



Soutenance de stage de 2^e année

Speeding up LHCb software through compilation optimization

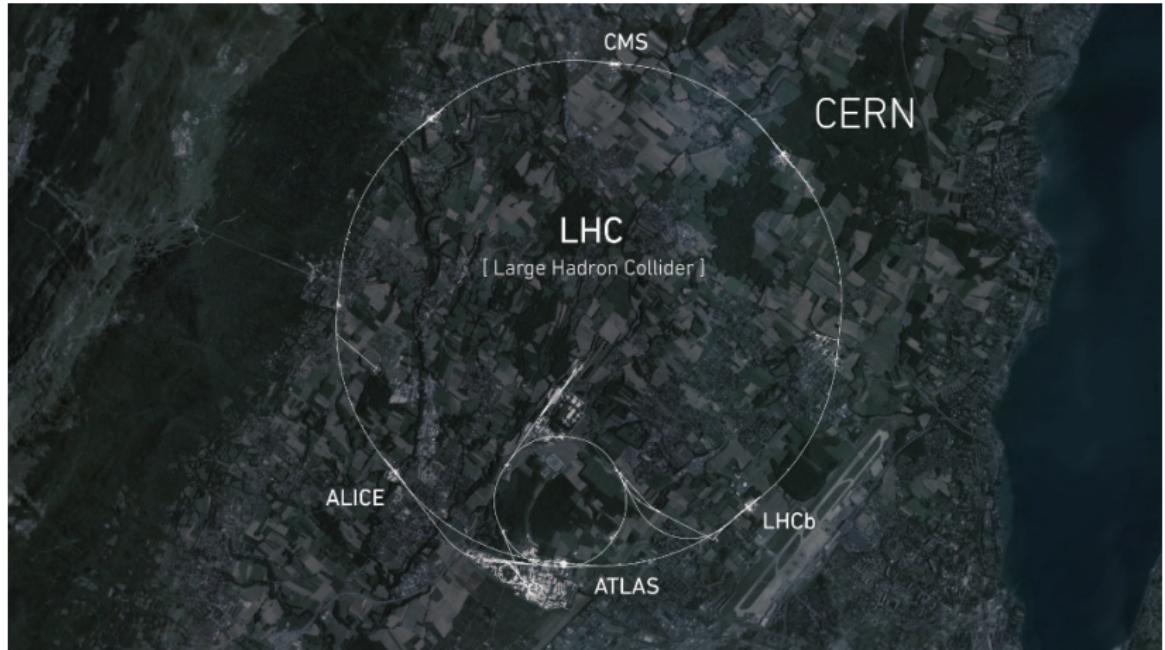
Oscar Buon

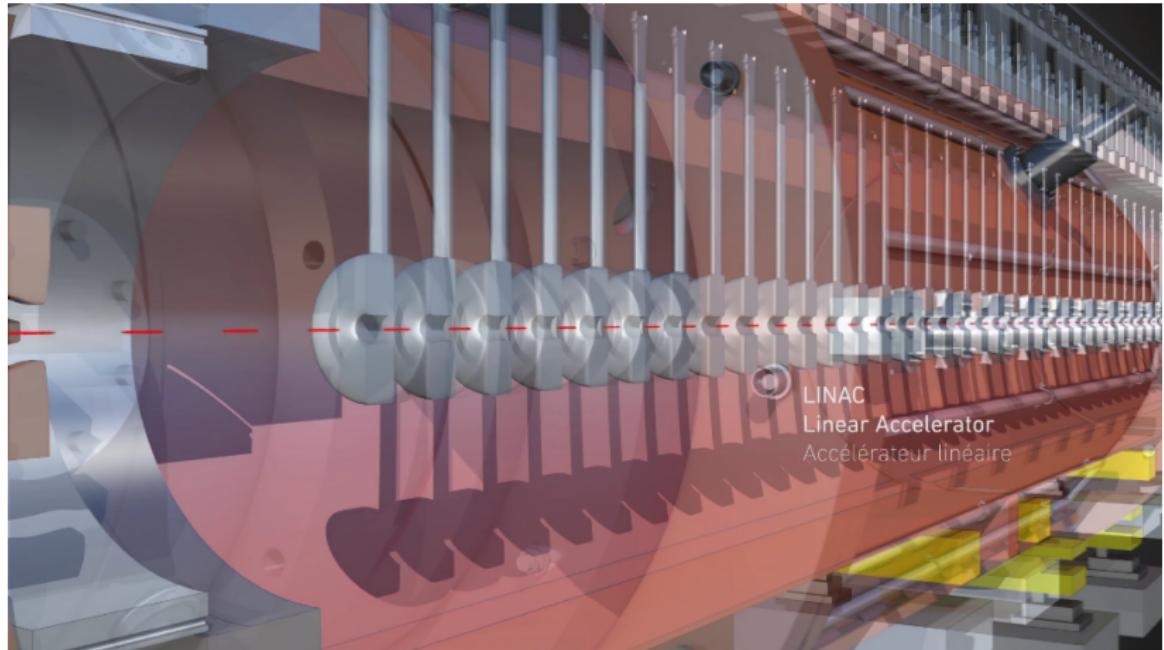
4 septembre 2023

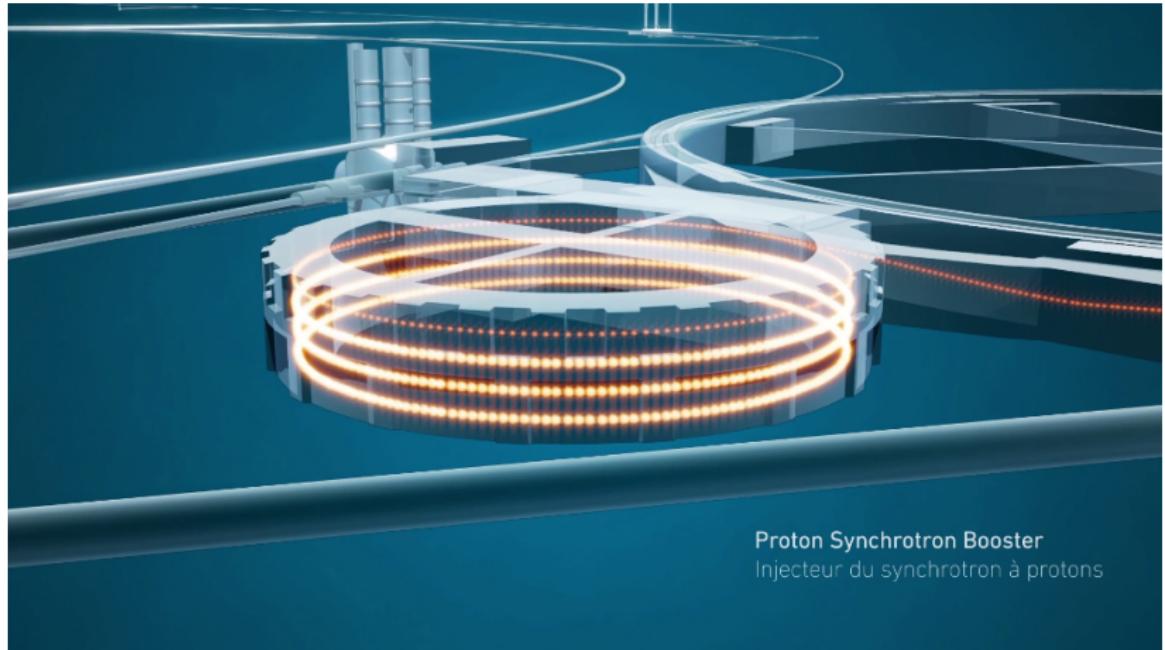
Jury : Sébastien Ponce, Mamadou Kanté, Claude Mazel

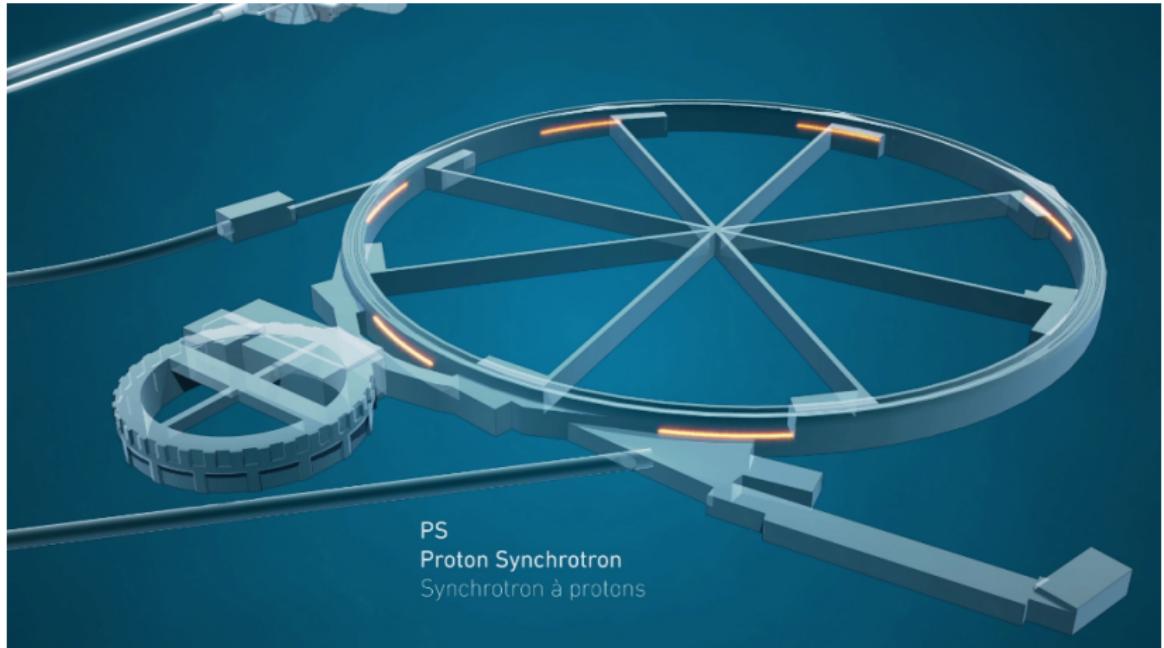




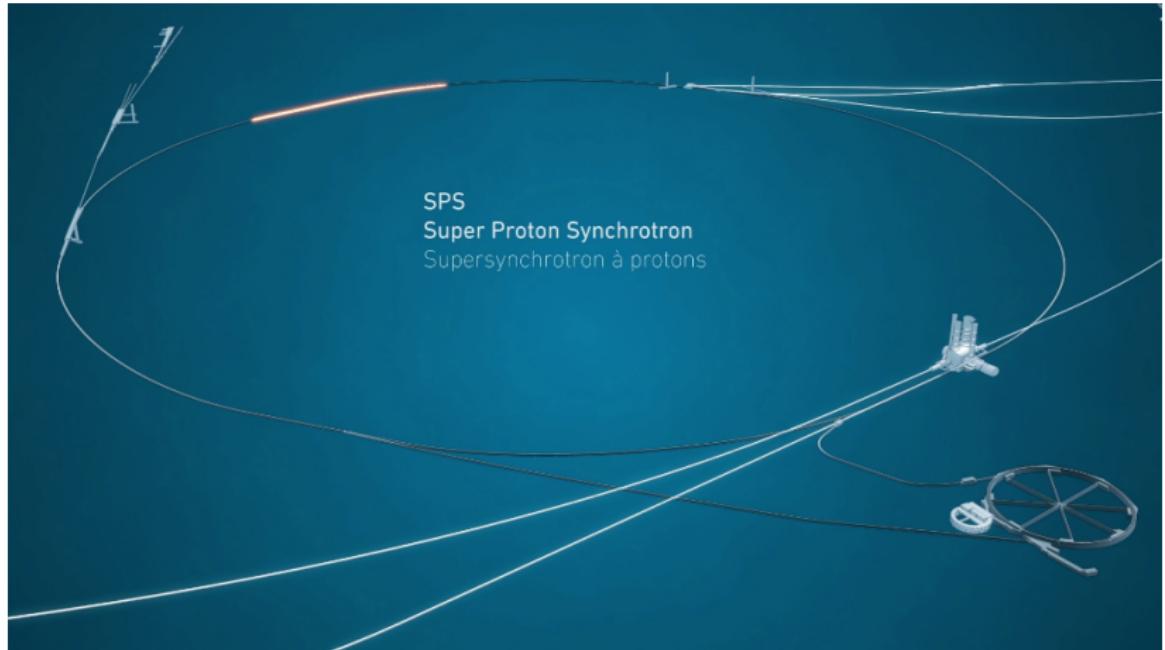


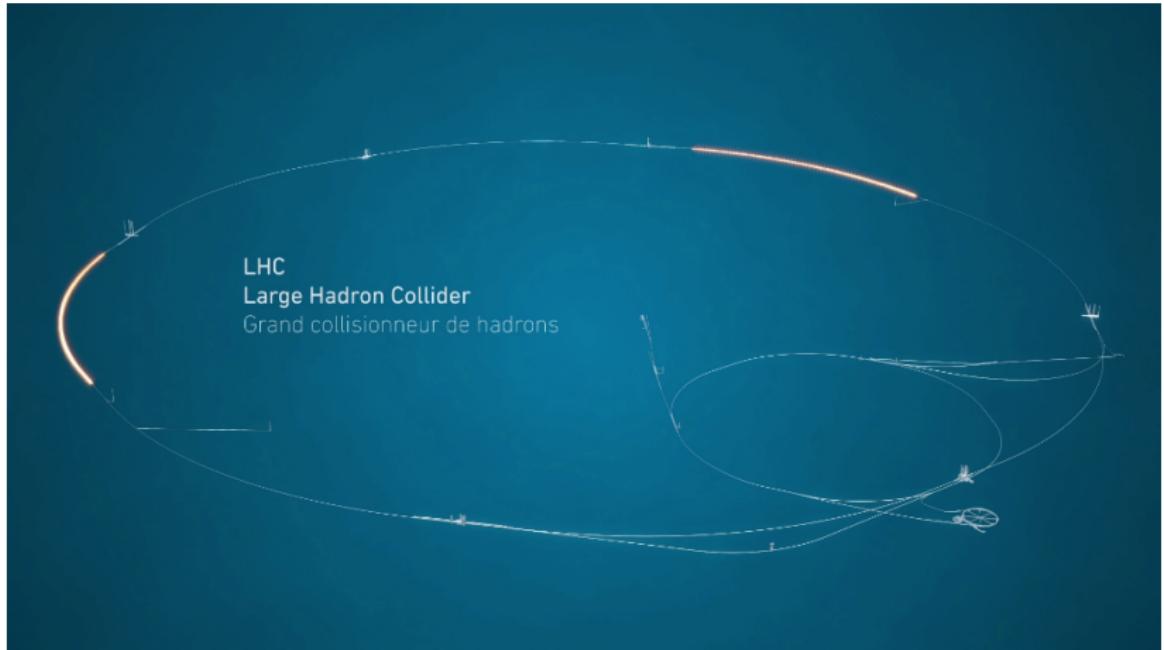


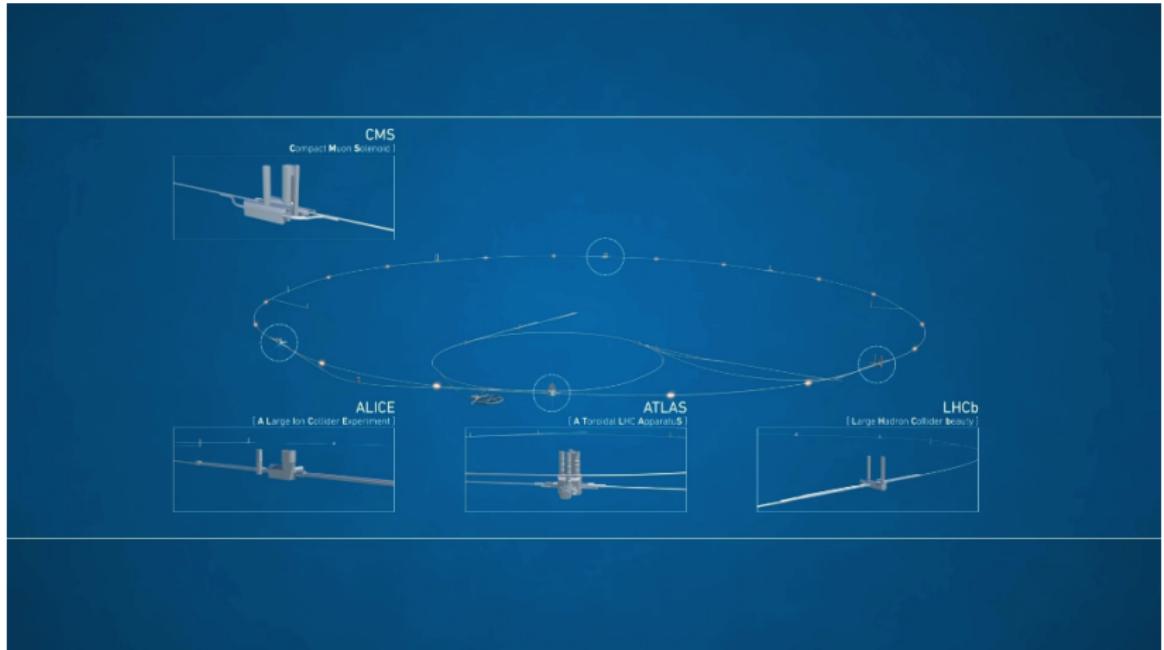




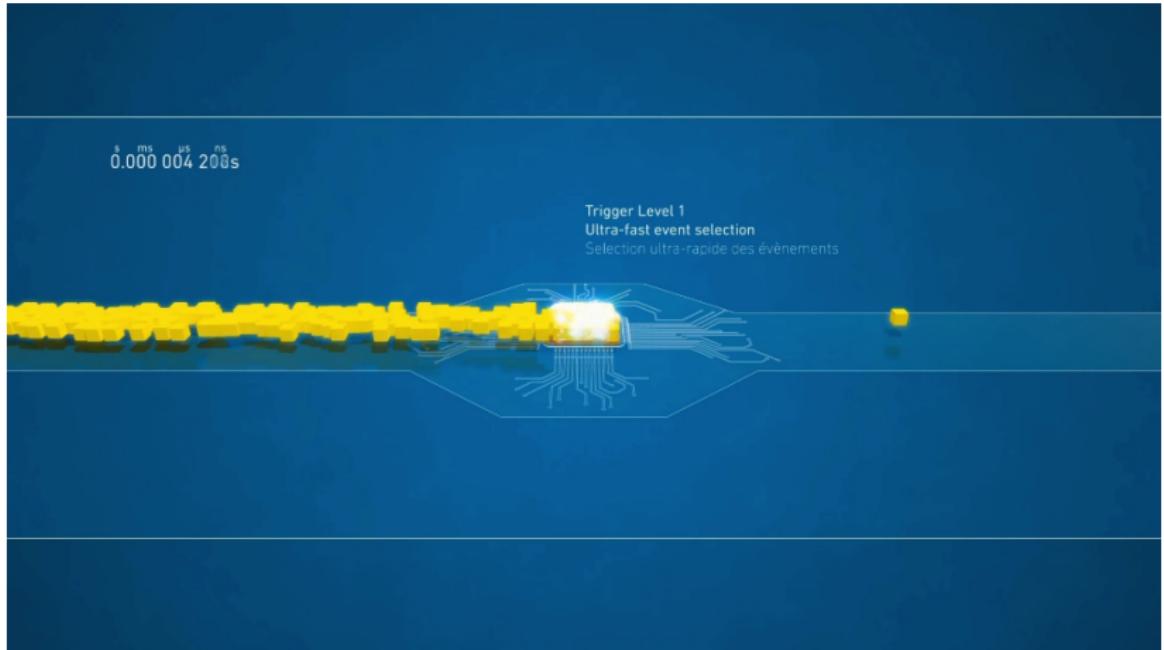
PS
Proton Synchrotron
Synchrotron à protons



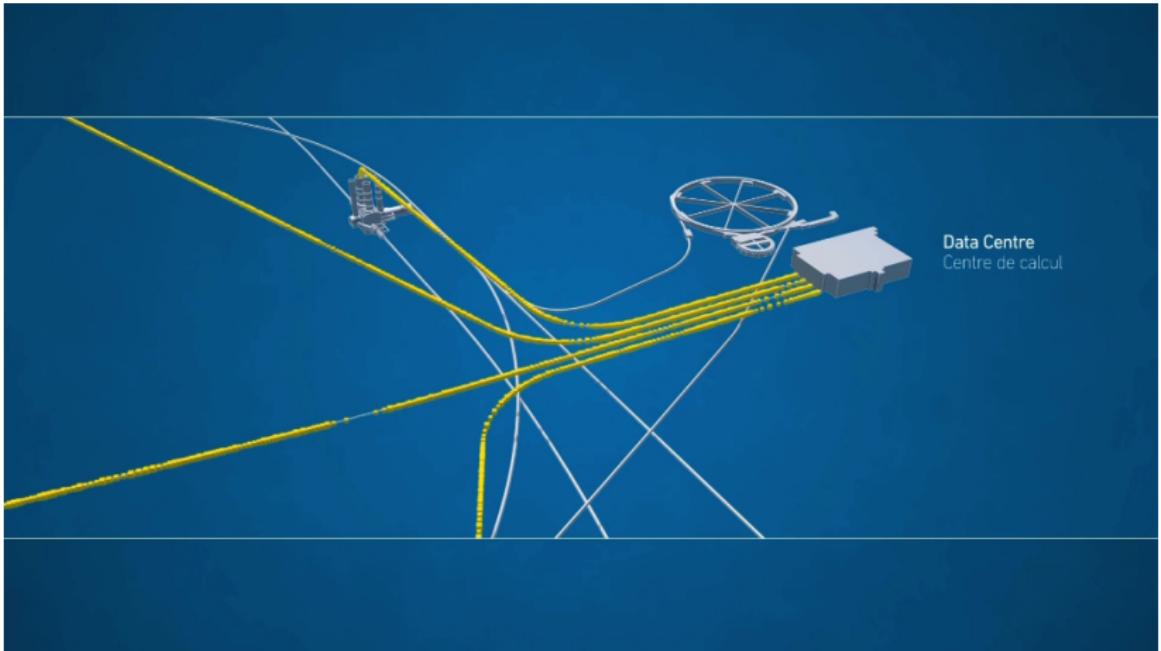


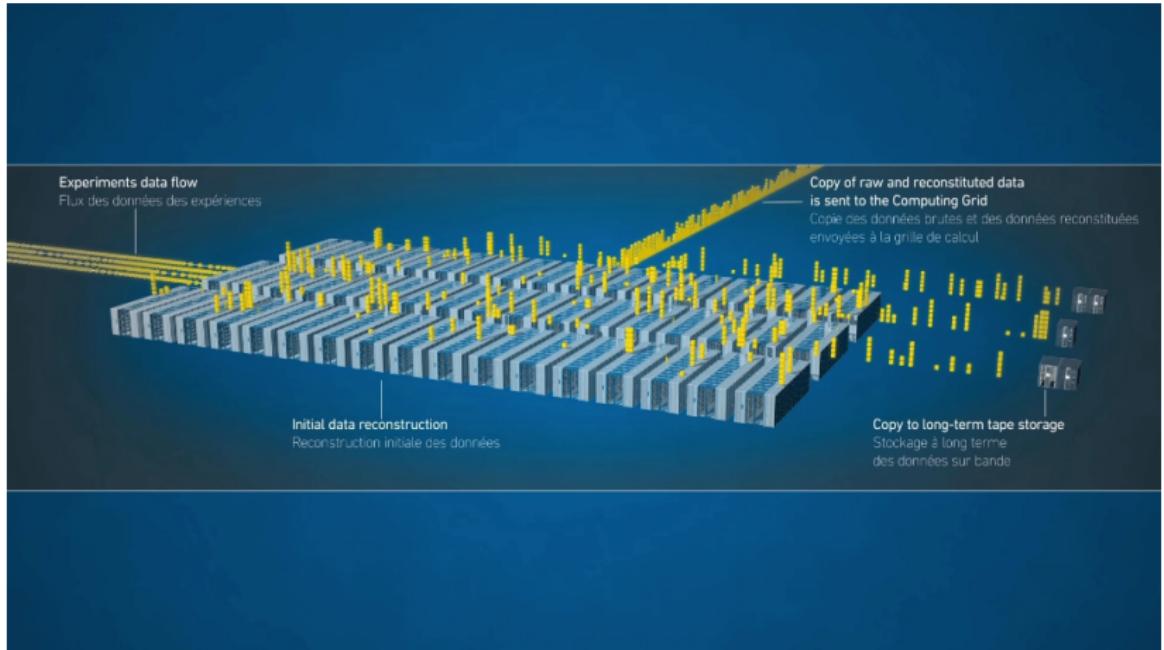






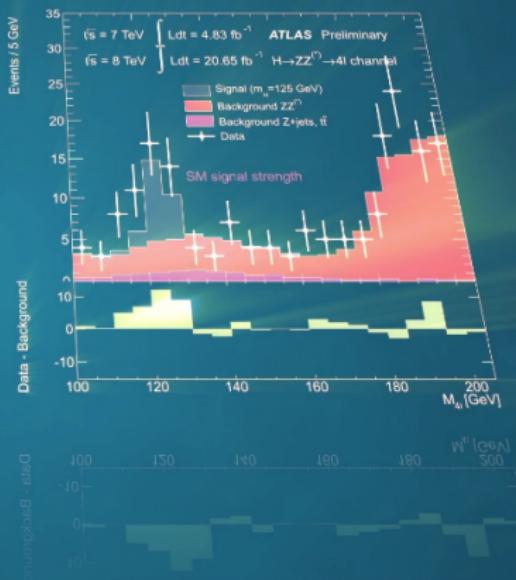
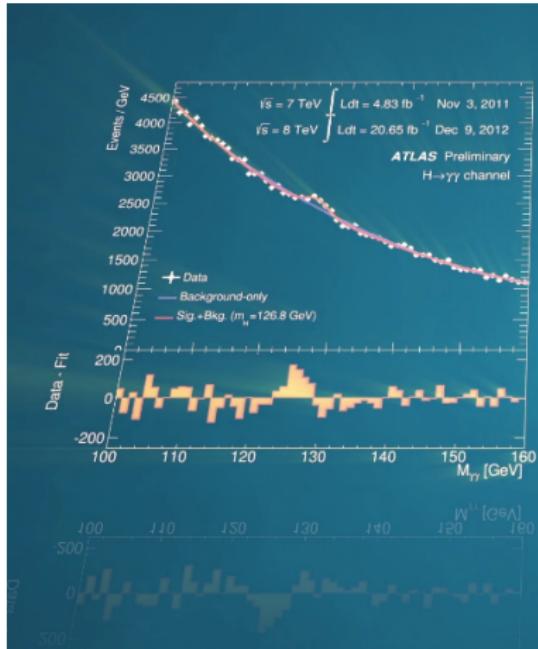








Introduction



LHCb



Computing group

- Gère l'infrastructure logicielle de LHCb ;
- Maintient les programmes de traitement des données issues du détecteur.

Pile LHCb

- Root
- Gaudi
- LHCb
- Lbcom
- Rec
- Allen
- Moore
- ...

Objectif du stage

- Le traitement a un coût important ;
- Besoin d'accélérer / optimiser les calculs ;
- Plusieurs méthodes envisagées.

1 Bibliothèques et modules statiques

- Types de bibliothèques
- Bibliothèques statiques
- Modules statiques

2 Profile-guided optimization

- Profile-guided optimization
- Link-time optimization
- Étapes de compilation finales

3 Fast-math

- Arithmétique des nombres flottants
- Exactitude de certains calculs
- Fast-math

1 Bibliothèques et modules statiques

- Types de bibliothèques
- Bibliothèques statiques
- Modules statiques

2 Profile-guided optimization

- Profile-guided optimization
- Link-time optimization
- Étapes de compilation finales

3 Fast-math

- Arithmétique des nombres flottants
- Exactitude de certains calculs
- Fast-math

Explications

- Programmes découpés en morceaux ;
- Facilité de développement & raisons historiques ;
- Plusieurs centaines.

Types de bibliothèques

- STATIC : archive de fichiers objets qui sont fusionés à l'exécutable au moment du link.
- SHARED : fichier .so (Linux) qui est automatiquement linké à l'exécutable au démarrage du programme. Permet à plusieurs exécutables d'être linké à une même bibliothèque installée sur le système.
- MODULE : pareil qu'une bibliothèque partagée à la différence qu'un module est chargé sur demande via la fonction dlopen.

Bibliothèques statiques dans Gaudi

- Gaudi utilise CMake pour compiler.
- Nouvelle fonction CMake dans Gaudi pour créer les bibliothèques statiques :
 - Adaptée de celle pour les bibliothèques partagées.
 - `gaudi_add_library` ⇒ `gaudi_add_static_library`.
 - Remplacer SHARED par STATIC dans l'appel de la fonction CMake `add_library`.

Static modules

Il faut en plus linker ave :

- **-Wl,--whole-archive**
 - sans quoi le linker enlève ce qui semble être du code inutile.
- **-Wl,--allow-multiple-definition**
 - car -Wl,--whole-archive fait que des symboles sont inclus plusieurs fois.
- **-Wl,--export-dynamic**
 - permet aux foncteurs d'accéder aux symboles.

Ces options ne sont pas recommandées. Il peut encore y avoir des bugs, notamment avec les foncteurs.

Autres limitations de ce prototype

- L'exécutable final passe de $\sim 20kB$ à $2.5GB$ (opt+g).
- La méthode standard pour démarrer un test avec Python ne marche pas.
 - Il faut démarrer l'exécutable en lui donnant un fichier de configuration en paramètre.

```
build/run ./build/Gaudi/Gaudi/Gaudi_static options.json
```

Gain de performance

Optimisation	Amélioration	Intervalle de confiance (2σ)
Static	0.87%	±0.60%

1 Bibliothèques et modules statiques

- Types de bibliothèques
- Bibliothèques statiques
- Modules statiques

2 Profile-guided optimization

- Profile-guided optimization
- Link-time optimization
- Étapes de compilation finales

3 Fast-math

- Arithmétique des nombres flottants
- Exactitude de certains calculs
- Fast-math

Profile-guided optimization

Le compilateur utilise des heuristiques pour optimiser certains éléments :

- Inlining
- Ordonancement des blocs
- Allocation des registres
- Spéculation sur les appels virtuels
- Séparation du code mort

Ces optimisations peuvent multiplier la rapidité d'exécution. Mais certains heuristiques peuvent être faux.

Une meilleure manière pour le compilateur serait de connaître le comportement à l'exécution. Principe du PGO :

- Compiler le programme avec instrumentation.
- Le faire tourner pour créer des profiles (compteurs).
- Recomplier le programme avec les profiles.

Link-time optimization

- Un programme C/C++ est composé de translation units.
 - 1 translation unit \approx 1 fichier .c/.cpp.
- Par défaut, le compilateur ne peut pas optimiser à travers plusieurs translation unit.
- Le LTO permet au linker d'effectuer des optimisations qui prennent en compte l'ensemble des translation units.

Étapes de compilation finales

- Compiler avec `-fprofile-generate` ;
- Faire tourner le programme sur des données représentatives ;
- Recompile le programme avec
`-flto -fprofile-use -fprofile-correction`.

Gain de performance

Optimisation	Amélioration	Intervalle de confiance (2σ)
LTO	0.17%	$\pm 1.12\%$
LTO & PGO	6.74%	$\pm 1.44\%$
Static LTO & PGO	6.88%	$\pm 0.83\%$

1 Bibliothèques et modules statiques

- Types de bibliothèques
- Bibliothèques statiques
- Modules statiques

2 Profile-guided optimization

- Profile-guided optimization
- Link-time optimization
- Étapes de compilation finales

3 Fast-math

- Arithmétique des nombres flottants
- Exactitude de certains calculs
- Fast-math

Arithmétique des nombres flottants

$$\begin{aligned}(0.75)_{10} &= (+7.5 \times 10^{-1})_{10} \\&= (+11 \times 2^{-2})_2 \\&= (\underbrace{+}_{\text{signe}} \underbrace{1.1}_{\text{mantisse}} \times 2^{\overbrace{-1}^{\text{exposant}}})_2 = (0.11)_2\end{aligned}$$

Cas non-nominaux

$$\begin{aligned}(0.8)_{10} &\approx (1.1001100110011001101 \times 2^{-1})_2 \\ &\approx (0.80000019073486328125)_{10}\end{aligned}$$

La représentation est une approximation du nombre.

IEEE 754

- Formats 32 bits (~ 7.2 digits) et 64 bits (~ 15.9 digits) ;
- Règles spéciales :
 - Nombres dénormalisés (très proches de 0) ;
 - Valeurs spéciales : -Inf, +Inf, NaN ;
 - Règles d'approximation ;
 - Gestion des exceptions.

Égalité entre flottants

- Ne jamais tester l'égalité de deux flottants.
- Options GCC `-Wfloat-equal`. Ex. dans LHCb :
 - `0 == ip`
 - `0.5 == ip`
 - `lhs.m_energy == rhs.m_energy`

```
if (f(x) != 0.)  
    return 5./f(x);  
else  
    ...
```

- Peut-être que les deux $f(x)$ ne seront pas les mêmes,
- à cause de différentes optimisations.

```
float y = f(x);
if (y != 0.)
    return 5./y;
else
    . . .
```

- Si $f(x)$ est proche de 0 alors $5./y$ peut valoir +Inf.

Utiliser `isfinite` après le calcul.

```
float result = 5./f(x);
if (std::isfinite(result))
    return result;
else
    ...
```

Principe

- Ensemble d'options.
- Optimisations mathématiquement valides mais ne respectant pas le standard :
 - $a + b - a = b$.
 - Avec $a = 10^6$ et $b = 10^{-6}$,
Le standards donne 0, fast-math donne 10^{-6} .
- Résultats différents mais pas plus faux que sans `fast-math`.

<https://godbolt.org/z/jqz8P8vjj>

Options

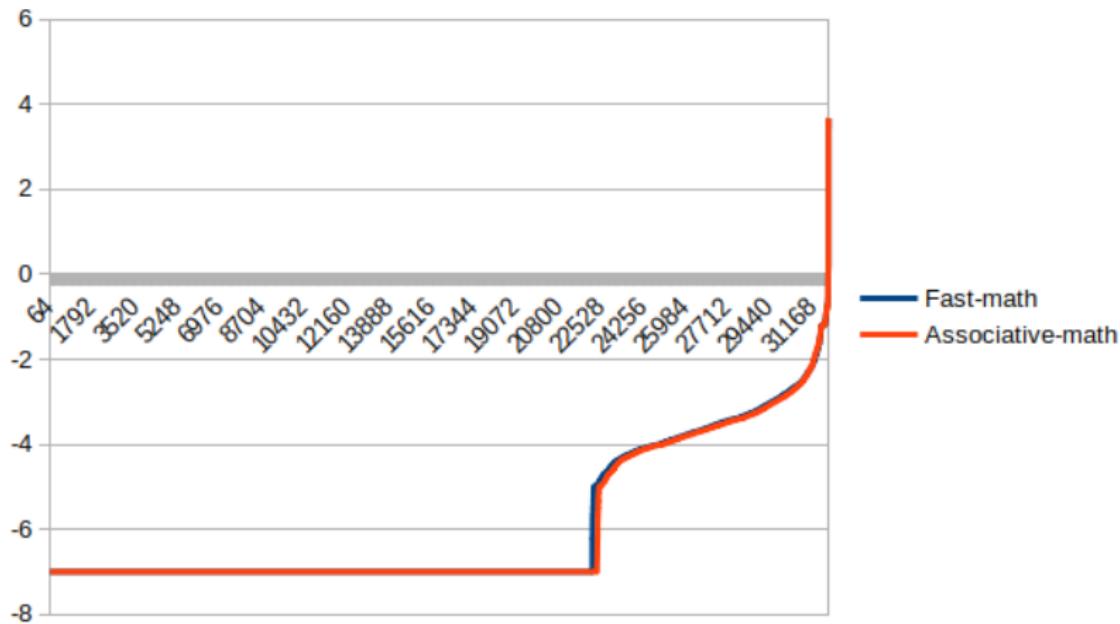
- no-math-errno : ne pas utiliser la variable `errno`.
- no-signaling-nans : désactive le signalement de certains NaNs.
- no-trapping-math : désactive certaines interruptions causées par des flottants.
- finite-math-only : suppose que tous les flottants sont finis.
- no-signed-zeros : ignore la différence entre $+0.0$ et -0.0 .
- associative-math : suppose que les flottants sont associatifs et donc que les opérations peuvent être réordonnées.
- reciprocal-math : autorise de remplacer x/y par $x \times (1/y)$.
- unsafe-math-optimizations : autres optimisations pouvant impacter les approximations.

Gain de performance

Optimisation	Amélioration	Intervalle de confiance (2σ)
Fast-math ¹	5.06%	±0.98%
Associative-math (PGO)	4.73%	±1.51%
Fast-math ¹ + PGO	6.74%	±1.44%
	11.02%	±0.98%

1. sans finite-math-only et unsafe-math-optimizations

Erreur relative des compteurs



Conclusion

- Utiliser des bibliothèques et modules statiques ne semble pas améliorer les performances et mène à des bugs.
- Utiliser LTO & PGO rend le programme plus rapide.
- Fast-math améliore les performances et permet de détecter des instabilités numériques.

Apports du stage

- Travail au sein d'un laboratoire de recherche ;
- Appris beaucoup de choses, notamment sur le fonctionnement bas niveau de Linux et du C++ ;
- Pratique de l'anglais.

Merci