



Architecture matérielle CPU

Hadrien Grasland

2024-02-01

Vous avez dit « architecture » ?

- x86, ARM, POWER = architecture *de jeu d'instruction* (ISA)
- Zen 4, Sapphire Rapids = *micro-architecture* (μ arch) d'une puce
- ISA = **Contrat** entre le code binaire et la machine
 - Public, partagé par de nombreux processeurs
 - Compatibilité descendante, évolue peu hors extensions
- μ arch = **Implémentation** de ce contrat au niveau matériel
 - En partie secret, spécifique à une génération de puces
 - Varie d'une puce à l'autre, évolue vite sans compatibilité

Pourquoi s'intéresser à la μ arch ?

- **Dicte ce qui est performant**, par exemple :
 - Quels opérations arithmétiques prennent moins de cycles ?
 - Quels accès mémoire maximisent la bande passante ?
 - Comment mieux tirer parti des différents caches ?
- Malgré la variabilité, des **tendances de fond** persistent
 - Les choix qualitatifs (ex : superscalaire) sont très stables
 - Les ordres de grandeur quantitatifs varient lentement
 - Beaucoup de points communs même entre ISA différentes

Optimiser les accès mémoire

Posons le problème

- Mettons la perf mémoire en perspective par rapport au calcul
- Exemple : Noeud de calcul avec 2x AMD EPYC 7702
 - **Calcul flottant** : 2 sockets x 64 coeurs x 2 GHz x 2 FMA/cy x 16 calculs f32/FMA = **8.2×10^{12} calculs f32/sec**
 - **Bande passante RAM** : 2 sockets x 204,8 GB/s = 409,6 GB/s = **1.0×10^{11} f32 transférés/sec**
- Interprétation : Pour chaque f32 échangé avec la RAM, il faut effectuer **≥ 80 calculs f32**, sinon la RAM limite la perf !

D'où ça vient ?

- Tout support de stockage doit faire un compromis entre...
 - **(Non-)Volatilité** (préservé le contenu sans électricité)
 - **Performance** (débit et surtout latence)
 - Efficacité **énergétique**
 - **Densité** (GB/m²)
 - **Capacité**
 - **Coût**
- Pour avoir un peu de tout, on utilise une **hiérarchie mémoire**

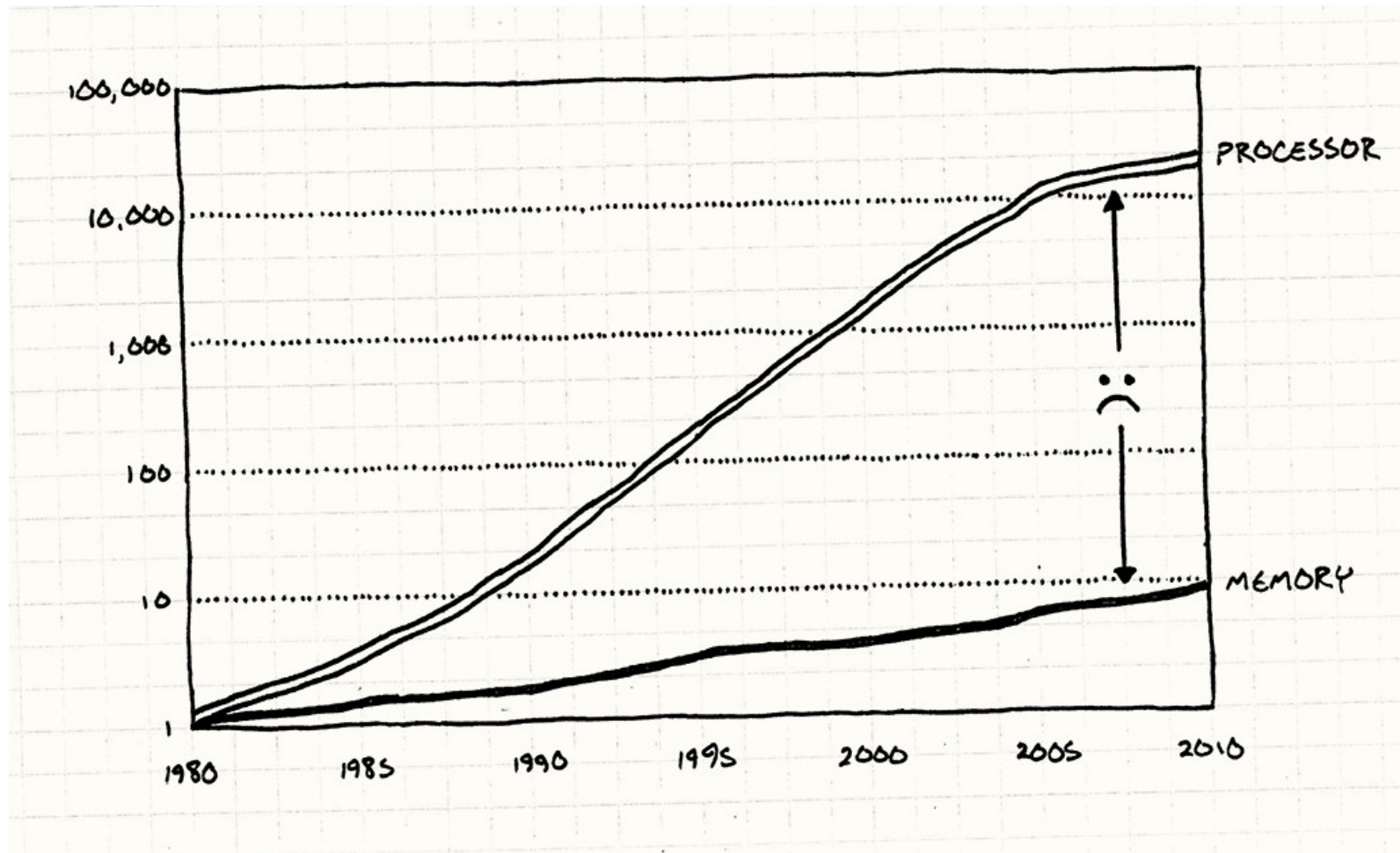
Hiérarchie mémoire x86

- Mémoires d'un système x86 moderne*, vues d'un coeur CPU :
 - Regs : 16x32B, **accès direct** (sans latence, débit infini)
 - L1d : 32 KiB **lat.** 4-8cy **débit** (2+1)x32B/coeur/cy
 - L2 : 512 KiB $\geq 12cy$ 32B/coeur/cy
 - L3 : 4 MiB/coeur $\sim 39cy$ **local** 32B/coeur/cy
 - RAM : ~ 2 GiB/coeur $\sim 10^2cy$ 1~16B/coeur/cy**
 - SSD : ~ 1 TiB $\sim 10^5cy$ 1~4B/**noeud**/cy**
 - HDD : ~ 10 TiB $\sim 10^7cy$ $10^{-2}\sim 10^{-1}B$ /noeud/cy**
 - NFS : Cf SSD/HDD + limites réseau + partage entre N noeuds

* Je me base principalement sur d'AMD, les specs Intel sont similaires

** On suppose un canal DDR5 pour 2-8 coeurs CPU + accès linéaires + fréquence 2-5 GHz

Si on simplifie un peu...



<http://gameprogrammingpatterns.com/data-locality.html>

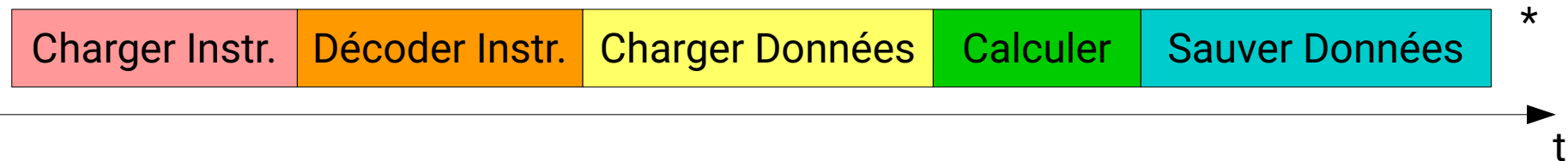
Tirer parti des caches

- Objectif : Données en L1 = Il suffit de faire **≥ 1 calcul / f32 lu !**
- Granularité : Le cache travaille par *ligne* de 64B alignés
 - Localité **spatiale** : Ce qui est utilisé ensemble est contigu
- Les nouvelles données chassent les anciennes (\sim LRU)
 - Localité **temporelle** : On réutilise rapidement chaque donnée
- Accès espacés \rightarrow Soucis de prédiction, associativité, TLB, ...
 - Privilégier les **accès linéaires**, sinon essayer le « **prefetch** »
- Attention aux **écritures parallèles** (\sim RWlock par ligne)

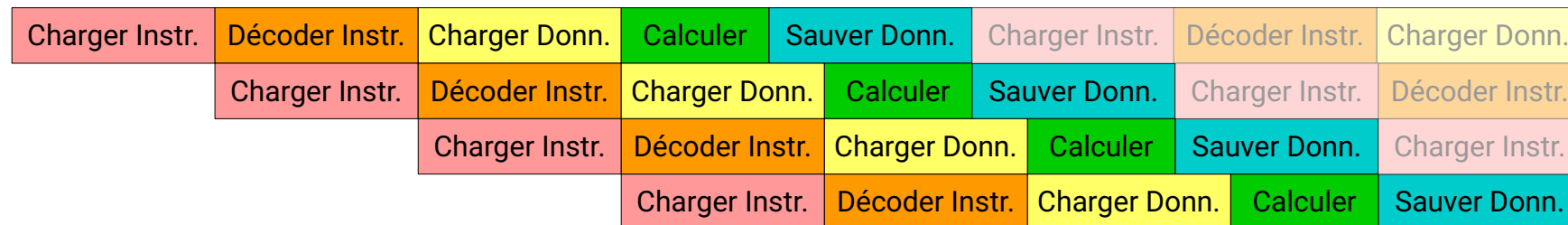
Parallélismes et concurrence

Le *pipelining*

- Une instruction CPU se décompose en plusieurs étapes :



- On peut les exécuter en parallèle (les circuits sont distincts)

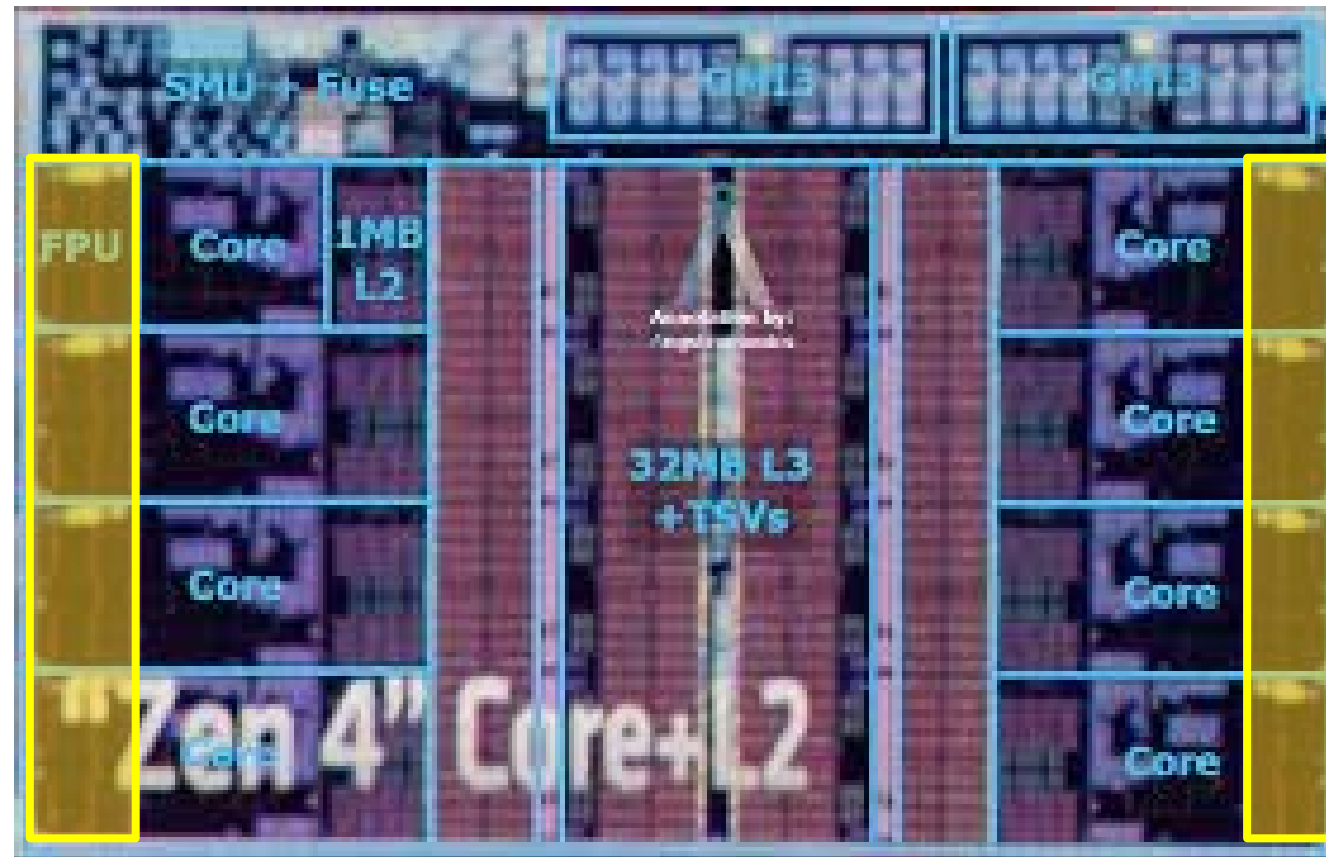


- Mais pour ça la logique du code doit être **prévisible** par le CPU
 - Il n'est pas si bête : prédiction de branches, spéculation...
 - Chaque erreur coûte cher (plusieurs dizaines de cycles)

* Ce schéma est très simplifié, en réalité il y a plutôt quelques dizaines d'étapes de traitement

Dédier plus de transistors au calcul

- Les unités de calcul utilisent relativement peu de transistors...



<https://twitter.com/SkyJuice60/status/1564454510382108672>

...on est donc tenté d'en rajouter plus en partageant le reste !

De la vectorisation au multicoeur

- **Vectorisation/SIMD** : Une instruction agit sur N données
 - Exemple : $a[0..4] \leftarrow b[0..4] + c[0..4] = 1$ instruction CPU
 - Le code machine doit utiliser des instructions spécifiques
- **Superscalaire** : Plusieurs instructions *indépendantes* d'un thread peuvent s'exécuter simultanément ou en concurrence
 - Le code doit s'y prêter : $(a + b) + (c + d) \neq (((a + b) + c) + d)$
- **SMT/Hyperthreading** : Masquer la latence (ex : lecture RAM) en basculant sur un autre *thread* du programme
- **Multicoeur** : Exécution simultanée de threads indépendants

Concurrence et masquage de latence

- Les instructions ont une **latence** plus ou moins élevée
 - En attendant, le CPU essaie d'exécuter d'autres instructions
- Un code qui en dépend trop se heurtera à diverses limites*
ex : nombre maximal d'instructions en vol
- Pour moins faire pression sur ces limites, on peut...
 - Moins dépendre de **mémoires haute latence** (L3, RAM...)
 - Eviter les **chaînes de dépendances** (ex : (((a + b) + c) + d)...))
 - Utiliser moins d'**embranchements** (if/else, switch, ...)

* Une différence entre les CPUs x86 et les GPUs et CPUs « HPC » type A64FX est que les CPUs x86 ont des limites plus généreuses, et tolèrent donc davantage le code non optimal.

Autres considérations

Coût de l'arithmétique

- Toutes les opérations flottantes n'ont pas le même coût*
 - ADD, SUB, MUL, FMA : 2 ops/cy SIMD optimal
 - DIV, SQRT : 6-8x plus **lent** SIMD lent
 - EXP, LOG : $\geq 10-12x$ FMA SIMD difficile**
 - SIN, COS : $\geq 20-22x$ FMA SIMD difficile**
 - ATAN : $\geq 44x$ FMA SIMD difficile**
- **Conséquences** : Préférer les opérations simples, réutiliser les inverses, identités trigonométriques plutôt que $\sin(\text{atan2}(x, y))$

* Mesuré sur CPU Haswell en 2015, a évolué un peu depuis sans que ça change l'ordre

** Non supporté par la libm, nécessite une bibliothèque dédiée. Peu efficace (utilise gather).

Problèmes de code classiques

- Attention, les compilateurs optimisent peu le **calcul flottant**
 - Raison : Change le résultat, au risque de le rendre incorrect
- Attention dans les **fonds de boucles**
 - Optimiser la localité spatiale et temporelle + accès linéaires
 - Peu de branchements (if, switch...), idéalement prévisibles
 - Pas d'orienté objet (perte d'*inlining*, tableaux de pointeurs...)
 - Avoir plusieurs chaînes de dépendances (cf annexe)
- Les **appels systèmes** coûtent de plus en plus cher

En conclusion

- Les processeurs modernes (CPU et GPU) sont complexes
 - C'est inévitable au vu des réalités sous-jacentes
- Comprendre leur fonctionnement permet d'adapter son code
 - Penser le CPU comme un allié qui a besoin d'un peu d'aide
- Ce n'est pas forcément incompatible avec la **portabilité**
 - Nombreux points communs entre x86, ARM, et les GPUs...
 - Différences : paramètres quantitatifs + instructions SIMD
 - Possible de s'en accommoder avec les bonnes abstractions

Références

- Le déséquilibre calcul/mémoire vu par l'auteur de BLAS & LAPACK :
<https://www.youtube.com/watch?v=cS00Tc2w5Dg>
- Publications des constructeurs
 - Fiches techniques type <https://ark.intel.com/content/www/us/en/ark.html>
 - <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
 - <https://www.amd.com/en/support/tech-docs>
- Travaux d'experts indépendants
 - « *Computer Architecture : A Quantitative Approach* » de Hennessy & Patterson
 - <https://www.agner.org/optimize/>
 - <https://en.wikichip.org> + pages wikipedia type « *List of AMD Ryzen processors* »
 - <https://uops.info/>
- Sites d'information spécialisés :
 - <https://www.anandtech.com/>
 - <https://www.nextplatform.com/>

Merci de votre attention !

Avoir plusieurs chaînes de dépendances

- Considérons un code qui somme un tableau de flottants
- Il est préférable d'éviter ce style de code...

```
float acc = 0.0;
for (size_t i = 0; i < N; ++i) acc += input[i];
```

- ...et préférer ce style* (en supposant que M divise N):

```
std::array<float, M> accs { 0.0, 0.0, ..., 0.0 };
for (size_t i = 0; i < N; i += M) {
    for (size_t j = 0; j < M; ++j) accs[j] += input[i+j];
}
// ...and then sum accs...
```

* Au niveau des algos STL préférer `std::reduce` à `std::accumulate` (ne fonctionne pas toujours). 21 / 21