Techniques de calcul haute performance sur GPU

Marc Morère – marc.morere@ipsa.fr

Quelques rappels importants

- CPU (Central Processing Unit)
  - Traitement de l'information séquentiel
  - Fonctionnel et versatile
  - Nombre de cœurs de l'ordre de la dizaine
  - Inadapté au calcul parallèle
  - Du point de vue du calcul GPU : c'est l'hôte

Quelques rappels importants

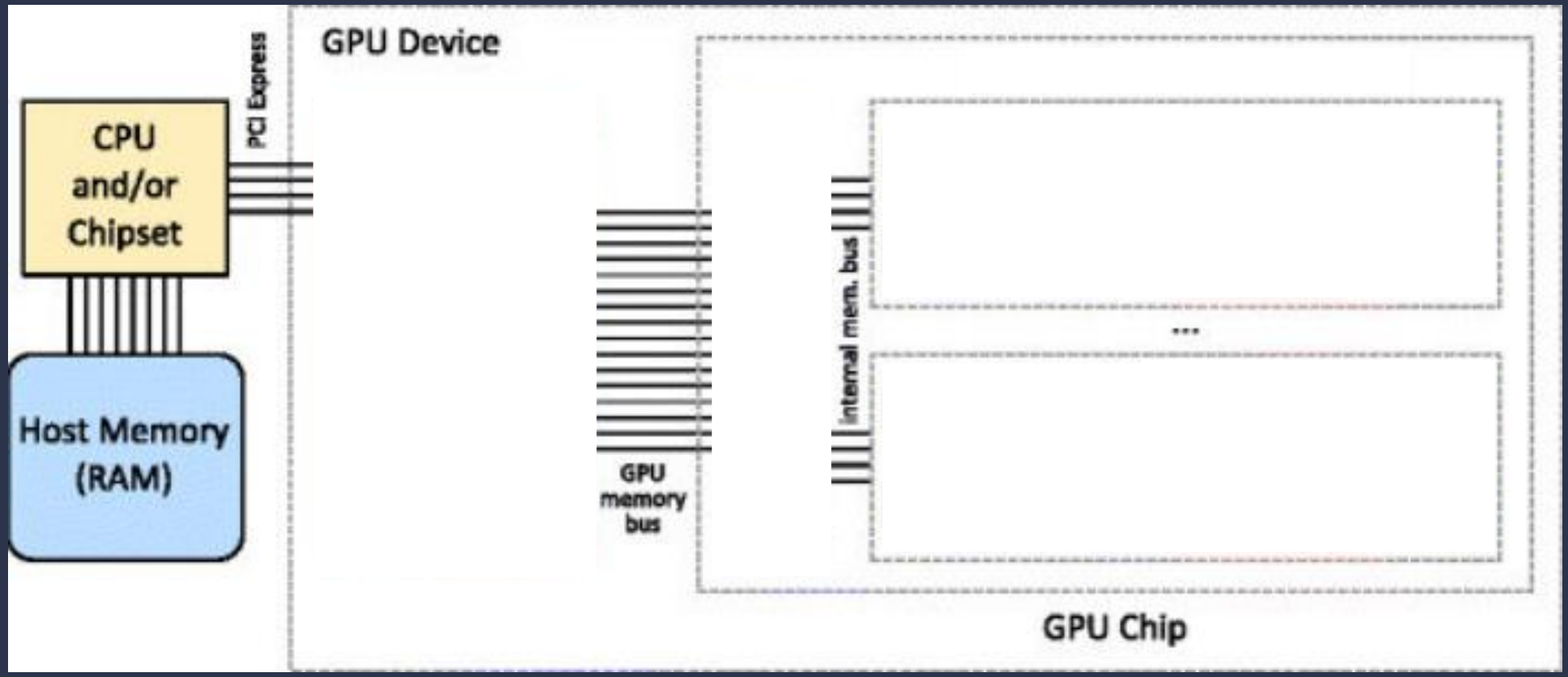
- GPU (Graphics Processing Unit)
  - Traitement de l'information parallèle
  - A l'origine créé pour le rendu graphique
  - Rapide et puissant
  - Utilise beaucoup de puissance électrique
  - Du point de vue du calcul GPU : c'est le device

Quelques rappels importants

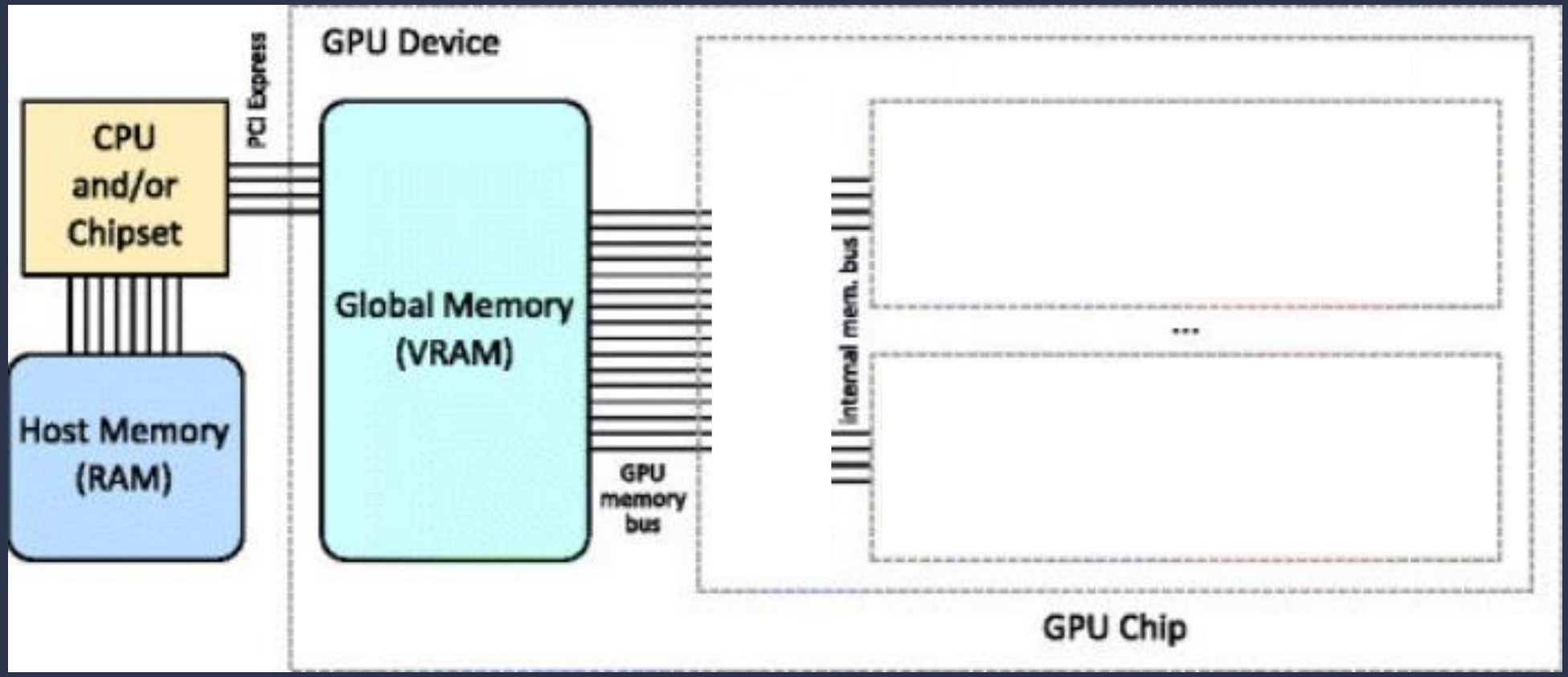
- Taxonomie de Flynn
- SISD (Single instruction on single data)
  - Aucun parallélisme
- SIMD (Single instruction on multiple data) Et pour le GPU ? SIMT
  - Parallélisme au niveau de la mémoire
- MISD (Multiple instructions on single data)
  - Donnée unique traitée par plusieurs unités de calcul en même temps. Peu utilisé en pratique
- MIMD (Multiple instructions on multiple data)
  - Plusieurs unités de calcul traitent des donnés différentes, chacune avec une mémoire distincte
  - (particulièrement utilisé sur les superordinateurs)

CPU + GPU

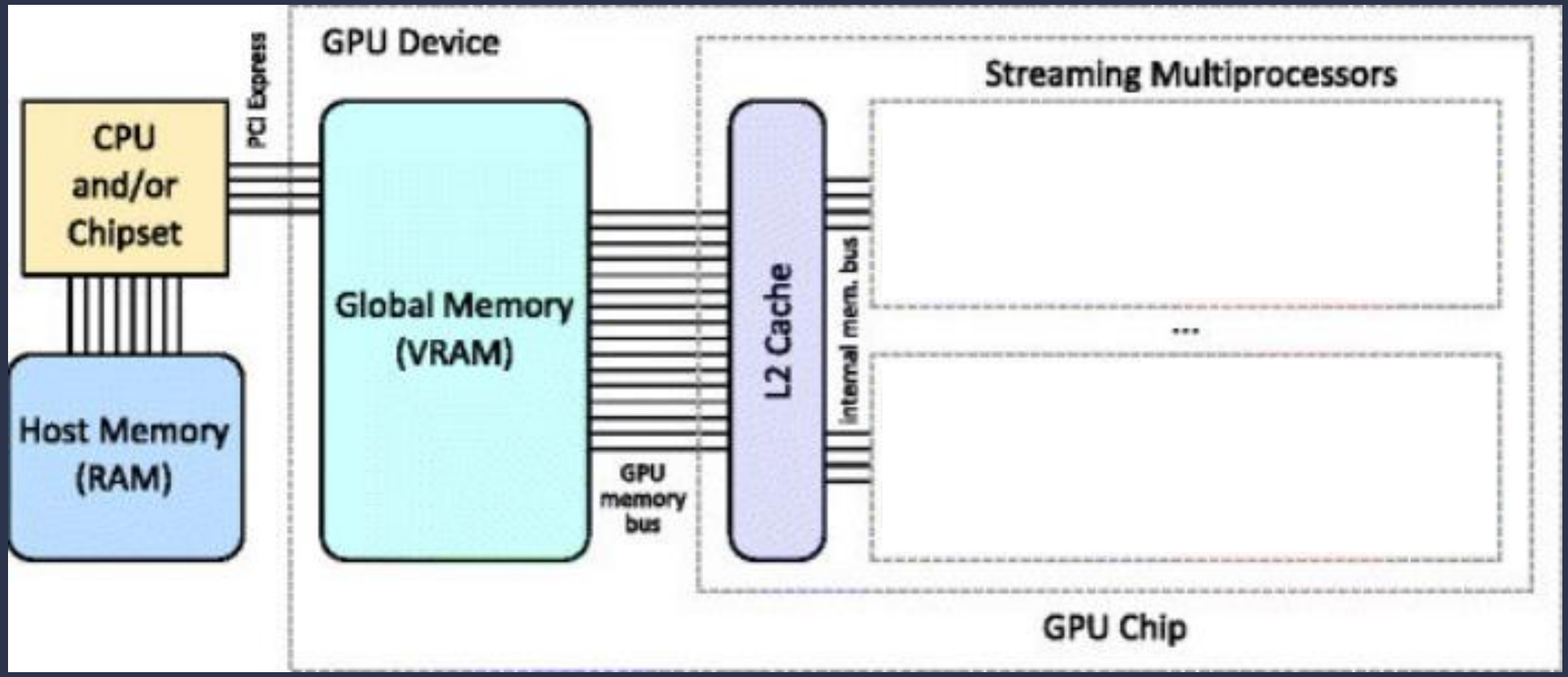
CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
GPU
memory
bus
internal mem. bus
...
GPU Chip

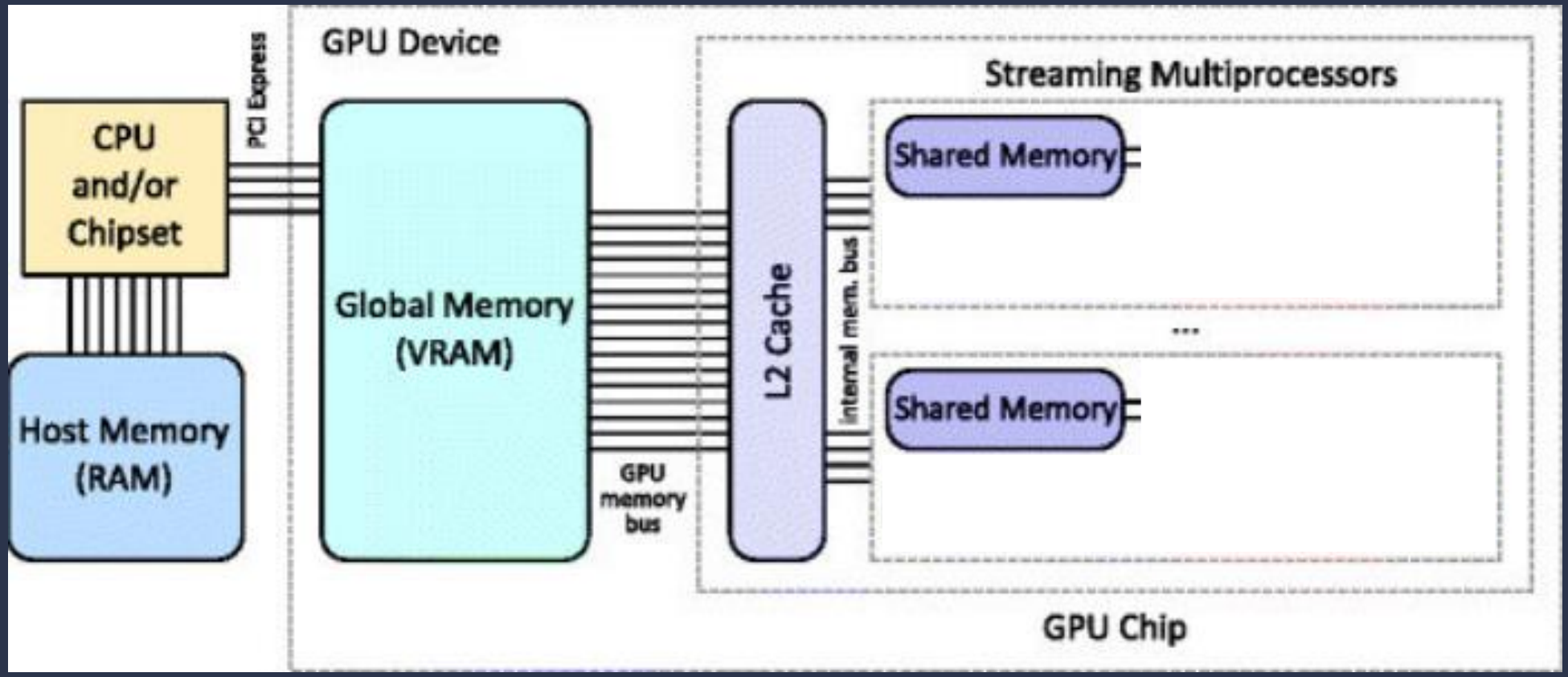
CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
Global Memory
(VRAM)
GPU
memory
bus
internal mem. bus
...
GPU Chip

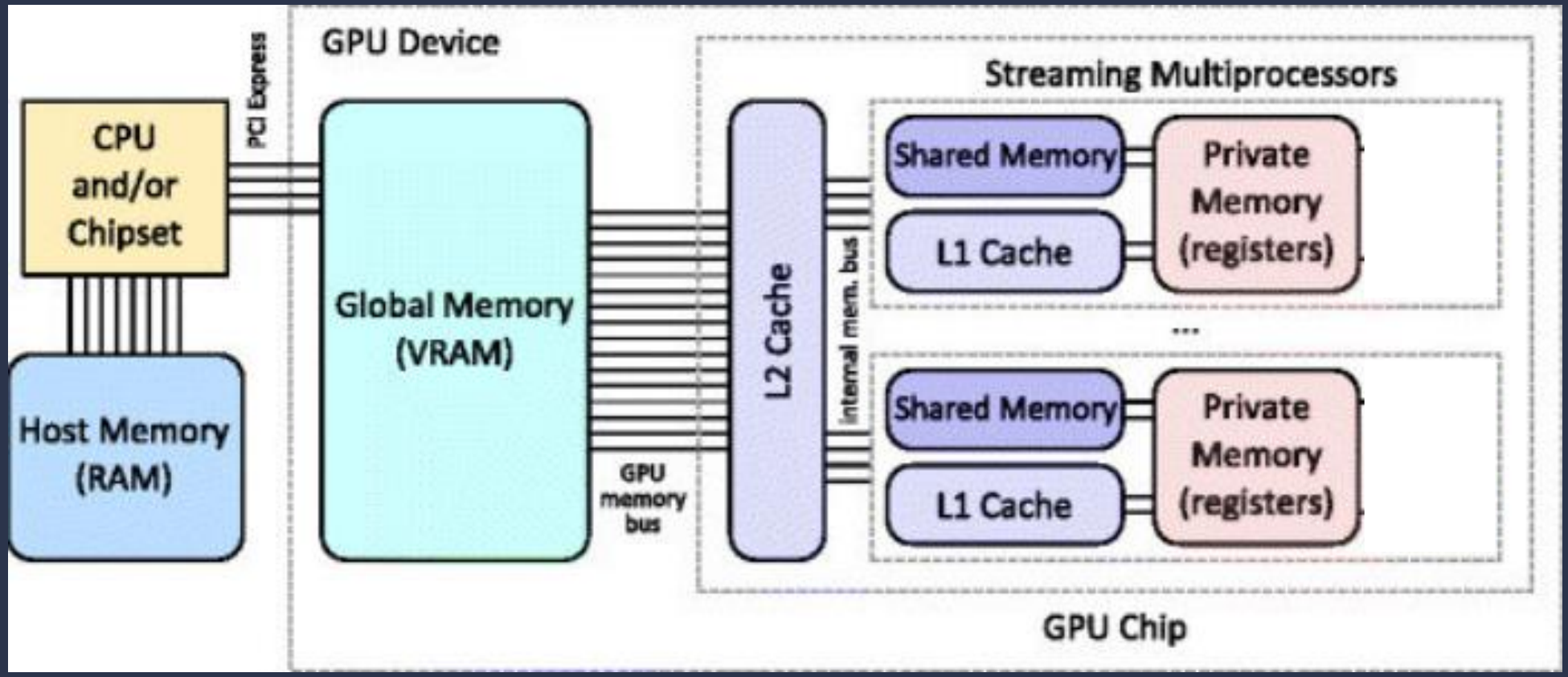
CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
Global Memory
(VRAM)
GPU
memory
bus
L2 Cache
internal mem. bus
Streaming Multiprocessors
...
GPU Chip

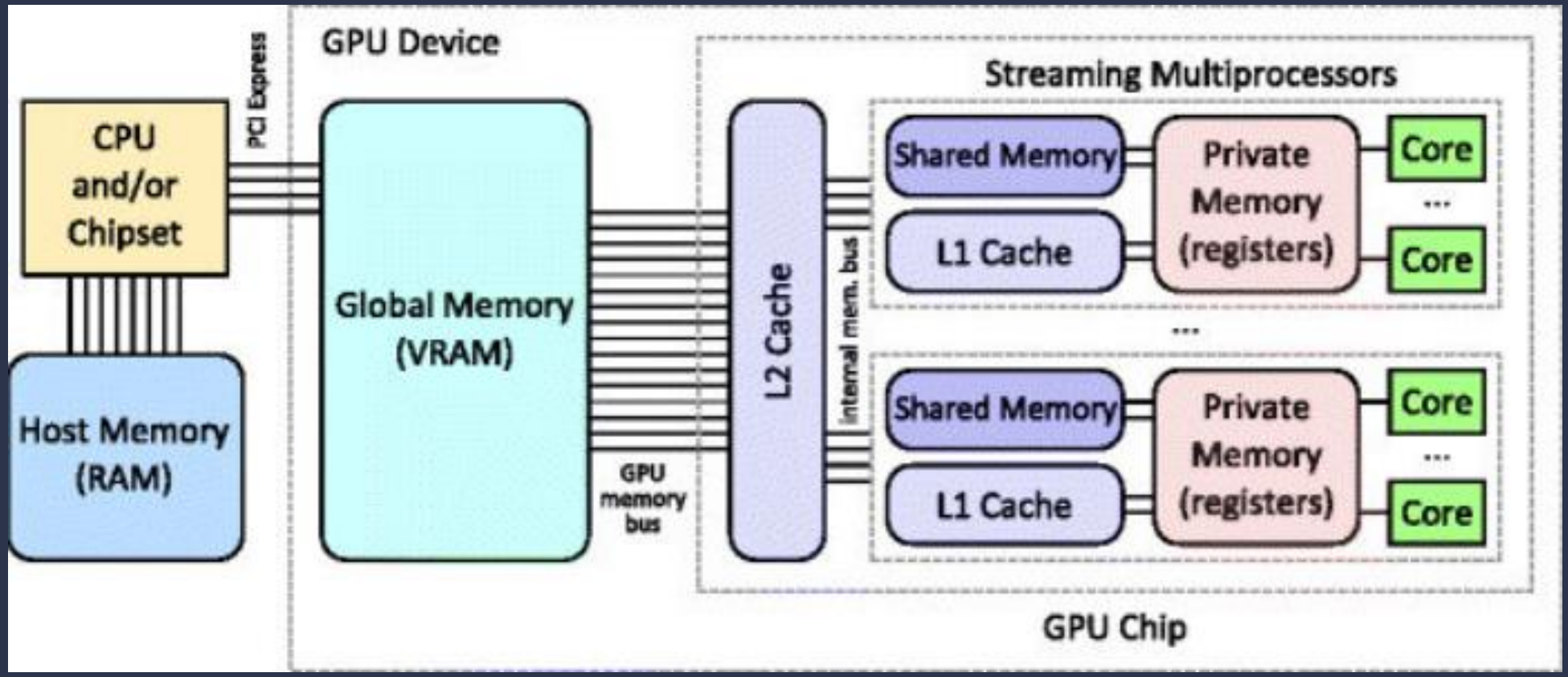
CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
Global Memory
(VRAM)
GPU
memory
bus
L2 Cache
internal mem. bus
Streaming Multiprocessors
Shared Memory
...
Shared Memory
GPU Chip

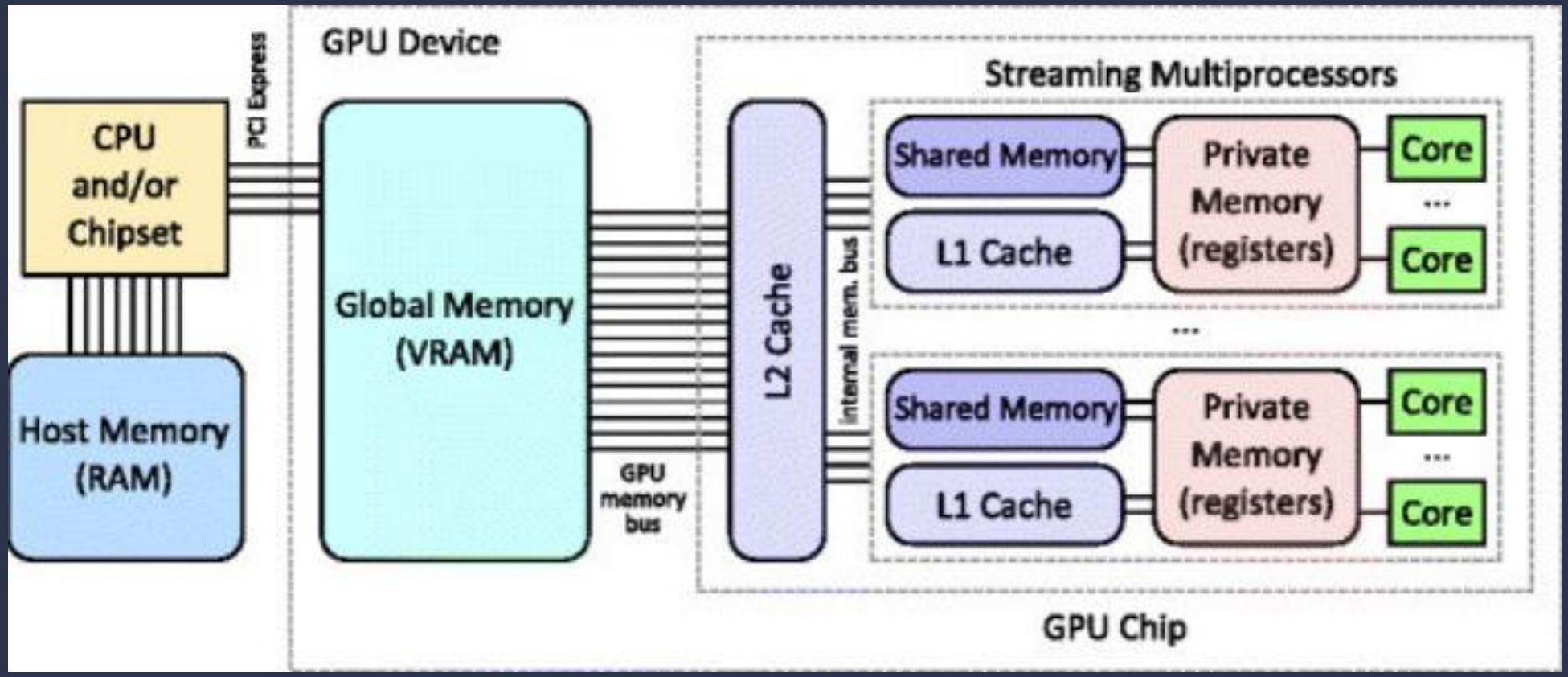
CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
Global Memory
(VRAM)
GPU
memory
bus
L2 Cache
internal mem. bus
Streaming Multiprocessors
Shared Memory
L1 Cache
Private
Memory
(registers)
...
Shared Memory
L1 Cache
Private
Memory
(registers)
GPU Chip

CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
Global Memory
(VRAM)
GPU
memory
bus
L2 Cache
internal mem. bus
Streaming Multiprocessors
Shared Memory
L1 Cache
Private
Memory
(registers)
Core
...
Core
...
Shared Memory
L1 Cache
Private
Memory
(registers)
Core
...
Core
GPU Chip

CPU + GPU


GPU Device
PCI Express
CPU
and/or
Chipset
Host Memory
(RAM)
Global Memory
(VRAM)
GPU
memory
bus
L2 Cache
internal mem. bus
Streaming Multiprocessors
Shared Memory
L1 Cache
Private
Memory
(registers)
Core
...
Core
...
Shared Memory
L1 Cache
Private
Memory
(registers)
Core
...
Core
GPU Chip

Zoom sur la mémoire partagée

- La mémoire partagée est séparée en segments, qu'on appelle banks.
- Généralement 32 banks par mémoire partagée aujourd'hui.
- Lorsque deux threads essaient d'accéder à la même bank au même moment, on parle de bank conflict.
- Si cela arrive, il y aura sérialisation.

Zoom sur les registres

- Un registre est un petit morceau de mémoire directement utilisé par le processeur.
- Mémoire la plus rapide (10x plus rapide que la mémoire partagée).
- Il y en a des dizaines de milliers dans chaque SM.

Un point sur la mémoire locale

- Son nom est trompeur : elle est cachée dans la mémoire globale... mais est directement accessible par les threads.
- Conséquence : elle est lente !
- S'y trouve tout ce qui n'est pas, ou ne rentre pas dans les registres.

Caches (L1, L2)

- Qu'est-ce qu'un cache ?

Caches (L1, L2)

- Qu'est-ce qu'un cache ?
  - Définition : Mémoire ultra-rapide entre le processeur (au sens large) et la mémoire principale stockant des données fréquemment utilisées.
  - But : Réduire les temps d'accès pour augmenter la performance en minimisant les appels à la mémoire principale.
  - Fonctionnement : Quand le GPU a besoin d'une donnée, il la cherche dans le cache avant d'aller dans le reste de la mémoire.

Cache L1

- Localisé directement dans chaque Streaming Multiprocessor (SM)
- Très rapide et de petite taille (environ 64 Ko)
- Usage : Stocke les données les plus fréquemment demandées par les threads du SM

Cache L2

- Cache global partagé entre tous les SM.
- Plus grand (par exemple 1-2 Mo) mais plus lent que le cache L1
- Usage : Stocke des données que plusieurs SM peuvent utiliser.

Utilisation des caches

- De manière générale, l'utilisateur ne choisit pas ce qu'il y a à stocker dans les caches !

Utilisation des caches

- Les données sont réunies dans les caches par cache lines (128 o)
- L'utilisateur a l'accès sur la gestion des accès mémoire, qu'on appelle memory transaction
- On parle de Memory Coalescing quand on regroupe les demandes mémoire ensemble. C'est quelque chose de difficile à optimiser.

Réunion de threads : le warp

- Le warp est un groupe de 32 threads.
- Tous les threads du warp exécuteront simultanément la même instruction (SIMT).

Réunion de threads : le warp

- Le warp est un groupe de 32 threads.
- Tous les threads du warp exécuteront simultanément la même instruction (SIMT).
- Quels intérêts ?

Réunion de threads : le warp

- Le warp est un groupe de 32 threads.
- Tous les threads du warp exécuteront simultanément la même instruction (SIMT).
- Quels intérêts ?
  - Réduction de la latence grâce au regroupement des threads
  - Optimisation des ressources du GPU (coalescence mémoire)

Opérations atomiques

- Opérations atomiques : Opérations sur une variable partagée qui sont exécutées de manière indivisible.
- Aucune interruption possible : Elles se réalisent en une seule étape, empêchant les autres threads d'accéder à la variable pendant l'opération

Opérations atomiques

- Évite les conflits d'accès aux données partagées
- Assure la cohérence des données dans les calculs parallèles, particulièrement pour les variables modifiées par plusieurs threads
- Limites :

Opérations atomiques

- Évite les conflits d'accès aux données partagées
- Assure la cohérence des données dans les calculs parallèles, particulièrement pour les variables modifiées par plusieurs threads
- Limites :
  - Ralentissement potentiel : Peut réduire les performances si trop d'opérations atomiques sont utilisées
  - Pas d'accès groupé : Contrairement à la coalescence mémoire, chaque opération atomique agit isolément, ce qui peut limiter l'efficacité

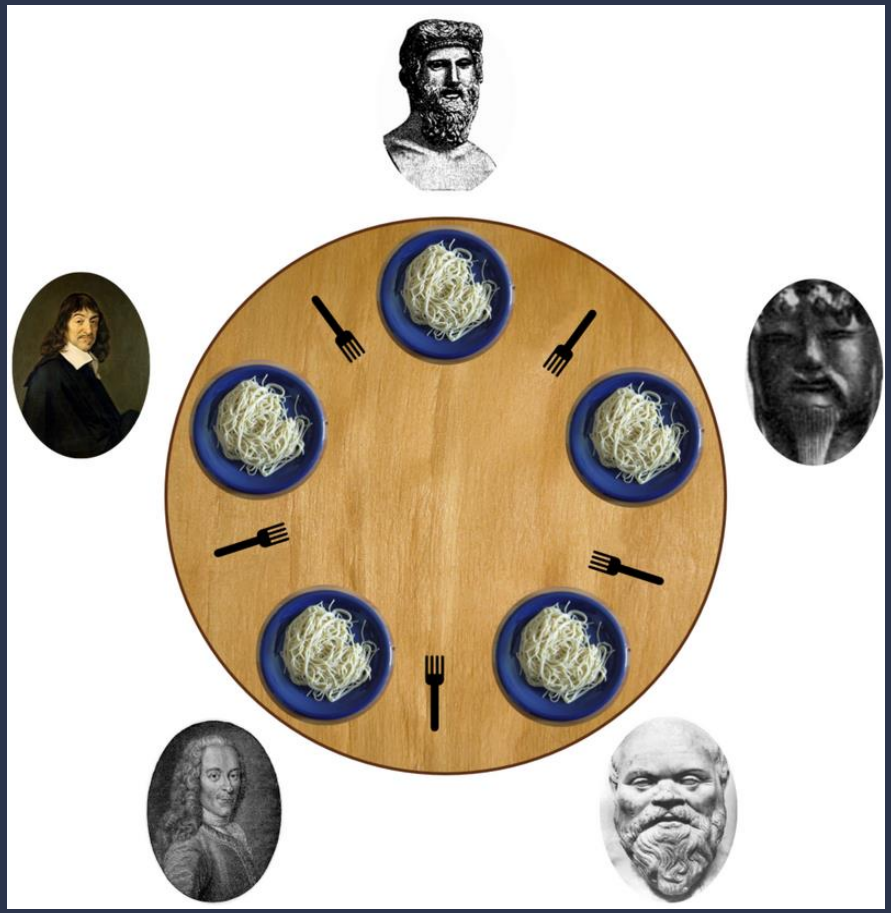
Synchronisation

Synchronisation

- En 1965, Edsger Dijkstra propose un problème de synchronisation simple : le dîner des philosophes.

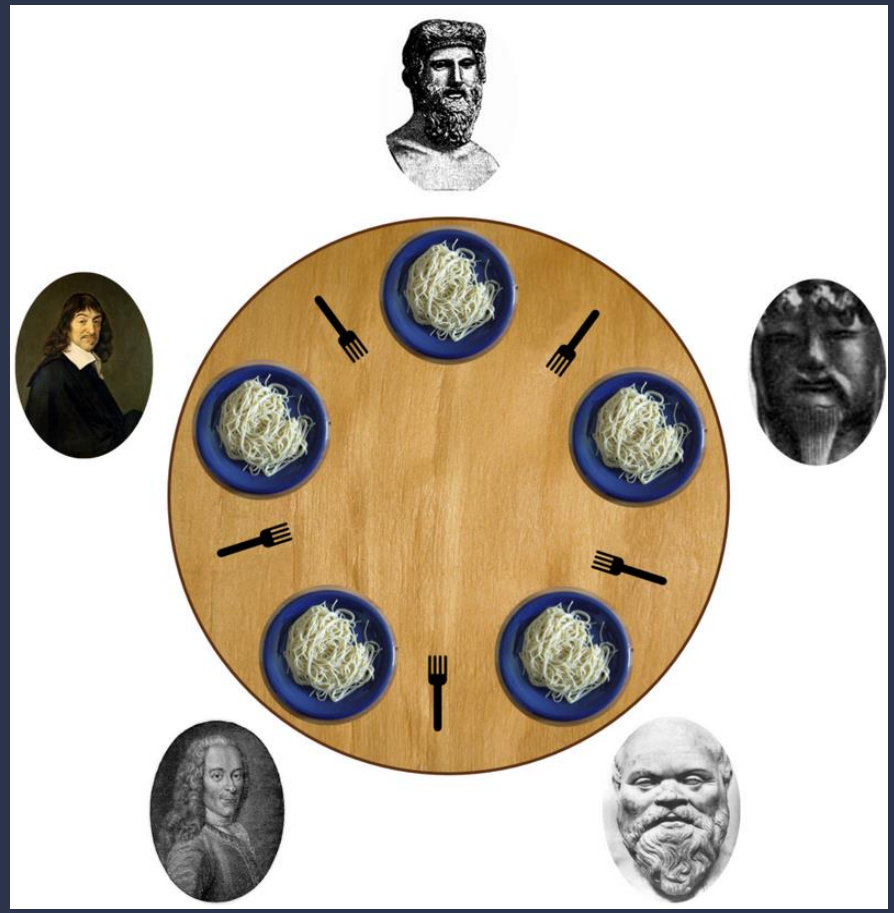
Synchronisation

- En 1965, Edsger Dijkstra propose un problème de synchronisation simple : le dîner des philosophes.
- 1 règle : un philosophe a besoin de deux fourchettes pour manger des spaghetti.



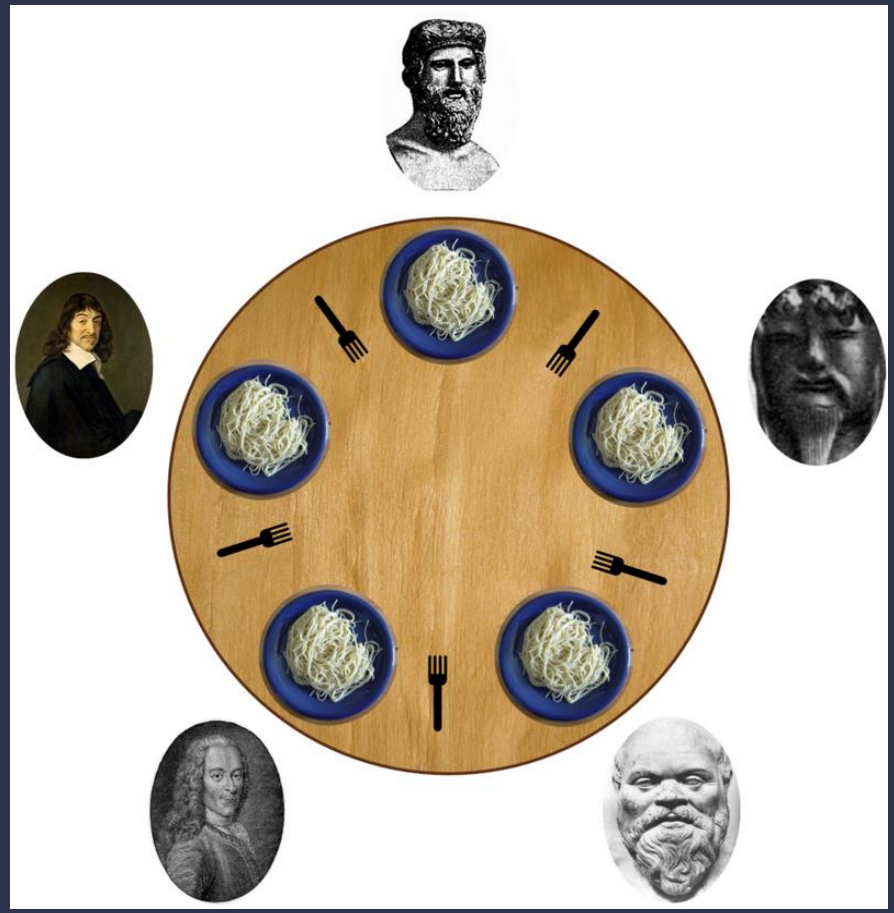
Synchronisation

- 3 états possibles par philosophe :
  - Penseur : le philosophe philosophe, pour une durée indéterminée.
  - Mangeur : le philosophe, pour une durée déterminée, utilise les deux fourchettes à sa gauche et à sa droite pour manger.
  - Affamé : le philosophe cherche à tout prix à attraper les fourchettes.



Synchronisation

- Il existe une solution parfaite pour un nombre pair de philosophes. Laquelle ?
- De manière générale, le problème peut rapidement finir en deadlock.
- Deadlock : le système est bloqué de manière stable, chaque acteur en attendant un autre.
- Que serait une situation de deadlock ici ?



Anti exemple

Pseudo code (mène au deadlock)

```
repeat forever:
  think()
  lock(left_fork)
  lock(right_fork)
  eat()
  unlock(right_fork)
  unlock(left_fork)
```

Propriétés recherchées

- **Sécurité** (safety) : jamais deux voisins ne mangent en même temps (mutual exclusion sur chaque fourchette).
- **Absence d'interblocage** (deadlock freedom) : il n'existe pas d'exécution où tout le monde attend pour toujours.
- **Absence de famine** (starvation freedom) : tout philosophe qui a faim finit par manger.
- **Equité** (fairness) : pas d'injustice systématique (ex. alternance raisonnable, FIFO, etc.).

Les 4 conditions de Coffman pour un deadlock (rappel théorique)

1. **Exclusion mutuelle** : chaque fourchette ne peut être détenue que par un philosophe à la fois.
2. **Possession et attente** (hold-and-wait) : un philosophe peut garder une fourchette en attendant l'autre.
3. **Non-préemption** : une fourchette ne peut pas être retirée de force.
4. **Attente circulaire** : il existe un cycle d'attente circulaire (chaque philosophe attend la fourchette d'un voisin).

Idée des solutions : casser au moins une de ces conditions.

Mutex (Mutual Exclusion)

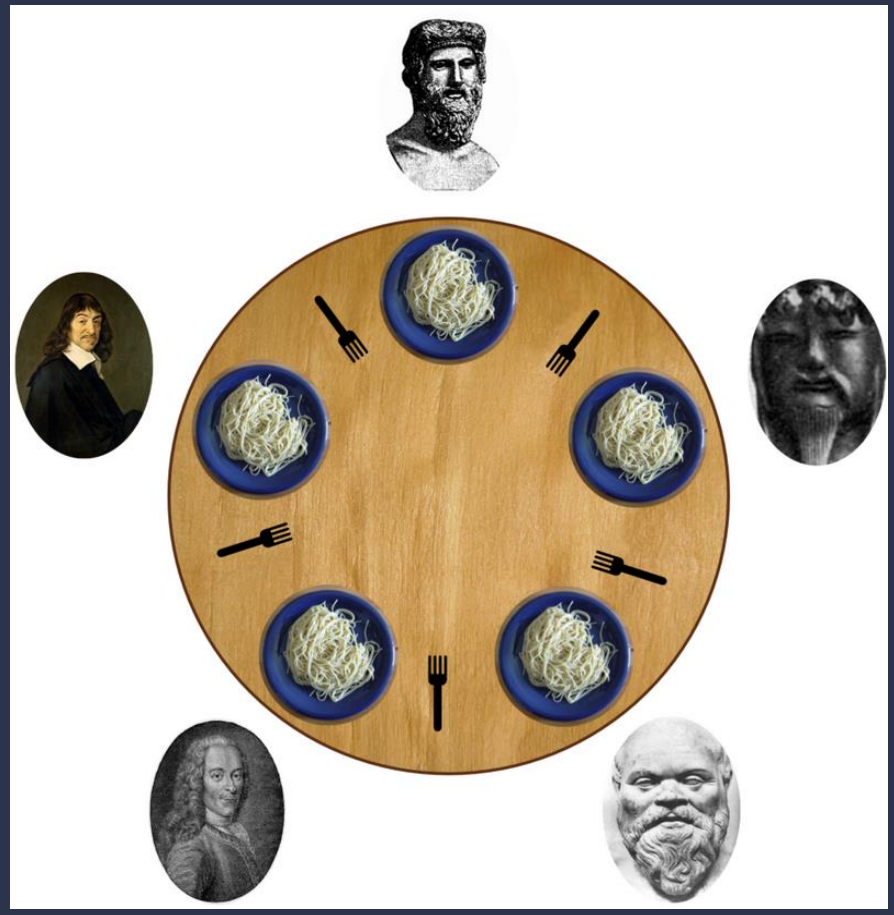
- Verrou binaire permettant de restreindre l'accès à une ressource partagée.
- Garantit qu'un seul thread accède à une ressource donnée à la fois.
- Deux états : lock, unlock.

Sémaphore

- Mécanisme de synchronisation basé sur un compteur pour gérer plusieurs accès à une ressource partagée
- Utilisé pour limiter le nombre de threads pouvant accéder à la ressource simultanément
- Deux actions : UP (libère de la disponibilité), DOWN (diminue la disponibilité).
- Le compteur peut être bloqué à une valeur minimale (0) et maximale.

Comment résoudre le problème des philosophes ?

- Un sémaphore de table (limité à 4)
- Un mutex par fourchette



Solution 1

Cassons l'attente circulaire → Hiérarchie de ressources

- **Idée** : imposez un **ordre total** sur les fourchettes (indice 0 < 1 < … < N-1). Chaque philosophe prend **toujours d'abord la fourchette de plus petit indice**, puis la plus grande.
- **Pourquoi ça marche (intuition)** : s'il y avait un cycle d'attente, les indices devraient **strictement augmenter** le long du cycle… ce qui est impossible puisque l'on revient au point de départ. → Pas d'attente circulaire ⇒ pas de deadlock.
- **Limite** : peut laisser de la **famine** si l'ordonnancement des fils/luttes est malchanceux (pas de FIFO implicite).

Solution 1

```
let a = min(left_fork_id, right_fork_id)
let b = max(left_fork_id, right_fork_id)
repeat forever:
  think()
  lock(fork[a])
  lock(fork[b])
  eat()
  unlock(fork[b])
  unlock(fork[a])
```

Solution 2

Cassons possession+attente ou limitons la contention → Arbitre/Garçon (waiter)

- **Idée** : un sémaphore (ou serveur central) limite à **N−1** le nombre de philosophes autorisés à tenter de s'asseoir/prendre des fourchettes simultanément.
- **Pourquoi ça marche (intuition)** : pour que le deadlock « chacun tient une fourchette et attend l'autre » survienne, il faudrait **N** philosophes engagés. En limitant à **N−1**, on force l'existence d'au moins un « trou » qui permet à quelqu'un d'obtenir ses deux fourchettes, de manger puis de libérer.
- **Limite** : selon l'implémentation, risque de **famine** sans politique d'équité (ajouter FIFO/queue).

Solution 2 (sémaphores)

```
semaphore room = N-1
semaphore fork[0..N-1] = {1,...,1}
philosopher i:

repeat forever:
    think()
    wait(room)              // demander la permission d'entrer dans la "zone critique étendue"
    wait(fork[left(i)])
    wait(fork[right(i)])
    eat()
    signal(fork[right(i)])
    signal(fork[left(i)])
    signal(room)
```

Solution 3

Garantir équité et absence de famine → Moniteur avec conditions

- **Idée** : gérer l'état de chaque philosophe (**thinking, hungry, eating**) dans un **moniteur**.

On n'autorise un philosophe à manger que si ses voisins ne mangent pas ; sinon, on le met en attente sur une **condition** dédiée. On réveille en priorité les voisins affamés quand on libère ses fourchettes.

Atout : avec une politique de réveil **FIFO** sur chaque self[i], on obtient **pas de famine**.

Solution 3 (style Tanenbaum)

```
monitor Dining:
  state[0..N-1] in {THINKING, HUNGRY, EATING}
  cond self[0..N-1]

 procedure test(i):
    if state[i] == HUNGRY and state[left(i)] != EATING and state[right(i)] != EATING:
      state[i] = EATING
      signal(self[i])

procedure pickup(i):
    state[i] = HUNGRY
    test(i)
    while state[i] != EATING:
      wait(self[i])

  procedure putdown(i):
    state[i] = THINKING
    test(left(i))
    test(right(i))
```

Solution 4

Solution distribuée sans arbitre : Chandy–Misra (1984)

- Chaque arête (voisinage) possède une **fourchette logique** détenue par **un** des deux voisins.
- Les fourchettes ont un état **propre/sale**. Après avoir mangé, une fourchette devient **sale**.
- Un philosophe **demande** la fourchette à son voisin lorsqu'il a faim ; un voisin **doit** céder une fourchette **sale** s'il n'en a pas besoin immédiatement.
- Les priorités implicites (via « propreté ») assurent que les demandes **finissent par être satisfaites → pas de deadlock, pas de famine**.

Solution 4

```
on request(fork(i,j)) from j:
  if not eating and (fork is dirty or I don't need it):
    send fork to j; mark fork clean

when hungry:
  request any missing forks
  wait until both forks held
  eat; mark both forks dirty
  reply to pending requests if any
```

Mise en pratique (test des machines)

- Installation de Visual Studio (2017 ou supérieur)

- Installation de Vulkan depuis LunarG

- Installation de GLFW (GL Frame Work : multi-platform library for OpenGL)

- Installation de GLM (OpenGL Mathematics : C++ library)