Introduction à l'optimisation

Hadrien Grasland

2022-10-25

Etape 0 : Qualité logicielle

- Toute modif d'un code peut introduire des erreurs…
  - …et un code optimisé tend à être moins intuitif/clair
- Précautions de base
  - **Gestion de version** (1 code qui marche = 1 commit)
  - **Tests**, si possible…
    - Sous-ensemble rapide (~secondes)
    - Simples à exécuter (1 commande)
    - Automatisables (intégration continue)

Etape 1 : Goulot d'étranglement

- Quelle est la **ressource matérielle** qui limite l'exécution ?
  - Haut niveau : CPU (utilisateur/OS), stockage, réseau, GPU
  - Bas niveau : ALUs int/FP, débits, latences, caches…
- Ce n'est pas forcément facile de trouver la réponse*
- Les outils & méthodes sont spécialisés pour une ressource
  - Si ce n'est pas la ressource limitante, résultats sans intérêt
  - Dans la suite, je vais supposer que c'est le CPU

* Je termine actuellemnt un bouquin de >100 pages A4 sur l'analyse de l'utilisation CPU…

…au sein d'une application

- Important de savoir **dans quel code** on passe son temps
  - Si on fait aller 100x plus vite une fonction qui prend 1 % du temps, on n'a gagné que 0,99 % de temps d'exécution…
  - L'optimisation a un coût humain, donc utile de la cibler
- Approches pour être fixé
  - Simple minutage hiérarchique (main() → sous-fonctions…)
  - Logiciels dédiés (VTune, perf, callgrind…)

Etape 2 : Etat de l'art

- Chercher des **sous-problèmes classiques** dans le code
  - Algèbre linéaire, transformée de Fourier, convolution…
- Réutiliser le **travail des autres** sur ces problèmes
  - Bibliothèques déjà écrites
  - Publications algorithmiques
    - Articles scientifiques
    - Livres de maths appliquées
    - StackOverflow, blogs...

Etape 3 : Optimisations « haut niveau »

- La logique générale est-elle optimale ? Cf mantras de Gregg :
  1. Don't do it
  2. Do it, but don't do it again
  3. Do it less
  4. Do it later
  5. ~~Do it when they're not looking~~
  6. ~~Do it concurrently~~
  7. Do it cheaper

Peu applicables en calcul batch !

Etape 4 : Organisation des données

- Souvent, un code limité par « le CPU » est limité par la RAM
- Raison : **Asymétrie puissance de calcul vs débits de données**
- Prenons le CPU Intel Xeon Silver 4210 utilisé sur MUST*
  - Puissance de calcul : $3{,}5.10^{11}$ calculs en double (f64)/sec
  - Débit max RAM : $1{,}2.10^{11}$ octets/sec = $1{,}4.10^{10}$ f64/sec
  - Donc il faut **≥49 calculs par flottant double précision** échangé avec la RAM pour ne pas être « memory-bound » !

* Je mets en annexe le calcul si vous voulez le refaire pour votre CPU favori

De l’importance des caches

- Peu de programmes font 50 calculs par flottant lu ou écrit
- Solution : tirer au mieux parti des **mémoires cache**
  (= petites mémoires rapides intégrées au CPU)
- Exemple du Xeon Silver 4210 de MUST* :
  - L1d : 32 ko/coeur, débit (2+1)x64 octet/cy = 422 Go/s/coeur
  - L2 : 1 Mo/coeur, débit 64 octet/cy = 140 Go/s/coeur
  - L3 : 14 Mo partagés, débit 64 octet/cy/coeur comme L2
  - Donc il faut juste faire **1 calcul/lecture L1d (ou 2/écriture)**

* Ces spécifications varient peu d’une génération CPU à l’autre

Propriétés des caches x86 (Intel, AMD)

- **Automatiques** (chaque lecture RAM passe par L3, L2, L1d)
- Travaillent sur des blocs alignés de 64 octets
  - Grouper les données par utilisation (localité **spatiale**)
- Amener de nouvelles données = supprimer les anciennes
  - Réutiliser rapidement une donnée (localité **temporelle**)
- **Associativité** limitée
  - Attention aux accès à des adresses mémoires séparées par une grosse puissance de 2 (ex : lignes d'un tableau 2D)

Et la latence ?

- Parfois, ce n’est pas le débit qui limite, mais la latence
  (= temps pour récupérer une donnée)
- Sujet plus complexe, mais la stratégie générale est de…
  - Rester dans les niveaux les plus rapides de cache
  - Eviter les accès indirects (ex : tableau de tableaux)
  - Globalement, réduire la longueur des chaînes d’opérations

Etape 5 : Complexité de la logique

- Les instructions conditionnelles (if, switch, etc) ont un coût
  - Le CPU ne peut en exécuter qu'une par cycle
  - Il doit prédire la condition, l'erreur coûte cher (15-20 cy[1])
  - Donc en avoir peu + conditions homogènes @ boucles
  - Il y a des techniques pour en utiliser moins (« branchless »)
- Les méthodes virtuelles (programmation objet) ont des coûts
  - Empêchent l'*inlining* → Latence, instructions en plus
  - Ok à haut niveau, mais à bannir du coeur du calcul

[1] Mesures sur CPU Intel « Haswell » : https://www.7-cpu.com/cpu/Haswell.html

Etape 6 : Efficacité du calcul scalaire

- Ordre de grandeur du coût d'opérations flottantes :
  - ADD, SUB, MUL, FMA (a*b+c) : 0,5 cycles*
  - DIV, SQRT : 3-4 cycles (6-8x plus lent)
  - EXP, LOG : ≥5-6 cycles (≥10-12x plus lent)
  - SIN, COS : ≥10-11 cycles (≥20-22x plus lent)
  - ATAN : ≥22 cycles (≥44x plus lent)
- Conséquences : Restez simples, réutilisez vos inverses, et rappelez-vous de vos identités trignonométriques !

* Le CPU peut exécuter deux opérations de ce type par cycle, hors quelques exceptions

Etape 7 : Vectorisation

- Bonne nouvelle : **1 instruction CPU travaille sur N données**
  - 2 x f64 ou 4 x f32* avec SSE (tous les CPUs x86 modernes)
  - 4 x f64 ou 8 x f32* avec AVX (~80-90 % des CPUs de WLCG)
- Mauvaise nouvelle : C'est pas simple à utiliser
  - Restreint à des opérations « simples » (ADD, FMA, …)
  - On doit vraiment faire la même chose pour chaque donnée
  - Très sensible au rangement des données en mémoire
  - Pas si facile d'être performant ET portable

* On voit qu'il est intéressant d'utiliser la simple précision là où c'est possible : calcul vectoriel 2x plus rapide doublé d'une réduction 2x de la bande passante mémoire, empreinte cache…

Approches pour vectoriser

- **Sous-traiter à quelqu'un d'autre** (bibliothèques vectorisées)
- Ecrire le code de sorte que le compilateur le fasse (difficile)
- Utiliser directement SSE, AVX (difficile, non portable)
- **Couche d'abstraction** (recommandé pour nouveaux calculs)
  - Approche 1 : Extension de compilateur (clang + GCC)
  - Approche 2 : Bibliothèque (MIPP, xsimd, Vc, eve, std::simd…)

Etape 8 : Parallélisation

- Rarement nécessaire en HEP, car **souvent déjà fait pour nous**
  - On ne doit pas battre le séquentiel, mais N evts en parallèle
  - Si ça tourne déjà Nx plus vite, pas/peu de gain à attendre !
- Seulement pertinent pour économiser une ressource partagée
  - Cache L3
  - Capacité & débit RAM
  - Stockage, réseau…
- Pas si simple (/!\ coûts + correction de la synchronisation)

Résumé du plan d’action

0. Avoir une infra de qualité logicielle solide
1. Vérifier ce qui limite les perfs
2. Réutiliser du code et des algos existants
3. Rendre l’algorithme plus intelligent
4. Bien ranger ses données
5. Simplifier la logique
6. Simplifier les opérations
7. Vectoriser les calculs
8. Paralléliser si nécessaire

Merci de votre attention !

Calcul puissance de calcul standardisée

- Produit de plusieurs spécifications: exemple Xeon Silver 4210
  - Nombre de coeurs : 10
  - Fréquence « base » : x 2,2 GHz
  - Largeur max vectorisation f64 : x 512/64 (AVX-512)
  - FMAs par cycle : x 1
  - Opérations par FMA : x 2 (par définition du FMA = a*b+c)

    = 352 GFLOP/s

Calcul bande passante max RAM

- Exemple pour le Xeon Silver 4210
  - Nombre de canaux mémoire du CPU : 6
  - Données transférées par cycle RAM : x 8 octets (DDR)
  - « Fréquence DDR» (ex : DDR4-**2400**) : x 2400 MHz

    = 115,2 Go/s
- Double précision = 64 bits = 8 octets : / 8 octets

  = 14,4 G f64/s

Et les GPUs dans tout ça ?

- Il y a de bonnes spécifications techniques sur Wikipedia EN
  - Nvidia : https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
  - AMD : https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units
- Prenons la Nvidia A100 utilisée sur MUST :
  - Puissance calcul f64 : $9{,}7.10^{12}$ opérations f64/sec
  - Débit VRAM : $1{,}6.10^{12}$ octets/sec = $1{,}9.10^{11}$ f64/sec
  - Donc il faut **≥50 opérations par f64** échangé avec la VRAM