9

Fondements des structures de données

Avant d'explorer les structures de données complexes, il est nécessaire de comprendre comment Go organise les données en mémoire au niveau le plus fondamental.

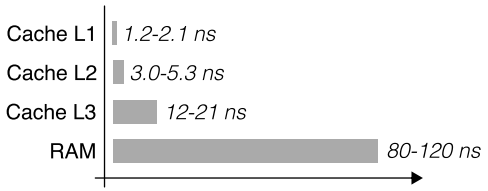
9.1. Hiérarchie mémoire et localité

L'organisation des données influence directement les performances via les mécanismes de cache du processeur. Comprendre cette hiérarchie est fondamental pour optimiser les structures de données et leurs modèles d'accès.

Principe de localité

La performance des programmes modernes dépend largement de leur capacité à exploiter les différents niveaux de la hiérarchie mémoire. Cette hiérarchie, du plus rapide au plus lent, comprend les registres CPU, les caches L1/L2/L3, la mémoire principale (RAM), et le stockage secondaire. La Figure 9.1 présente les ordres de grandeur des temps d'accès à chaque niveau de cette hiérarchie.

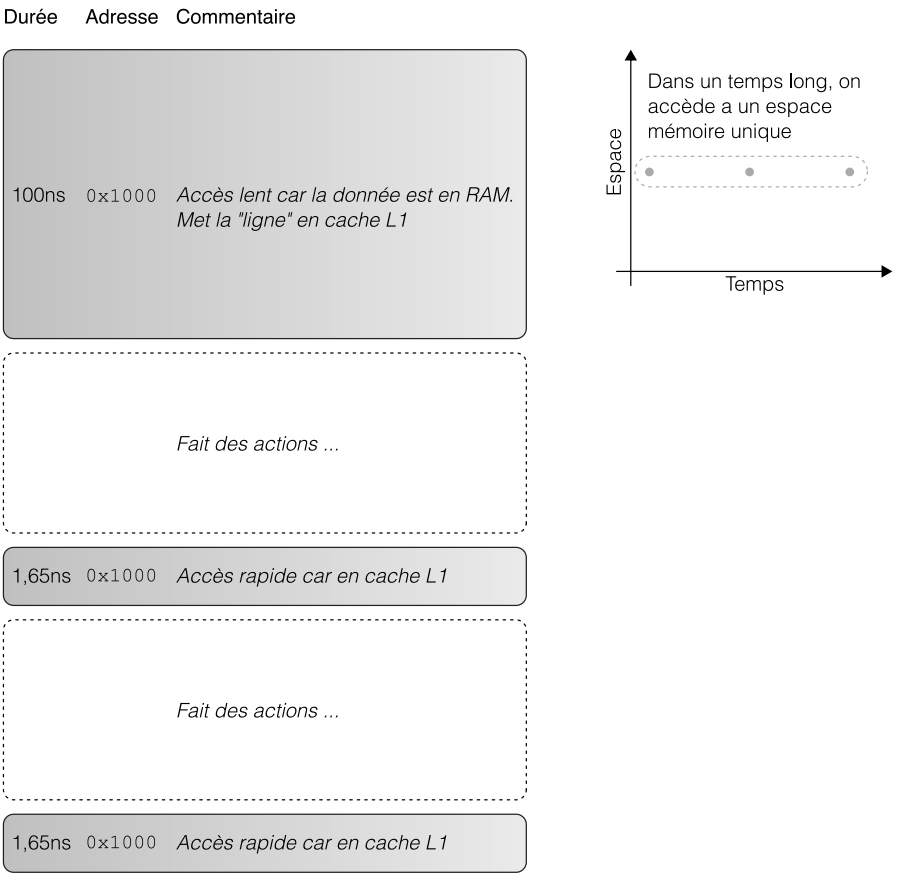
Figure 9.1 : *Temps d'accès dans la hiérarchie mémoire*



La localité temporelle repose sur le principe selon lequel les données récemment accédées ont une forte probabilité d'être réutilisées rapidement. Cette propriété justifie l'existence des caches : conserver les données fraîchement utilisées dans une mémoire

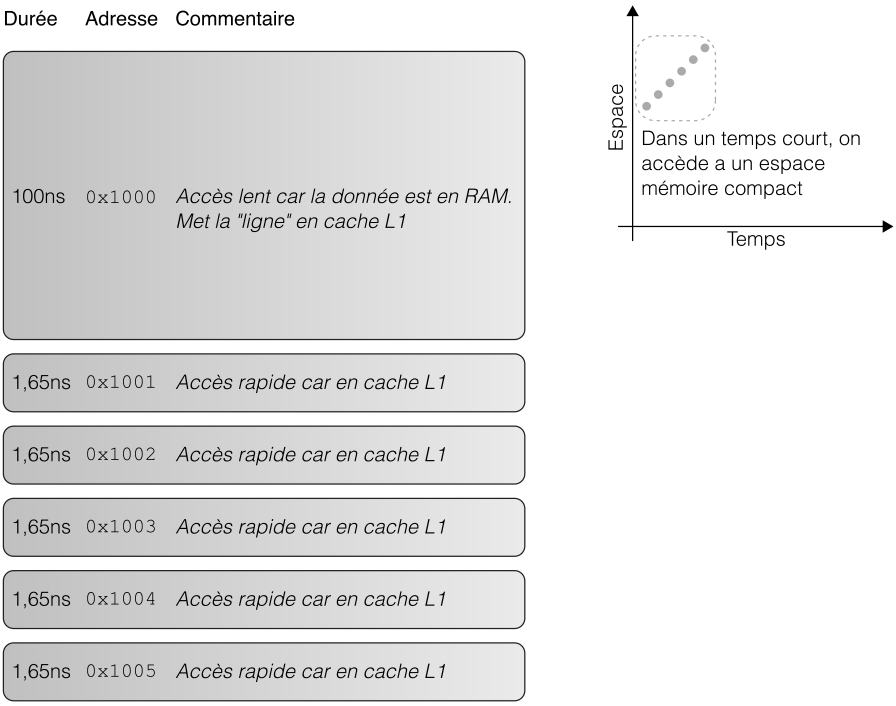
ultra-rapide améliore considérablement les performances globales. La Figure 9.2 illustre ce principe en montrant comment les accès répétés à une même zone mémoire bénéficient de la mise en cache.

Figure 9.2 : *Localité temporelle*



La localité spatiale désigne la tendance des programmes à accéder à des données stockées à des emplacements mémoire contigus. Les processeurs exploitent cette propriété en chargeant des blocs entiers plutôt que des données isolées. La Figure 9.3 démontre comment l'accès à un élément d'un tableau déclenche le chargement des éléments adjacents.

Figure 9.3 : *Localité spatiale*



Un parcours séquentiel d'un tableau d'entiers offre une excellente localité spatiale : chaque accès à un élément bénéficie du préchargement des éléments adjacents dans la ligne de cache. À l'inverse, un parcours avec des sauts importants (par exemple, lire un élément sur dix) dégrade la localité spatiale car les données intermédiaires chargées en cache ne sont pas utilisées, gaspillant la bande passante mémoire et réduisant l'efficacité du cache. L'annexe Démonstrations techniques et benchmarks > Section 11, Localité mémoire et performance des accès présente une démonstration pratique de cet impact : un parcours avec sauts peut être plus de 10 fois plus lent qu'un parcours séquentiel sur les mêmes données.

Lignes de cache et performance

Les processeurs modernes chargent la mémoire par blocs de 64 octets appelés lignes de cache. Accéder à un seul octet déclenche le chargement des 63 octets adjacents, rendant les accès suivants dans cette zone quasi gratuits.

Cette caractéristique influence directement la conception des structures de données. Une structure de 65 octets s'étend sur deux lignes de cache, doublant potentiellement le coût d'accès par rapport à une structure équivalente de 64 octets.

Coût des défauts de cache

Un défaut de cache (cache miss) peut coûter 100-300 cycles CPU contre 1-3 cycles pour un succès de cache (cache hit). Cette différence énorme justifie l'importance accordée à l'organisation des données pour minimiser les défauts de cache.

Pour mieux comprendre ces écarts, voici les ordres de grandeur des temps d'accès dans la hiérarchie mémoire moderne :

- cache L1 : 1-3 cycles (dédié à chaque cœur physique) ;
- cache L2 : 8-12 cycles (dédié à chaque cœur physique) ;
- cache L3 : 30-50 cycles (partagé entre tous les cœurs du processeur) ;
- mémoire principale (RAM) : 200-400 cycles (via contrôleur mémoire).

Ces chiffres expliquent pourquoi les processeurs investissent massivement dans des caches de plus en plus sophistiqués et pourquoi l'organisation des données devient un facteur déterminant des performances.

Les patterns d'accès déterminent l'efficacité du cache. Les accès séquentiels bénéficient du *prefetching* automatique du processeur, tandis que les accès aléatoires ou dispersés peuvent saturer la bande passante mémoire.

Représentation mémoire des types primitifs

Chaque type primitif occupe un espace mémoire fixe défini par l'architecture cible. Cette taille détermine la plage de valeurs représentables et influence directement les performances.

Sur les architectures cibles, les types `int` et `uint` s'adaptent dynamiquement : 32 bits sur les systèmes 32 bits, 64 bits sur les systèmes 64 bits. Le type `uintptr` correspond quant à lui à la taille d'un pointeur sur l'architecture cible, soit l'espace nécessaire pour représenter une adresse mémoire. Cette adaptation optimise les accès mémoire pour l'architecture native.

Contrairement aux types adaptés, les types explicitement dimensionnés (`int32`, `int64`, etc.) conservent leur taille sur toutes les architectures, garantissant la portabilité des données.

Type	Taille	Notes
`bool`	1 octet	L'octet est la plus petite unité d'adressage du processeur. Utiliser un bit nécessiterait une logique de masquage complexe. Si l'optimisation mémoire est critique, le développeur peut implémenter des champs de bits avec des `uint` et des masques.
`int8`, `uint8`, `byte`	1 octet	Plage −128 à 127 pour `int8`, 0 à 255 pour `uint8` et `byte`. `byte` est un alias d'`uint8`
`int16`, `uint16`	2 octets	Plage −32 768 à 32 767 pour `int16`, 0 à 65 535 pour `uint16`
`int32`, `uint32`, `rune`	4 octets	Plage −2 147 483 648 à 2 147 483 647 pour `int32`, 0 à 4 294 967 295 pour `uint32`. `rune` est un alias d'`uint32` pour les points de code Unicode
`int64`, `uint64`	8 octets	Plage −9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 pour `int64`, 0 à 18 446 744 073 709 551 615 pour `uint64`
`int`, `uint`	Architecture	32 bits sur systèmes 32 bits, 64 bits sur systèmes 64 bits
`uintptr`	Architecture	Taille d'un pointeur, pour stocker des adresses mémoire
`float32`	4 octets	Précision simple (IEEE 754), environ 7 chiffres décimaux
`float64`	8 octets	Précision double (IEEE 754), environ 15 chiffres décimaux
`complex64`	8 octets	Deux `float32` (partie réelle + partie imaginaire)
`complex128`	16 octets	Deux `float64` (partie réelle + partie imaginaire)

Alignement mémoire et padding

L'alignement mémoire constitue une optimisation fondamentale imposée par l'architecture des processeurs modernes pour accélérer les accès aux données. L'alignement désigne la position en mémoire où commence une donnée. Une donnée est dite alignée si son adresse mémoire est un multiple de son alignement requis. Pour les types primitifs, cet alignement correspond généralement à leur taille : un `byte` peut être placé à n'importe quelle adresse (alignement sur 1), un `int32` doit être aligné sur 4 octets, un `int64` sur 8 octets. Pour les structures, l'alignement est déterminé par le champ ayant la plus grande exigence d'alignement. La structure entière doit ensuite être alignée sur cette valeur pour garantir que tous les champs restent correctement alignés dans un tableau de structures.