

Programmation Système Répartie et Concurrente

Master 1 Informatique – MU4IN400

Cours 10 : Concurrence Avancée

Promise, Future, Async

Introduction à OpenMP, CUDA

Yann Thierry-Mieg

Yann.Thierry-Mieg@lip6.fr

Plan

- On a vu au cours précédent :
 - Protobuf, un format et un outillage pour la sérialisation
 - Aujourd'hui : Concurrency le retour
 - Retour sur les outils thread : mutex, condition, atomic
 - Promise, Future
 - Async, packaged task
 - Références :
 - « Design Patterns », le GOF Gamma, Helm, Vlissides, Johnson
 - « C++ concurrency in Action » Anthony Williams
 - Slides assemblées de plusieurs sources, citées dans les slides concernées
- En particulier Akim Demaille (EPITA) (pas mal de biblio citée)
- https://www.lrde.epita.fr/~theo/lectures/CXXA/cxxa_5.pdf
- https://www.lrde.epita.fr/~theo/lectures/CXXA/cxxa_6.pdf
- « C++ concurrency in Action » Anthony Williams
 - Slides assemblées de plusieurs sources, citées dans les slides concernées dont
 - Edwin Carlinet (EPITA), Ruud Van Der Pas (Sun), Cyril Zeller (NVidia)

Concurrence vs Parallélisme

- Concurrence

- Un Modèle de programmation
 - Différents flots de contrôle
- Vise la clarté
 - Pas nécessairement l'efficacité
- Certains langages entièrement basés sur la concurrence
 - GO, UrbiScript

- Parallélisme

- Exécution simultanée de calculs
 - Mécanisme physique
- Hyperthread, Multicore, GPU...

Mémoire : Thread vs Processus

- Processus
 - Chacun son espace d'adressage
 - Communication uniquement via des canaux dédiés (IPC)
 - Signaux
 - Tubes
 - Memory Map
 - Sockets
- Threads
 - Partagent l'espace d'adressage
 - Lectures et écritures concurrentes possibles
 - Les variables de la pile (appels) sont locales au thread (en général)
 - Accès concurrents
 - Attention aux collisions,
 - mutex, atomic..

Thread vs Taches Concurrentes

- Multithreading
 - Création explicite de thread
 - Création de thread, affectation du travail, join
 - Latence due à la création/destruction des thread
 - Problème de passage à l'échelle, trop de thread tue la mule
- Multitask :
 - Partitionner le travail en morceaux parallélisables
 - Notion de tâche à réaliser
 - Parallélisable = concurrent, pas *nécessairement* parallèle
 - Laisse le runtime décider de l'allocation aux threads disponibles
 - Connaissance de la machine (compilateur, librairie)
 - Passe mieux à l'échelle (massivement multicore)
 - Parallélisme à grain plus fin => plus de parallélisme potentiel

Communiquer entre Thread

- Mémoire partagée
 - Les threads accèdent en concurrence aux données partagées
 - Nécessité de synchronisations
 - Atomic
 - Mutex
 - Condition variable
- Passage de messages
 - En principe pas de partage mémoire
 - Files de messages, channels, boîte aux lettres...
 - Passe bien à l'échelle
 - Sur multi processus
 - Sur infrastructure distribuée
 - Mais plus lent que le partage mémoire direct
 - Copies, ou risque de réintroduction de data race.

Data Race

- Conflit d'accès
 - Deux (ou plus) threads accèdent en même temps au même emplacement mémoire
 - Au moins un d'entre eux est un écrivain
 - Résultats non définis
- Data Race
 - Un conflit d'accès non protégé
 - « les threads font la course »
 - Mauvaise idée, on ne contrôle pas qui gagnera
 - Synchronisations bloquantes pour corriger
 - Utilisation de mutex, lecteurs et écrivains partagent le lock
 - Possibilité de structures « lock free » cependant très difficiles à réaliser correctement

Modèle Mémoire

- Les différents processeurs
 - Ont une vision distincte de la mémoire
 - Chaque processeur a son propre cache
 - Les garanties pour la propagation entre caches sont relaxées
 - Des instructions dédiées (barrières mémoires) pour contrôler la cohérence
 - Accessibles en C++
- Les variables atomic
 - Disposent d'opération atomiques simples : incrément, compare and swap
 - Permettent l'introduction de barrières mémoire explicites
 - Sont la base de toutes les autres primitives de synchronisation

Absence de Data Race

- L'objectif est la cohérence « séquentielle »
 - Les langages/architectures modernes permettent de la contourner
 - Les raisonnements normaux s'écroulent

$X=0 ; Y=0$

(Thread A)

$X=1$

$R1=Y$

(Thread B)

$Y=1$

$R2=X$

Absence de Data Race

- L'objectif est la cohérence « séquentielle »
 - Les langages/architectures modernes permettent de la contourner
 - Les raisonnements normaux s'écroulent

$X=0 ; Y=0$

(Thread A)

$X=1$

$R1=Y$

(Thread B)

$Y=1$

$R2=X$

- Ici on peut terminer avec : $R1=R2=0$
- L'absence de Data Race garantit la cohérence séquentielle
 - Donc toujours prévenir les conflits d'accès

Risques liés à la concurrence

- Nombre exponentiel d'entrelacements possibles
 - Si N processus indépendants, chacun pouvant prendre K états possibles, N^K états du programme entier
 - Difficile pour le raisonnement
 - Difficile à vérifier/contrôler pour les outils
- Erreurs et bugs de concurrence
 - Data race, violation de section critique, deadlocks, famines
 - Difficile à trouver
 - Difficile à reproduire (tests !)
 - Difficile à debugger

PROMISE, FUTURE

Simplifier les échanges entre threads

- Deux threads collaborent :
 - Thread principal lance un thread annexe en lui déléguant le calcul d'un résultat
- Plusieurs problèmes se posent :
 - Il faut des données partagées entre les threads, en particulier le résultat du calcul produit par le thread annexe
 - Il faut prévoir un mécanisme de notification, pour que le thread principal puisse attendre si nécessaire que le thread annexe ait fini
- Une solution propre et générique pour ce type de cas d'utilisation ?
 - `future<T>` : un résultat, donné au thread principal, « `T get()` » est bloquant tant que le résultat n'est pas disponible.
 - `promise<T>` : la promesse que doit remplir le thread annexe, « `void set_value(T val)` » rend le future associé disponible et débloque.

Programmation par Tâches

- Lancer un traitement puis obtenir sa réponse
 - Calculée par un autre thread
 - Directement en mémoire
 - Nécessité de synchronisation
- Comment communiquer le résultat de la tâche
 - Un canal de communication<T> standard pour échanger un résultat
 - Un état partagé, variable(s) accédées par les deux threads
 - **promise** : on doit la tenir, et écrire dedans le résultat
 - **Extrémité en écriture**
 - `set_value(T)`
 - **future** : on obtient la valeur au bout d'un moment
 - **Extrémité en lecture**
 - `wait()`, `get()`
 - Le **future** est bloquant tant que la **promise** n'est pas encore tenue

Promise/Future

```
void hello(std::promise<std::string>& prm)
{
    std::string res{"Hello from future"};
    prm.set_value(res);
}

int main()
{
    std::promise<std::string> prm;
    std::future<std::string> fut(prm.get_future());
    std::thread t(hello, std::ref(prm));
    std::cout << "Hello from main\n";
    std::cout << fut.get() << '\n';
    t.join();
}
```

Promise / Future

- Permet aussi de logger des exceptions
 - `set_exception(ex)` au lieu de `set_value(val)` sur une promise
 - Le `get()` sur le future associé va lever l'exception
 - Utile même en mono thread
- L'état partagé n'est accessible via le future qu'une seule fois
 - Est consommé par `get()`, s'il n'est pas trivial
 - Réalisé avec **`std::move`** évitant une copie
- Si on souhaite au contraire attendre à plusieurs un résultat
 - **`shared_future`** peut être construit à partir de **`future`**
 - Invalide l'objet future sous jacent, il ne faut plus utiliser que le shared

Opérations du future

- `valid()`
 - Vrai si le future est encore utilisable (associé à un état partagé)
- `get()`
 - Rend la valeur associée
 - Bloquant si elle n'est pas disponible
- `wait()/wait_for()/wait_until()`
 - Attente bloquante ou avec timeout
- `share()`, ou simplement construire un `shared_future<T>(f)`
 - Construire une version partagée
 - Invalide le future

Taches : async

- L'exemple est encore assez lourd syntaxiquement

```
void hello(std::promise<std::string>& prm)
{
    std::string res{"Hello from future"};
    prm.set_value(res);
}
```

```
int main()
{
    std::promise<std::string> prm;
    std::future<std::string> fut(prm.get_future());
    std::thread t(hello, std::ref(prm));
    std::cout << "Hello from main\n";
    std::cout << fut.get() << '\n';
    t.join();
}
```

Async

- Du code parasite
 - Sur la fonction appelée
 - Signature prend un « `promise<T>` » au lieu de `T`
 - Signature rend `void` (forcé par `thread`)
 - Nécessaire de `set_value` explicite
 - Sur le code appelant
 - Création de `thread`
 - `Join` explicite
- Solution :
 - **async** : même signature que **thread**
 - Engendre automatiquement une paire `promise/future`
 - Encapsule les invocations pour avoir des signatures « normales »
 - Encapsule la création/fin de `thread`
 - Gère le degré de **parallélisme** indépendamment de la **concurrency**

Hello revisité

```
#include<future>
```

```
#include<iostream>
```

```
std::string hello()
```

```
{
```

```
    return "Hello from future";
```

```
}
```

```
Int main()
```

```
{
```

```
    auto future =std::async(hello);
```

```
    std::cout <<"Hello from main\n";
```

```
    std::cout << future.get() <<'\n';
```

```
}
```

Politiques d'exécution

```
template< class Function, class... Args>  
result_type async(Function&& f, Args&&... args);  
  
template< class Function, class... Args >  
result_type async(launch policy, Function&& f, Args&&... args);
```

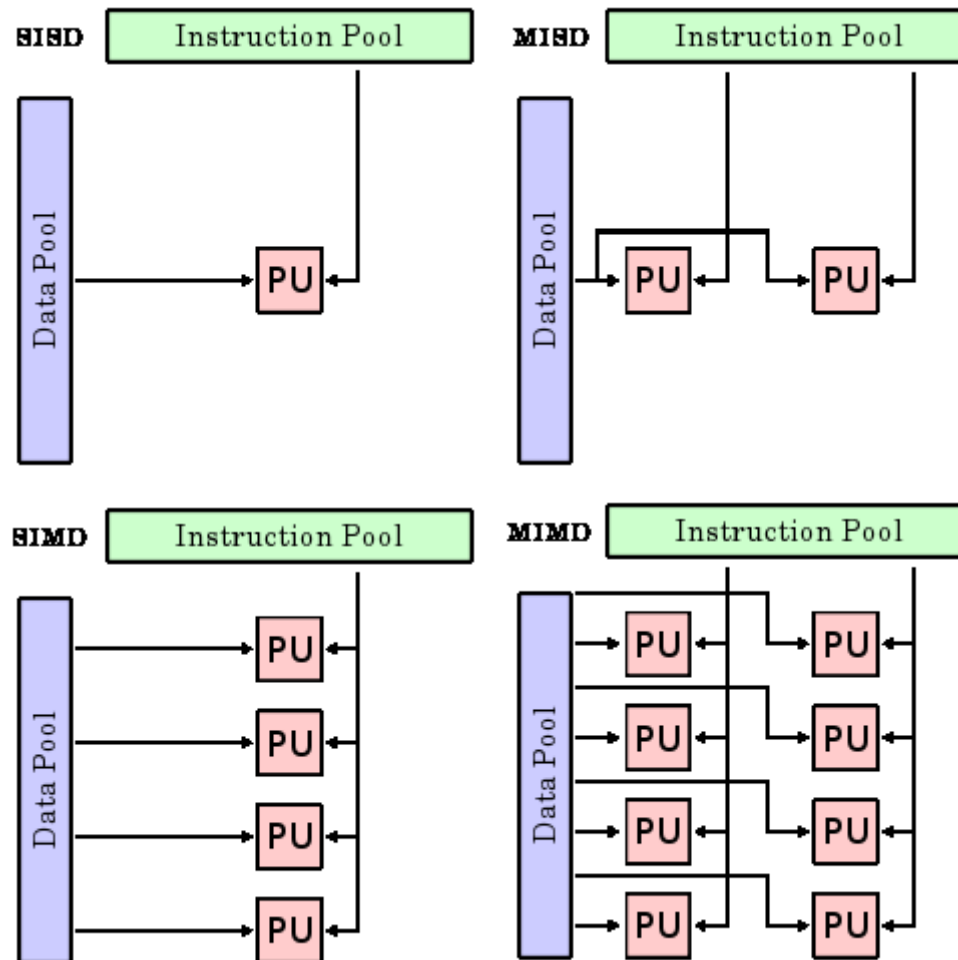
- On peut spécifier une politique
 - **launch::async** : exécution dans un nouveau thread
 - **launch::deferred** : le future qui est retourné n'est pas concurrent
 - L'invocation à `get()` va faire calculer le résultat par le thread appelant
 - **launch::deferred | launch::async** : défaut, laisser le runtime décider

Packaged_task

- Élément support pour la construction de async
 - Encapsule une fonction dans une paire future/promise
 - Pas de création de thread ou de concurrence
 - Construction similaire à async ou thread
 - Fonction + arguments en nombre et type arbitraire

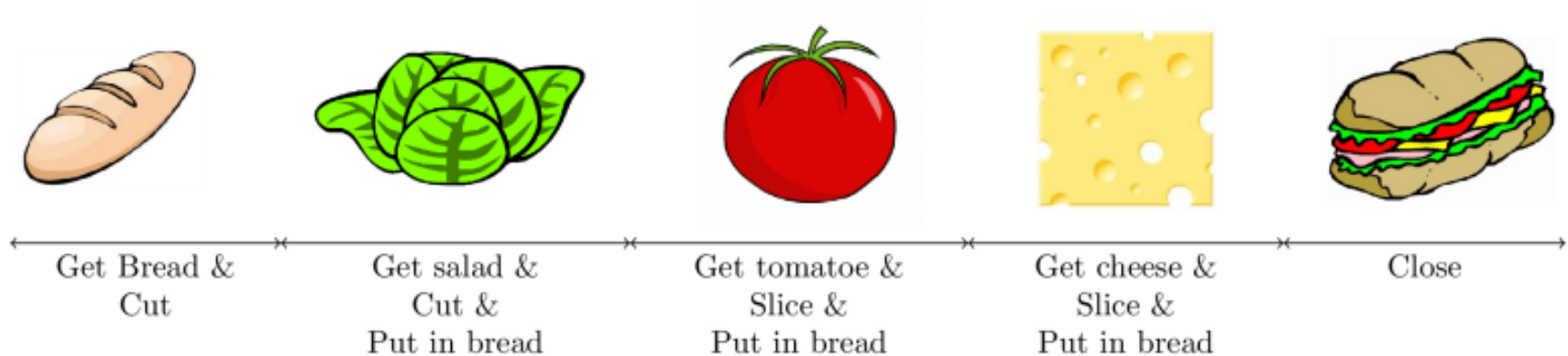
PARALLELISME MATERIEL

Modèles de Parallélisme (Flynn)



- SISD : Séquentiel
- MISD : (rare) redondance
- MIMD : multi thread
- SIMD : Single Instruction Multiple Data
 - Vectorisation

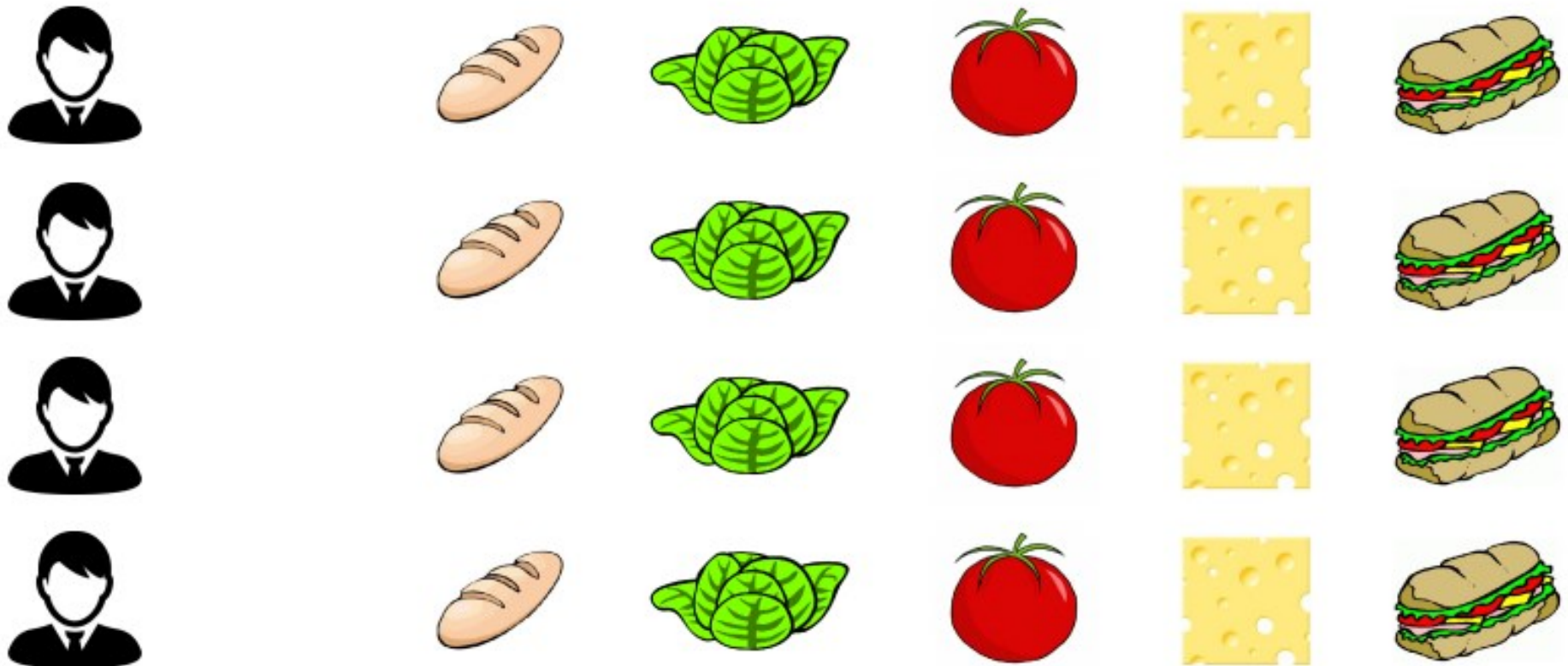
The burger factory assembly line



How to make several sandwich as fast as possible ?

MIMD approach

4 workers make 1 sandwich each:

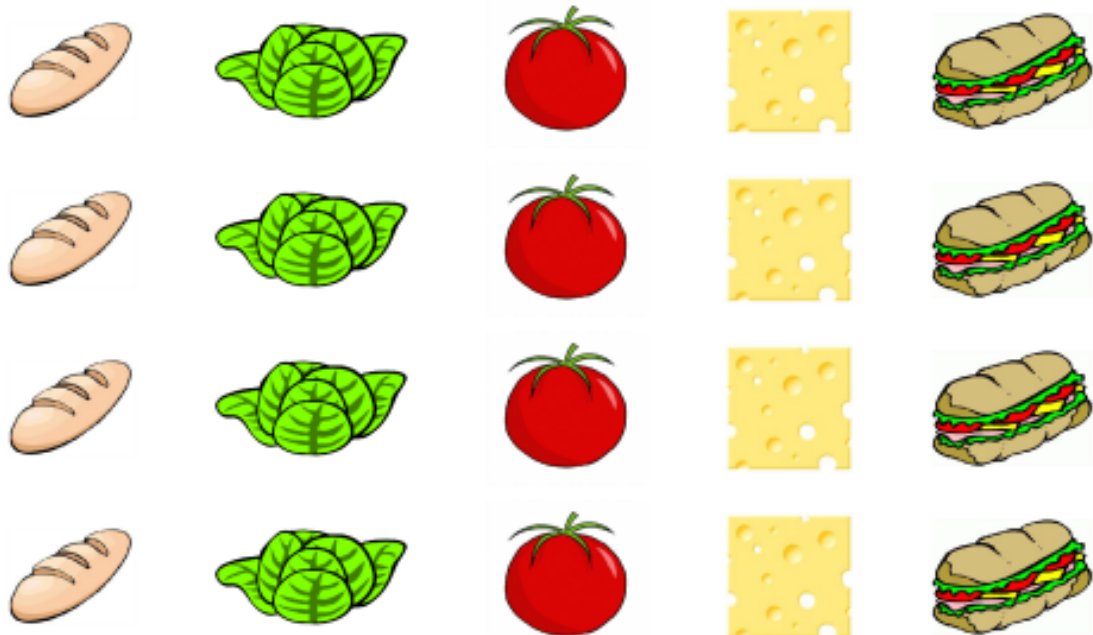


- Expected speed-up: 400%

SIMD approach

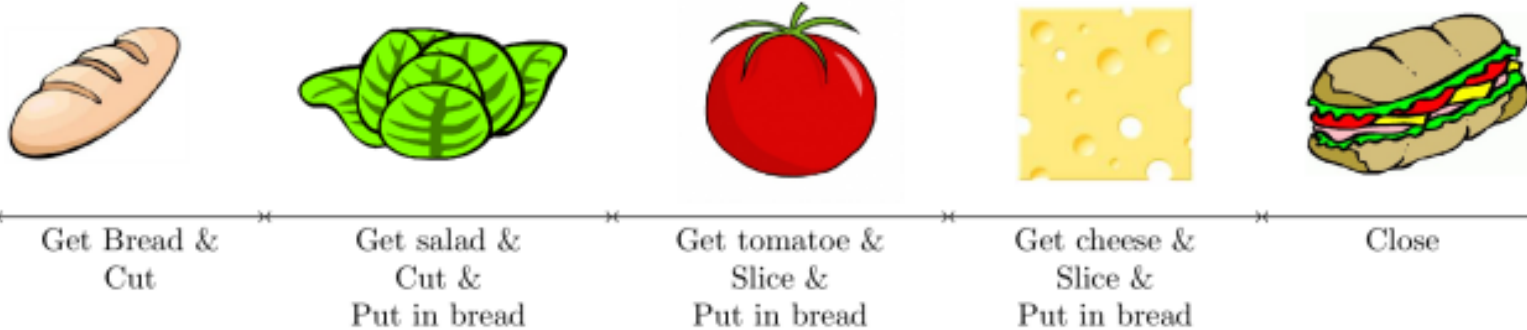
A single worker makes 4 sandwiches with all operations in the same time:

•



Expected speed-up: 400%

Exécution Out-of-Order



Pipeline:

Instructions

- | | |
|-----------------------|--------------------------|
| 1. Get Bread | 7. Slice tomatoes |
| 2. Cut Bread | 8. Put tomatoes in bread |
| 3. Get Salad | 9. Get cheese |
| 4. Cut Salad | 10. Slice cheese |
| 5. Put Salad in bread | 11. Put cheese in bread |
| 6. Get Tomatoes | 12. Close |

- Tant que l'ordre des instructions sur chaque produit est respecté
 - Possible de réordonner

Exécution OoO



Suppose we have we have three arms usable in the same time:

- Unit 1. can grab products
 - Unit 2. can cut/slice stuffs
 - Unit 3. can do other stuffs
-
- Utiliser l'existence physique d'unités différentes au niveau matériel

Exemple Revisité

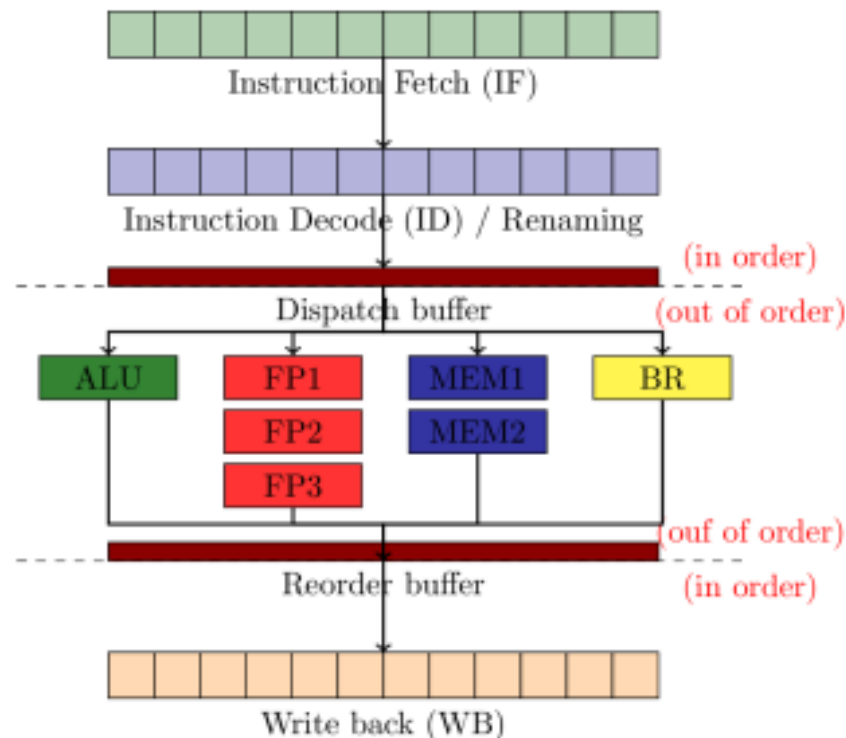
Cycle	Load Unit	Slice/Cut Unit	Generic Unit
1	1. Get Bread		
2	3. Get Salad	2. Cut Bread	
3.	5. Get Tomatoes	4. Cut Salad	
4.	8. Get Cheese	7. Cut tomatoes	6. Put Salad in bread
5.		10. Cut cheese	9. Put Tomatoes in bread
6.			11. Put cheese in bread
7.			12. Close

- On a un parallélisme niveau instructions : Instruction Level Parallelism ILP
 - On parle de pipeline, vitesse mesurée en instructions par cycle, ici 1.7

En pratique : pipeline RISC

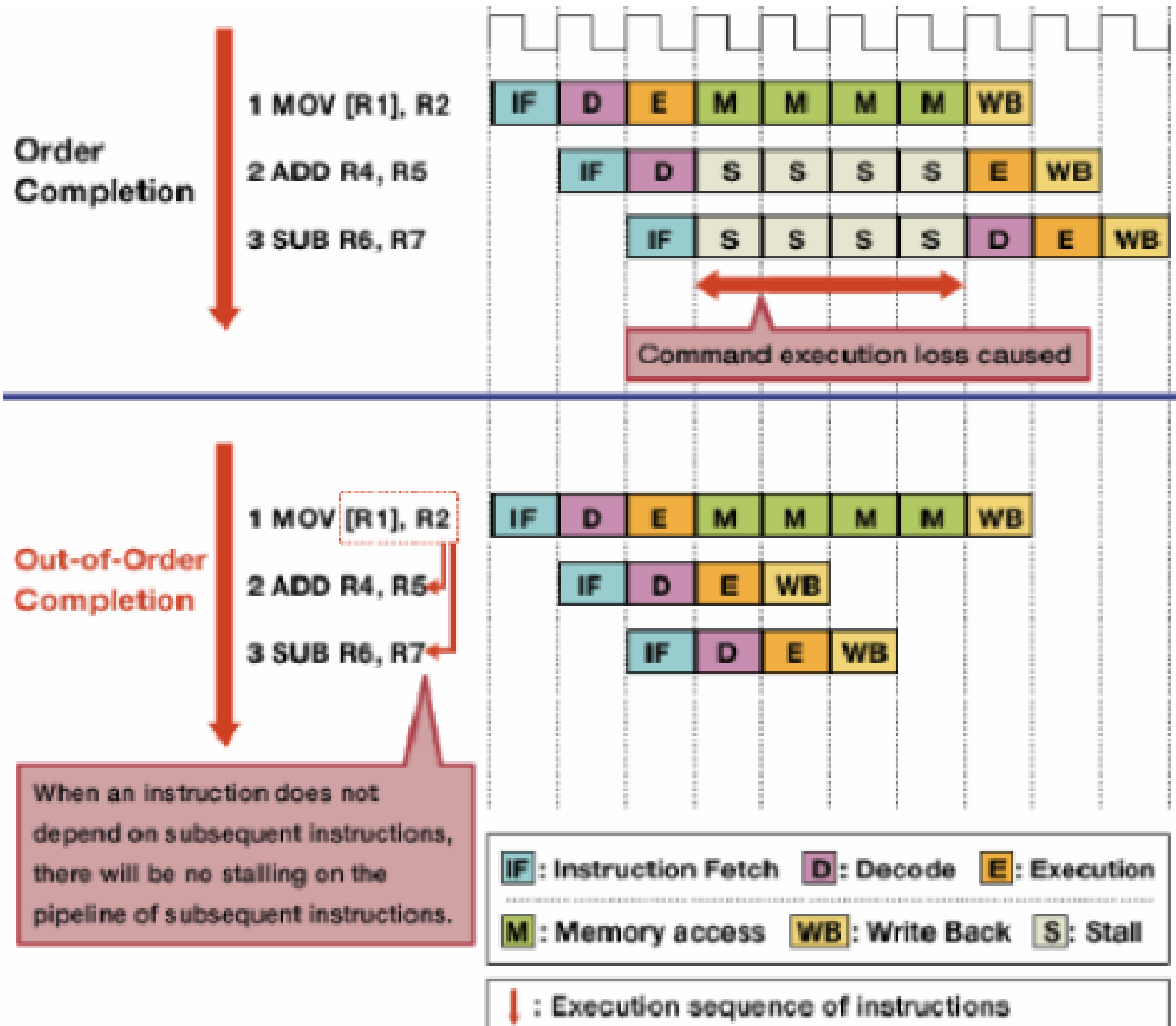
Pipeline RISC:

- IF: Instructions are serially fetched
- ID: Instructions are decoded (ID) and register are affected
- EX: Instruction Executed by an Unit (EX):
- MEM: LOAD or STORE from memory
- WB: Store a result in a register



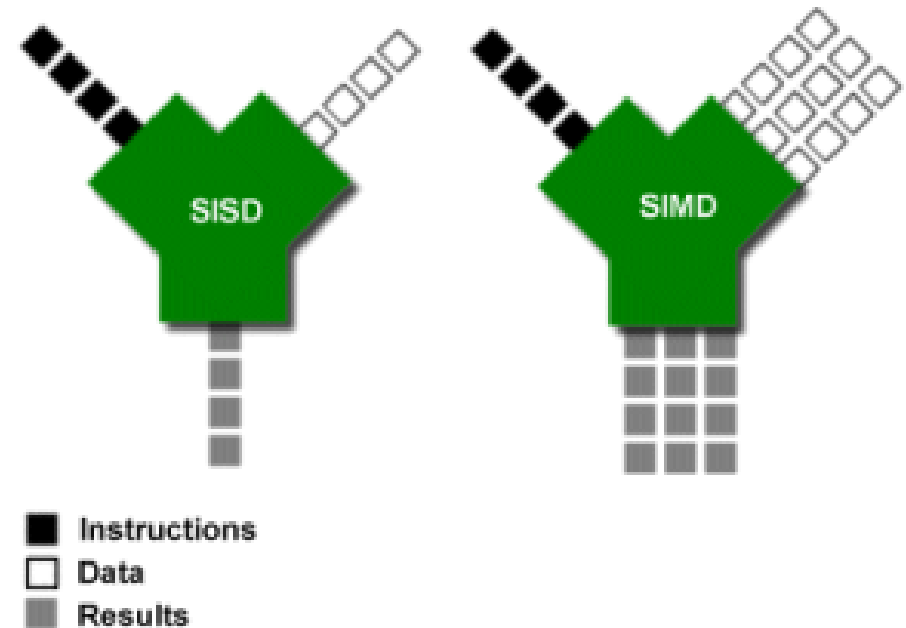
Pipeline

- Une seule « couleur » à la fois
- Si pas de dépendance de données
 - Réordonner
 - Instruction 2 faite avant la fin de 1



Profiter des pipelines

- Organiser les données
 - Données contigues : vector, string, deque
 - Extrêmement important, payer les copies nécessaires
- Eviter les branchements (if)
 - Invalide la prochaine instruction
 - Les CPU modernes utilisent tous du « branching prediction »
 - Cf Meltdown, Spectre...
 - Cependant si on peut restructurer le code c'est mieux



Instructions Super scalaire

- La plupart des CPU proposent de l'ILP
- Les GPU sont tous sur ce modèle vectoriel

Processor	Instructions	Register size
x86	SSE	128
	AVX(2)	256
	AVX512	512
ARM	NEON	128
IBM POWER	Altivec	128
	QBX	256
SPARC	HPC-ACE	128
	HPC-ACE2	256

Utilisation explicite des instructions

- En utilisant explicitement les instructions

`<mmintrin.h>` MMX, `<xmmintrin.h>` SSE, `<emmintrin.h>` SSE2, `<pmmintrin.h>` SSE3. [Ref](#)

```
__m64 _mm_abs_pi16 (__m64 a)
__m64 _mm_abs_pi32 (__m64 a)
__m64 _mm_abs_pi8  (__m64 a)
```

- En déclarant des types particuliers + opérateurs habituels

```
typedef float float4 __attribute__((ext_vector_type(4)));
typedef float float2 __attribute__((ext_vector_type(2)));

float4 x, y;
x += y
```

A l'aide d'OpenMP

- Un jeu de pragma pour activer la vectorisation

```
#pragma omp simd  
for (int i = 0; i < N; ++i)  
    a[i] += s * b[i];
```

- Le compilateur sait aussi vectoriser
 - Il faut lui faire confiance, mais ne pas hésiter à lui mâcher le travail
 - Désambiguer l'aliasing
 - Glisser des directives par endroits

OPEN MP

<https://www.openmp.org/>

Open Multi-Processing

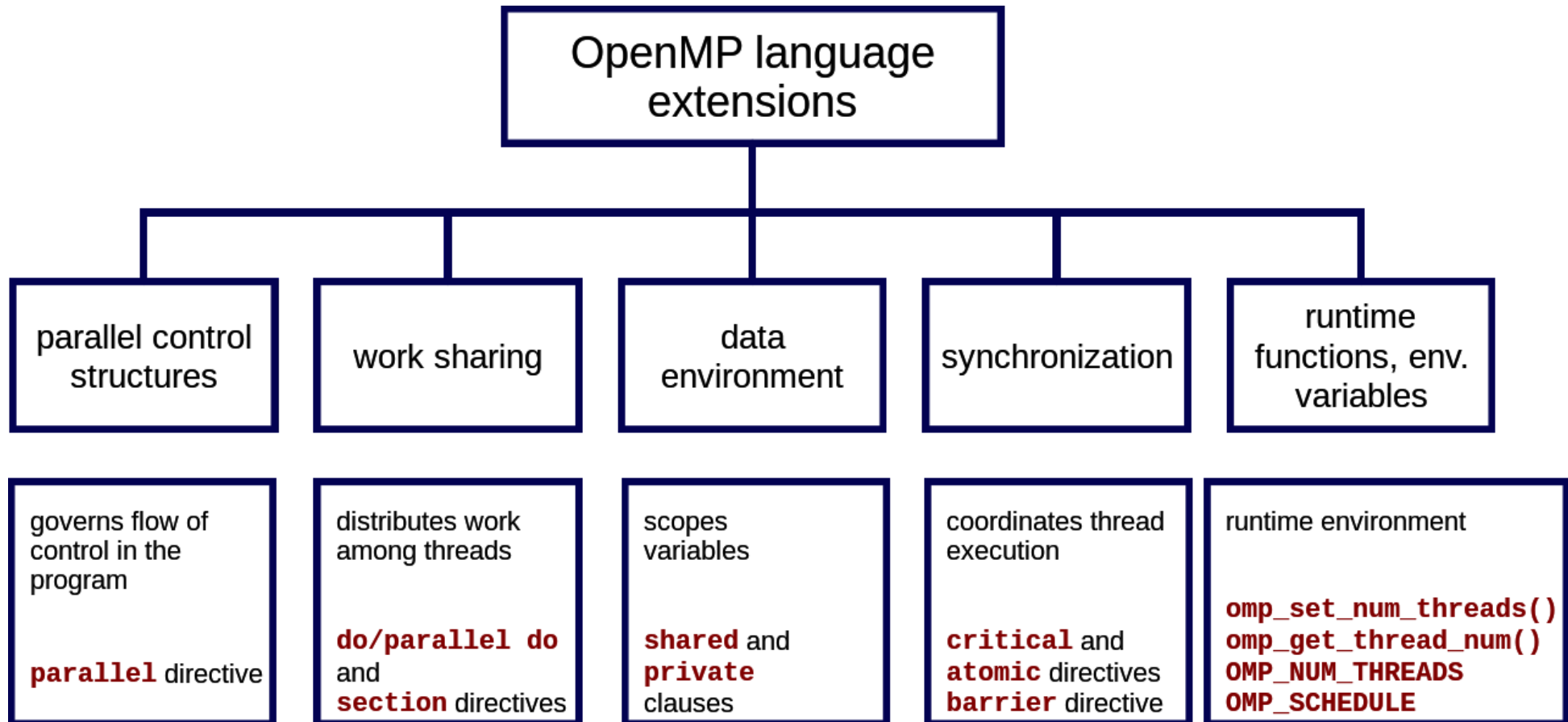
Open MP

- Une API ouverte supportée par de nombreux industriels
 - AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, Texas Instruments, Oracle Corporation, ...
 - « le » standard en C, C++, Fortran
- Consiste en un jeu de pragma `#omp`
 - Active la vectorisation; permet la concurrence sous diverses formes
 - Compatible MPI : Message Passing Interface pour le distribué (grid)
- Flexible et relativement simple
 - À utiliser avec discernement
 - Cf conseils précédents sur la localité des données
- Pourquoi utiliser OpenMP
 - Le compilateur ne peut pas faire tout tout seul
 - Identifier le parallélisme « safe » sans dépendances de données ?
 - Identifier la bonne granularité pour paralléliser ?

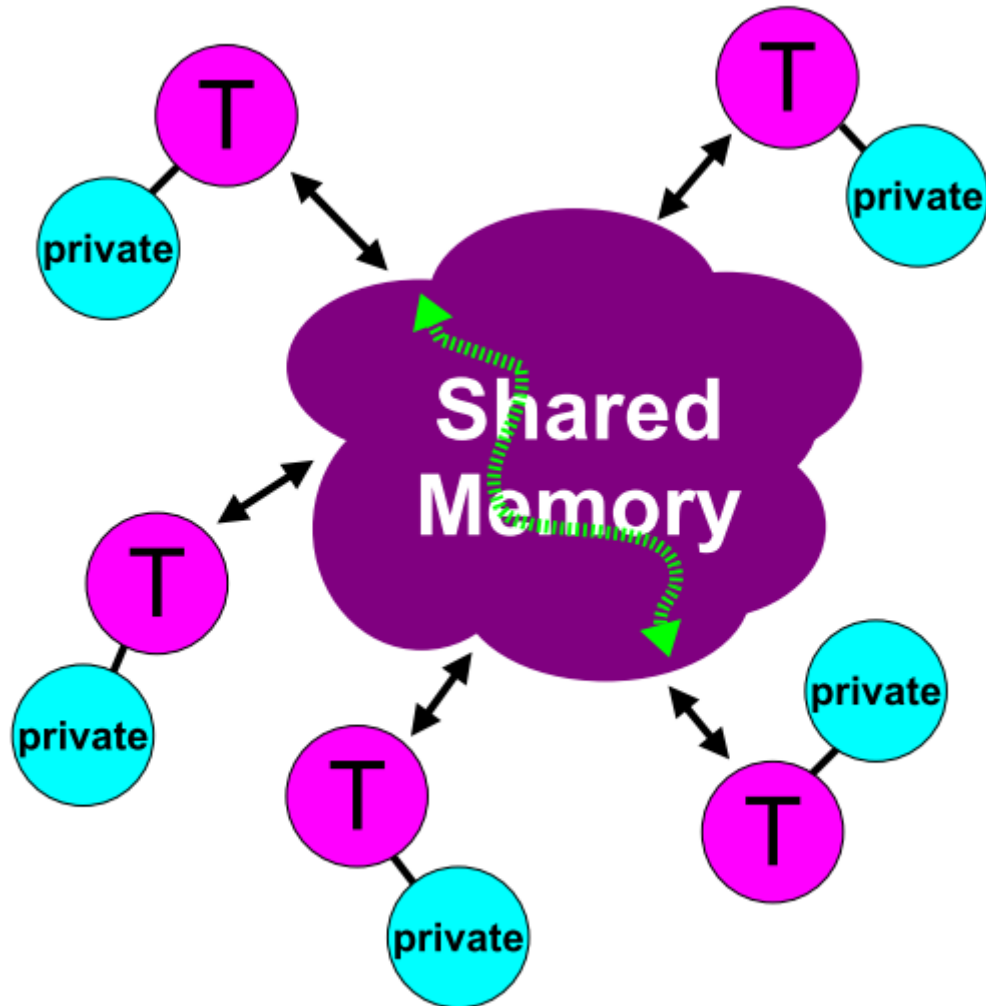
OpenMP

- Avantages notables sur e.g. utilisation directe SSE
 - Supporté par de nombreux compilateurs sur de nombreux matériels
 - Un « standard » mature et portable
 - Relativement peu intrusif dans le code
- OpenMP propose du parallélisme grain très fin
 - Modèle mémoire et concurrence
 - Très léger par rapport aux threads
 - Bien adapté aux architectures modernes massivement multicore

OpenMP : vue d'ensemble



Modèle mémoire OpenMP

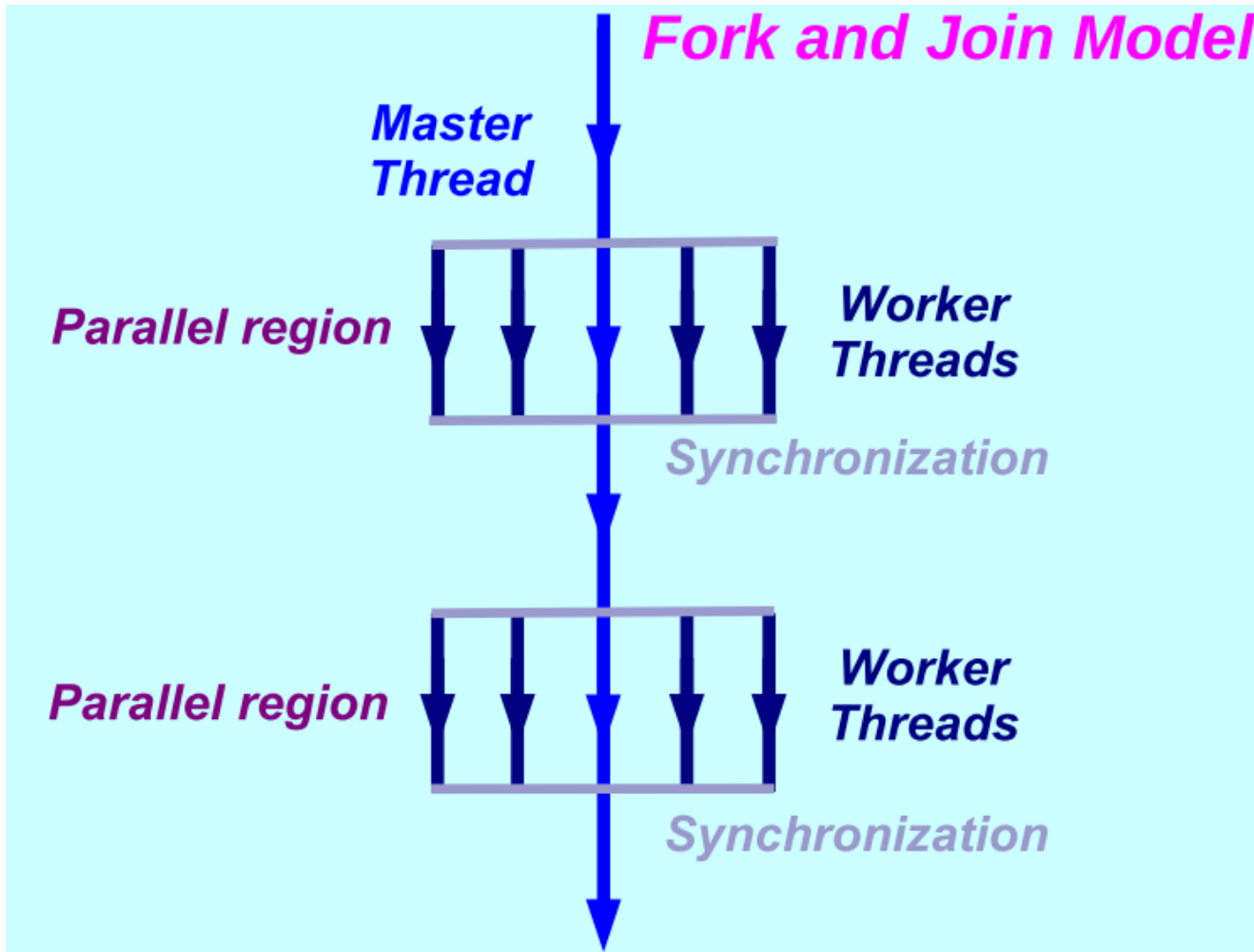


- Modèle similaire aux threads
 - accès partagé à une mémoire globale
- Chaque thread dispose de données privées/locales
 - Accès exclusif par le propriétaire
- Les transferts de données sont transparents
- Les synchronisations sont principalement implicites

Mémoire OpenMP

- Deux types de mémoire distingués Shared vs Private
- Shared
 - Une seule instance de la donnée
 - Tous les threads peuvent y accéder en concurrence
 - Il existe des primitives de synchronisation
 - Les changements sont visibles de tous, mais pas nécessairement tout de suite
 - Modèle mémoire relaxé
- Private
 - Chaque thread détient **une copie** de la donnée (cf mmap MAP_PRIVATE)
 - Aucun autre thread ne peut y accéder

Modèle d'exécution OpenMP



- Des zones en parallèle
 - E.g. for sur un tableau
- Des zones séquentielles
- Synchronisations implicites entre les zones

Exemple simple

For-loop with independent iterations

```
for (int i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

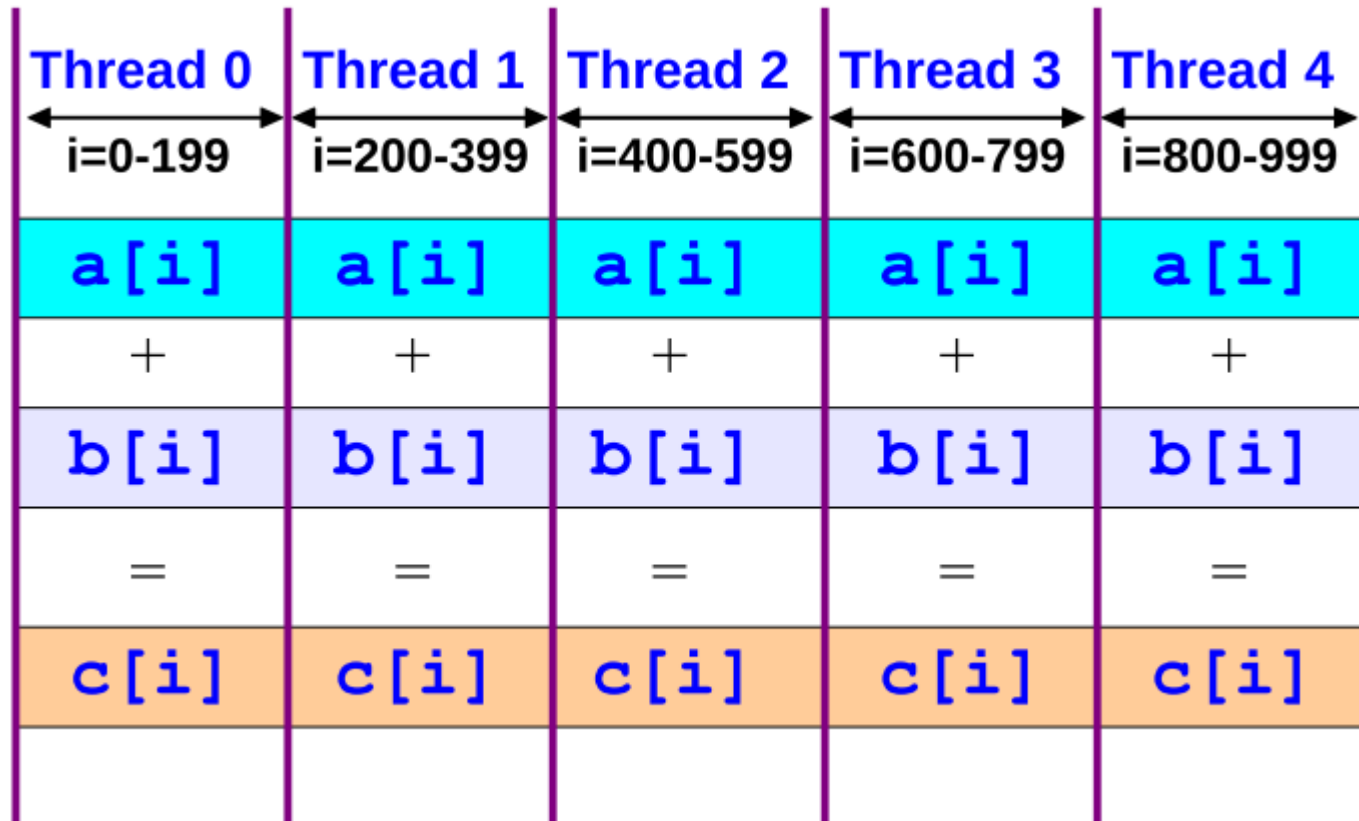
For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for  
for (int i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c  
% setenv OMP_NUM_THREADS 5  
% a.out
```

- On ajoute un pragma : « parallel for »
- Le nombre de threads est positionné via l'environnement
- Le reste est automatique

Execution en parallèle



- Le « parallel for » découpe la boucle en portions égales

Terminologie OpenMP

- L'exécution se divise en un **Master** et des **Worker**
- Une **région parallèle** est un bloc exécuté simultanément par plusieurs threads
 - Le thread master port l'ID 0 zéro
 - En entrant dans une région, on fixe un nombre de thread
 - Les régions ne doivent pas être imbriquées
 - Reste possible mais support assez limité
 - Retour sur l'idée de préparer le terrain pour le SIMD
 - On peut garder les zones parallèles avec un « if »
 - Si false, on exécute séquentiellement
 - Contrôle du grain du parallélisme
- Le partage du travail à réaliser entre les threads
 - Est à la charge du « work-sharing construct »
 - Politiques par défaut OK

Région Parallèle

```
#pragma omp parallel [clause[,] clause] ...]  
{  
    "this is executed in parallel"  
} (implied barrier)
```

- Un bloc de code exécuté en parallèle/simultanément par les workers

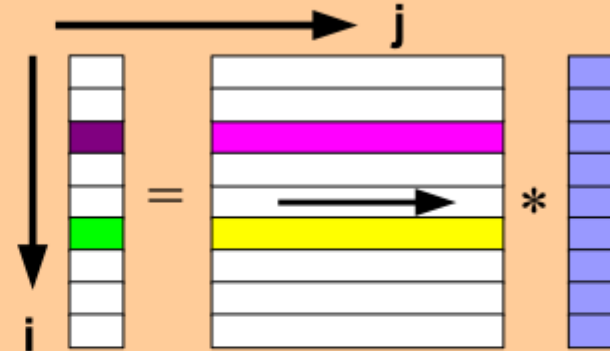
Clauses : if, private, shared

```
#pragma omp parallel if (n > threshold) \  
    shared(n,x,y) private(i)  
{  
    #pragma omp for  
    for (i=0; i<n; i++)  
        x[i] += y[i];  
} /*-- End of parallel region --*/
```

- « if (expr) »
 - Exécuter séquentiellement si false
- « private (list) »
 - Pas de version « globale » de la variable, références par thread
 - Valeurs indéfinies au début et après le bloc parallèle
- « shared (list) »
 - Variables accédées en concurrence dans l'espace d'adressage global

Produit Matrice Vecteur

```
#pragma omp parallel for default(none) \
    private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```



TID = 0

for (i=0,1,2,3,4)

i = 0

sum = \sum b[i=0][j]*c[j]

a[0] = sum

i = 1

sum = \sum b[i=1][j]*c[j]

a[1] = sum

TID = 1

for (i=5,6,7,8,9)

i = 5

sum = \sum b[i=5][j]*c[j]

a[5] = sum

i = 6

sum = \sum b[i=6][j]*c[j]

a[6] = sum

Barrières

```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];
```

```
for (i=0; i < N; i++)  
    d[i] = a[i] + b[i];
```

- On exécute ces deux boucles dans un bloc parallèle
 - Problème potentiel si la division du travail n'est pas homogène entre les deux boucles

Barrières

```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];
```

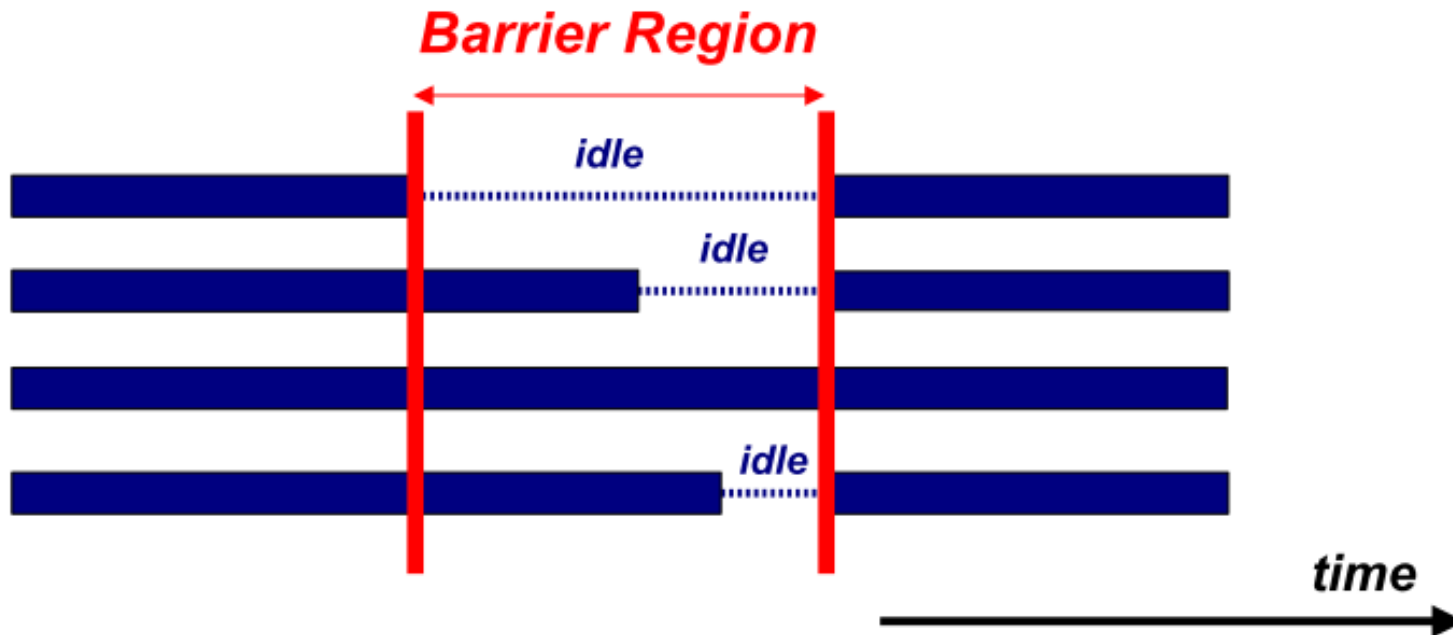
wait !

barrier

```
for (i=0; i < N; i++)  
    d[i] = a[i] + b[i];
```

- On exécute ces deux boucles dans un bloc parallèle
 - Problème potentiel si la division du travail n'est pas homogène entre les deux boucles
- Ajout d'une barrière de rendez vous
 - Synchronisation classique à N threads (cf le partiel :D)
 - Attendre que tous les threads aient atteint la barrière avant de continuer

Barrière OpenMP



- Syntaxe

```
#pragma omp barrier
```

Fin de bloc non contrainte : **nowait**

- Permet de minimiser les synchronisation **en fin** de bloc
 - Pas de synchro explicite ou implicite en entrée de bloc
 - Avec **nowait**, à la fin du bloc, le master peut continuer dès que l'un des workers termine

```
#pragma omp for nowait  
{  
    :  
}
```

Exemple plus complet

```
#pragma omp parallel if (n>limit) default(none) \
    shared(n,a,b,c,x,y,z) private(f,i,scale)
{
```

```
    f = 1.0;
```

```
#pragma omp for nowait
```

```
    for (i=0; i<n; i++)
        z[i] = x[i] + y[i];
```

```
#pragma omp for nowait
```

```
    for (i=0; i<n; i++)
        a[i] = b[i] + c[i];
```

```
#pragma omp barrier
```

```
    ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
    ....
```

```
} /*-- End of parallel region --*/
```

Statement is executed
by all threads

parallel loop
(work is distributed)

parallel loop
(work is distributed)

synchronization

Statement is executed
by all threads

parallel region

Partage du travail

<pre>#pragma omp for { }</pre>	<pre>#pragma omp sections { }</pre>	<pre>#pragma omp single { }</pre>
---	--	--

- Trois options principales pour définir le partage
 - « for »: découpe une boucle en segments
 - « sections »: on a plusieurs blocs de code constituant des sections
 - « single »: exécution une seule fois, sans que ça bloque le reste

for

```
#pragma omp for [clause[,] clause] ...]  
  <original for-loop>
```

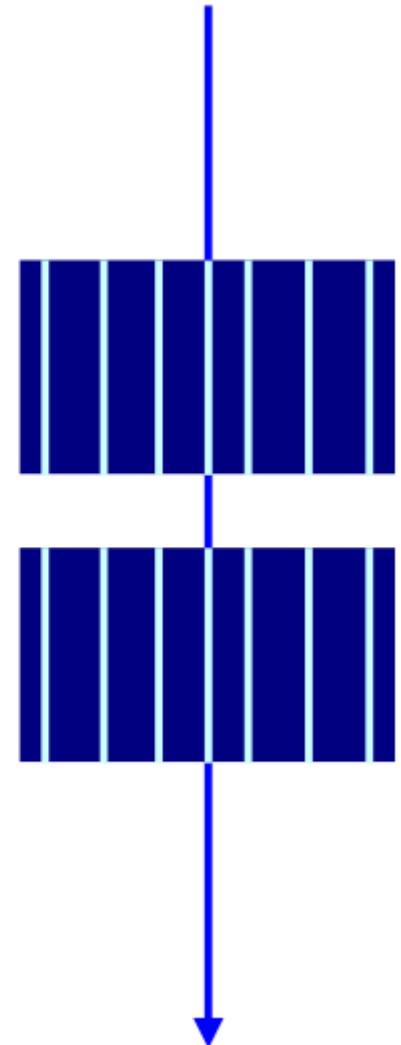
- Découpe des itérations sur les threads

for : exemple

```
#pragma omp parallel default(none)\
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /* -- End of parallel region -- */
    (implied barrier)
```



Section

```
#pragma omp sections [clause(s)]  
{  
    #pragma omp section  
        <code block1>  
    #pragma omp section  
        <code block2>  
    #pragma omp section  
        :  
}
```

- Découpe de blocs de code sur les threads
- NB : peut se combiner avec le for

Section : exemple

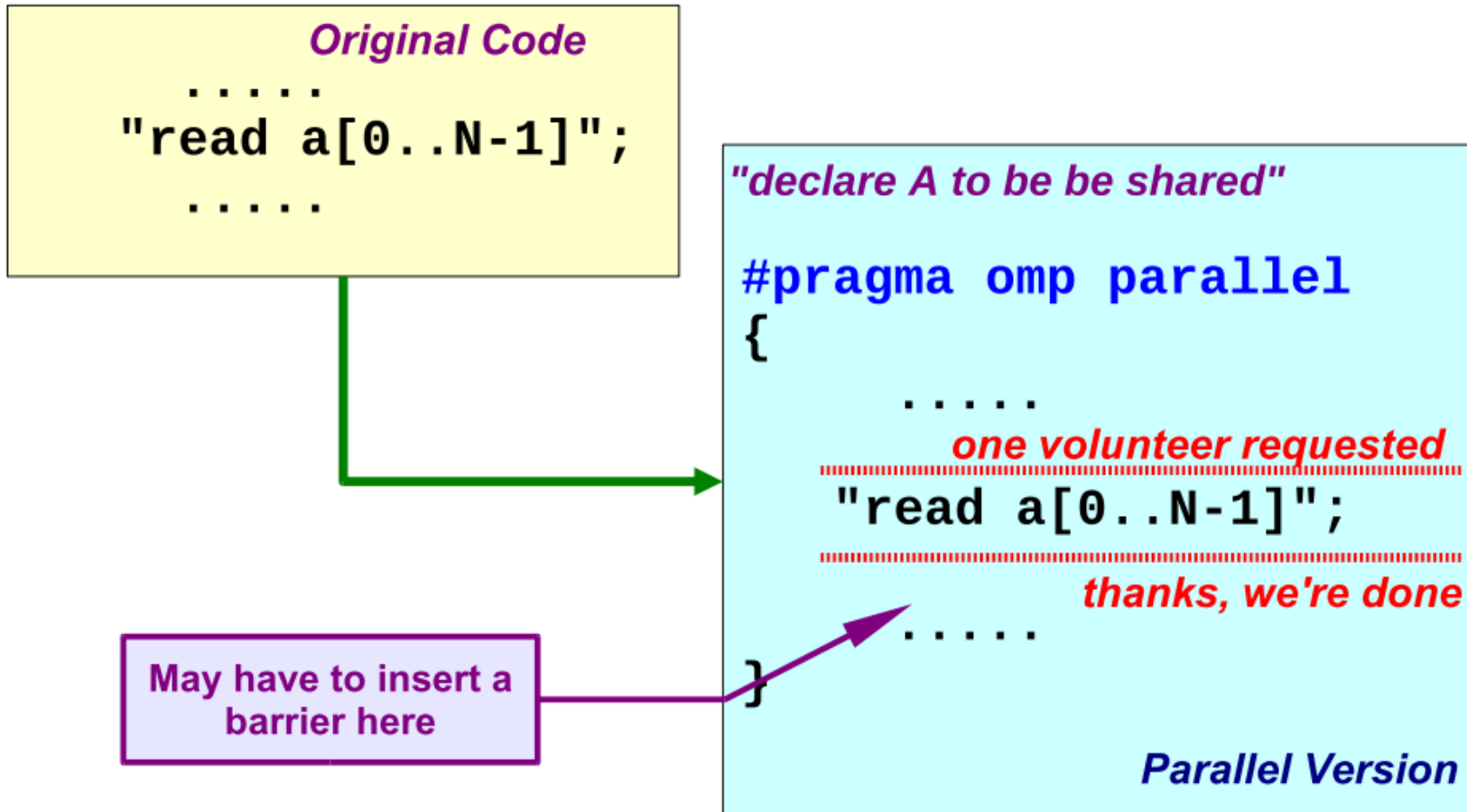
```
#pragma omp parallel default(none)\
    shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

        #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/
} /*-- End of parallel region --*/
```

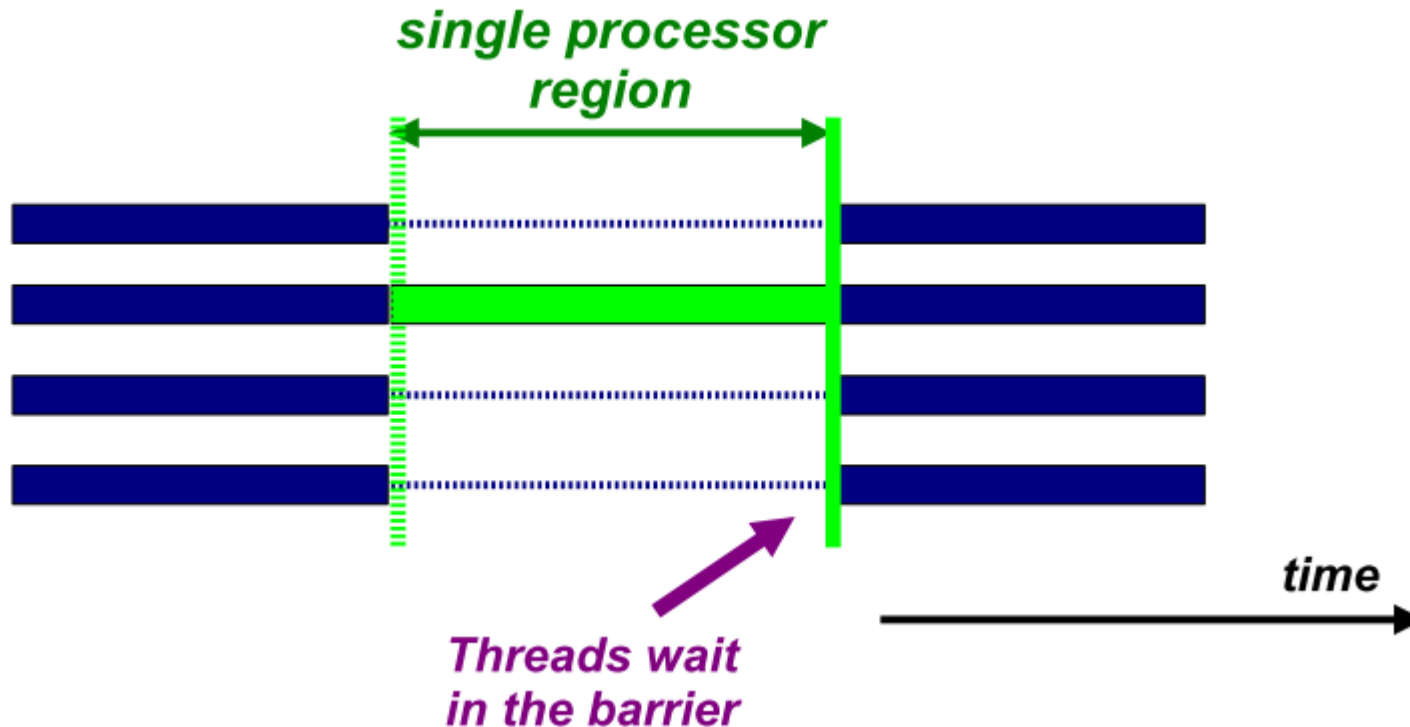


Single



- Pratique pour les I/O, les initialisations, destructions...

Single



- Barrière implicite
 - Le code « single » pourrait vite devenir un goulot d'étranglement
 - On veut qu'un seul thread exécute, mais pas d'autre barrière (en fin de bloc)

Single, Master

```
#pragma omp single [private][firstprivate] \  
                  [copyprivate][nowait]  
{  
    <code-block>  
}
```

```
#pragma omp master  
{<code-block>}
```

*There is no implied
barrier on entry or
exit !*

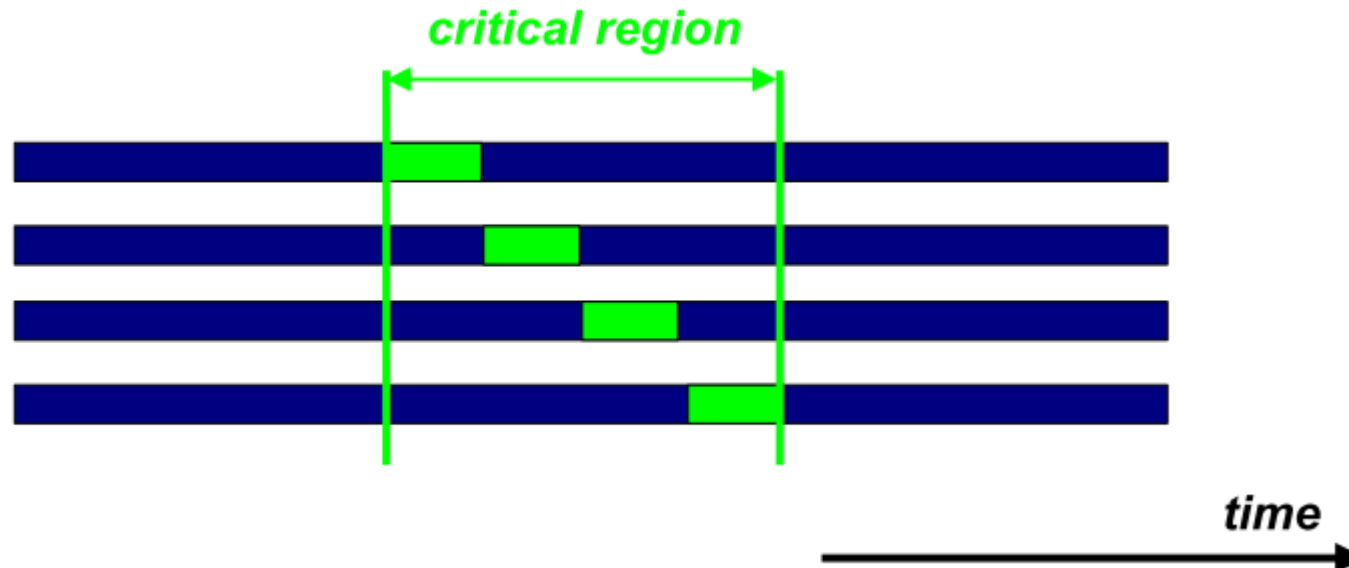
- Deux variantes
 - Single : un seul thread exécute
 - Master : le master exécute

Synchronisations explicites

```
for (i=0; i < N; i++){  
    ..... one at a time can proceed  
    .....  
    sum += a[i];  
    .....  
    ..... next in line, please  
}
```

- Utilisation de section critiques pour l'accès aux variables partagées

Sections critiques



- Parfois nécessaire pour accéder aux données
 - Pénalité importante en performances

Critical vs Atomic

- Section critique : tous les threads exécutent le code, mais un par un

```
#pragma omp critical [(name)]  
{<code-block>}
```

*There is no implied
barrier on entry or
exit !*

- Atomic : les opérations load/store sont garanties atomiques
 - Attention, pas tout à fait atomic de c++11

```
#pragma omp atomic  
<statement>
```

*This is a lightweight, special
form of a critical section*


```
#pragma omp atomic  
a[indx[i]] += b[i];
```

Pour aller plus loin : task

- Facilite l'organisation du code
 - Proche des coroutines

```
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

OpenMP Task is specified
here
(executed in parallel)



Conclusion OpenMP

- La clé du parallélisme fin sur multi cœur
 - Exploitation de la vectorisation
 - Relativement portable
 - API standard en C, C++ et Fortran
- OpenMP permet de profiter au maximum des CPU
 - Le mapping vers GPU est plus récent (OpenMP 4.5)
 - Permet l'exécution en parallèle sur GPU

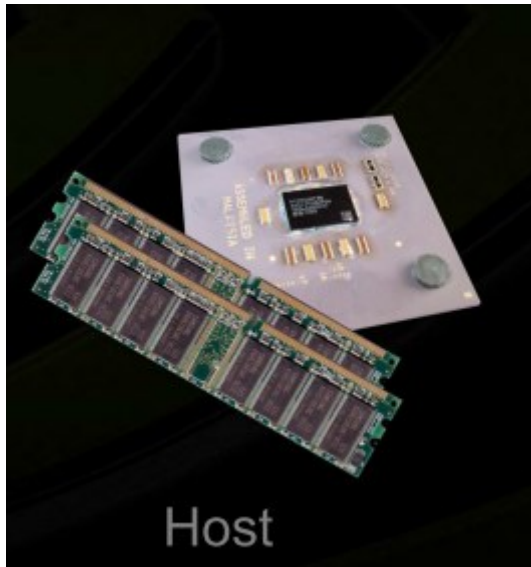
```
void vadd2(int n, float * a, float * b, float * c)
{
    #pragma omp target map(to:n,a[0:n],b[0:n]) map(from:c[0:n])
    #if defined(__INTEL_COMPILER) && defined(__INTEL_OFFLOAD)
        #pragma omp parallel for simd
    #else
        #pragma omp teams distribute parallel for simd
    #endif
    for(int i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

CUDA

GPU, CPU

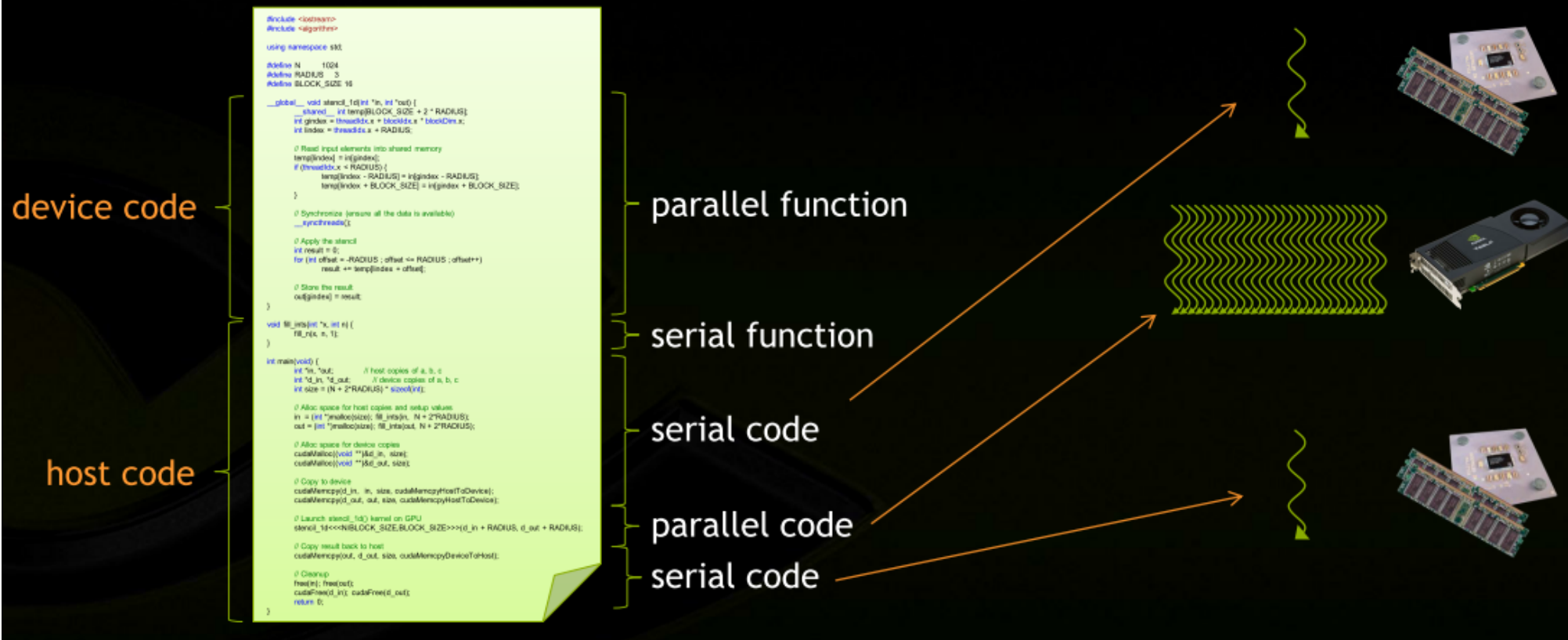
- L'évolution matérielle mène aux cartes « graphiques »
 - Massivement parallèle
 - Mémoire interne très rapide
 - Traitements très simples, synchronisés SIMD
- Accéder au GPU ?
 - Utilisation d'API spécifiques
 - CUDA est un des standards (cf aussi OpenCL) proposé par Nvidia
 - API en C/C++
 - Permet d'accéder théoriquement à une puissance énorme de calcul

Host vs Device



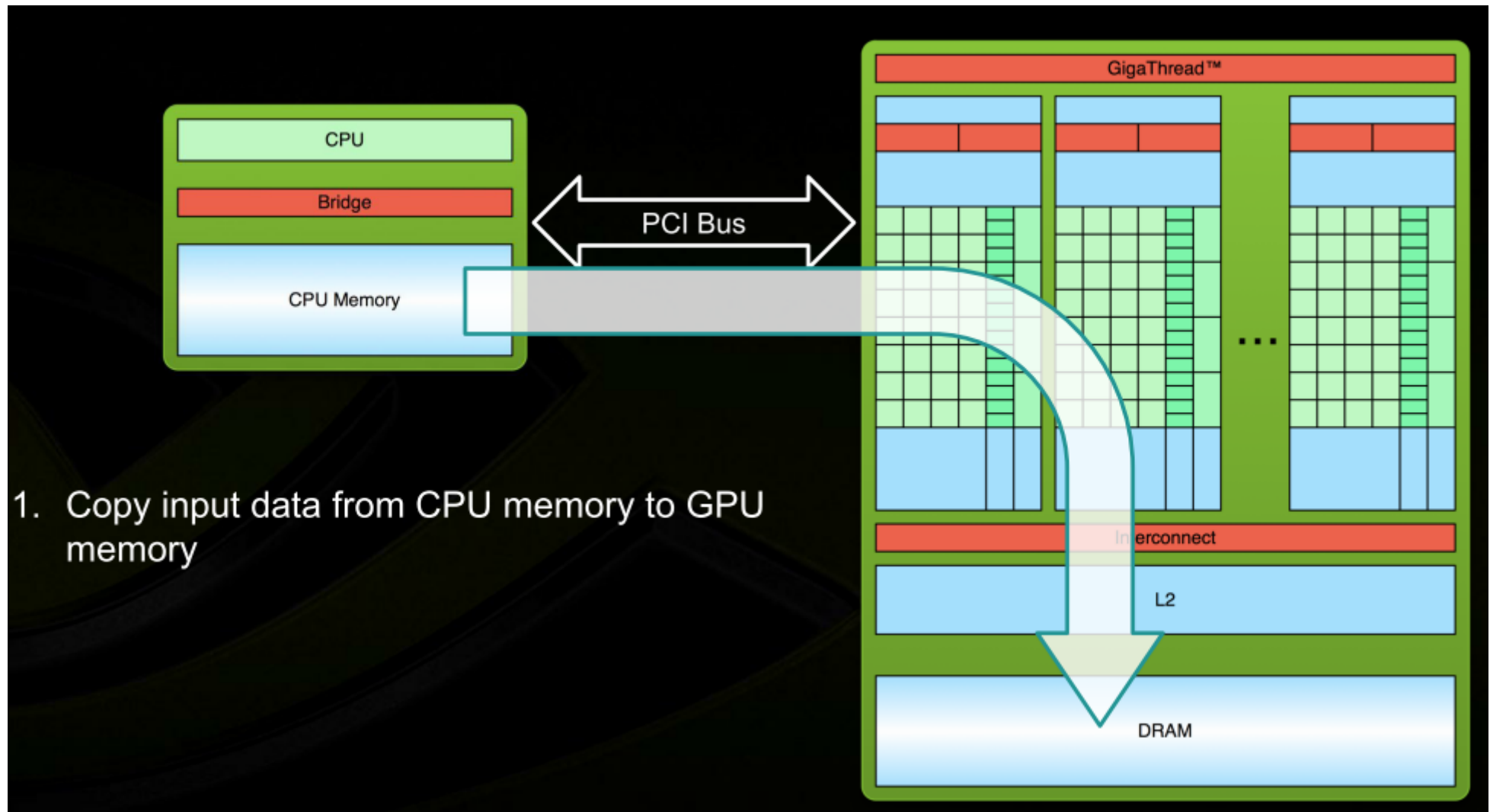
- Nécessité de matérialiser l'échange
 - 1. copier dans la mémoire du device
 - 2. exécuter SIMD le code
 - 3. ramener les données en mémoire principale

Modèle similaire à OpenMP

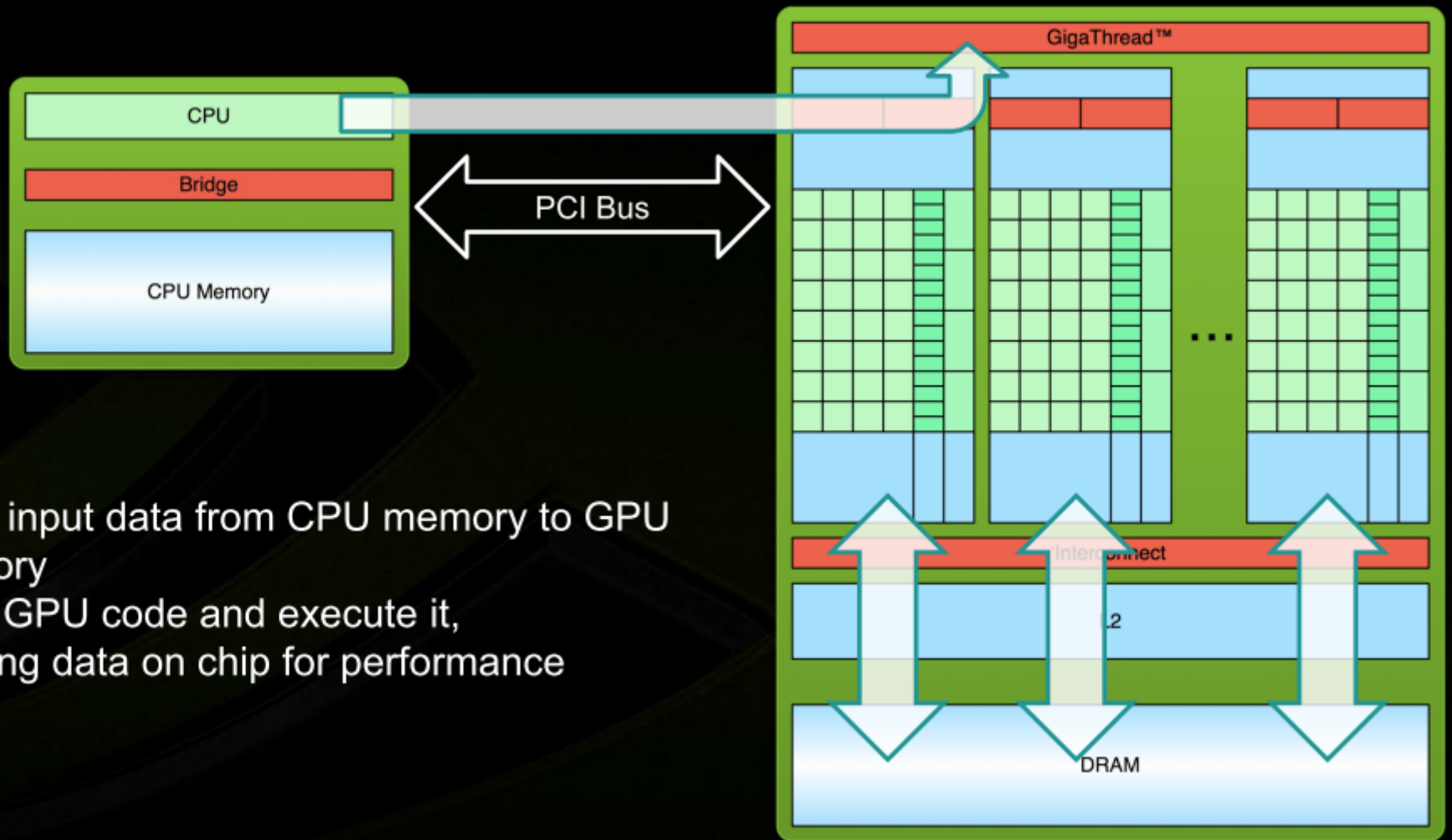


- Alternance entre portions séquentielles (sur host) et parallèles (sur device)
 - Le code host invoque le code device pour rentrer dans des sections parallèles

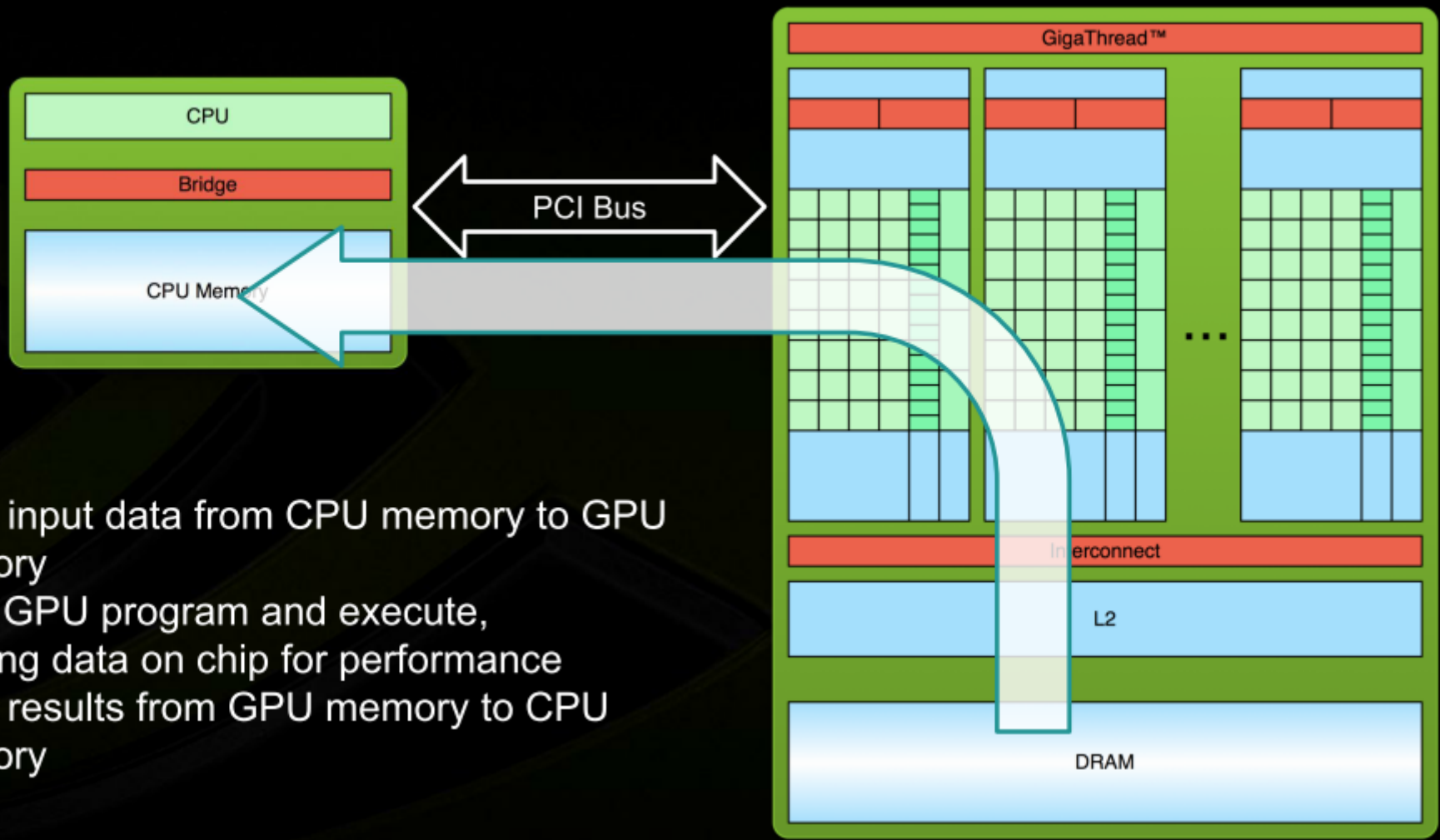
Flot d'opérations



Flot d'opérations



Flot d'opérations



Hello World

- Basiquement : on compile avec le compilateur nvidia
 - Ici une seule portion séquentielle...

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

Output:

```
$ nvcc  
hello_world.cu  
$ a.out  
Hello World!  
$
```

Hello World v2

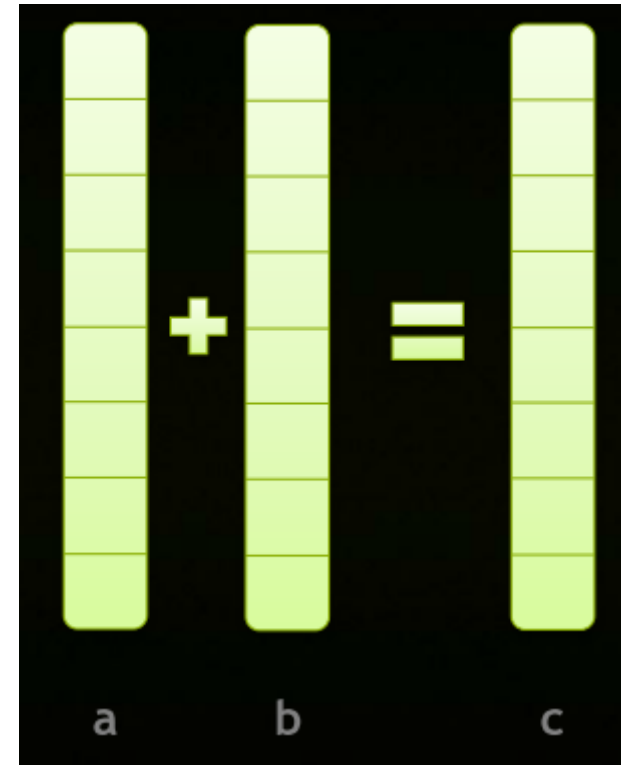
- Avec du code pour interagir avec le device

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Nouvelle syntaxe :
 - `__global__` : une fonction exécutée sur le device, appelée dans le code
 - **Nvcc sépare les fonctions en deux groupes**
 - `<<<1,1>>>`
 - **Définit un appel du code hôte vers le device (oui, il y a trois <<<)**
 - **Retour sur les valeurs plus loin**

Addition de vecteurs

- On souhaite ajouter deux vecteurs
 - En parallèle
- On prépare une fonction pour le GPU



```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

Allocation mémoire

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Ici on passe des pointeurs à la fonction pour le device
 - **Doivent** être valables dans l'espace device
 - Nécessité d'allouer explicitement sur le device
- Modèle mémoire en deux parties
 - Pointeurs « host » ne peuvent être déréférencés que dans le code host, copie/lecture OK partout
 - Pointeurs « device » valables pour être déréférencés dans l'espace du device uniquement
- Allocations et copie : `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - `cudaMemcpy` est bidirectionnel entre mémoire host et device

Retour sur l'exemple

```
int main(void) {  
    int a, b, c;           // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

Main (2)

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```


Configuration du parallélisme

- On pose directement à l'invocation
 - N = nombre d'exécutions en parallèle



The diagram illustrates the configuration of parallelism in a code snippet. It shows two lines of code on a black background. The first line is `add<<< 1, 1 >>> () ;`. A green arrow points from the first `1` to the `N` in the second line, which is `add<<< N, 1 >>> () ;`. The `N` is highlighted in orange.

```
add<<< 1, 1 >>> () ;  
      ↓  
add<<< N, 1 >>> () ;
```

Lancement en parallèle

- Chaque invocation en parallèle = 1 bloc
 - L'ensemble des blocs est la grille (grid)
 - Chaque invocation à son propre indice de bloc
 - Permet d'accéder à son élément propre du tableau

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

Block 0

`c[0] = a[0] + b[0];`

Block 1

`c[1] = a[1] + b[1];`

Block 2

`c[2] = a[2] + b[2];`

Block 3

`c[3] = a[3] + b[3];`

Retour sur le main (tableau)

```
#define N 512

int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Main (2)

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
```

CUDA : conclusion

- Il existe aussi des threads en CUDA
 - Permet la coopération / synchronisations entre tâches
- Modèle CUDA
 - Mémoire séparée pour les traitements parallèles
 - cudaMalloc + copies
 - Importance des allocations contigües
 - Modèle d'exécution host vs device : l'host invoque des fonctions dans le device
 - Compilation séparée des parties
- Des bibliothèques existent qui combinent tous ces avantages
 - E.g; Thrust, une API C++ proche de la STL, qui mappe du code concurrent vers
 - OpenMP, Intel TBB ou NVidia CUDA
 - Cf aussi les outils comme TensorFlow

UE Conclusion

- Contenu dense
 - À digérer avec du travail personnel
 - Corrigés seront disponibles pour réviser
 - On ne sortira pas des sentiers battus en TD ou TME
- L'essentiel pour l'examen (sur feuille)
 - Les threads et les synchronisations :
 - Création, detach, join
 - mutex, condition
 - Future, promise
 - Processus et IPC
 - Signaux, Pipe
 - Shared Memory et Sémaphore IPC
 - Sockets
 - modèles connectés client serveur
 - datagramme
 - Savoir se plier à une organisation OO du code fournie