  - Communicate completion of queued work
  - Arbitrary number of Events per Device
- Accelerator
  - Provides Kernel its current work index, access to shared memory and atomics, etc
- Buffer: like std::shared_ptr that knows size



Fermilab

What we wanted to achieve

- **Requirement: portable code between CPU and NVIDIA GPU**
  - Other GPU vendors (AMD, Intel) added bonus
- Take advantage of Alpaka-CPU and Alpaka-CUDA modules defining the same (or very close) algorithm
  - Was not the case with CUDA modules: functionality was split in different ways in (legacy) CPU and CUDA modules
- Be able to run Alpaka-CPU and Alpaka-CUDA modules in the same process
  - Allows comparing outputs event-by-event or object-by-object
- Minimize the amount of code we need to compile with device compiler
  - Once upon a time we got bit by `nvcc` not supporting C++17 yet
- Portable configuration
  - Single configuration file describes the behavior on all platforms
  - Framework uses a hash of the configuration to segregate data

Backend definition

- Alpaka's "backend" "concept" is somewhat loosely defined
- In CMSSW, for each backend we define several type aliases specifying
  - Platform, Device, Queue, Event, Accelerator types
  - Queue can be blocking or non-blocking
    - We use blocking queue for CPU, and non-blocking for CUDA and ROCm
  - Currently enabled: CPU serial, CUDA, ROCm
  - We also define a macro to specify a namespace for all per-backend user code

```
#ifdef ALPAKA_ACC_GPU_CUDA_ENABLED
namespace alpaka_cuda_async {
  using namespace alpaka_common;

  using Platform = alpaka::PltfCudaRt;
  using Device = alpaka::DevCudaRt;
  using Queue = alpaka::QueueCudaRtNonBlocking;
  using Event = alpaka::EventCudaRt;

  template <typename TDim>
  using Acc = alpaka::AccGpuCudaRt<TDim, Idx>;
  using Acc1D = Acc<Dim1D>;
  using Acc2D = Acc<Dim2D>;
  using Acc3D = Acc<Dim3D>;

}  // namespace alpaka_cuda_async

#define ALPAKA_ACCELERATOR_NAMESPACE alpaka_cuda_async
```

Framework module code

```
namespace ALPAKA_ACCELERATOR_NAMESPACE {

  class TestAlpakaProducer : public global::EDProducer<> {
  public:
    TestAlpakaProducer(edm::ParameterSet const& config)
        : deviceToken_{produces()}, size_{config.getParameter<int32_t>("size")} {}

    void produce(edm::StreamID sid, device::Event& event, device::EventSetup const&) const override {
      // run the algorithm, potentially asynchronously
      portabletest::TestDeviceCollection deviceProduct{size_, event.queue()};
      algo_.fill(event.queue(), deviceProduct);

      // put the asynchronous product into the event without waiting
      event.emplace(deviceToken_, std::move(deviceProduct));
    }
    const device::EDPutToken<portabletest::TestDeviceCollection> deviceToken_;
    const int32_t size_;

    // implementation of the algorithm
    TestAlgo algo_;
  };
```

Framework module code

```
namespace ALPAKA_ACCELERATOR_NAMESPACE {
```

Backend-specific namespace guarantees unique symbols

```
class TestAlpakaProducer : public global::EDProducer<> {
public:
  TestAlpakaProducer(edm::ParameterSet const& config)
      : deviceToken_{produces()}, size_{config.getParameter<int32_t>("size")} {}

  void produce(edm::StreamID sid, device::Event& event, device::EventSetup const&) const override {
    // run the algorithm, potentially asynchronously
    portabletest::TestDeviceCollection deviceProduct{size_, event.queue()};
    algo_.fill(event.queue(), deviceProduct);

    // put the asynchronous product into the event without waiting
    event.emplace(deviceToken_, std::move(deviceProduct));
  }
  const device::EDPutToken<portabletest::TestDeviceCollection> deviceToken_;
  const int32_t size_;

  // implementation of the algorithm
  TestAlgo algo_;
};
```

Framework module code

```
namespace ALPAKA_ACCELERATOR_NAMESPACE {

  class TestAlpakaProducer : public global::EDProducer<> {
  public:
    TestAlpakaProducer(edm::ParameterSet const& config)
        : deviceToken_{produces()}, size_{config.getParameter<int32_t>("size")} {}

    void produce(edm::StreamID sid, device::Event& event, device::EventSetup const&) const override {
      // run the algorithm, potentially asynchronously
      portabletest::TestDeviceCollection deviceProduct{size_, event.queue()};
      algo_.fill(event.queue(), deviceProduct);

      // put the asynchronous product into the event without waiting
      event.emplace(deviceToken_, std::move(deviceProduct));
    }

    const device::EDPutToken<portabletest::TestDeviceCollection> deviceToken_;
    const int32_t size_;

    // implementation of the algorithm
    TestAlgo algo_;
  };
```

Some of the behavior depends on the backend. These components are also defined in `ALPAKA_ACCELERATOR_NAMESPACE`

Fermilab

Directory structure and compilation

```
HeterogeneousCore/AlpakaTest/
├── BuildFile.xml
├── interface/
│   │
│   │
│   └── AlpakaESTestData.h
├── plugins/
│   │
│   │
│   │
│   │
│   ├── BuildFile.xml
│   └── TestAlpakaAnalyzer.cc
└── src/
    │
    │
    └── ES_AlpakaESTestData.cc
```

- Each CMSSW package may result in
  - 0 or 1 shared library
    - Other libraries can depend on
  - Any number (usually 1 if any) of plugin libraries
    - Other libraries **may not** depend on
    - Framework loads the plugins that provide the components defined by the job configuration file
- CMSSW package has subdirectories that have specific meaning
  - `interface`: public interface of the shared library
  - `src`: implementation of the shared library
  - `plugins`: implementation of the plugin library/libraries

Directory structure with Alpaka

```
HeterogeneousCore/AlpakaTest/
├── BuildFile.xml
├── interface/
│   ├── alpaka/
│   │   └── AlpakaESTestData.h
│   └── AlpakaESTestData.h
├── plugins/
│   ├── alpaka/
│   │   ├── TestAlgo.dev.cc
│   │   ├── TestAlgo.h
│   │   └── TestAlpakaProducer.cc
│   ├── BuildFile.xml
│   └── TestAlpakaAnalyzer.cc
└── src/
    ├── alpaka/
    │   └── ES_AlpakaESTestData.cc
    └── ES_AlpakaESTestData.cc
```

- Added `alpaka/` subdirectory to all `interface/`, `src/`, `plugins/`
- The code in the `alpaka/` subdirectory is compiled once for each Alpaka backend, and results a separate shared object
  - Shared libraries
    - One Alpaka-independent library
    - One Alpaka-dependent library per backend
  - Plugin libraries
    - One Alpaka-independent plugin library
    - One Alpaka-dependent plugin library per backend
- The code in `alpaka/` subdirectory generally enclosed in `ALPAKA_ACCELERATOR_NAMESPACE`

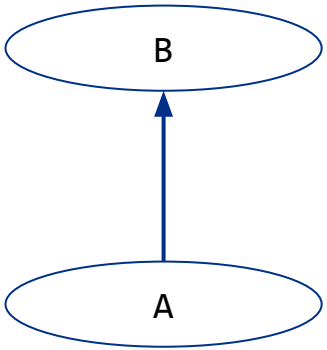
Compilation of Alpaka-dependent code

```
HeterogeneousCore/AlpakaTest/
├── BuildFile.xml
├── interface/
│   ├── alpaka/
│   │   └── AlpakaESTestData.h
│   └── AlpakaESTestData.h
├── plugins/
│   ├── alpaka/
│   │   ├── TestAlgo.dev.cc
│   │   ├── TestAlgo.h
│   │   └── TestAlpakaProducer.cc
│   ├── BuildFile.xml
│   └── TestAlpakaAnalyzer.cc
└── src/
    ├── alpaka/
    │   └── ES_AlpakaESTestData.cc
    └── ES_AlpakaESTestData.cc
```

- There are files that need to be compiled for each backend, but do not contain any device code
  - E.g. framework module definition
  - Have **`.cc`** suffix, compiled with the host compiler (`gcc`)
  - Are allowed to call host-side API, e.g. allocate memory
- There are files that need to be compiled for each backend and contain device code
  - Algorithm implementation
    - Device functions, kernel launches
  - Have **`.dev.cc`** suffix, compiled with the device compiler of the backend (`gcc`, `nvcc`, `hipcc`)
- Linked into one shared object according to the rules of the corresponding device compiler

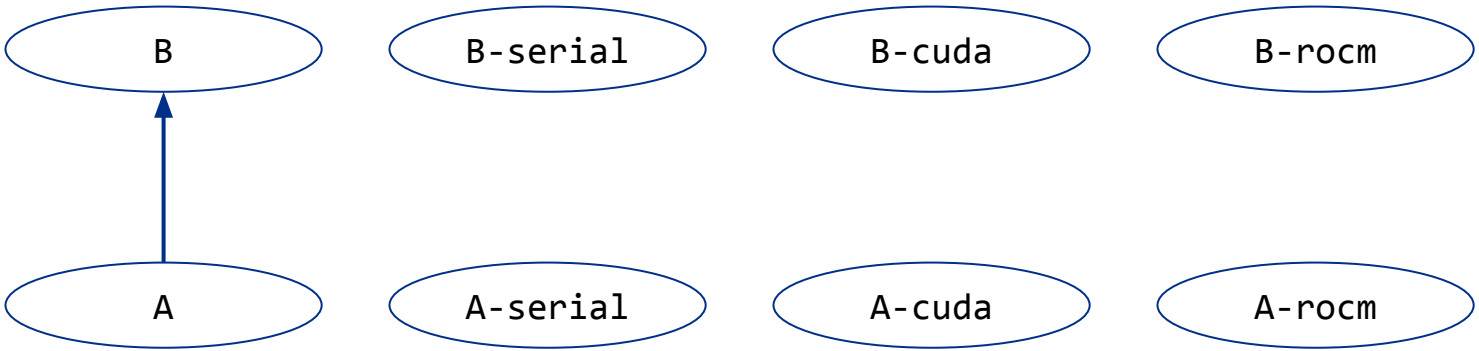
Handling dependencies on Alpaka-enabled packages

- Package A depends on package B
  - Shared library of A is linked against the shared library of B



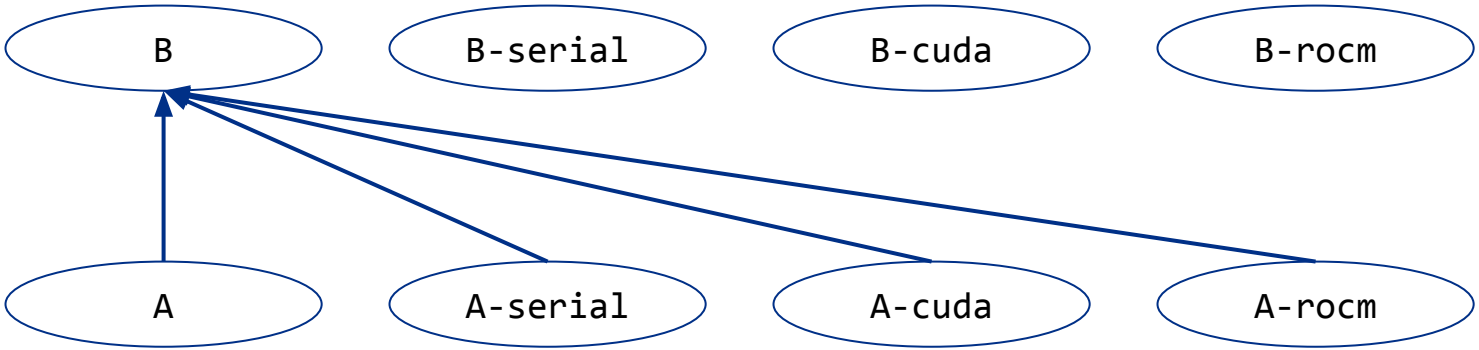
Handling dependencies on Alpaka-enabled packages

- Alpaka-enabled package A depends on Alpaka-enabled package B
  - Alpaka-independent shared library of A is linked against the Alpaka-independent shared library of B



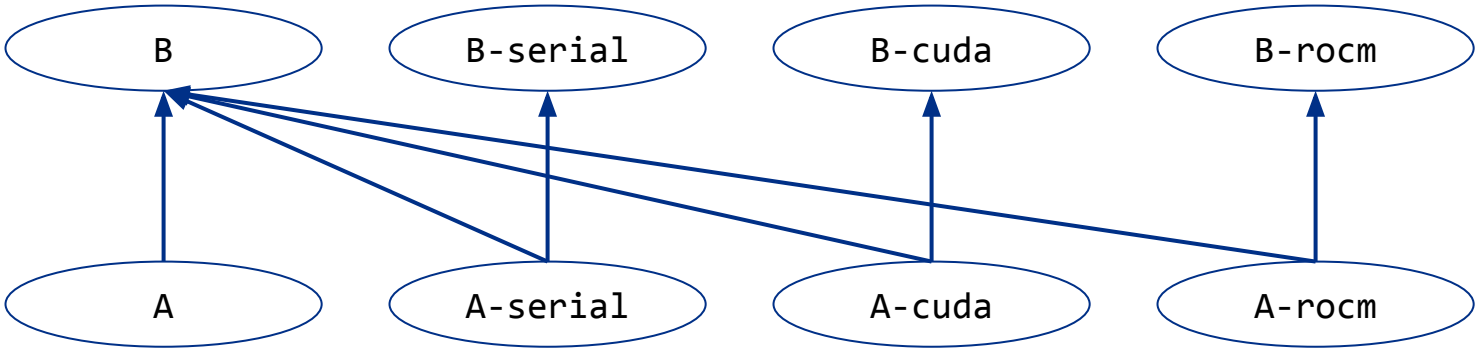
Handling dependencies on Alpaka-enabled packages

- Alpaka-enabled package A depends on Alpaka-enabled package B
  - Alpaka-independent shared library of A is linked against the Alpaka-independent shared library of B
  - Each alpaka-dependent shared library of A is linked against
    - Alpaka-independent shared library of B



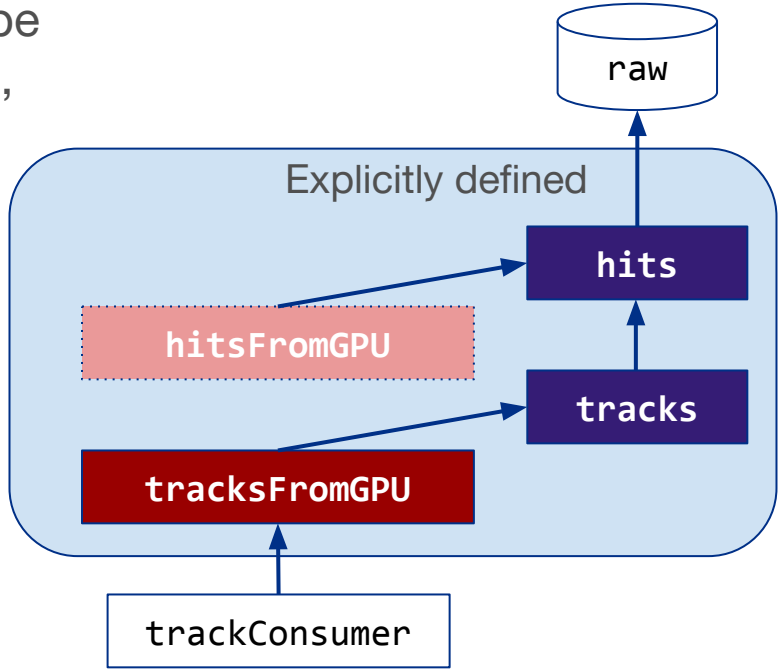
Handling dependencies on Alpaka-enabled packages

- Alpaka-enabled package A depends on Alpaka-enabled package B
  - Alpaka-independent shared library of A is linked against the Alpaka-independent shared library of B
  - Each alpaka-dependent shared library of A is linked against
    - Alpaka-independent shared library of B
    - Alpaka-dependent shared library of B of the same backend



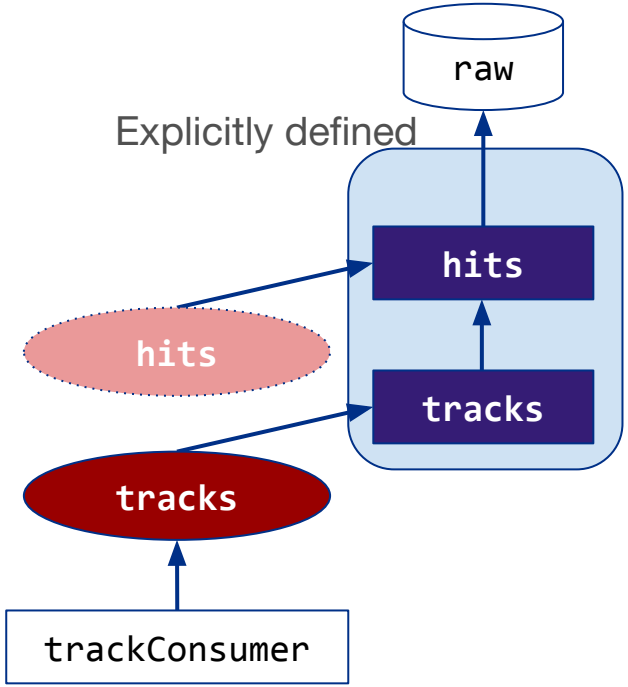
Implicit event data product copy from device to host

- Our CUDA framework module pattern required all data transfers between host and device to be explicitly implemented by the user as modules, and specified in the job configuration
  - Tedious to develop and maintain



Implicit event data product copy from device to host

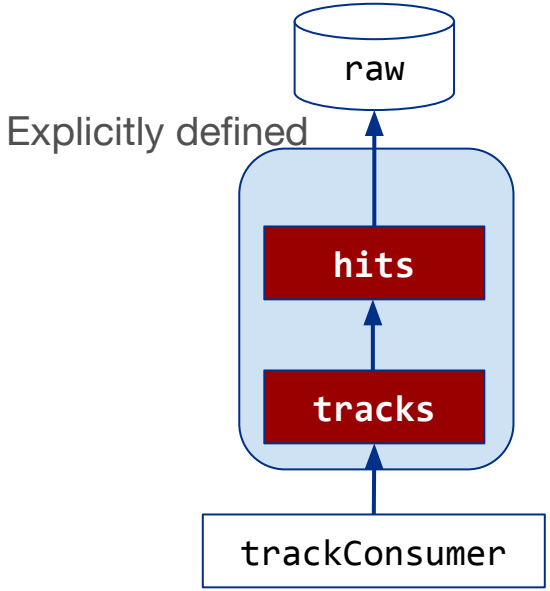
- Constraints placed in the Alpaka module pattern allowed to make some of the device-to-host transfers implicit
  - Alpaka module is considered to produce its data products in “device memory space”
  - Framework registers another “node” in the module DAG to copy the device-side data product to the host
    - User still needs to define the function for the copy
      - But at least the copy code is now placed close to the data format class definition
    - The node copying the data is scheduled and run only if some other module consumes the host-side data
  - Same label used for both device-side and host-side data products
    - Device-side data product has different, wrapped type



Fermilab

Implicit event data product copy from device to host

- Constraints placed in the Alpaka module pattern allowed to make some of the device-to-host transfers implicit
  - In CPU serial backend the copying is avoided
    - “Device-side” data product is used directly by all host-side consumers



Unified configuration for CPU and non-CPU algorithms

- Want jobs for a workflow to be able to run at any site
- Want same configuration for all jobs in a workflow
  - Be agnostic to the kind of hardware being used for a given job
  - Hash of configuration already used by framework to segregate data from different workflows
- Earlier with CUDA wanted to be able to keep CPU and GPU algorithms separate
- Now want to unify the CPU and GPU algorithms as much as feasible
  - To minimize maintenance effort and chances for e.g. configuration mistakes
- Use provenance tracking to store the choice of technology along the Event
  - Framework already tracks the input data of each module Event-by-Event
- Such workflows need to be validated with all technology permutations

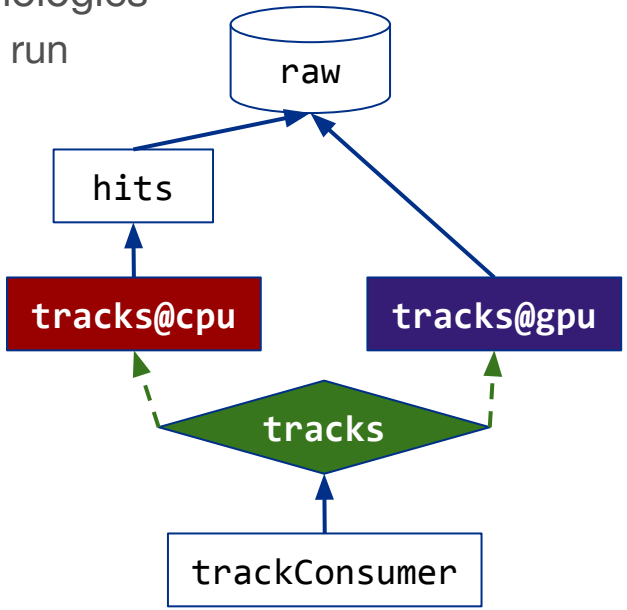
Semi-portable configuration with “SwitchProducer”

- SwitchProducer
  - Allows specifying multiple modules associated to same module label
  - At runtime picks one to be run based on available technologies
    - Consumers dictate which producers are scheduled and run

```
hits = Producer(“HitsProducer”,
    input = “raw”
)

tracks = SwitchProducer(
    cpu = Producer(“TrackProducer”,
        input = “hits”),
    gpu = Producer(“TrackProducerGPU”,
        input = “raw”)
)

trackConsumer = Producer(“TrackConsumer”,
    input = “tracks”
)
```



Semi-portable configuration with “SwitchProducer”

- SwitchProducer
  - Allows specifying multiple modules associated to same module label
  - At runtime picks one to be run based on available technologies
    - Consumers dictate which producers are run

```
hits = Producer(“HitsProducer”,
    input = “raw”
)

tracks = SwitchProducer(
    cpu = Producer(“TrackProducer”,
        input = “hits”),
    gpu = Producer(“TrackProducerGPU”,
        input = “raw”)
)

trackConsumer = Producer(“TrackConsumer”,
    input = “tracks”
)
```

Problem: need to define (and construct) the modules for all platforms. What if some platform does not support (all) GPUs?

For example

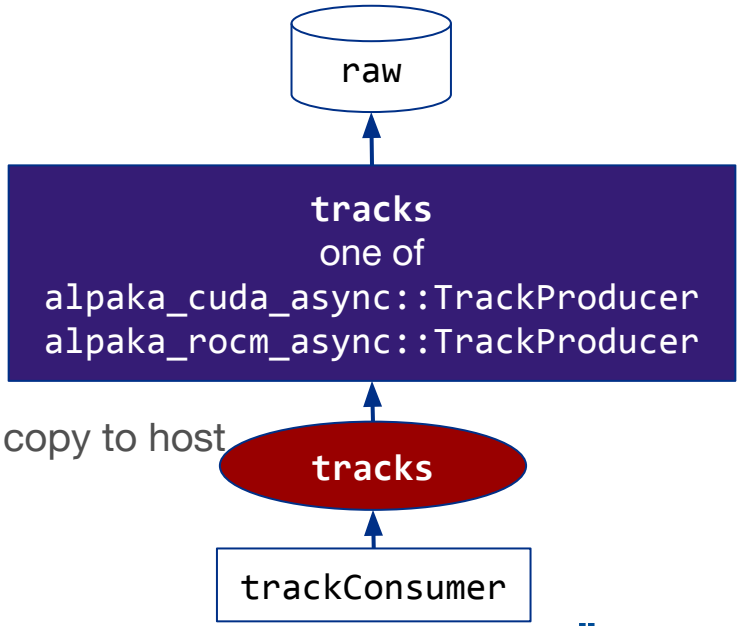
- CUDA doesn’t support a new GCC version yet
- We haven’t bothered with ROCm support on ARM or PPC

Portable configuration with “module type resolver”

- Added several customization points in the module plugin loading code
  - Decorate relevant module C++ types in configuration with **@alpaka** suffix
  - At runtime pick one concrete module C++ type based on worker node hardware
- Mechanism extensible for other technologies with hardware-specific backends as well
  - E.g. Tensorflow or other ML inference

```
tracks = Producer(“TrackProducer@alpaka”,
    input = “raw”
)

trackConsumer = Producer(“TrackConsumer”,
    input = “tracks”
)
```

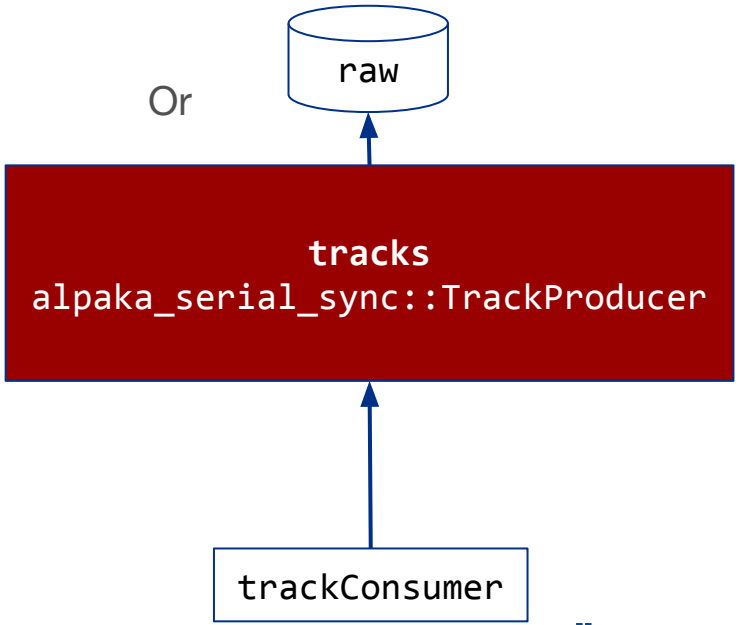


Portable configuration with “module type resolver”

- Added several customization points in the module plugin loading code
  - Decorate relevant module C++ types in configuration with **@alpaka** suffix
  - At runtime pick one concrete module C++ type based on worker node hardware
- Mechanism extensible for other technologies with hardware-specific backends as well
  - E.g. Tensorflow or other ML inference

```
tracks = Producer(“TrackProducer@alpaka”,
    input = “raw”
)

trackConsumer = Producer(“TrackConsumer”,
    input = “tracks”
)
```



Summary

- Reviewed how Alpaka is used in CMSSW framework
  - Want to have as unified overall look and feel between the backends as feasible
    - Configuration, framework module code, algorithm code, data structures
  - Additional layer of framework module base classes to deal with data product memory spaces, synchronization, differences between backend behavior
  - Separate non-Alpaka host code, Alpaka host-only code, and Alpaka device code into separate files
    - Non-Alpaka code and Alpaka code linked into separate shared libraries
  - Implicit event data product copies
- We don’t have much operational experience yet with this pattern
  - Expect to evolve based on experience
  - We have some ideas how to improve and automate some things more

Spares

External worker mechanism

- Replace blocking waits with a callback-style solution
- Traditionally the algorithms have one function called by the framework, `produce()`
- That function is split into two stages
  - `acquire()`: Called first, launches the asynchronous work
  - `produce()`: Called after the asynchronous work has finished
- `acquire()` is given a reference-counted smart pointer to the task that calls `produce()`
  - Decrease reference count when asynchronous work has finished
  - Capable of delivering exceptions