Optimisation des corrélations

Hadrien Grasland 2021-02-25

Précédemment

- Analyse des performances CPU
- Optimisation de la localité spatiale
- Optimisation de la localité temporelle du 1er flux
- Première vectorisation du calcul
- Premières études sur la localité du 2e flux
  - Un problème plus complexe que prévu !

Ces 2 dernières semaines…

- Priorité : Produire un démonstrateur CPU complet
  - Entrées aléatoires → Sorties vérifiées
  - Formats de données entrée/sortie proches de l'idée initiale
  - On oublie pour l'instant le passage à l'échelle à >8 flux
  - Meilleure vectorisation et localité des accumulateurs
  - Relativement facile à compiler

Optimisation de la vectorisation

Un mot sur les registres

- Les registres sont la mémoire de travail du CPU
  - L'arithmétique se fait toujours au sein des registres*
  - Passage mémoire <-> registres par opérations *load/store*
- Ils sont peu nombreux et spécialisés. Pour la vectorisation**...
  - SSE : 16 registres 128 bits (processeurs 64-bit)
  - AVX : 16 registres 256 bits
  - AVX-512 : 32 registres 512 bits

* Sous x86 l'arithmétique en mémoire est décomposée en *load/store* + arithmétique en registre.
** Je passe ici sous silence un monde de complexité invisible (*aliasing*, *renaming*…).

Localité en registres

- Rappel : un corrélateur naïf a une intensité arithmétique de 1
- C'est insuffisant pour saturer le matériel
  - CPU Intel actuel* : 1 cycle = 2 FMA, 2 *load* et 1 *store*
  - Avant : 2 cycles = 2 mul, 2 add, et (2 *load* ou 1 *load* + 1 *store*)
  - Donc il faut **minimum** 2 FLOP par *load* et 4 FLOP par *store*
- Solution : Maintenir les données dans les registres CPU
  - ...ce qui a nécessité un passage à la vectorisation explicite

* Depuis la microarchitecture Haswell, sortie en 2013. Je me focalise ici sur les processeurs haut de gamme, les puces d'entrée de gamme peuvent avoir des limitations supplémentaires.

Que peut-on garder en registre ?

- Si on itère sur des paquets à fréquences et flux constants
  - …alors on utilise toujours les **même accumulateurs**
  - Cela permet d'augmenter le nombre d'opérations par *store*
- Si, ayant choisi un flux de référence, on itère sur les autres flux
  - …alors on utilise toujours les **mêmes données de référence**
  - Cela permet d'augmenter le nombre d'opérations par *load*
- Ces deux approches peuvent être combinées

En images

Registre 1	Registre 2	Registre 3
Registre 4	Registre 5	Registre 6

Registre 7	Registre 8	Registre 9
Registre 10	Registre 11	Registre 12

Registre 13	Registre 14	Registre 15

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Registre 7	Registre 8	Registre 9
Registre 10	Registre 11	Registre 12

Registre 13	Registre 14	Registre 15

- **Load flux 1**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Registre 13	Registre 14	Registre 15

- **Load flux 1**
- **Load flux 2**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Re*Re Paquet 1	Registre 14	Registre 15

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Re*Re Paquet 1	Im*Im Paquet 1	Registre 15

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Re*Re Paquet 1	Im*Im Paquet 1	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Re*Re Paquet 2	Im*Im Paquet 1	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Re*Re Paquet 2	Im*Im Paquet 2	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Re*Re Paquet 3	Im*Im Paquet 3	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**
- **… etc …**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Acc. global Re	Im*Im Paquet 3	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**
- **… etc …**
- **Load acc. global Re**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Acc. global+local Re	Im*Im Paquet 3	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**
- **… etc …**
- **Load acc. global Re**
- **Accumulation**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Acc. global+local Re	Im*Im Paquet 3	Acc. local Re

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**
- **… etc …**
- **Load acc. global Re**
- **Accumulation**
- **Store acc. global Re**

- **1 *store* pour 6 produits et 6 sommes => Bien**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 2, Paquet 1, Re	Flux 2, Paquet 2, Re	Flux 2, Paquet 3, Re
Flux 2, Paquet 1, Im	Flux 2, Paquet 2, Im	Flux 2, Paquet 3, Im

Acc. global+local Im	Im*Re Paquet 3	Acc. local Im

- **1 *store* pour 6 produits et 6 sommes => Bien**

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**
- **… etc …**
- **Load acc. global Re**
- **Accumulation**
- **Store acc. global Re**
- **… idem partie Im …**

En images

Flux 1, Paquet 1, Re	Flux 1, Paquet 2, Re	Flux 1, Paquet 3, Re
Flux 1, Paquet 1, Im	Flux 1, Paquet 2, Im	Flux 1, Paquet 3, Im

Flux 3, Paquet 1, Re	Flux 3, Paquet 2, Re	Flux 3, Paquet 3, Re
Flux 3, Paquet 1, Im	Flux 3, Paquet 2, Im	Flux 3, Paquet 3, Im

Acc. global+local Im	Im*Re Paquet 3	Acc. local Im

- **Load flux 1**
- **Load flux 2**
- **Re*Re paquet 1**
- **Im*Im paquet 1**
- **Accumulation**
- **Re*Re paquet 2**
- **Im*Im paquet 2**
- **… etc …**
- **Load acc. global Re**
- **Accumulation**
- **Store acc. global Re**
- **… idem partie Im …**
- **Load flux 3**

- **1 *store* pour 6 produits et 6 sommes => Bien**
- **~10 *loads** pour 12 produits et 12 sommes => OK**

* 6 *loads* flux2, 2 *loads* accumulateur et 6 * Nflux/Nvis = 12 / (Nflux + 1) *loads* de flux1 amortis

En pratique

- Avec un format de données idéal pour le CPU, on obtient
  - 35 GFLOPS (66% max) sur un coeur de i3-3220
  - 92 GFLOPS (38%* max) sur un coeur de i9-10980XE
  - 1.5 TFLOPS (62%* max) sur 16 coeurs de i9-10980XE
- Mais l'idéal du CPU n'est hélas pas le nôtre…
  - Coefficients de Fourier désentrelacés
  - Stockage contigu dans l'ordre de lecture / écriture

* Il y a une différence d'efficacité car 1/il faut plusieurs coeurs pour saturer la bande passante mémoire et 2/la fréquence d'horloge baisse en parallèle (perte du « turbo » + surchauffe AVX-512).

Coût des données entrelacées

- Finalement, les données entrelacées ne coûtent pas trop cher :
  - Sur 1 coeur i3-3220, on passe à 29 GFLOPS (-17%)
  - Sur 1 coeur i9-10980XE, on passe à 88 GFLOPS (-4%)
  - Sur 16 coeurs i9-10980XE, on passe à 1,3 TFLOPS (-11%)
- Optimisation : garder les données désentrelacées en cache
  - Prix à payer : Plus d'utilisation du cache L1 liée aux flux…
  - …mais à 8 flux, on n'en manque pas, donc ça se passe bien

Coût des accès non contigüs

- Remettre les FFTs dans le « bon » ordre est plus coûteux :
  - Sur 1 coeur i3-3220, on passe à 24 GFLOPS (**-17%**)
  - Sur 1 coeur i9-10980XE, on passe à 83 GFLOPS (-6%)
  - Sur 16 coeurs i9-10980XE, on passe à 0,9 TFLOPS (**-31%**)
- Pourquoi est-ce si cher ?
  - En accès discontinu, il faut aider le CPU avec du « prefetch »
  - Ce sont des instructions en plus qui ne sont pas du calcul
  - Et ça interfère avec le *prefetching* automatique du CPU…

Format de données actuel

- Je suis donc revenu à des FFTs entrelacées et contigües
  - Devrait faciliter la vie pour le calcul des FFTs en DAQ
  - A long terme, étudier viabilité d'un post-traitement DAQ ?
    - J'ai tenté un pré-traitement, mais ce n'est pas rentable…
- J'ai gardé les optimisations qui peuvent être faites en offline
  - Espacement entre les FFTs pour mieux utiliser les caches*
  - Rangement des FFTs par ordre de lecture lors du calcul

* Plus de détails sur demande, mais en gros, les caches CPU n'aiment pas du tout qu'on fasse des accès mémoire dont les adresses sont séparées par une grosse puissance de 2.

Autres travaux

Maturation du code

- Sélection automatique de la plupart des paramètres ajustables
  - Possibilité de prendre la main avec -DPARAM=<valeur>
- Portabilité vers d'autres compilateurs que GCC
  - Le C++ standard ne sait pas tout faire (cf SIMD, prefetch…)
  - Avec clang, je peux garantir des performances comparables
  - Avec les autres, je peux juste garantir que ça fonctionne*
- Meilleur benchmarking (calcul FLOPS, GB/s, *warmup*, stats…)

* Moyennant un support correct du C++17, dans l'état actuel du code.

Validation numérique

- Deux sources possibles d'instabilité numérique dans le calcul
  - Soustraction de nombres proches → Perte de précision
  - Accumulation non bornée sur les paquets
- J'ai vérifié que ça se passe bien sur un signal semi-réaliste
  - Chaque flux est une somme de résonances lorentziennes
  - Une résonance « physique » d'amplitude A dans tous les flux
  - En *float*, accord à 10^{-5} A près du calcul avec la théorie

Prochaines étapes

- Modulariser le code → Mais où le ranger dans les dépôts git ?
- Plus généralement, faciliter la maintenance future du code*
- Adapter à la parallélisation de PAON-4 (actuellement OpenMP)
- Permettre le passage à l'échelle à >8 flux
- Mieux tirer parti du FMA et des 32 registres de AVX-512
- Faire le portage GPGPU

* Actuellement, plusieurs régions du code doivent être maintenues synchrones au niveau algorithmique, j'aimerais trouver un moyen de les synchroniser de façon plus automatique.

Merci pour votre attention !

Rôle du *padding* (1)

- Un cache CPU est organisé en N_B **banques**. En simplifiant...
  - Adresse mémoire 1 ⟶ Banque cache 1
  - Adresse mémoire 2 ⟶ Banque cache 2
  - ...
  - Adresse mémoire N ⟶ Banque cache N
  - Adresse mémoire N + 1 ⟶ Banque cache 1
  - Adresse mémoire N + 2 ⟶ Banque cache 2
  - ...

Rôle du *padding* (2)

- Les banques ont un impact visible sur la performance
  - Les accès mémoire à différentes banques sont parallélisés
- Les banques ont aussi une associativité finie
  - Pas plus de M lignes de cache par banque (8 chez Intel)
- Et si les accès mémoires sont séparés d'un multiple de N_B ?
  - Tout va sur la première banque de cache !
  - Donc zéro parallélisme et 8 lignes de cache (512 o) max

Rôle du *padding* (3)

- Pour limiter ces conflits de banques, on introduit du *padding*
  - Ainsi, ce qui était à l'adresse A se retrouve à A + P
  - Si $A = k*N_B$ alors $A + P = k*N_B + P$
  - Donc $(A + P) \% N_B = P \% N_B$ et non plus 0
  - Avec P bien choisi, on répartit donc les accès entre banques
- Concrètement, si on accède à F fréquences FFT par paquet…
  - Alors on accède à S = 2*F*sizeof(float) octets par paquet
  - Pour éviter les conflits entre paquets, il faut P ≥ S octets

Modèles de vectorisation

- Le problème de fond :
  - C++17 n'a pas de support standard de la vectorisation*
  - La vectorisation automatique des compilos n'est pas fiable
- Approche 1 : Extensions SIMD du compilateur (GCC, clang)
  - Portable entre matériels, non portable entre compilateurs
- Approche 2 : Bibliothèques d'intrinsèques (xsimd, MIPP, …)
  - Portabilité matérielle limitée (typiquement x86 + ARM)

* …et le support qui est introduit dans la STL en C++20 avec « unseq » est basé sur la vectorisation automatique, autrement dit il n'a presque aucun intérêt.