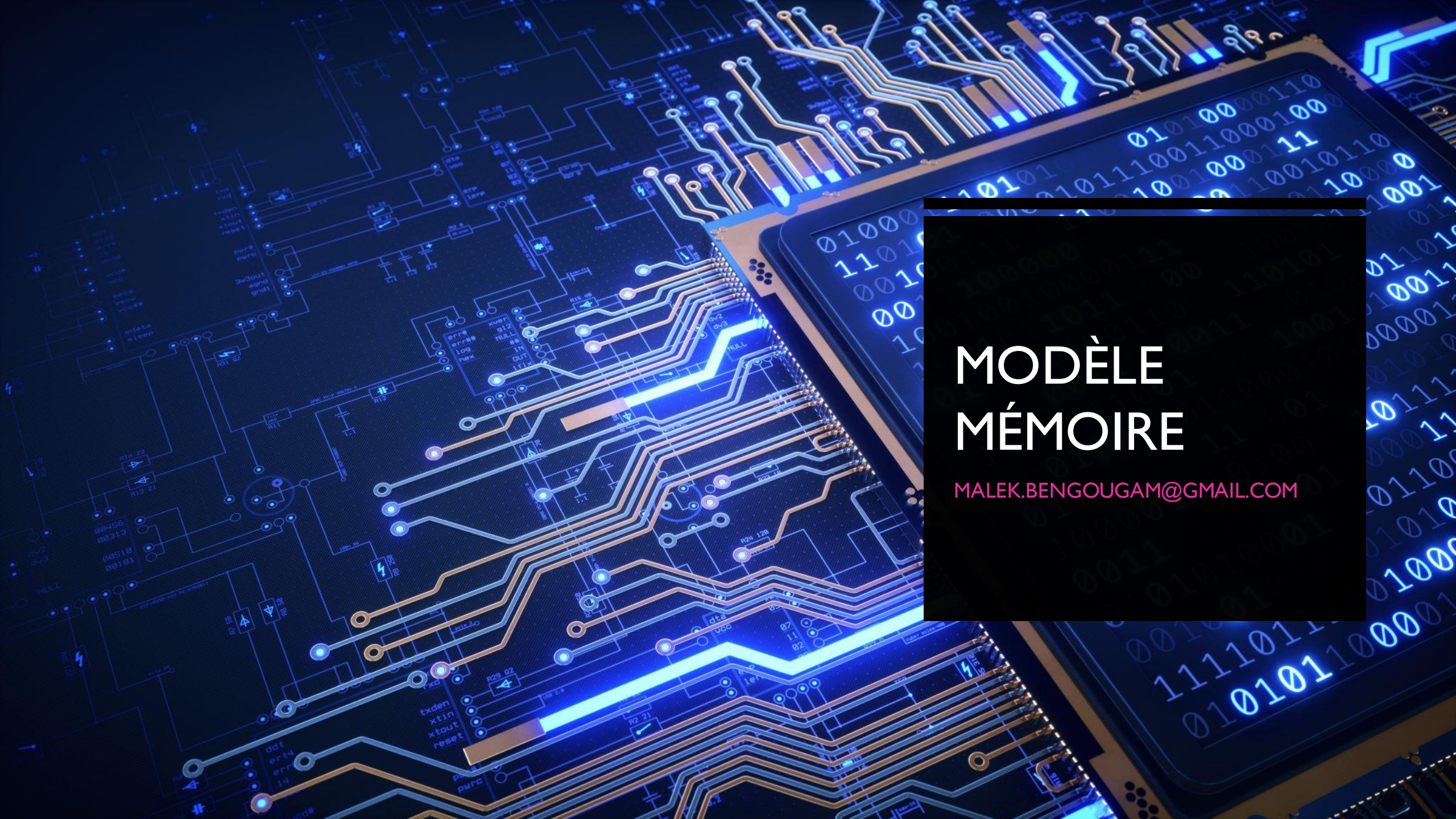

MODÈLE MÉMOIRE

MALEK.BENGOUGAM@GMAIL.COM

MODELE MÉMOIRE (MEMORY MODEL)

- Souvent occulté par les non-initiés mais fondamental dans un système multi processeur (SMP) ou multi thread.
- Objectif : laisser le maximum de latitude au compilateur afin de pouvoir optimiser le programme
  - Tirer parti de l'architecture de la machine
- On retrouve un modèle mémoire bien défini dans les langages modernes prenant en compte la concurrence
  - Java, C11 / C++11 et au-delà, Rust, Go, Haskell, …
- Un modèle mémoire n'est pas obligatoire mais contribue à définir des bonnes lignes de conduites implicites

POURQUOI UN MODÈLE ?

- Les optimisations effectuées par le compilateur peuvent influencer l'**ordre** de lecture et d'écriture des variables partagées et provoquer des "race condition" (compétitions ou course critiques).
  - En pratique le compilateur est libre de réordonnancer l'ordre des opérations sur les variables à la compilation.
  - Le but étant de saturer au maximum le pipeline du CPU et permettre un parallélisme d'instruction.

- Une solution classique : les mutexs mais risque de contention
  - Nécessite une implication plus importante du programmeur afin de s'assurer que les verrous se déverrouillent dans tous les cas.

- Un modèle mémoire spécifie quels cas de figure sont implicitement géré par une synchronisation (appelée "barrier" ou "fence" en anglais, donc une idée de cloture –mais non fermée).
  - On n'est pas encore dans le tout automatique, le compilateur a tout de même besoin d'informations.

ORDONNANCEMENT MÉMOIRE (MEMORY ORDER)

- La complexité d'un modèle mémoire dépend en grande partie de l'architecture CPU.
- il s'agit de la façon dont un CPU accède à la mémoire et de sa capacité à réordonnancer les opérations en mémoire afin de maximiser la bande passante (bandwidth) notamment vis-à-vis des caches et banques mémoires.
  - On a déjà parlé précédemment des réordonnancement effectué par le compilateur, là il s'agit de l'exécution par le CPU
- On dit alors qu'un processeur est 'in-order' lorsqu'il l'exécution est purement séquentielle, 'out-of-order' sinon.
- Pour faire simple, il existe 3 grandes consistances de modèles mémoires :
  - Séquentiel, toutes les lectures et écritures se font dans l'ordre
  - Assoupli (relaxed), plusieurs variantes (loads after loads, loads after stores, stores after stores, stores after loads [x86-64])
  - Faible (weak), totalement arbitraire, seules les barrières mémoires (memory barrier) peuvent forcer un ordre

QUELLES VALEURS SONT POSSIBLES POUR X ET Y (EN X86-64) ?

CPU0

- Initialement A et B valent tous deux 0

- A= 42
- X = B

CPU1

- Initialement A et B valent tous deux 0

- B = 24
- Y = A

ON PEUT OBTENIR X = Y = 0 ! POURQUOI ?

- Une lecture peut arriver avant qu'une écriture dans un autre espace mémoire (cache d'un autre CPU) ait été effective en mémoire (RAM).

- Effectivement cela peut se résoudre en définissant les variables X et Y comme "**atomiques**", ce qui force l'écriture en RAM en insérant des barrières mémoires (memory fence) et implique une **cohérence** entre le cache et la ram.

- C'est le seul vrai cas problématique sur une architecture x86-64 (IA32 ou AMD64, mais pas IA64 (itanium) !).
  - Il en va autrement sur les autres architectures telles ARMv7/ARM64 (la plupart des mobiles), PowerPC etc...

ATTENTION À "VOLATILE" EN C / C++

- Il s'agit d'un mot clé du C/C++ initialement prévu pour gérer les cas d'accès direct à un processeur / contrôleur externe comme si il était adressable en mémoire système (on parle alors de "memory mapping").
- En pratique 'volatile' indique au compilateur d'éviter les optimisations et réordonnancements pour les variables concernées.
- Il n'y a cependant aucune garantie de cohérence de cache ici …
- … sauf que Visual Studio et d'autres compilateurs sur les architectures x86-64 ont implicitement glissé des fences
- Imaginez un programmeur qui porte son code multi thread sur une architecture complexe PowerPC (PS3 / Xbox360) sans ces barrières mémoires !

VARIABLES ATOMIQUES ET RÉORDONNANCEMENT

- Dans le cas non séquentiel et dans un cadre multiprocesseurs/coeurs, les variables atomiques ont les propriétés suivantes:
- Acquire - s'applique aux instructions lisant en mémoire (lecture seule ou lecture-écriture), se dit read-acquire
  - Empêche le réordonnancement de toutes les instructions qui lisent ou écrivent en mémoire qui **suivent** l'instruction
- Release – s'applique aux instructions écrivant en mémoire (écriture seule ou lecture-écriture), se dit write-release
  - Empêche le réordonnancement de toutes les instructions qui lisent ou écrivent en mémoire qui **précèdent** l'instruction

LES VARIABLES ATOMIQUES EN C++11

- `#include <atomic>`
- `atomic<int> X(0);`

- On peut alors utiliser X comme une variable standard, operator=, operator++ etc...
- mais pour plus de contrôle on a les instructions load() et store() qui prennent une des valeurs suivantes:

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst   // Par défaut "sequentially consistent ordering", le plus coûteux aussi
};
```

...TAKE IT EASY.

- Memory_order_relaxed offre les avantages de l'atomicité (load après store d'une variable spécifique) tout en permettant un réordonnancement des autres accès mémoire.

```
#include <atomic>

std::atomic<uint64_t> valeur(0);

void storeValeur()
{
    valeur.store(0x42424242, std::memory_order_relaxed);
}

uint64_t loadValeur()
{
    return valeur.load(std::memory_order_relaxed);
}
```

EXEMPLE ORIGINEL REVU ET CORRIGÉ EN C++11

X = 0 est maintenant impossible

- `atomic<int> A(0);`
- `A.store(42);`
- `int X = B.load();`

Y = 0 est maintenant impossible

- `atomic<int> B(0);`
- `B.store(24);`
- `int Y = A.load();`

LES BARRIÈRES EN C++11

On émule une variable atomique avec 2 memory fences

- `atomic_thread_fence(memory_order_acquire); // avant les lectures`
- `A = 42; // non atomique, sequentiel`
- `Y = B;  // non atomique, sequentiel`
- `atomic_thread_fence(memory_order_release); // après les écritures, peut être suffisant dans certains cas`

- `atomic_thread_fence(memory_order_acquire); // avant les lectures`
- `B = 24; // non atomique, sequentiel`
- `X = A;  // non atomique, sequentiel`
- `atomic_thread_fence(memory_order_release); // après les écritures, peut être suffisant dans certains cas`

UNE VARIABLE INTERMEDIAIRE ATOMIQUE COMME SIGNAL

signal.store() garanti l'ordre des écritures qui précèdent. Force une synchronisation "happens-before"

- `atomic<int> signalA(0);`
- `A = 42; // non atomique`
- `signalA.store(1, memory_order_release); // signal quand A est modifiée`
- `int g = signalB.load(memory_order_acquire); // TRY, pas garantie`
- `if (g != 0)`
- `    X = B;`
- `// loop sur signalB si besoin`

- `atomic<int> signalB(0);`
- `B = 24; // non atomique`
- `signalB.store(1, memory_order_release); // signal quand B est modifiée`
- `int g = signalA.load(memory_order_acquire); // TRY, pas garantie`
- `if (g != 0)`
- `    Y = A;`
- `// loop sur signalA si besoin`

RÉFÉRENCES

- https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973
- https://www.khronos.org/blog/vulkan-has-just-become-the-worlds-first-graphics-api-with-a-formal-memory-model.-so-what-is-a-memory-model-and-why-should-i-care
- https://fr.wikipedia.org/wiki/X86
- https://en.wikipedia.org/wiki/Memory_model_(programming)
- https://en.wikipedia.org/wiki/Memory_ordering
- https://www.cl.cam.ac.uk/~pes20/weakmemory/
- https://en.cppreference.com/w/cpp/atomic/memory_order
- https://en.wikipedia.org/wiki/Memory_barrier
- https://www.kernel.org/doc/html/latest/process/volatile-considered-harmful.html
- https://preshing.com/20120930/weak-vs-strong-memory-models/

OBJECTIFS PREMIERES SEANCES

- Emetteur de particules multi-threadé (via un thread pool)
- Les particules rebondissent sur un plan
  - simulation simple type Euler ou autre, avec une gestion de force et accélération (gravité...)
  - Voir https://www.richardlord.net/blog/presentations/physics-for-flash-games.html pour des informations plus générales
- Rendu simple en OpenGL 3.x
  - Utilisation du Geometry Shader pour le rendu du sol et particules