

ARCHITECTURES DE CALCUL MODERNES

...et pourquoi un développeur devrait s'en soucier

Hadrien Grasland

LAL – Orsay



La fin d'une époque

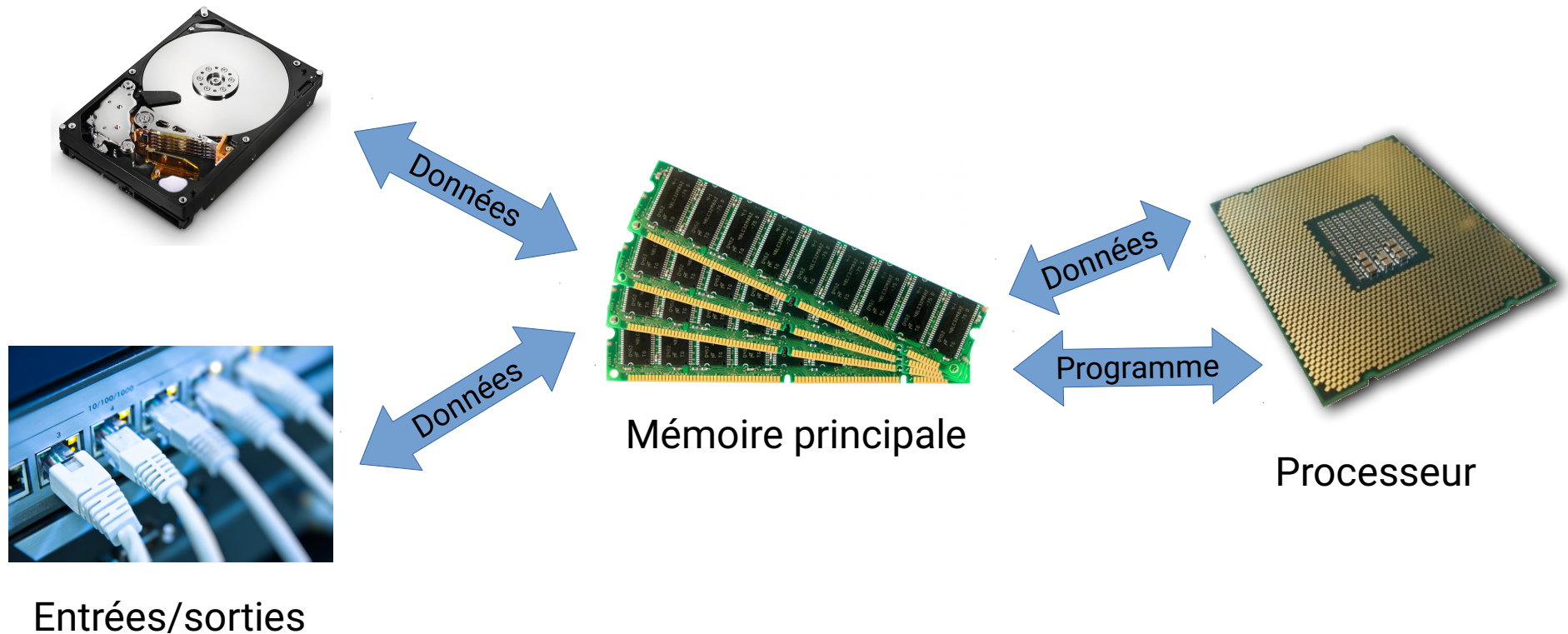
- Nous sortons d'une ère de performance "gratuite" où...
 - Il suffisait de racheter du matériel pour accélérer son code
 - Ou bien de mettre à jour / mieux configurer son compilateur
 - Et donc, on enseignait peu l'art de la performance logicielle (à part quelques bases de complexité algorithmique)

La fin d'une époque

- Nous sortons d'une ère de performance "gratuite" où...
 - Il suffisait de racheter du matériel pour accélérer son code
 - Ou bien de mettre à jour / mieux configurer son compilateur
 - Et donc, on enseignait peu l'art de la performance logicielle (à part quelques bases de complexité algorithmique)
- ...mais il faut de nouveau se retrousser les manches
 - Le matériel moderne est riche en pièges de performance
 - Beaucoup de fonctionnalités nécessitent un support explicite
 - Il est donc important d'en avoir un bon **modèle mental**

Modèle de von Neumann

- Le modèle mental classique des programmeurs
- Assez représentatif des premiers CPUs (~1960)



Von Neumann en action

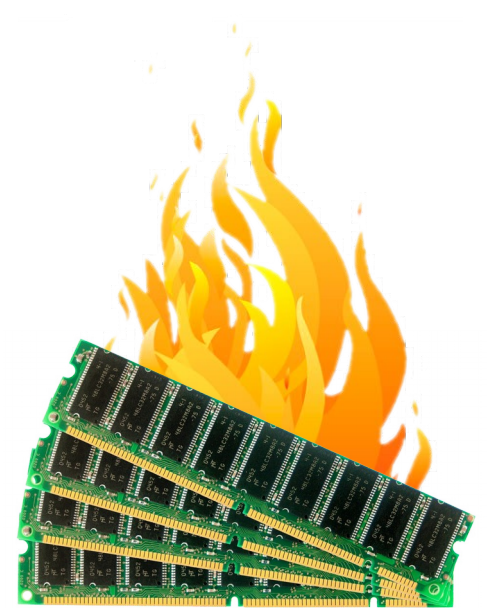
- Calculons la norme d'un vecteur 2D $n = \sqrt{x^2 + y^2}$
 - $x \leftarrow$ Entrée
 - $x2 \leftarrow x * x$
 - $y \leftarrow$ Entrée
 - $y2 \leftarrow y * y$
 - $n2 \leftarrow x2 + y2$
 - $n \leftarrow \text{sqrt}(n2)$
 - Sortie $\leftarrow n$

Von Neumann en action

- Calculons la norme d'un vecteur 2D $n = \sqrt{x^2 + y^2}$
 - $x \leftarrow$ Entrée (Mémoire, E/S)
 - $x^2 \leftarrow x * x$ (Mémoire, processeur)
 - $y \leftarrow$ Entrée (Mémoire, E/S)
 - $y^2 \leftarrow y * y$ (Mémoire, processeur)
 - $n^2 \leftarrow x^2 + y^2$ (Mémoire, processeur)
 - $n \leftarrow \text{sqrt}(n^2)$ (Mémoire, processeur)
 - Sortie $\leftarrow n$ (Mémoire, E/S)

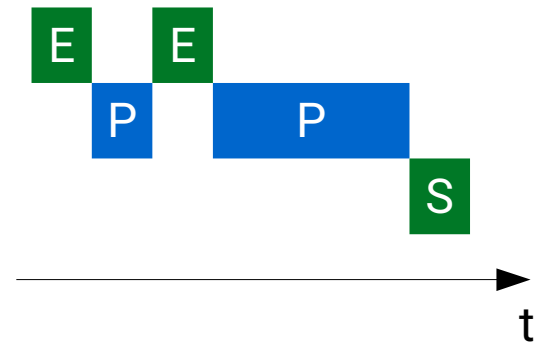
Von Neumann en action

- Calculons la norme d'un vecteur 2D $n = \sqrt{x^2 + y^2}$
 - $x \leftarrow$ Entrée (Mémoire, E/S)
 - $x^2 \leftarrow x * x$ (Mémoire, processeur)
 - $y \leftarrow$ Entrée (Mémoire, E/S)
 - $y^2 \leftarrow y * y$ (Mémoire, processeur)
 - $n^2 \leftarrow x^2 + y^2$ (Mémoire, processeur)
 - $n \leftarrow \text{sqrt}(n^2)$ (Mémoire, processeur)
 - Sortie $\leftarrow n$ (Mémoire, E/S)



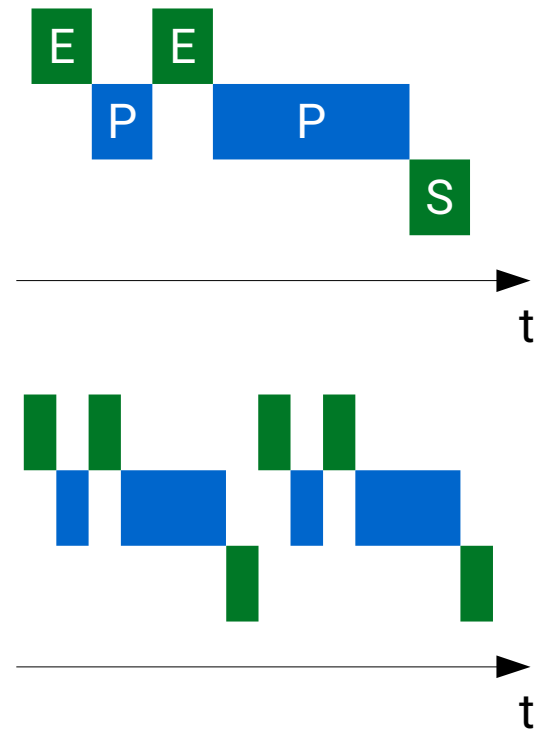
Von Neumann en action

- Calculons la norme d'un vecteur 2D $n = \sqrt{x^2 + y^2}$
 - $x \leftarrow$ Entrée (Mémoire, E/S)
 - $x^2 \leftarrow x * x$ (Mémoire, processeur)
 - $y \leftarrow$ Entrée (Mémoire, E/S)
 - $y^2 \leftarrow y * y$ (Mémoire, processeur)
 - $n^2 \leftarrow x^2 + y^2$ (Mémoire, processeur)
 - $n \leftarrow \text{sqrt}(n^2)$ (Mémoire, processeur)
 - Sortie $\leftarrow n$ (Mémoire, E/S)



Von Neumann en action

- Calculons la norme d'un vecteur 2D $n = \sqrt{x^2 + y^2}$
 - $x \leftarrow$ Entrée (Mémoire, E/S)
 - $x2 \leftarrow x * x$ (Mémoire, processeur)
 - $y \leftarrow$ Entrée (Mémoire, E/S)
 - $y2 \leftarrow y * y$ (Mémoire, processeur)
 - $n2 \leftarrow x2 + y2$ (Mémoire, processeur)
 - $n \leftarrow \text{sqrt}(n2)$ (Mémoire, processeur)
 - Sortie $\leftarrow n$ (Mémoire, E/S)



Limites de ce modèle

- Une machine de von Neumann “pure” serait **très inefficace**
 - La mémoire principale et les E/S limitent la performance
 - Les ressources systèmes s’attendent mutuellement

Limites de ce modèle

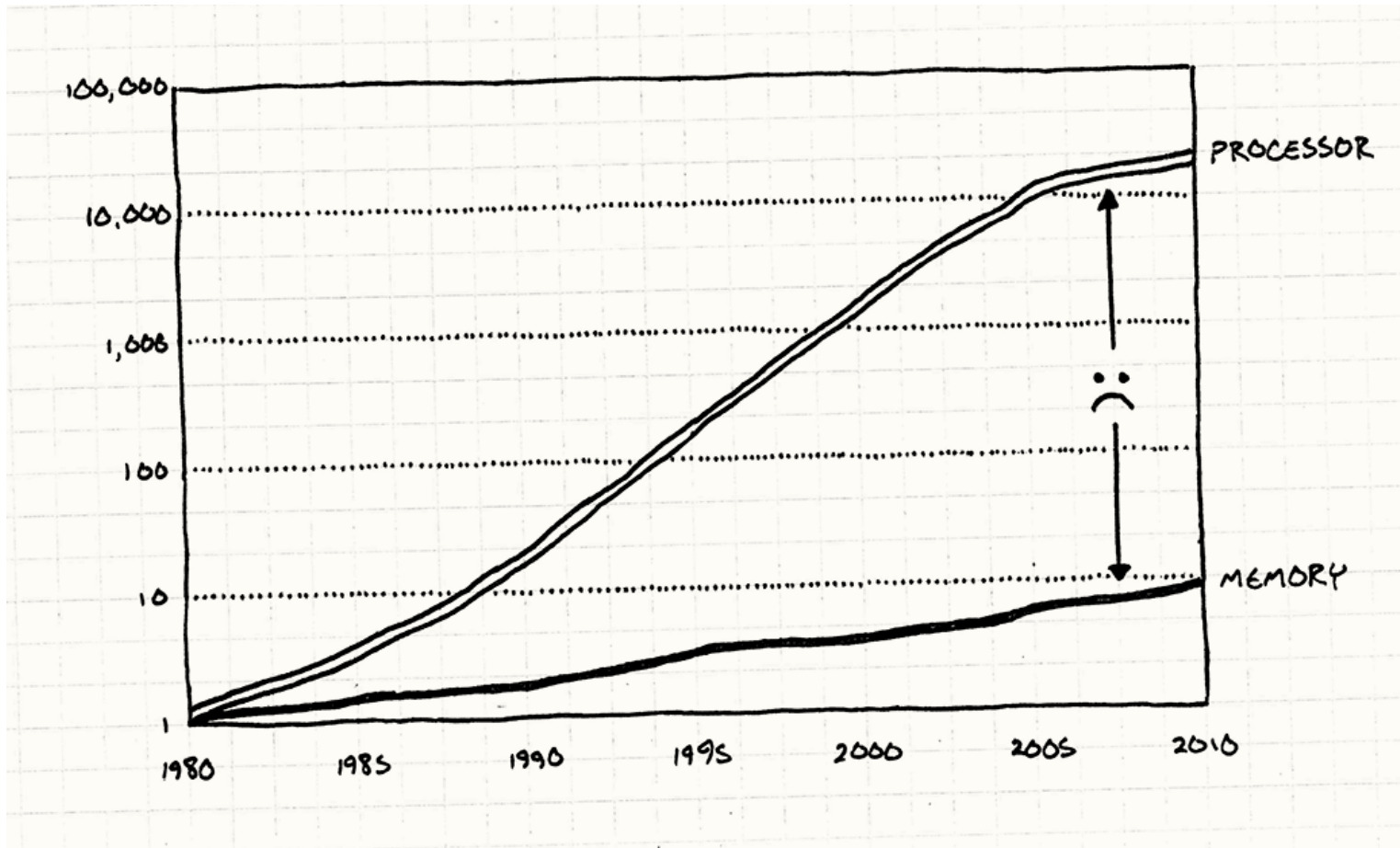
- Une machine de von Neumann “pure” serait **très inefficace**
 - La mémoire principale et les E/S limitent la performance
 - Les ressources systèmes s’attendent mutuellement
- La performance du CPU n’y dépend que de sa **fréquence**
 - Qui a des limites physiques (thermiques), déjà atteintes

Limites de ce modèle

- Une machine de von Neumann “pure” serait **très inefficace**
 - La mémoire principale et les E/S limitent la performance
 - Les ressources systèmes s’attendent mutuellement
- La performance du CPU n’y dépend que de sa **fréquence**
 - Qui a des limites physiques (thermiques), déjà atteintes
- Comment dépasser ces limites?
 - Utiliser des **mémoires plus rapides** (caches)
 - Avoir des **entrées-sorties autonomes** (DMA, non-bloquant)
 - Faire **plusieurs calculs à la fois** (parallélisme, concurrence)

Accélérer la mémoire

Un goulot d'étranglement



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

Comment aller plus vite ?

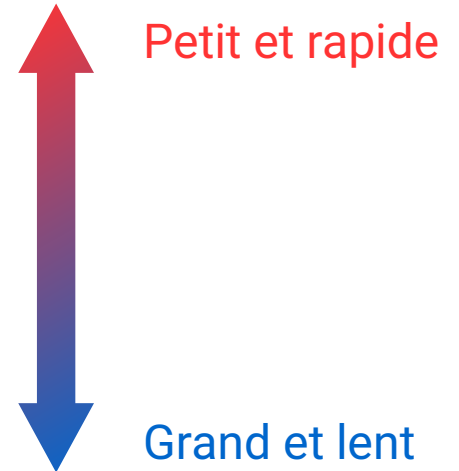
- Le programmeur voudrait une mémoire qui soit...
 - Aussi rapide que le CPU (débit & latence)
 - Vaste (capacité) et bon marché (capacité/€)
 - Non-volatile, économe en énergie...
- Mais le monde est fait de compromis
 - Plus rapide = plus petit, plus cher, plus gourmand...

Comment aller plus vite ?

- Le programmeur voudrait une mémoire qui soit...
 - Aussi rapide que le CPU (débit & latence)
 - Vaste (capacité) et bon marché (capacité/€)
 - Non-volatile, économe en énergie...
- Mais le monde est fait de compromis
 - Plus rapide = plus petit, plus cher, plus gourmand...
- Solution: Combiner **différents types** de mémoires

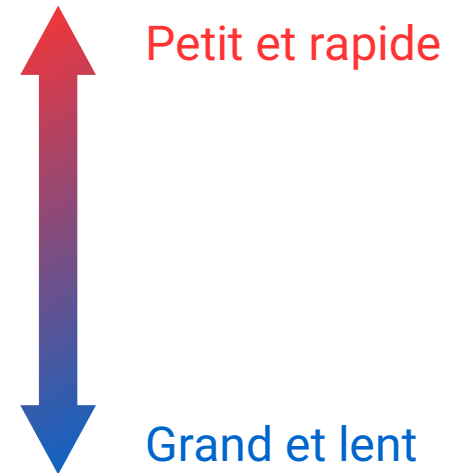
Hiérarchie mémoire

- L'ordinateur moderne combine de nombreuses mémoires:
 - Registre CPU (1 cycle)
 - Caches (~3, 15, 45 cycles)
 - DRAM principale (> 100 cycles)
 - Disque dur, SSD flash (> 10^5 cycles)
 - Accès internet aux données (> 10^7 cycles)



Hiérarchie mémoire

- L'ordinateur moderne combine de nombreuses mémoires:
 - Registre CPU (1 cycle)
 - Caches (~3, 15, 45 cycles)
 - DRAM principale (> 100 cycles)
 - Disque dur, SSD flash (> 10^5 cycles)
 - Accès internet aux données (> 10^7 cycles)
- Enjeu: **Placer** les données au meilleur endroit possible
 - Si possible avant même d'en avoir besoin
 - En minimisant l'intervention du programmeur



Principe d'un cache

- On copie une information dans une mémoire plus rapide
 - Les lectures sont ainsi accélérées
 - Mais les écritures doivent être synchronisées
 - Conséquence: **les écritures (propagées) coûtent cher**

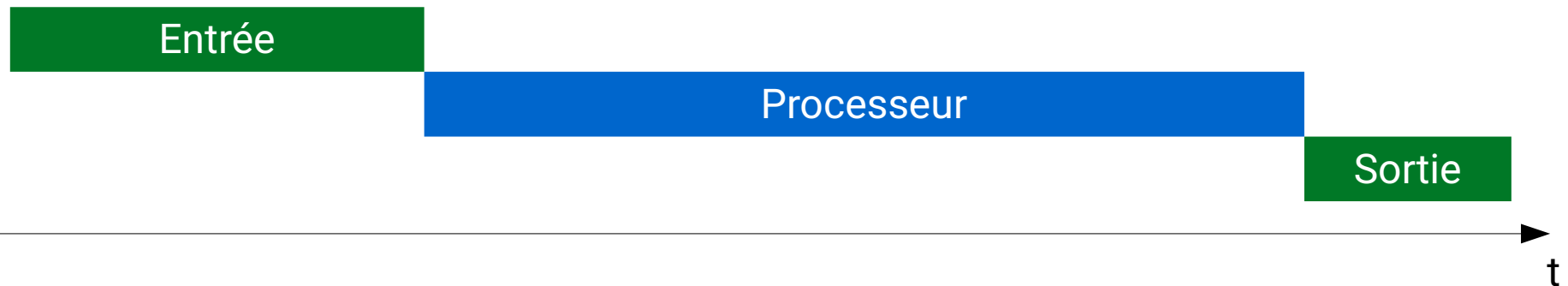
Principe d'un cache

- On copie une information dans une mémoire plus rapide
 - Les lectures sont ainsi accélérées
 - Mais les écritures doivent être synchronisées
 - Conséquence: **les écritures (propagées) coûtent cher**
- La sélection de l'info en cache se base sur 3 éléments
 - Instructions du programmeur (code ordinaire ou "prefetch")
 - Localité **temporelle** (la donnée N re-servira bientôt)
 - Localité **spatiale** (une requête pour N suivie d'une pour N+1)

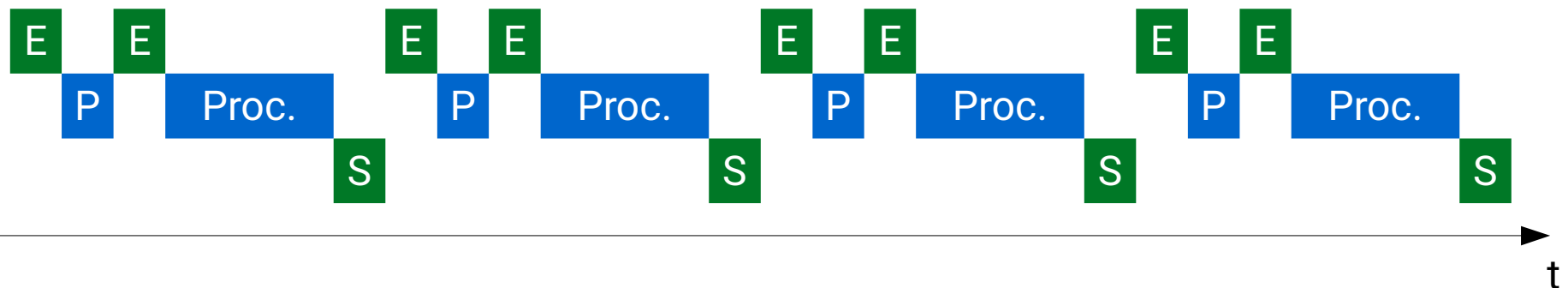
Ne pas attendre les E/S

Effet des entrées/sorties

- Un style de programme en exécution:



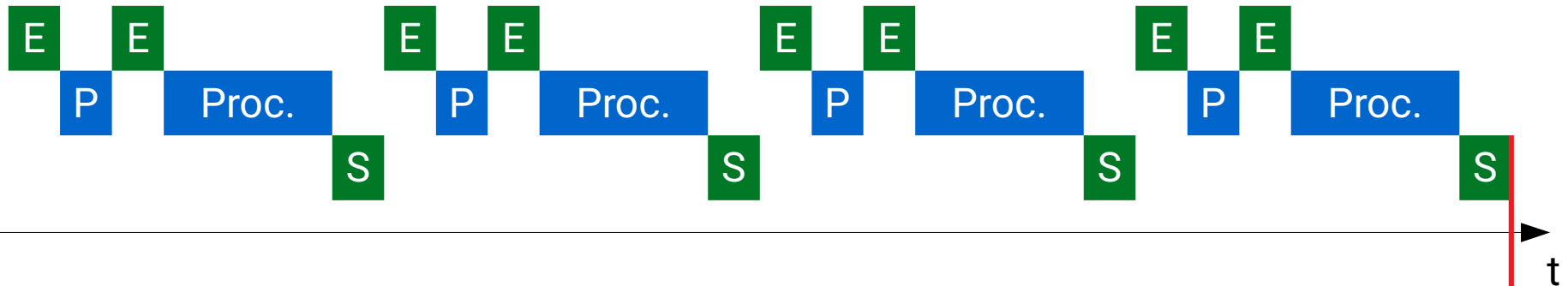
- Un autre style de programme:



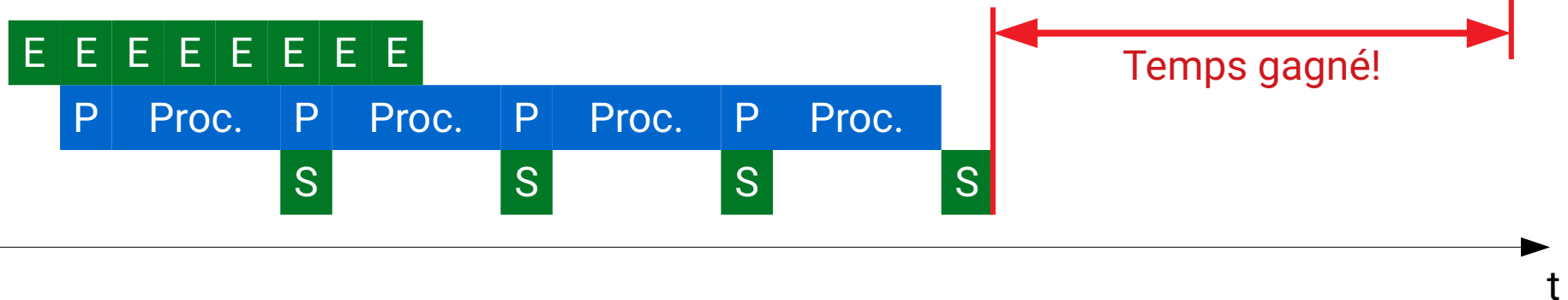
- Quelle différence? Et comment faire mieux?

Intérêt des E/S autonomes

- Avec des entrées/sorties autonomes, ceci...



- ...peut devenir cela:



- Le matériel le permet (DMA), mais sous conditions

Conditions d'E/S autonomes

- Les entrées doivent être commandées à l'avance
 - Par le programmeur (requêtes **asynchrone**)
 - Ou par le système d'exploitation (lectures **spéculatives**)
 - Sur le fond, c'est très similaire à la gestion d'un cache

Conditions d'E/S autonomes

- Les entrées doivent être commandées à l'avance
 - Par le programmeur (requêtes **asynchrone**)
 - Ou par le système d'exploitation (lectures **spéculatives**)
 - Sur le fond, c'est très similaire à la gestion d'un cache
- Le programme ne doit pas attendre inutilement le matériel
 - Ne demander que ce dont on a besoin (à l'OS/lib d'optimiser)
 - Avoir des tampons d'entrée/sortie assez grands
 - Préférer les interfaces **non bloquantes** (epoll, boost::asio...)
 - Isoler les vieilles APIs bloquantes dans des threads

Parallélismes

Un problème de riche

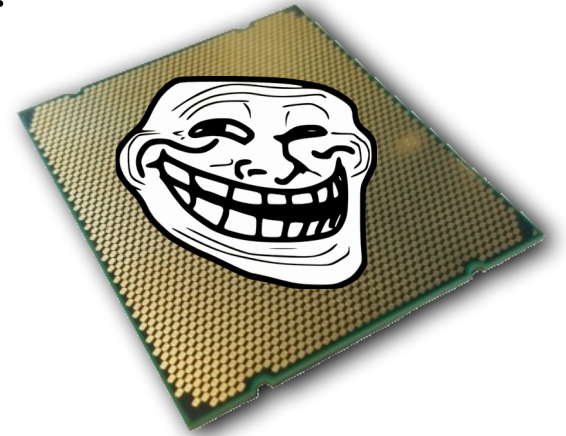
- Il arrive parfois que ce soit le **processeur** qui est limitant
 - Ne pas oublier de mesurer ce qui se passe!

Un problème de riche

- Il arrive parfois que ce soit le **processeur** qui est limitant
 - Ne pas oublier de mesurer ce qui se passe!
- Comment calculer plus vite?
 - Augmenter la fréquence
 - Traiter plus de données à la fois
 - Traiter plusieurs tâches en parallèle
 - Traiter chaque tâche plus efficacement

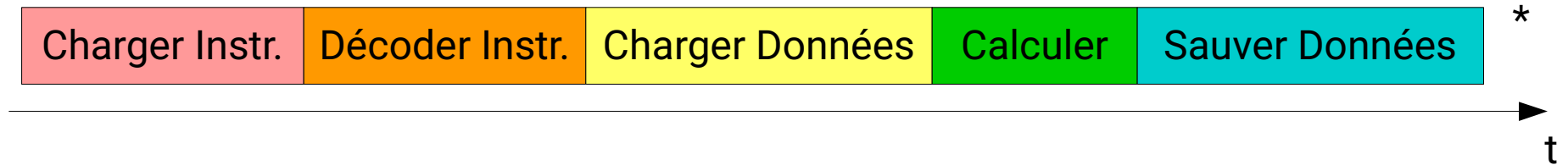
Un problème de riche

- Il arrive parfois que ce soit le **processeur** qui est limitant
 - Ne pas oublier de mesurer ce qui se passe!
- Comment calculer plus vite?
 - ~~Augmenter la fréquence~~ (~1980-2005)
 - Traiter plus de données à la fois
 - Traiter plusieurs tâches en parallèle
 - Traiter chaque tâche plus efficacement



Le pipelining

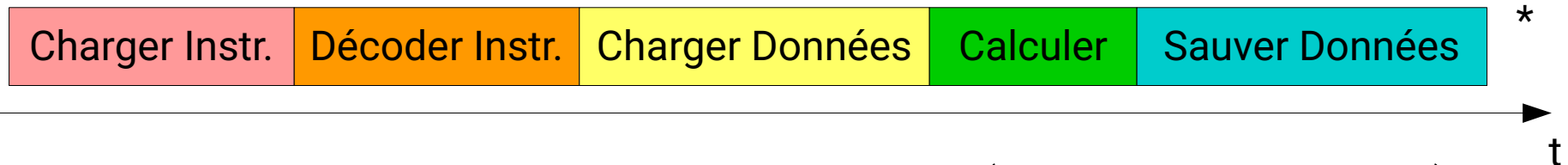
- Une instruction CPU se décompose en plusieurs parties...



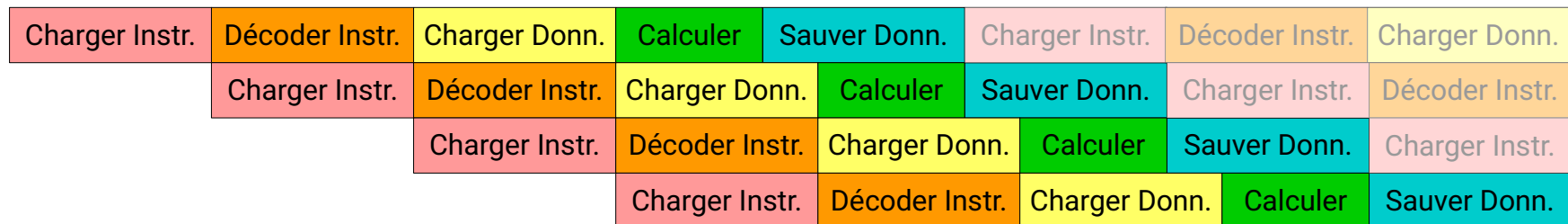
* Modèle très simplifié, une instruction d'un CPU moderne peut s'exécuter en plusieurs dizaines, voire centaines d'étapes!

Le pipelining

- Une instruction CPU se décompose en plusieurs parties...



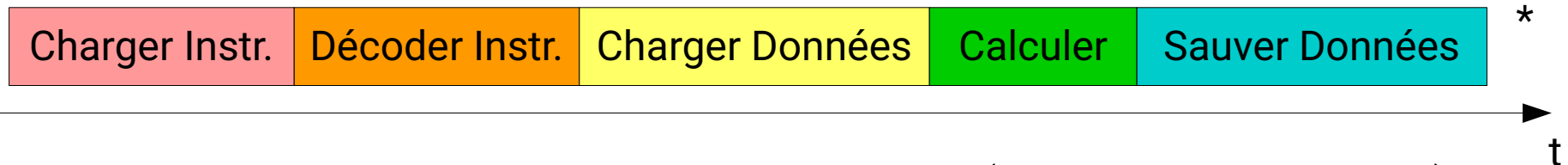
- ...qui peuvent s'exécuter en parallèle (circuits distincts)



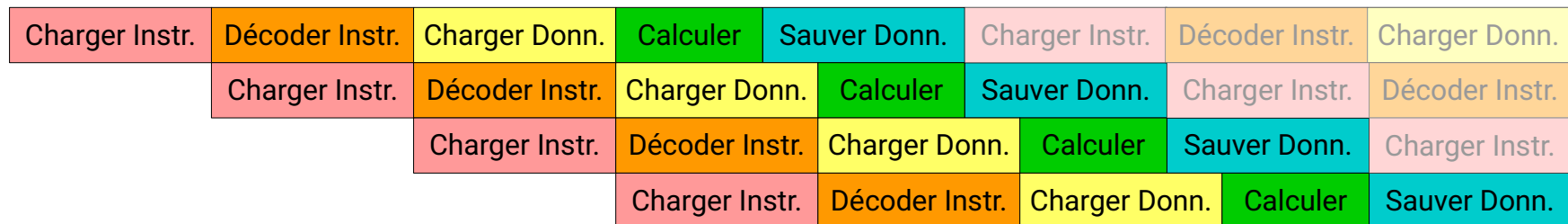
* Modèle très simplifié, une instruction d'un CPU moderne peut s'exécuter en plusieurs dizaines, voire centaines d'étapes!

Le pipelining

- Une instruction CPU se décompose en plusieurs parties...



- ...qui peuvent s'exécuter en parallèle (circuits distincts)

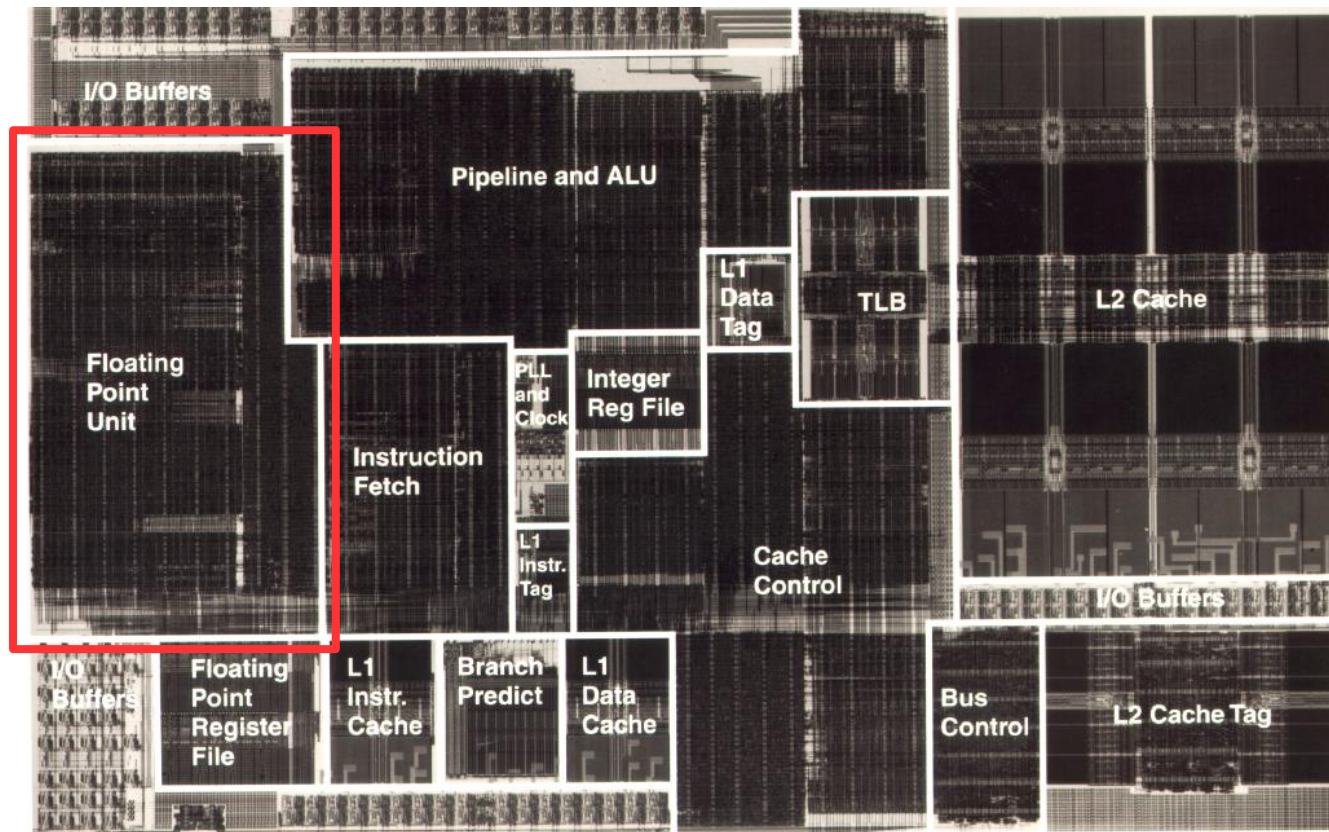


- Requier: Logique de code simple, prévisible par le CPU
 - Contournements: Prédiction de branche, spéculation...
 - Limite: Un “if” mal prédit coûte cher (parfois ~300 cycles!)

* Modèle très simplifié, une instruction d'un CPU moderne peut s'exécuter en plusieurs dizaines, voire centaines d'étapes!

Et le “vrai” calcul parallèle ?

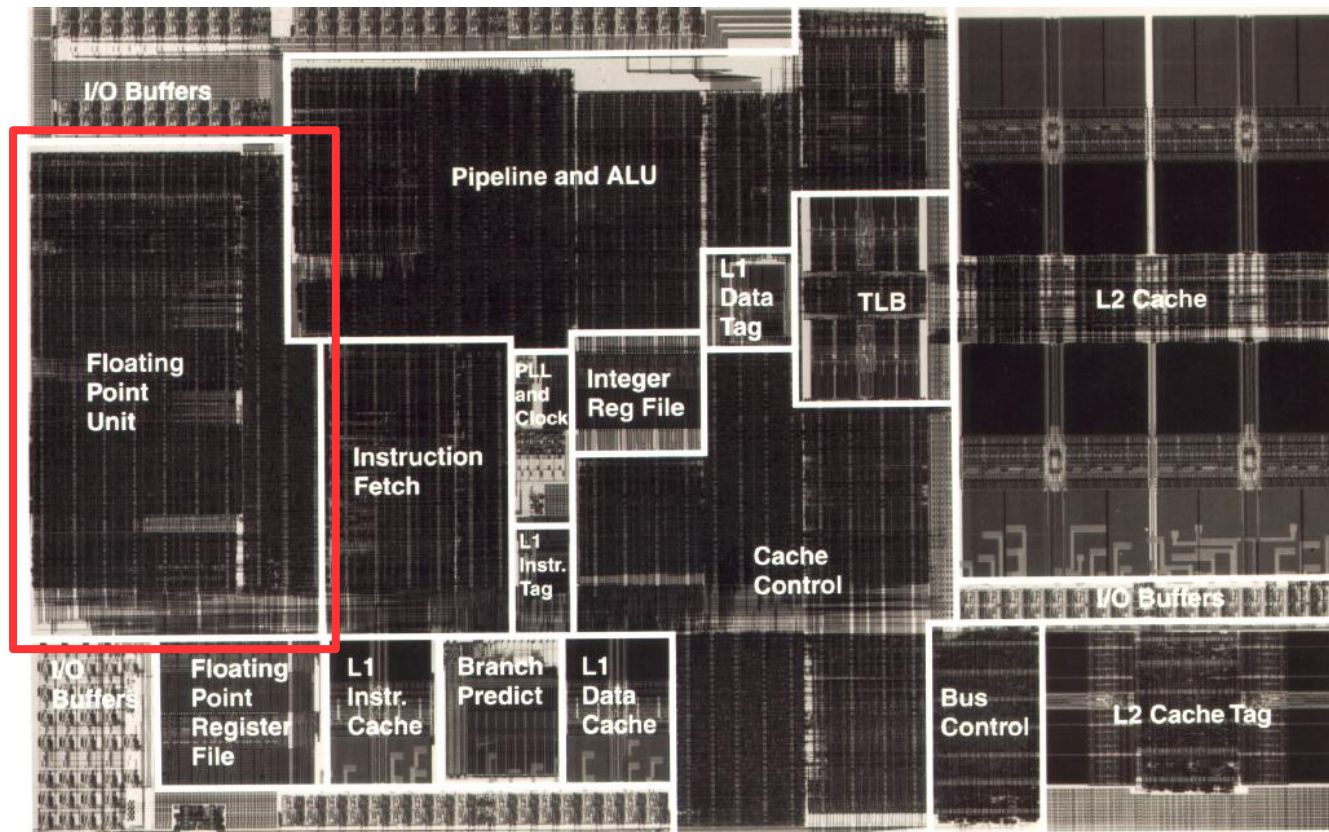
- Dans un CPU, les unités de calcul ne coûtent pas si cher...



https://en.wikichip.org/wiki/exponential_technology/x704

Et le “vrai” calcul parallèle ?

- Dans un CPU, les unités de calcul ne coûtent pas si cher...



https://en.wikichip.org/wiki/exponential_technology/x704

...on peut très bien en rajouter tout en partageant le reste

De la vectorisation au multicoeur

- Vectorisation: Une opération agit sur plusieurs données
 - Exemple: $a[0..4] = b[0..4] + c[0..4]$ → Une instruction CPU
 - Le code machine doit utiliser explicitement ces instructions

De la vectorisation au multicoeur

- Vectorisation: Une opération agit sur plusieurs données
 - Exemple: $a[0..4] = b[0..4] + c[0..4] \rightarrow$ Une instruction CPU
 - Le code machine doit utiliser explicitement ces instructions
- Superscalaire: Plusieurs instructions **indépendantes** d'un même programme simultanément
 - Le code doit s'y prêter: $(a+b) + (c+d) \neq (((a+b)+c)+d)$

De la vectorisation au multicoeur

- Vectorisation: Une opération agit sur plusieurs données
 - Exemple: $a[0..4] = b[0..4] + c[0..4] \rightarrow$ Une instruction CPU
 - Le code machine doit utiliser explicitement ces instructions
- Superscalaire: Plusieurs instructions **indépendantes** d'un même programme simultanément
 - Le code doit s'y prêter: $(a+b) + (c+d) \neq (((a+b)+c)+d)$
- SMT/Hyper-threading: Remplir les "trous" du pipeline (ex: requête cache) avec des instructions d'une autre tâche

De la vectorisation au multicoeur

- Vectorisation: Une opération agit sur plusieurs données
 - Exemple: $a[0..4] = b[0..4] + c[0..4] \rightarrow$ Une instruction CPU
 - Le code machine doit utiliser explicitement ces instructions
- Superscalaire: Plusieurs instructions **indépendantes** d'un même programme simultanément
 - Le code doit s'y prêter: $(a+b) + (c+d) \neq (((a+b)+c)+d)$
- SMT/Hyper-threading: Remplir les "trous" du pipeline (ex: requête cache) avec des instructions d'une autre tâche
- Multi-coeur: Exécution simultanée de plusieurs tâches

Le retour du coprocesseur

- Les CPU modernes sont très complexes
 - Fort accent sur la performance du code séquentiel
 - Gros caches, longs pipelines, prédictions, spéculations...
 - Autant de transistors perdus pour le calcul!
- C'est inutile pour des calculs simples (ex: algèbre linéaire)
 - Emergence de matériel de calcul simplifié (DSPs, GPUs...)
 - Priorités: Débit mémoire, vectorisation et parallélisme
 - Nouveau problème: Communications avec le CPU

En guise de conclusion

Résumons

- Le modèle de von Neumann ne nous dit pas tout
 - Le matériel informatique moderne va **beaucoup** plus loin
- Il faut plus qu'un bon compilateur pour exploiter tout ça
 - **Cache**: Structures de données optimisées pour la localité
 - **DMA**: Interfaces non-bloquantes ou threads d'E/S dédiés
 - **Pipeline, coprocesseur**: Simplification de la logique du code
 - **Vectorisation**: Bibliothèques spécialisées (BLAS, Eigen...)
 - **Coeurs, threads**: Décomposition des programmes en tâches

Que peut-on faire? (1)

- Tout d'abord, ne pas nuire
 - Préparer un bon jeu de tests avant d'optimiser
 - Gare aux pointeurs (chaînage, tableau de tableau, vtables)
 - Attention aux interfaces... (E/S bloquantes, travail inutile)
 - ...et aux infrastructures aussi (interpréteurs, piles de VMs, ramasse-miettes, "Global Interpreter Lock")
- Penser au matériel quand on programme
 - Est-ce que j'exploite les vecteurs, les coeurs, le DMA...?
 - Mes outils ont-ils une implémentation efficace?
 - Et si c'est le cas, est-ce celle que je l'utilise?

Que peut-on faire? (2)

- Savoir à quoi on passe son temps pour se focaliser dessus
 - perf (Linux), Instruments (macOS), WPA/xPerf (Windows)
 - VTune (Intel), CodeXL (AMD), NVVP (Nvidia)
- Ecrire des programmes à plus haut niveau
 - Troquer ses boucles pour des concepts plus abstraits (algèbre linéaire, itérateurs, programmation fonctionnelle...)
 - Ils permettent davantage d'optimisations automatiques
 - Bien utilisés, ils clarifient aussi le code (et donc ses soucis)
- Pour des nombres absolus, aller voir la spec matérielle

Pour finir

- La loi de Moore a autorisé un certain laisser-aller
- Le mur des GHz nous pousse à revoir nos habitudes
- Les débuts sont difficiles, mais l'expertise grandit vite :)
- Quelques références, hélas très copieuses:
 - Hennessy & Patterson, "Computer Architecture: A Quantitative Approach" (architecture matérielle)
 - www.brendangregg.com (analyse de performances Linux)

Merci de votre attention