Fighting Acts build bloat

Hadrien Grasland 2020-07-20

We have a build problem

- Like all « modern C++ » projects, Acts builds slowly
  - On 2020-05-20, a full build* took **1h30 of seq. CPU time**
  - **Some tests take *minutes*** to build → bad for dev. iterations
- More importantly, however, the build uses a lot of RAM
  - On 2020-05-20, the record was CKF tests @ **7,4 GB RSS**
  - Ergo, **can’t use all cores** on a typical dev machine
- Some work was done in the past, but more is needed

* RelWithDebInfo build, using GCC 9.3.1 on Linux, i7-4720HQ CPU, everything but CUDA enabled

Setting a goal

- **« Acts should build with all cores on Moritz' laptop »**
  - 4 threads, 8 GB of RAM, assume 1-2GB used by system
  - Actually a fairly typical mid-end development machine
  - By that metric, Acts should stay below **1.5 GB/process**

Identifying the culprits

- Offenders are easy to spot with a system monitor*
  - ...but good to cross-check with GNU time for extra precision
  - Error on the peak with ~2s polling can be 200-300 MB
- On 2020-05-20, those processes were >4GB :
  - **CKF tests (7.4 G)**
  - **TrkFdAlgTrkFdFunc (6.9 G)**
  - **KF tests (6.8 G)**
  - **FitAlgFitFunc (6.4 G)**
  - **AMVFinder tests (4.1 G)**
  - **GainMatrixUpd tests (4.1 G)**

* More recently, Paul made https://github.com/paulgessinger/cmakeperf and added it to CI

Telling what’s going on

- Compiler profiling is sadly a bit of a pain
  - Most give you a **per-pass breakdown**, which is useless
  - **External profilers** like perf won’t help you either
    - Require debug symbols, compiler impl knowledge
    - No tracing information about method parameters
  - **Templight** requires a custom clang build + is hard to use
  - Thankfully, clang 9+ has **`-ftime-trace`**…

-ftime-trace

- Clang 9+ feature contributed by a Unity3D developer*
- Gives **fine-grained, hierarchical compiler time profiles**
  - Source pass (#include other preprocessor) :
    - Which **top level headers** take a lot of time to process
    - Why they do so (transistive inclusion, eager templates…)
  - PerformPendingTemplateInstantiations pass :
    - Which **templates** take a lot of time to instantiate
    - Which other templates they transitively instantiate

* https://aras-p.info/blog/2019/01/16/time-trace-timeline-flame-chart-profiler-for-Clang/

Wait… *time* profiles ?

- Unfortunately, nothing like `-ftime-trace` for memory usage
  - So, must make do with what we have…
- **Assumption :** Using a lot of RAM <=> Taking a lot of time
  - => : Reasonable expectation, data takes time to process
  - <= : Less obvious (think alloc/free cycle), turned out to hold
- **Assumption** : GCC and clang have similar perf characteristics
  - Again, not obvious but turned out to hold well enough

Using -ftime-trace

- Get the command line used to build the .cpp file
  - Simple way* : touch cpp file and re-run « make »
- Adjust it
  - « g++ » → « clang++ »
  - « -std=gnu++17 » → « -std=c++17 »
  - Add -ftime-trace flag
- Run it → A JSON file is produced next to the .o file
- Open Chrom(e|ium)**, go to « chrome://tracing », feed it the file

* Clever way : Have CMake generate a « compilation database » and parse it

** Could use SpeedScope before, but unfortunately they improved input sanitization…

Demo : CKF test build analysis

(End of may) Conclusions

- Two major contributors to CKF tests build time :
  - Huge std::variant from **Acts’ Measurement** mechanism
  - Lots and lots and lots of **Eigen templates**
- Decided to focus on reducing Eigen bloat because…
  - It was the biggest contributor
  - I have an old axe to grind with that lib anyway

Eigen characteristics

- The good : First-class support for **small matrices**
  - No heap allocation when size is statically known
  - Methods can be inlined (though codegen isn't great*)
- The bad : Some features have a large **complexity cost**
  - Expression templates
  - CRTP-style inheritance
  - Block<MatrixType>
  - Dynamic-sized matrices
  - Row-major support

* An intern of ours once wrote a small prototype library which is multiple times faster than Eigen at low-dimensional matrix multiplication and inversion to back up this claim

A bothersome feature

- **Expression templates** are a special kind of evil
  - « a*b + c » isn't just « a*b » and « a+b »
    - Type is like Sum<Product<M1, M2>, M3>
    - Construct Matrix from this → Expression is evaluated
  - Consequences :
    - **Combinatorial explosion** of types/constructors
    - **Lifetime issues** (who got bitten by « auto » in Eigen?)
    - **Bad compiler optimization** (CSE takes a hit)
    - **Incomprehensible execution profiles**
    - All to avoid temporaries… that compiler optimize out !

Blocking the bother

- I tried to **inhibit expression templates** by…
  - Building wrappers for Eigen types
  - Replicating most of the Eigen API on the wrappers…
  - …but returning matrices from operators, not expressions
- Took me about a month of work
  - Net result : **-0.3 GB to -1.0 GB** per compilation unit :-(
  - Not awful, but not worth adding 6 kLoC to Acts yet…

Meanwhile, on master…

- At end of June, I rebased the finished wrapper on master…
- …whose build profile had changed a lot wrt late May !
  - CKF tests : 5.9 G (-1.5)
  - **EvDatView test : 5.7 G (NEW)**
  - KF test : 5.7 G (-1.1)
  - TrkFdAlgTrkFdFunc : 5.6 G (-1.3)
  - FitAlgFitFunc : 4.8 G (-1.6)
  - GainMatUp test : 3.4 G (-0.7)
  - AMVFinder test : 3.3 G (-0.8)
- Exact origin unknown, bisecting would be too expensive…
  - But good surprise was welcome, and motivating !

Finding more fat

- Without expression templates, the **build profile is clearer**
  - Complex ops (e.g. matrix inversion, geometry, Cholesky…) obviously not helped by wrapping
  - But still surprisingly high contribution of add, mul, etc.
  - Cause turned out to be **large-scale use of Block and Map**
    - …which are actually Block<Matrix> and Map<Matrix>
    - …which, combined with CRTP, re-instantiates all the code
    - So I tried to switch to an extractBlock/setBlock design

…and even more

- Per se, **changing block API was not enough**
  - Still needed many Matrix constructor instances (1/block)
  - So I accepted the necessity of rewriting the impl too…
  - …and similarly rewrote the impl of every other simple matrix operation with a big impact on KF test build profile
- Having to go there was unfortunate, but effective :
  - CKF tests : 4.3 G (-1.6)
  - FitAlgFitFunc : 3.9 G (-0.9)
  - TFAlgTFFunc : 3.7 G (-1.9)
  - EvDatView test : 3.7 G (-2.0)
  - KF test : 3.4 G (-2.3)
  - Everyone else <3 GB

Current status

- Can likely gain even more by **replacing more** Eigen impls
  - Geometry, matrix inversion, and Cholesky are quite bad
  - …but more work to rewrite than addition/multiplication
- Can that alone take us down to <1.5 G ? Not sure…
  - I suspect **Measurement variant** will need some love too
- Also, will need **better impls** to beat Eigen at runtime
  - Tried auto-vectorizable loops… but that didn’t work out
  - I don’t expect SIMD impls to cost more… but must prove it

Summary

- We still have a build problem (but it got better in June)
- Eigen is a very significant part of it
  - Though Measurement variant should be investigated too
- We can go far with a piecewise rewrite of Eigen…
  - …but I still need to prove that at equivalent runtime perf
  - Also, the new impls are really specialized for Acts' needs
    - Can't contribute them to Eigen, room for a simpler BLAS

Thanks for your attention !