CMSSW Framework Status and Roadmap

Christopher Jones, Matti Kortelainen

Fermilab Frameworks Workshop

5 June 2023

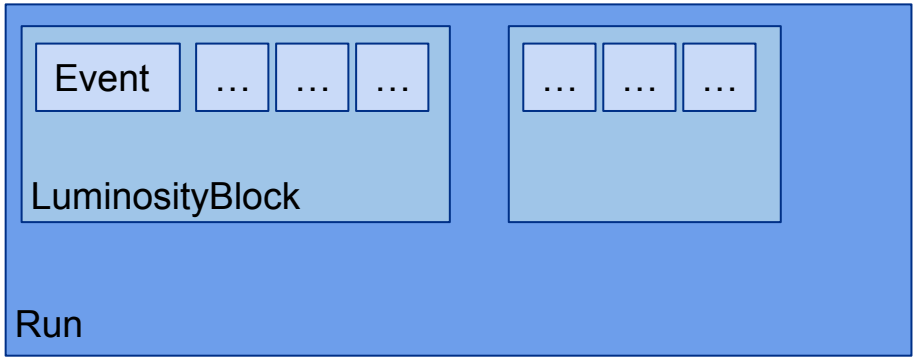
Introduction and outline

- CMS' data processing framework (CMSSW) is multithreaded with Intel oneTBB
  - Main, but not only, motivation was to reduce the memory requirement per CPU core
  - Initial multithreading development and subsequent efficiency improvements have been the main development theme for the framework on the past decade
- Focusing here on high-level overview of select features:
  - Data transition system
  - Conditions system
  - Module categories
  - Configuration with python
  - Levels of concurrency
  - I/O
  - Facilities for accelerator use
  - Future plans

Data transition system

Manages processing of collision data

- Hierarchy
  - Event: smallest atomic unit of data
  - LuminosityBlock
    - Corresponds to *art*'s SubRun
    - Collection of Events
    - Corresponds to ~23.3 s of data
    - Smallest atomic unit for Intervals of Validity in the conditions system
  - Run
    - Collection of LuminosityBlocks
    - Corresponds to several hours of data
    - E.g. trigger menu does not change

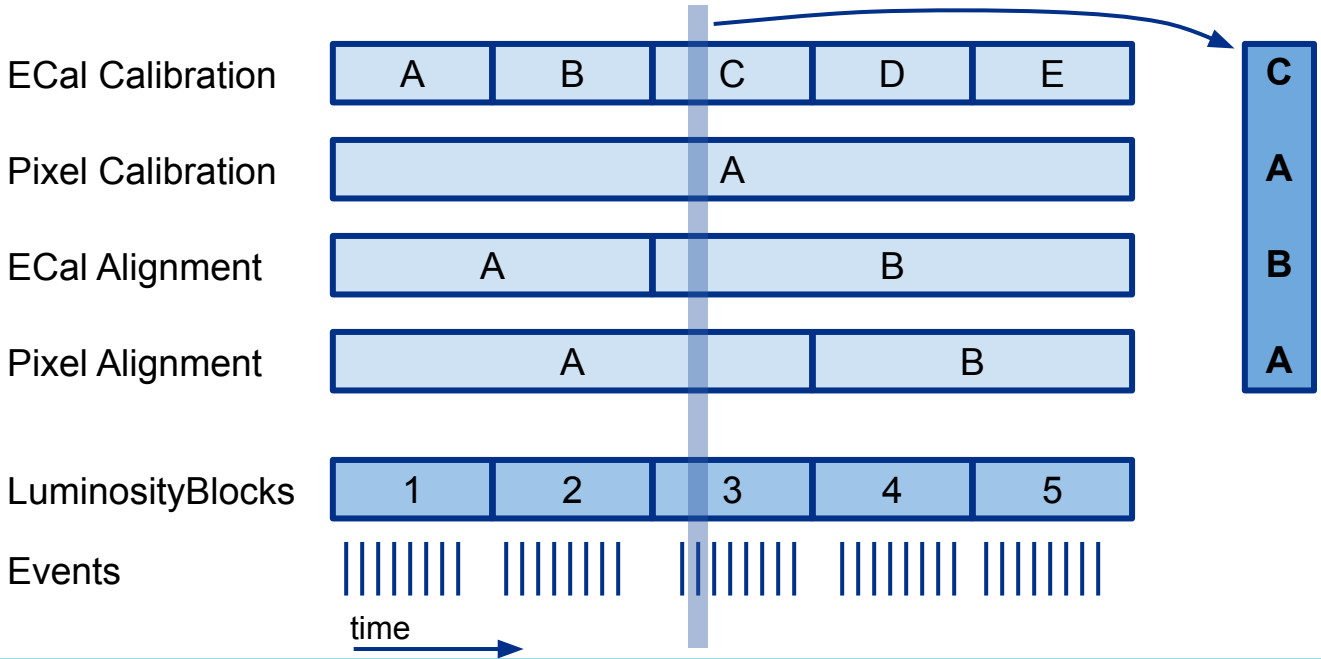


- Orthogonally: ProcessBlock
  - Process-level data storage

Conditions system

Conditions data: calibrations, geometry, etc

- Data can have many versions that have Interval of Validity (IOV)
- Provides uniform access mechanism to all data constrained by IOV



An “EventSetup” is formed from Records that have an IOV that overlaps with the moment in time being studied

Fermilab

Module categories

- Data transition system (similar to *art*)
  - Source: produce the transitions (event, lumi, run), possibly also data products
  - Analyzers: can access data products
  - Producers: can access and produce data products
  - Filters: can access (and produce) data products, and stop the processing of an Event in a trigger Path
  - Output modules: stores data products into a file
- Conditions system
  - Sources: produce the IOV transitions for the Records
  - Modules: produce data products into the Records
- Services
  - Can register callbacks for the transitions in either system
  - Must not affect physics results

Configuration with Python

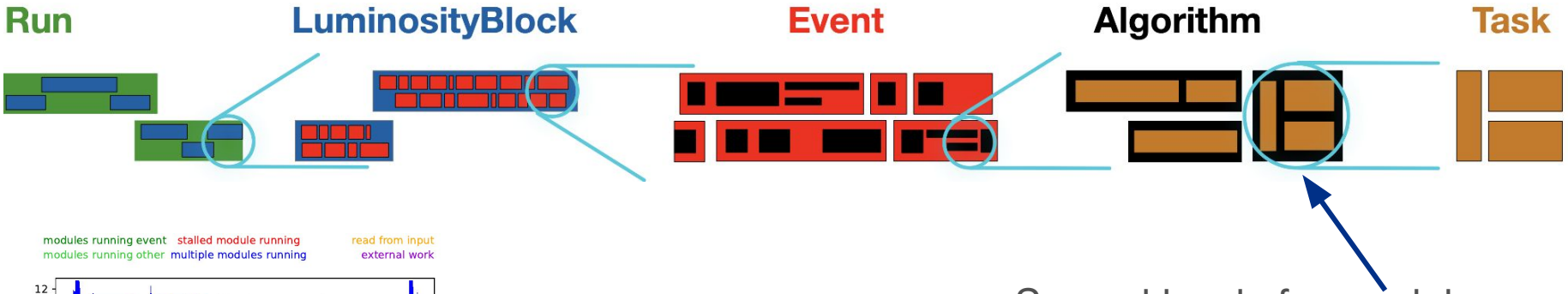
- CMSSW jobs are configured directly with Python
  - Powerful, avoids additional configuration generator layer
- Can use programming constructs directly in the configuration
  - ifs, loops, functions, etc
  - E.g. `argparse` to pass command line arguments
  - Need some policies and conventions to avoid overcomplicating the configurations
- Any configuration file can be “dumped” to see expanded result of the program

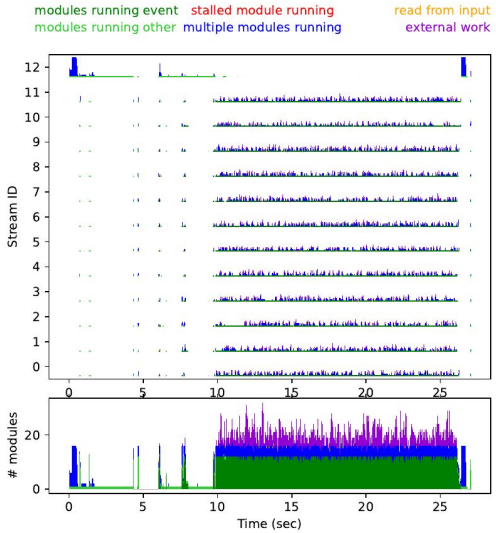
```
import FWCore.ParameterSet.Config as cms

process = cms.Process("TEST")

process.maxEvents = 100
process.options.numberOfThreads = 4

process.source = cms.Source("PoolSource",
    fileNames = cms.untracked.vstring("file:input.root")
)

process.producer = cms.EDProducer("TestProducer",
    value = cms.int32(42)
)

from Some.Package.analyzer_cfi import analyzer
process.analyzer = analyzer.clone(source = "producer")

process.out = cms.OutputModule("PoolOutputModule",
    fileName = cms.untracked.string("output.root")
)

process.task = cms.Task(process.producer)
process.path = cms.Path(process.analyzer, process.task)
process.endpath = cms.EndPath(process.out)
```

Some threading nomenclature (in this talk)

- "Thread safe"
  - Same object (member function) can be called simultaneously from multiple threads
  - C++11 expects operations on `const` objects to be thread-safe
    - Either bitwise `const`, or internally synchronized
- "Thread friendly"
  - Same object can be called from single thread at a time, but different objects can be called simultaneously from different threads
    - No mutable global state such as writable class-`static` or file-`static` variables
- "Thread hostile"
  - Same and different objects can be called only from single thread at a time
    - E.g. objects do unprotected writes to shared date
- "Thread efficient"
  - E.g. single mutex for all functions is safe, but not efficient

Available concurrency levels in CMSSW





Tools to monitor how efficiently the concurrent events are processed by the modules

- Several levels for module thread-friendliness
  - Module declares its thread-friendliness level by the base class it inherits from
    - I.e. each level has its own base class, and a part of the API is specific to the level

Fermilab

Levels of module thread-friendliness

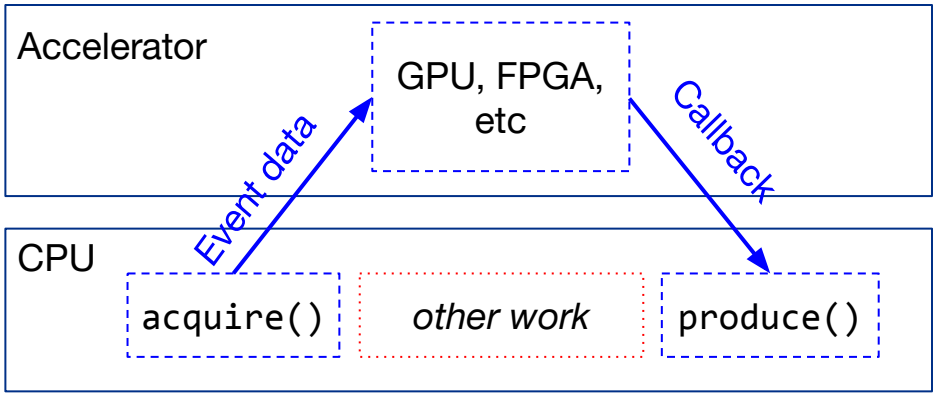
- "Global": module is thread safe (*art*'s "shared" with call `async<Event>()`)
  - Framework has exactly one instance of the module whose member functions may be called concurrently
  - Most efficient
- "Stream": module is thread friendly (*art*'s "replicated")
  - Framework has one replica of the module for each concurrent Event ("stream")
  - For each concurrent event, only one of the replicas is used
    - I.e. member functions of any one module instance are not called concurrently
  - CPU-wise as efficient as "Global", but uses more memory
- "One": module is thread hostile (*art*'s "shared" with call `serialize(resources_)`)
  - Framework has one instance of the module, and serializes the calls to the module
  - Becomes very inefficient at high thread count
  - Necessary ability to deal with e.g. non-thread-safe 3rd party libraries

Input and output

- I/O in CMSSW
  - Online (DAQ) outputs simple custom file format, where data for each event is serialized with ROOT as a single blob
    - Converted to a ROOT file in Tier-0
  - Offline processing: ROOT files (`TTree`) for input and output
- XRootD is used to stream data from remote data centers
- Framework uses ROOT dictionaries for reflection also outside of I/O
- On HL-LHC timescale we are working with the ROOT team to make their new `RNTuple` storage format usable in CMSSW for CMS' data formats
  - We have a prototype for `RNTuple`-using output module for the "analysis ntuple" (NanoAOD)
  - Expect to result in smaller files and higher threading efficiency on high thread counts

Compute accelerators

- CMS' general approach has been to
  - Implement general mechanisms, that are agnostic of accelerator specifics, in the framework
  - Implement accelerator specific code as a layer between the framework and user code
  - Allow gradual adoption, keep rest of codebase unchanged
- Generic mechanism for "outside of CMSSW" work called "external worker"
  - Allows CPU thread to do other work
    - Once external work finishes, new task added to TBB
  - Used for
    - Direct GPU usage via CUDA/Alpaka
    - SONIC (ML inference as a service)
    - GeantV integration exercise



CHEP19 doi:10.1051/epjconf/202024505009

Fermilab

Direct accelerator usage with CUDA/Alpaka

- Main use case has been CMS' High Level Trigger (HLT)
- Some defining characteristics
  - Chains of modules that keep the data in GPU memory, minimizing synchronization
  - Ability to run a configuration on "any hardware" ("portable configuration")
  - Implemented as a layer on top of the framework
    - Framework itself stayed independent of any accelerator technology
- CUDA support was added in 2020
  - Used in production at HLT since 2022 data taking
- CUDA code is being migrated to use Alpaka portability library
  - Technically have ability to use AMD GPUs, but so far not really tested
- Expect to retire direct CUDA in timescale of a year or so

Future work on accelerator support

- Main theme is efficient use of accelerators
- Ability for asynchronous execution in Conditions system modules
  - First use case: copy calibration data from CPU to GPU memory asynchronously
  - Framework-level support already exists, Alpaka-level support to be done
- Record information on worker node hardware in the data files
  - Stored module provenance information is not good enough to describe the past behavior anymore
- Want to automate the ability of deleting temporary Event data products after they are no longer needed
  - Could be useful especially for low-memory GPUs
- Develop better mechanisms to deal with “memory spaces” of data products
  - Want to be able to handle both discrete memory and unified memory cases
- Want to investigate batching of data from multiple Events to reduce overheads

Spares

More details on 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