Build optimization in ACTS

Paul Gessinger
CERN
2022-07-05

Talk road map

- cmakeperf: monitor time and memory usage per compilation unit
- Closer look at mitigation in code:
  - Refactor `Acts::Measurement` to reduce template complexity
  - Move some very heavy Eigen operations into separate compilation units
  - Refactor fitters to reduce complexity of templates
  - Lessons learned from this and application to EDM rework

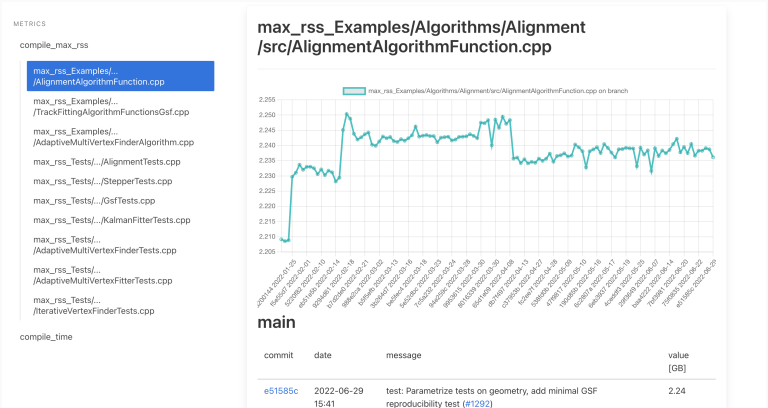
cmakeperf

- Small python utility to measure build resource usage
  - Source code: https://github.com/paulgessinger/cmakeperf
  - `pip install cmakeperf`
- Consumes `compile_commands.json` that cmake can generate
- By default it will give you a measurement for each compilation unit, can output to CSV for further processing

```
cmakeperf collect build/compile_commands.json -o perf.csv
I will write output to perf.csv
[  1/631,   0.2%] [  499.03M] [    8.53s] - Core/src/EventData/NeutralTrackParameters.cpp
[  2/631,   0.3%] [  476.18M] [    7.02s] - Core/src/EventData/PrintParameters.cpp
[  3/631,   0.5%] [  495.81M] [    8.02s] - Core/src/EventData/TrackParameters.cpp
[  4/631,   0.6%] [  482.86M] [    6.02s] - Core/src/EventData/TransformationBoundToFree.cpp
[  5/631,   0.8%] [  491.34M] [    6.52s] - Core/src/EventData/TransformationFreeToBound.cpp
[  6/631,   1.0%] [  743.49M] [   13.03s] - Core/src/EventData/CorrectedTransformationFreeToBound.cpp
# ...
```

Monitoring over time

- Have job in ACTS which runs `cmakeperf` on each commit to main (example)
- Extra infrastructure to extract top resource consumers and store info over time
- Can produce history plots over time (what Hadrien showed before already)
- Also: live-dashboard here



Code mitigations

Measurement rework I

- Measurements in ACTS: subspace of sensor-local track parameters

$$\vec{x}_{\text{sub}} = \begin{pmatrix} l_0 \\ l_1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \vec{x}_{\text{full}}.$$

- Transform between two representations via *projectors*
- Goal: do as much at compile time as possible. So: calculate projectors at compile time
- Consequence: Measurement type was templated on **local parameters**, packaged into a giant variant for type erasure which contained **all possible permutations** ($\mathcal{O}(2^N - 1)$)

```
template <typename source_link_t, ParID_t... params>
class Measurement;
```

- Pseudo-code of how this type was generated at compile-time

Measurement rework II

```
comb = [()]
for i in range(0, N):
  comb = comb + [c + (i,) for c in comb]
comb = comb[1:] # remove first element `()`
```

- Very big type, instantiations were very heavy
- First refactoring: move from Boost MPL (pre-variadic templates) to Boost hana, without changing the algorithm itself
  - Gave us some improvement but not enough
- Second refactoring: template measurement on number of local dimensions

```
template <typename indices_t, size_t kSize>
class Measurement;
```

  - Reduces variant size to N (instead of $2^N - 1$)
  - Caveat: additional runtime operations. Did not find enough runtime benefit to keep fully templated approach

Move Eigen instantiations wherever possible

- Eigen instantiations are generally **heavy**
- ACTS code was not careful enough where to put them
  - Often put in headers, sometimes without good reason
- Refactoring of `SourceLink` (itself out of scope) allowed to move a few heavy-hitters into separate compilation units: `GainMatrixUpdater`, `GainMatrixSmoother`
  - These run in the Kalman Filter so have deeply nested Eigen expression templates
  - These compilation units immediately used 1.3G / 500M of memory on their own
  - Exacerbated effect in ACTS build / CI: (C)KF used in many different unit tests
  - Reduced number of variations of templates that had to be instantiated
- Reduced memory consumption by $\mathcal{O}(1.5G)$ for some compilation units
- Have been looking to find more cases like this

Example: GainMatrixSmoother

This clocks in at about 500M of memory usage:

```
// Gain smoothing matrix
BoundMatrix G = ts.filteredCovariance() * prev_ts.jacobian().transpose() *
                prev_ts.predictedCovariance().inverse();
//          ^ 6x6 matrix

// Calculate the smoothed parameters
ts.smoothed() =
    ts.filtered() + G * (prev_ts.smoothed() - prev_ts.predicted());

// And the smoothed covariance
ts.smoothedCovariance() =
    ts.filteredCovariance() -
    G * (prev_ts.predictedCovariance() - prev_ts.smoothedCovariance()) *
        G.transpose();

ts.smoothedCovariance() = smoothedCov;
```

Example: GainMatrixUpdater

Removing this *saves* about 600M during compilation:

```
const auto H = projector.template topLeftCorner<kMeasurementSize, eBoundSize>().eval();
ParametersVector res = calibrated - H * predicted;
return (res.transpose() * ((calibratedCovariance +
  H * predictedCovariance * H.transpose())) .inverse() * res).eval()(0, 0);
```

Reduce fitter template complexity

- Fitters have extension mechanism: functionality plugged in by user to customize
- Want this to be low-overhead: so template arguments

```
template <typename propagator_t, typename updater_t = VoidKalmanUpdater,
          typename smoother_t = VoidKalmanSmoother>
class KalmanFitter;
```

- Existence of template parameter by itself no problem. But more variations of instantiated types: not scalable
- Solution: refactor this to runtime functions
  - Added `Delegate` type (sort of like `std::function`, but without ownership)
  - Can compile down to a simple function pointer call or even inline target function
- Reduced template parameters to

```
template <typename propagator_t> class KalmanFitter;
```

Lessons learned

- Some compile time programming might not be worth the extra resources
- Compile time resource consumption needs to be a constant consideration (especially if you're using Eigen)
- Example: when potentially relevant changes are made, benchmark compile time performance along side runtime performance
  - See #1262 where I did this

	file	max_rss_main	max_rss_feat	relative
0	CombinatorialKalmanFilterError.cpp	1.28614	1.24109	96.4968
1	MeasurementSelector.cpp	864.469	879.059	101.688
2	KalmanFitterError.cpp	0.569344	0.008192	1.43885
3	GainMatrixUpdater.cpp	1280.87	1300.28	101.515
4	GainMatrixSmoother.cpp	513.376	487.125	94.8865
5	GsfError.cpp	0.008192	0.585728	7150
6	GsfUtils.cpp	628.855	609.251	96.8827
7	TrackFindingAlgorithmFunction.cpp	794.558	765.542	96.3482
8	TrackFittingAlgorithmFunction.cpp	818.065	798.306	97.5847
9	AlignmentAlgorithmFunction.cpp	1369.24	1329.99	97.133
10	CombinatorialKalmanFilterTests.cpp	1087.35	1230.05	113.123
11	GsfTests.cpp	1642.13	1686.95	102.729
12	KalmanFitterTests.cpp	1232.11	1282.43	104.085
13	GainMatrixUpdaterTests.cpp	383.144	437.797	114.264
14	GainMatrixSmootherTests.cpp	329.216	413.655	125.649
15	GsfComponentMergingTests.cpp	455.721	440.185	96.5909

Backup

Delegate type

- Extension mechanism uses `Delegate` type (PR #1059). Why?
- `Delegate<int(int,float)>` is replacement to `std::function<int(int,float)>` but with only one level of indirection
- Supports free function pointers, and pointers to member functions + instance pointer (**Does not assume ownership!**)
- Potentially interesting for other use cases (e.g. seeding)

```
int sumImplementation(int a, int b) {
  return a + b;
}
Delegate<int(int, int)> sum;
sum.connect<&sumImplementation>();
int c = sum(2, 2); // = 4
```

```
struct Subtractor {
  int x;
  int subtract(int v) const { return v - x; }
};
Delegate<int(int)> s;
Subtractor inst{5};
// subtract assumes no ownership
s.connect<&Subtractor::subtract>(&inst);
int y = subtract(10); // = 5
```