Tracking summary

Paul Gessinger
CERN
2024-07-17

What is ACTS?

- Experiment-independent toolkit for track reconstruction applications
- Modern architecture and code, unit tested, continuous integration
- Minimal external dependencies, easy to build
- Robust concurrency through thread-safety by design
- Community platform for R&D across various experiment

Goals

- Provide established algorithms in a modern package
- Provide testbed for R&D activities including new algorithms, machine learning, heterogeneous computing

My EP R&D involvement

- Joined EP R&D in April 2021 as a Fellow
- Worked 3 years on the ACTS project
- Presentations in this forum:
  - 2021-05-26, 2021-12-15, 2022-06-15, 2022-11-09, 2023-03-15, 2023-11-15
- Conference contributions:
  - CTD2023 (talk)
  - CHEP2023 (talk, paper)
  - ACAT2021 (poster)

Direct developments

- ACTS paper in 2021
- Python bindings for the Examples
- Reproducibility tests via ROOT hashes
- Build-time memory consumption monitoring
- New (C)KF extension mechanism
- Type-erased `SourceLink`
- CI-bridge + GPU CI setup
- Robust physics monitoring
- DD4hep integration rework (dropped `ActsExtension` requirement)
- High-level track EDM + backend abstraction
- Major documentation update and push
- PODIO track EDM backend including dynamic columns
- Floating point exception monitoring

New since November 2023

- Much improved EDM4hep reading
- Major vertexing refactor (build-time memory reduction)
- First public ATLAS+ACTS performance results
- Synthesis of geometry work (ongoing)

Python bindings in the Examples

Python bindings for the examples

ACTS examples

- ACTS ships with a set of **examples** to show assembly of a track reconstruction chain
- Ships with a minimal event processing framework: not intended for production

- **Previously:** large number of executables for different purposes: controllable via command line arguments
- Drawback: large number of options for everything, expose almost all configuration via CLI arguments
- Added python bindings to example classes: allows writing simply python scripts to run example payloads
  - **Advantage:** can follow configuration flow, understand what is actually happening
- Deprecated and finally removed executables in this year!

Python script example

```
detector, trackingGeometry, decorators = acts.examples.GenericDetector.create()
field = acts.ConstantBField(acts.Vector3(0, 0, 2 * u.T))
rnd = acts.examples.RandomNumbers()

s = acts.examples.Sequencer(
    events=100, numThreads=-1, logLevel=acts.logging.INFO
)

s.addReader(someParticleInput) # e.g. particle gun, pythia8 ...

selector = acts.examples.ParticleSelector(level=acts.logging.INFO,
  inputParticles=inputParticles, outputParticles="particles_selected")
s.addAlgorithm(selector)

alg = acts.examples.FatrasSimulation(
    level=acts.logging.INFO, randomNumbers=rnd, trackingGeometry=trackingGeometry,
    magneticField=field, generateHitsOnSensitive=True, # + input/output collections
)
s.addAlgorithm(alg)

s.addWriter(someWriter) # e.g. CSV, ROOT, ...
```

Continuous integration developments

Reproducibility tests at the python level

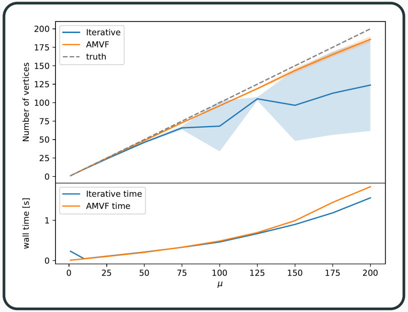
- *Old* C++ example executables were largely **untested**
- Added tests for many examples implemented in python
- Cover use cases: Magnetic field writing, digitization, HepMC3 recording, FATRAS, geometry construction, material recording/mapping/ validation, particle gun, propagation tests, Pythia8 input, seeding, truth tracking, CKF track finding
- Tests run in CI, check multi-threaded execution succeeds, asserts outputs in some cases
- Added **reproducibility tests**: ROOT outputs are *hashed*, current test results are compared against stored hash
- Hashes are ordering independent. Can test
  - **Multi-threaded** reproducibility
  - Functional regressions (same output as before)

Physics performance monitoring

- **Has become one of the main ways to test new developments now!**
- Implemented workflows:
  - Truth tracking with KF and GSF
  - CKF + seeding, truth estimation, truth smearing + vertexing
  - CKF on $t\bar{t}$ event at pile-up 200
  - Fatras and G4 simulation
  - Dedicated vertexing workflow
  - + more
- Runs histogram comparisons
- Further development planned to make this more parameterizable

Physics performance monitoring

- **Has become one of the main ways to test new developments now!**
- Implemented workflows:
  - Truth tracking with KF and GSF
  - CKF + seeding, truth estimation, truth smearing + vertexing
  - CKF on $t\bar{t}$ event at pile-up 200
  - Fatras and G4 simulation
  - Dedicated vertexing workflow
  - + more
- Runs histogram comparisons
- Further development planned to make this more parameterizable



Physics performance monitoring

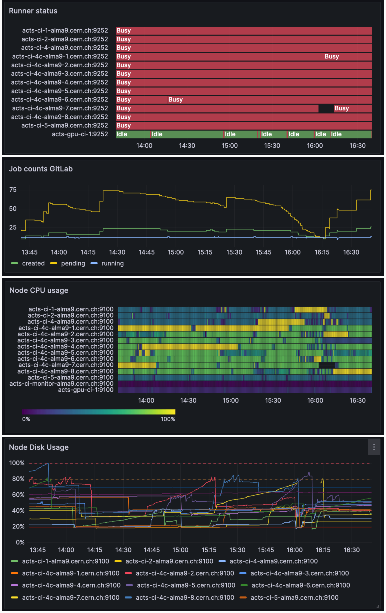
- **Has become one of the main ways to test new developments now!**
- Implemented workflows:
  - Truth tracking with KF and GSF
  - CKF + seeding, truth estimation, truth smearing + vertexing
  - CKF on $t\bar{t}$ event at pile-up 200
  - Fatras and G4 simulation
  - Dedicated vertexing workflow
  - + more
- Runs histogram comparisons
- Further development planned to make this more parameterizable

📊: Physics performance monitoring for `bede82e`

✅ CKF truth_smeared
✅ IVF truth_smeared
✅ AMVF truth_smeared
✅ Track Summary CKF truth_smeared
✅ Seeding truth_estimated
✅ CKF truth_estimated
✅ IVF truth_estimated
✅ AMVF truth_estimated
✅ Track Summary CKF truth_estimated
✅ Seeding seeded
✅ CKF seeded
✅ IVF seeded
✅ AMVF seeded
✅ AMVF (+grid seeder) seeded
✅ Track Summary CKF seeded
✅ Seeding orthogonal
✅ CKF orthogonal
✅ IVF orthogonal
✅ AMVF orthogonal
✅ Track Summary CKF orthogonal
✅ Ambisolver seeded
✅ Ambisolver orthogonal
✅ Seeding ttbar
✅ CKF ttbar
✅ Ambisolver
✅ Track Summary CKF ttbar
✅ AMVF ttbar
✅ AMVF (+grid seeder) ttbar
✅ Particles ttbar
✅ Vertices ttbar
✅ Truth tracking (GSF)
✅ Truth tracking
✅ Truth tracking (GX2F)
✅ Particles fatras
✅ Particles geant4

CI-bridge and GPU CI setup

- Have custom setup to run CI jobs on CERN resources
  - Currently $\mathcal{O}(10)$ slots running in VMs + 1 physical machine with a GPU attached
- Runs our FPE monitoring jobs (see next slides)
- Runs LCG-based jobs (more robust CVMFS access)
- VM monitoring using Prometheus + Grafana



FPE monitoring

FPE monitoring

```
volatile float j = 0.0;
volatile float r
  = 123 / j;  // MARK: fpeMask(FLTDIV, 1, #1234)

// MARK: fpeMaskBegin(FLTINV, 1, #2348)
m_err_eLOC0[ipar].push_back(
  std::sqrt(
    covariance(
      Acts::eBoundTime, Acts::eBoundTime)));
// MARK: fpeMaskEnd(FLTINV)
```

- FPE monitoring introduced as a plugin
- Uses custom infrastructure to enable FPE trap signal and handle
- Signal handler collects stack trace in async-safe way and records it
  - FPE locations are deduplicated based on the top-most stack frame **source code location!**
- Sequencer can be configured to ignore (**mask**) FPEs
- If configured: Sequencer terminates job when FPE is encountered (and not masked)
- At end of job: FPE are accumulated per algorithm and reported
  - Job is still failed if unmasked FPE are encountered

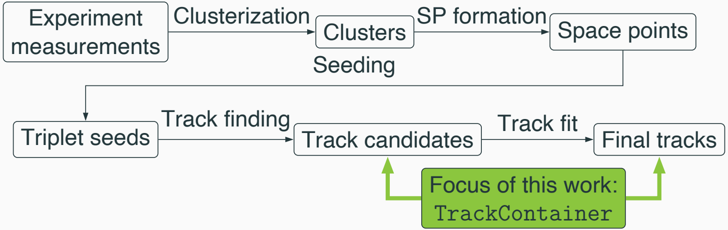
FPE masking

- Mask is a combination of source file and line range
  - Matching is performed from bottom to top in stack frame: if any frame matches mask, FPE is considered masked
  - Limit number of FPEs per event (but keep in mind that FPE state has to be reset manually)
- Typically want to **fix** the FPE, masking does not mask them outside of Examples
- Statistical process: if you get them depends on the workflow / inputs

```
FPE masks
FpeMask(Core/include/Acts/Utilities/GaussianMixtureReduction.hpp:88, FLTUND ≤ 1)
    const auto &[weight_l, pars_l, cov_l] = projector(cmp);

    cov += weight_l * cov_l;  // MARK: fpeMask(FLTUND, 1, #2347)

    ActsVector<D> diff = pars_l - mean;

/Users/pagessin/dev/acts/Core/include/Acts/Utilities/GaussianMixtureReduction.hpp

FpeMask(Core/include/Acts/Utilities/GaussianMixtureReduction.hpp:(198, 202], FLTUND ≤ 1)
  std::apply([&](auto... dsc) { (wrap(dsc), ...); }, angleDesc);

  // MARK: fpeMaskBegin(FLTUND, 1, #2347)
  const auto cov =
      gaussianMixtureCov(components, mean, sumOfWeights, projector, angleDesc);
  // MARK: fpeMaskEnd(FLTUND)

  return RetType{mean, cov};
}
/Users/pagessin/dev/acts/Core/include/Acts/Utilities/GaussianMixtureReduction.hpp

FpeMask(Core/include/Acts/Vertexing/AdaptiveMultiVertexFinder.ipp:(119, 123], FLTUND ≤ 1)
      }
    }
    // MARK: fpeMaskBegin(FLTUND, 1, #2590)
    bool keepVertex = isGoodVertex &&
                      keepNewVertex(vtxCandidate, allVerticesPtr, fitterState);
    // MARK: fpeMaskEnd(FLTUND)
    ACTS_DEBUG("New vertex will be saved: " << keepVertex);

    // Delete vertex from allVertices list again if it's not kept
/Users/pagessin/dev/acts/Core/include/Acts/Vertexing/AdaptiveMultiVertexFinder.ipp

FpeMask(Examples/Io/Root/src/RootTrackStatesWriter.cpp:(519, 533], FLTINV ≤ 1)

          // track parameters error
          // MARK: fpeMaskBegin(FLTINV, 1, #2348)
          m_err_eLOC0[ipar].push_back(
              std::sqrt(covariance(Acts::eBoundLoc0, Acts::eBoundLoc0)));
          m_err_eLOC1[ipar].push_back(
              std::sqrt(covariance(Acts::eBoundLoc1, Acts::eBoundLoc1)));
          m_err_ePHI[ipar].push_back(
              std::sqrt(covariance(Acts::eBoundPhi, Acts::eBoundPhi)));
          m_err_eTHETA[ipar].push_back(
              std::sqrt(covariance(Acts::eBoundTheta, Acts::eBoundTheta)));
          m_err_eQOP[ipar].push_back(
              std::sqrt(covariance(Acts::eBoundQOverP, Acts::eBoundQOverP)));
          m_err_eT[ipar].push_back(
              std::sqrt(covariance(Acts::eBoundTime, Acts::eBoundTime)));
          // MARK: fpeMaskEnd(FLTINV)

          // track parameters pull
          m_pull_eLOC0[ipar].push_back(
```

EDM4hep + PODIO integration

Tracks



- We ❤️ Tracks!
- High-level Track EDM added to ACTS
- Developments closely linked to ATLAS effort to build on top of `xAOD` infrastructure
- **But**: want EDM to fully generic as a first-class ACTS data type

`Acts::TrackContainer`

- New **client-focused** Event Data Model object as primary output of tracking
- Models track properties and gives access to track states for more information
- **Fully decoupled interface** seen by ACTS and client consumers from the backend implementation
  - Backend can be fully experiment-specific
- Can be augmented with additional columns, both at track and track-state level
- `VectorMultiTrajectory` + `VectorTrackContainer`: ACTS default purely transient[1] backends
- Generalized holder type: can optionally own or only reference backends
  - Optionally also via smart-pointers like `std::shared_ptr`
- Mirrors const-ness of backends

[1] Plan to make it persistable in the future

`Acts::TrackContainer` interface & access

- The basic unit is the container: tracks and track states are only views into the container
  - Modeled with an associated *proxy* type holding a pointer + index into the container
  - Track has a start index into the track state container ($\sim$ flat linked list of track states)

```
auto tracks = factory();

auto track = tracks.makeTrack();

track.parameters() = BoundVector::Zeros();
doSomethingWithTheTrack(track);

track.nMeasurements() = 8;
track.template component<unsigned int, "nMeasurements"_hash>(); // equivalent

ProxyAccessor<unsigned int> nMeasurements("nMeasurements"); // use accessor for convenience
assert(nMeasurements(track) == track.nMeasurements());
```

Dynamic columns

- Fitters have different requirements / information to attach to tracks + track states
- Both container interfaces support dynamic columns
- `Vector{TrackContainer,MultiTrajectory}` support this through type-erasue by inheritance
  - Backends are free to implement this: ATLAS `xAOD` backend will use decorations

```cpp
auto tracks = factory();

tracks.template addColumn<float>("col_a"); // create a new column of type float
assert(tracks.hasColumn("col_a"_hash));

auto track = tracks.getTrack(tracks.addTrack());
track.template component<float>("col_a") = 42.42f;
assert((t.template component<float, "col_a"_hash>() == 42.42f));

ProxyAccessor<float> answer("col_a");
assert(answer(track) == 42.42f);
answer(track) = 99.0f;
```

Migration to `Acts::TrackContainer`

- Migrated all Core fitters to operate on `Acts::TrackContainer`
- Fitter accepts mutable container as input, returns track proxy
- CKF fill track container, returns vector of track proxies

```
auto fitter = makeFitter(); // KF, GSF, GX2F
auto track = fitter.fit(sourceLinks.begin(), sourceLinks.end(), initialParameters,
         options, tracks);
auto finder = makeFinder(); // CKF
auto foundTracks = finder.findTracks(initialParameters, options, tracks);
```

- Also working to migrate all of our examples code to `TrackContainer`

```
for (auto track : tracks) {
  for (auto state : track.trackStates()) {
    if (not state.typeFlags().test(Acts::TrackStateFlag::MeasurementFlag)) {
      continue;
    }
    // do something with measurement
  }
}
```

`EDM4hep` conversion

- `EDM4hep` has `edm4hep::Track` & `edm4hep::TrackState`
  - Uses the LCIO parametrization $d_0, z_0, \phi, \tan\lambda, \Omega$ (ACTS uses $l_0, l_1, \phi, \theta, q/p, t$)
  - Track states are described using perigee parameters only (ACTS uses varying local parametrization + link to geometry object)

Direct & transparent backend in EDM4hep not feasible

- **Required contract cannot be fulfilled**:
  - No stable references to native parametrization
  - Loss of on-surface hit position
- Instead: **Full (lossy) conversion** to and from EDM4hep tracks implemented (and in turn is backend agnostic)

`PODIO` backend: `ActsPodioEdm`

Goal

- Demonstrate **ability to integrate** with an external IO solution like `PODIO`
- This is not an alternative to `EDM4hep`, but help us understand requirements

- Specify ACTS EDM in `PODIO`-yaml in *plugin*
- Implemented `ActsPodioEdm::Track` + `ActsPodioEdm::TrackStates`
  - Use *components* to produce stable references to fulfill backend contract
  - Recent update in `PODIO` made this less brittle
  - Auxiliary data types for dense columns overallocated storage for measurements
  - Experiment-aware translation helper for surfaces and uncalibrated measurements
- **Full IO roundtrip implemented and tested, Kalman Filter can run on this without modifications**

Experiment interface

- `PODIO` backend still supposed to be experiment agnostic
- Experiment-knowledge needed to persist otherwise transient information

Surfaces

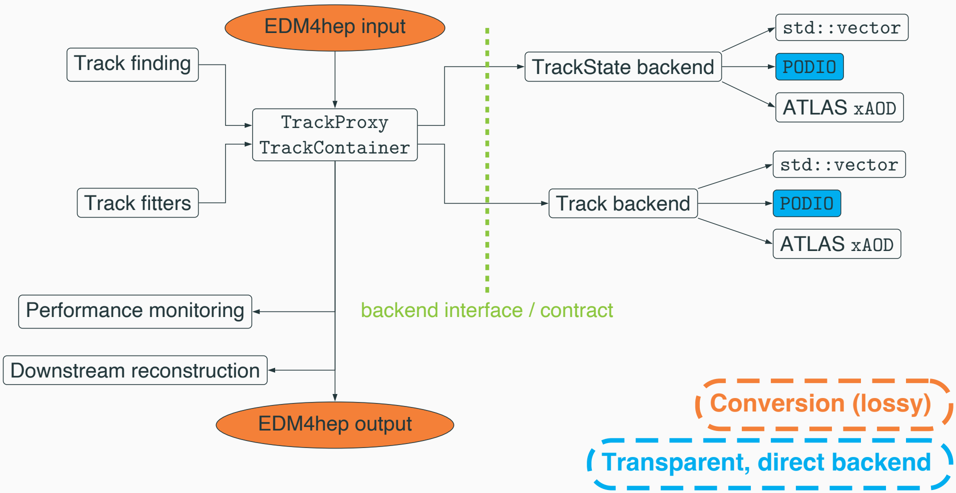
- Two types: part of detector geometry, *ad-hoc* surfaces
- Encode known surfaces as identifiers, serialize ad-hoc surfaces
- Make no assumptions on identification model

Measurements

- ACTS uses strong type-erasure for experiment-specific input measurements
- Cannot serialize type-erased measurements automatically

- Factorized to experiment-specific helper class to implement these conversions

Backend overview



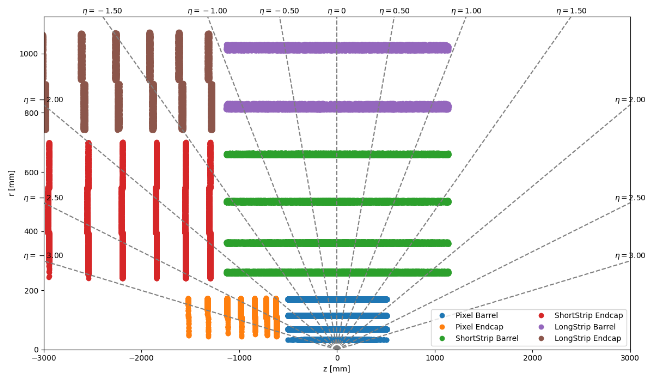
Improved EDM4hep inputs

- **Goal:** simulate outside of ACTS (e.g. `ddsim`), run tracking in ACTS
- Previous status: had disjoint measurement, particle, simhit readers
  - **Problem:** read internally inconsistent info
  - ACTS uses flat particle sequence with intricate barcodes
  - EDM4hep uses tree structure of particle objects
  - Missing connection between particles / hits

feat: Add uber EDM4hep reader (#2939) Merged

- Added single reader from EDM4hep
- Reader correctly assembles ACTS' flat particle container, vertex + particle ID from EDM4hep particle tree
- Correctly splits up into *generator*, *initial* and *final* particles
- Associates hits to ACTS surface equivalents from DD4hep
- Critical for truth matching downstream of reconstruction

`ddsim` inputs read into ACTS from ODD



- Positive endcap already missing after `ddsim`: issue with ODD+`ddsim`
- Reading + processing works correctly

Vertexing refactoring

Vertexing refactoring

- Series of PRs to reduce build-time memory footprint of vertexing (#2842 Merged)
- Vertexing code was heavily templated, leading to large memory consumption
- Refactored large fraction of the vertexing code to avoid templates

- feat: Propagator optionally inherits from BasePropagator #2874
- refactor!: Vertex InputTrack becomes concrete type #2876
- refactor!: Untemplate `Vertex` #2877
- refactor!: Untemplate `VertexInfo` and `VertexingOptions` #2878
- refactor!: Remove `input_track_t` template parameters #2880
- refactor!: Use `Delegate` for parameter extraction #2881
- refactor!: Use BasePropagator interface in vertexing #2886
- refactor!: Use `Delegate` for track linearizers #2946
- feat!: Add `IVertexFinder` interface, use in vertexing #2948
- refactor: Remove `VertexFitterConcept` #2951
- refactor: `KalmanVertex(Track)Updater` interface change #2955
- refactor!: Hard-code vertex fitter, finder + density combinations #2952
- refactor: Move large parts of Vertexing to `.cpp` files #2953
- refactor!: `ImpactPointEstimator` moves to cpp file #2971
- refactor!: Move and Grid Density finders to cpp #2973

Vertexing refactoring

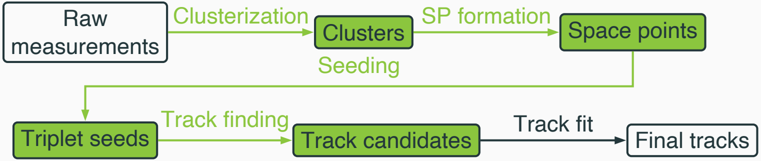
- Custom concrete type-erased type for input tracks
- Use delegate for parameter extraction (experiment interface)
- Add `IVertexFinder` interface to allow for different vertex finders (nested for seeding)
- Use `switch` dispatch in separate compilation units to hide **heavy** Kalman math
- Untemplate many classes, split into header and implementation files

Result

Build-time $\approx$ **-40%** memory consumption $\approx$ **-20%**, runtime performance **unaffected**

First public ATLAS+ACTS performance results

ACTS integration into ATLAS software



Implemented with ACTS!

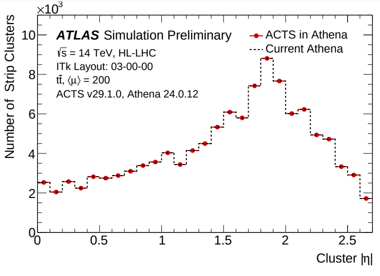
- Adding implementations using ACTS tools
- Add option allowing running a tracking chain with pieces using ACTS
- At the same time: enable ACTS output Event Data Model (EDM) to use ATLAS IO infrastructure
  - Converters for validation to allow reusing robust tooling in place
  - No conversions foreseen for final configuration
- At this time: **full tracking chain using ACTS available!**
  - Post-processing currently using non-ACTS components

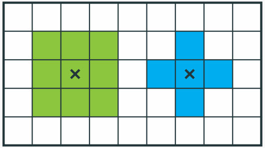
ACTS Clusterization

PLOTS-IDTR-2023-04

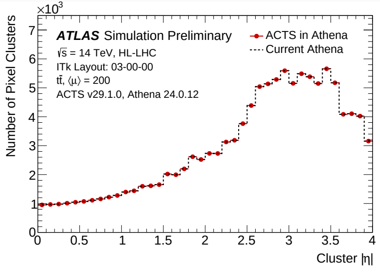
- Reimplementation of pixel and strip clustering
  - Based on prior ATLAS implementation, with some modifications
- Number of clusters and cluster sizes agree with current ATLAS SW
- Slightly favorable timing compared to current ATLAS SW

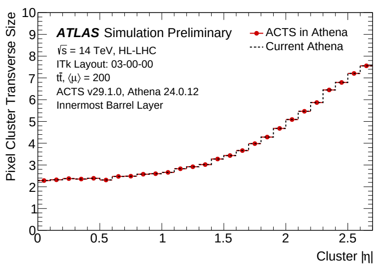
Strips



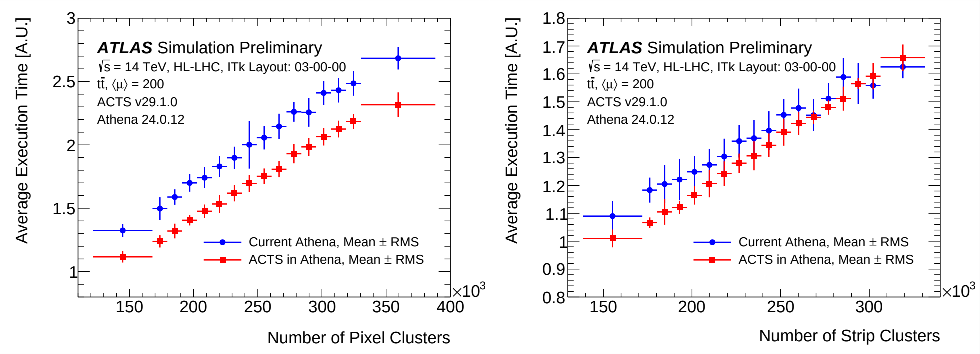


Pixel





ACTS Clusterization timing

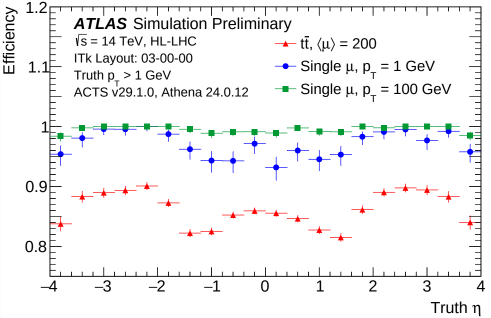


- ACTS Clusterization faster than previous Athena implementation
- Pixel: timing differences constant vs. event complexity
- Strips: ACTS implementation has larger speedup at lower complexity

ACTS tracking efficiency

PLOTS-IDTR-2023-04

- Simulation of $t\bar{t}$ events with pileup with standard Athena workflow
- Configured ACTS tracking chain: clusterization, space-point formation, seeding, combinatorial track finding using Combinatorial Track Finder (CKF)
- CKF is a complete reimplementation!
- Outputs converted to standard ATLAS tracks
  - Ambiguity resolution (without refitting) using non-ACTS tools
  - Standard performance validation toolchain

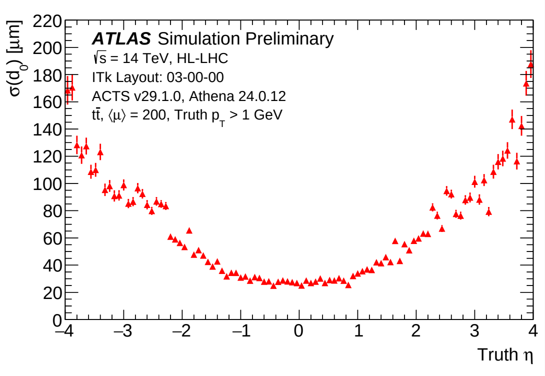


- Single μ efficiencies reasonably high (note: reconstruction requires $p_{\mathrm{T}} > 900$ MeV at central η!)
- $t\bar{t}$ efficiency within striking distance target performance

All of this is pending thorough optimization & tuning!

ACTS resolution

- Transverse impact parameter resolution directly from the ACTS CKF (no refitting)
- Produces results compatible with the standalone Kalman Filter
  - Caveats: no smoothing, no in-fit measurement calibration
  - KF independently validated against the ATLAS workhorse: the global χ^2 fitter
- Resolution here mainly due to composition of track population under study



Conclusion

- Three productive years with EP R&D
- Great exchange with other WPs, especially calorimetry on OpenDataDetector!
- ACTS library saw lots of developments during that time!
- Monitoring and testing infrastructure has greatly expanded
  - **Reproducibility tests** for all workflows
  - Increased **trust** in results obtained
  - Critical for ATLAS integration
- In parallel: support of developments for GPU tracking

What's next?

- Transitioned to LD Staff with WP2 (ATLAS) in the Next Generation Triggers project
- Will keep working on ACTS, keep in touch!