IJCLab Irène Joliot-Curie Laboratoire de Physique des 2 Infinis

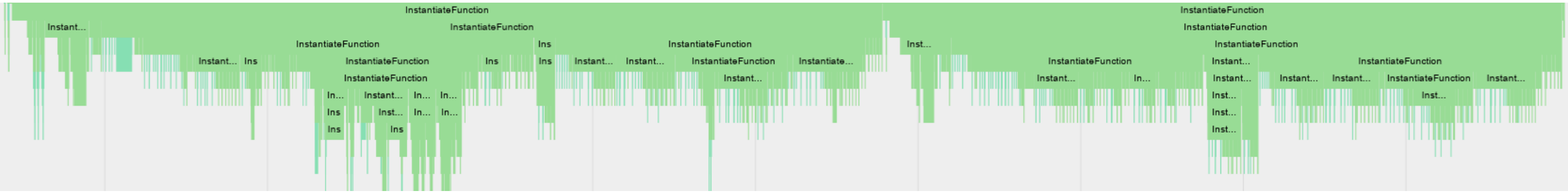
cnrs

université PARIS-SACLAY

Université de Paris

Ceci n'est pas un logo.

Mais si on est partis pour se faire les 3/4 du slide en logos, autant aller jusqu'au bout de l'idée !



Profilage de compilation en C/++

Hadrien Grasland

2020-11-19

Problèmes de compilation

- Parfois, les compilations C++ ont des soucis. Prenons Acts :
  - Certains tests : **2min*** → Itérations lentes…
  - Compilation complète : **1h30 CPU**, mais parallélisable ?
- Sauf que pour paralléliser, il faut suffisamment de RAM
  - Quand j'ai commencé, un des tests montait à **7,4 Go** RSS
  - **Limite la parallélisation** sur une machine standard
- Clairement, il est temps de faire un peu de ménage !

* Non parallélisable, car les compilateurs C/++ actuels ne parallélisent qu'entre fichiers source.

Savoir ce qui se passe

- Pas si simple de profiler une compilation pathologique
  - La **durée des passes** (frontend, backend…) ne sert à ~rien
  - Les **profileurs externes** (ex : perf) n'aident pas non plus
    - Il faut bien connaître l'implémentation du compilateur
    - Cf passes : on sait ce qui est gourmand, pas pourquoi
  - **Templight** nécessite un clang patché + pénible à utiliser
  - Heureusement, dans clang >=9, il y a **`-ftime-trace`**…

-ftime-trace

- Fonctionnalité clang 9+ contribuée par un dev Unity3D*
- Enfin des profils temporels **hiérarchiques** et à **grain fin** !
  - Passe « `Source` » : `#include`, préprocesseur en général
    - Quels **en-têtes** prennent du temps à être traités ?
    - Pourquoi ? (inclusions transitives, *templates* stricts…)
  - Passe « `PerformPendingTemplateInstantiations` » :
    - Quels ***templates*** prennent du temps à s'instancier ?
    - Quels autres *templates* ils instancient ce faisant ?

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

Des profils *temporels* ?

- Il n'y a pas d'équivalent à `-ftime-trace` pour l'utilisation RAM
  - Peut-on optimiser le profil RAM juste avec ce profil CPU ?
- **Supposition 1 :** Utiliser de la RAM <=> Prendre du temps
  - => : Raisonnable, ça prend du temps de traiter ces données
  - <= : Pas certain, mais s'est assez bien vérifié jusqu'ici
- **Supposition 2** : Optimiser pour clang => Optimiser pour GCC
  - Pas certain*, encore une fois assez bien vérifié jusqu'ici

* Face aux test pathologiques Acts, clang consomme quand même 2x moins de RAM que GCC…

Mode d'emploi de -ftime-trace

- Obtenir la ligne de commande du compilateur
  - Méthode simple* : toucher le fichier cpp, relancer `make`
- L'adapter pour utiliser clang en mode profilage
  - `g++` → `clang++`
  - Ajouter l'option `-ftime-trace`
- Exécuter → On obtient un fichier JSON à côté du fichier objet
- Prendre Chrome**, ouvrir ce JSON avec « chrome://tracing »

* Méthode sophistiquée : Demander une « compilation database » à CMake et la déchiffrer.

** Avant, on pouvait utiliser l'excellent SpeedScope… mais pour l'instant c'est cassé.

Démo : Profil d’un test pathologique

Epilogue

- Avec cet outil, on comprend mieux les problèmes d'Acts
  - Parfois, abus du polymorphisme statique (*templates*)
  - Souvent, **explosion combinatoire** de méta-programmes
  - Eigen se révèle être particulièrement pathologique*
- Également testé lors de développements ROOT 7
  - A révélé des **erreurs de conception d'interface***
  - C'est mieux de s'en rendre compte avant stabilisation !

* Précisions disponibles sur demande.

Merci de votre attention !

Soucis d'API histos ROOT 7 (RHist)

- RHist permet l'utilisation de différents types d'axes
  - Fill() doit être rapide → On veut avoir `RHistImpl<Axis…>`
  - Mais ergonomiquement gênant pour l'utilisateur ?
  - Donc `RHist<DIM>` implémenté avec `RHistImpl<Axis…>`
- Problème : Cette API implique des explosions combinatoires !
  - RHist() doit instantier tous les RHistImpl() possibles
  - Add() doit gérer toutes les paires de RHistImpl possibles
- Finalement, mieux vaut `RHistBase<DIM>` → `RHist<Axis…>`

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…