Acts build issues (feat. Eigen)

Hadrien Grasland 2020-07-20

A build problem

- Like all « modern C++ » projects, Acts builds slowly
  - When I started, 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
  - When I started, the record was CKF tests @ **7,4 GB RSS**
  - So, **can't use all cores** on a normal dev machine
- Some work done on this 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

- Typical mid-range laptop : 4 threads, 8 GB of RAM
  - Hard to get system below 1-2GB : 6 GB left for the build
  - To use all threads, Acts build must stay <**1.5 GB/process**
- How far are we ? On Acts master@b244d0d5 (Aug 27) :
  - 5 processes have a peak RSS in the 5,4-5,8 GB range
  - 16 processes are in the 2,1-3,4 GB range
  - 15 processes are in the 1,6-2,0 GB range

Telling what’s going on

- Compiler profiling is sadly a bit of a pain
  - Usually you get a **per-pass breakdown**, which is useless
  - **External profilers** like perf won’t help you either
    - Require debug symbols, compiler impl. knowledge
    - Lacks 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, we must live with two assumptions…
- **Assumption 1 :** Using a lot of RAM <=> Taking a lot of time
  - => : Reasonable expectation, data takes time to process
  - <= : Not obvious (think alloc/free cycle)
- **Assumption 2** : GCC & clang have similar perf. characteristics
  - Not obvious (clang uses ~2x less RAM)

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++ »
  - 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 it doesn't work anymore…

Demo : Initial CKF tests build profile

General observations

- Direct problem : Code bloat from lots of small functions
  - « Death by 1000 cuts »… but some cuts deeper than others
  - ~50 % of LLVM IR codegen time spent on Eigen expresions
- Let’s look at template instantiations (~41s)
  - 21,6s (52 %) from Eigen types and methods
  - 10,2s (25 %) from a ridiculous std::variant over 63 types
- Decided to work on the Eigen issue first

Eigen characteristics

- The good : Decent support for **small matrices**
  - No heap allocation when size is statically known
  - Methods can be inlined (though codegen isn't great*)
- The bad : Other things that we pay for, but could live without
  - **Expression templates**
  - CRTP-style inheritance
  - Block<MatrixType>
  - Dynamic-sized matrices
  - Row-major support
  - Terrible code (e.g. no includes)

* 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
  - Basically, Eigen’s « a*b + c » isn’t just « x*y » and « x+y »
    - 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?)
    - **Less compiler optimizations** (CSE takes a hit)
    - **Incomprehensible build & execution profiles**
    - All to avoid temporaries… that compilers optimize out !

A workaround

- 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 : **0.3-1,0 GB** gain in large compilation unit
  - Not awful, but not worth maintaining 6 kLoC yet…

Searching for more…

- At least, w/o expression templates, **the build profile is clean**
  - Complex ops (e.g. matrix inversion, geometry, Cholesky…) obviously not helped by the wrapping strategy
  - Still a 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, thanks to CRTP, re-instantiates all the code
    - So I tried out an extractBlock/setBlock wrapper API

…and even more

- extractBlock/setBlock API over Eigen's impl isn't enough
  - Still needed many Matrix constructor instances (1/block)
  - So I accepted the necessity of **rewriting the impl** too…
  - …and transitively rewrote the impl of every simple matrix operation with a big impact on KF test build profile
- Having to go there was unfortunate, but effective :
  - For >5GB compilation units, benefits in the **1,0-2,1 GB** range
  - But now, I am responsible for runtime optimizations…

Current status

- These changes improve the situation, but not enough yet
  - 5 process in the 3,6-4,4 GB range (only one >4,0 GB)
  - 5 processes in the 2,1-2,3 GB range
  - 13 processes in the 1,6-2,0 GB range
- Could take it further, but other work must be done first
  - Runtime performance must be brought back ≥ Eigen
  - Maintaining this as an Acts dev branch is becoming painful
  - We still have that huge std::variant to take care of…

HSF questions

- Do we want to take the linear algebra effort outside Acts ?
  - Not super-keen on maintaining a small BLAS on our own
  - Easy to extract in a separate library if there is demand
  - Does this sound of interest to HSF ?
- Are we convinced that it is using the right approach ?
  - Wrapping is good for Eigen code reuse, bad for complexity
  - Exposing wrappers in the Acts API would feels wrong, but wrapping inside of function impls destroys ergonomics…

Thanks for your attention !