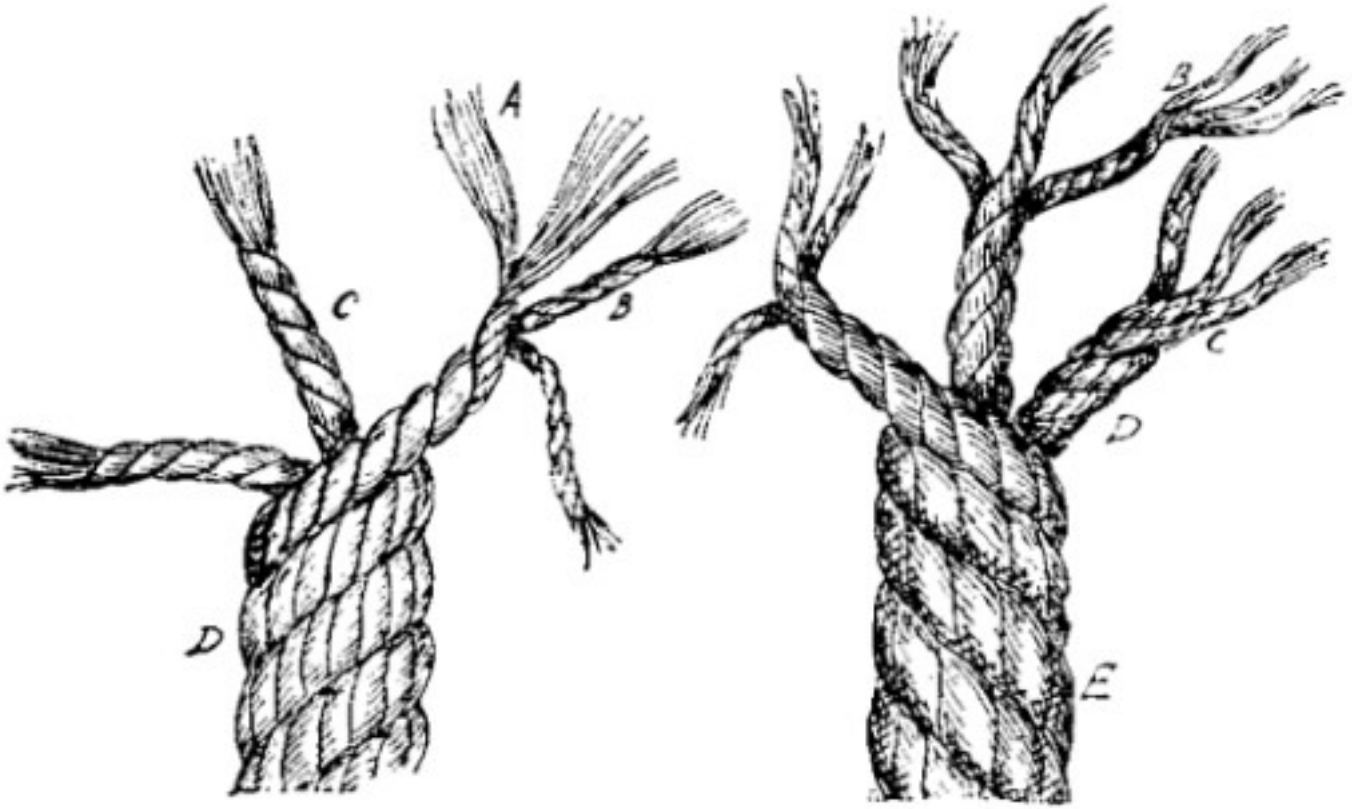
Introduction à la programmation parallèle multi-cœurs



Guillermo Andrade B. Guillermo.Andrade@inria.fr
Service d'Expérimentation et Développement

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
INRIA
centre de recherche RENNES - BRETAGNE ATLANTIQUE

Plan du cours

- 1ère partie : Multi-taches et processus légers
- 2ème partie : Machine parallèle et Multi-cœurs
- 3ème partie : GPUs et CUDA
- 4ème partie : OpenCL
- 5ème partie : Exercice de synthèse

Plan 1ère partie

- Introduction
  - Loi de Moore et processeurs
  - Vers la machine parallèle
- Multi-Taches
  - Processus
  - Ordonnanceur
  - Accès concourants
- Exercice

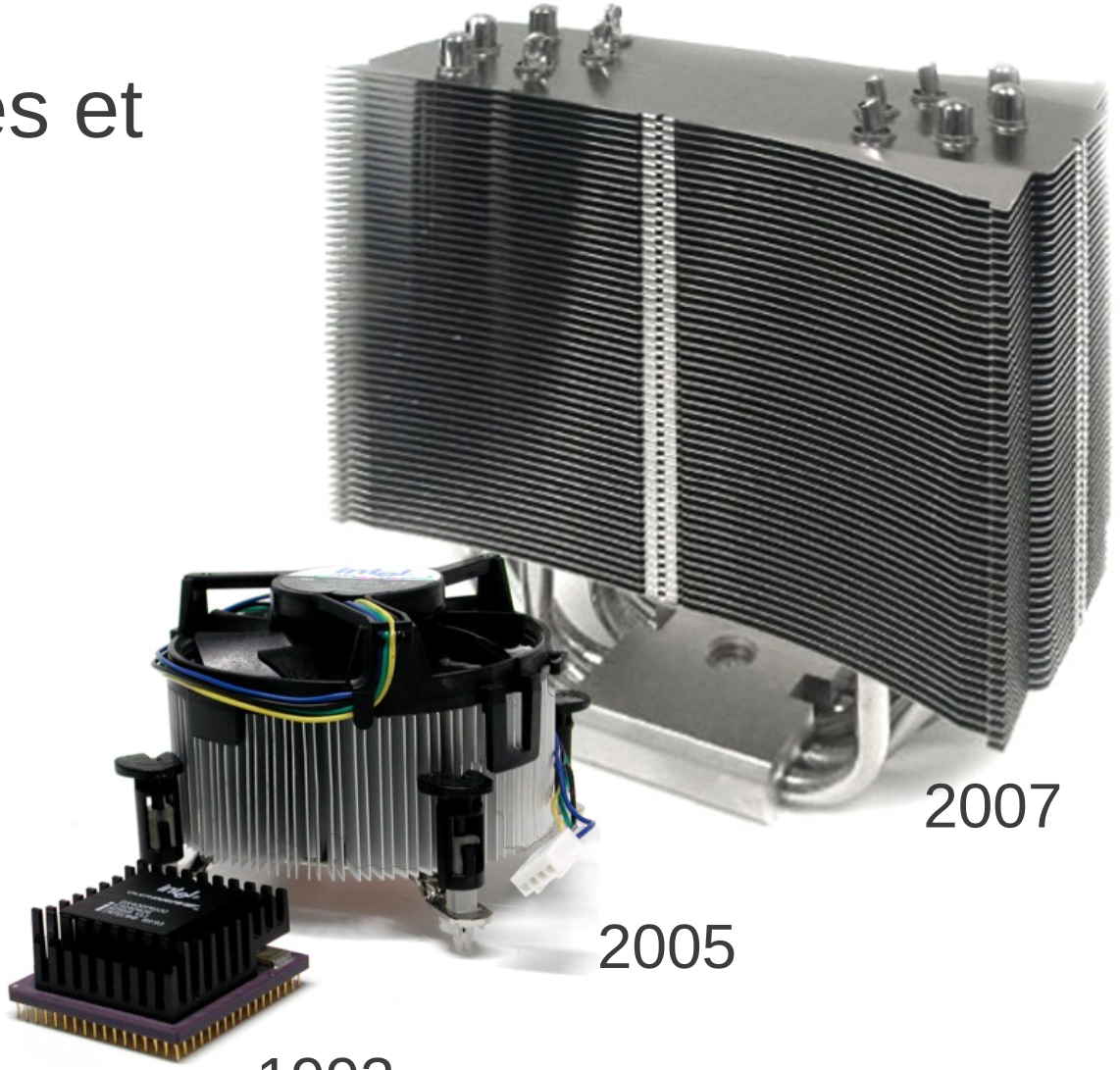
Loi de Moore et Processeurs

- Limites énergétiques et de refroidissement

  => limite de la fréquence de processeurs

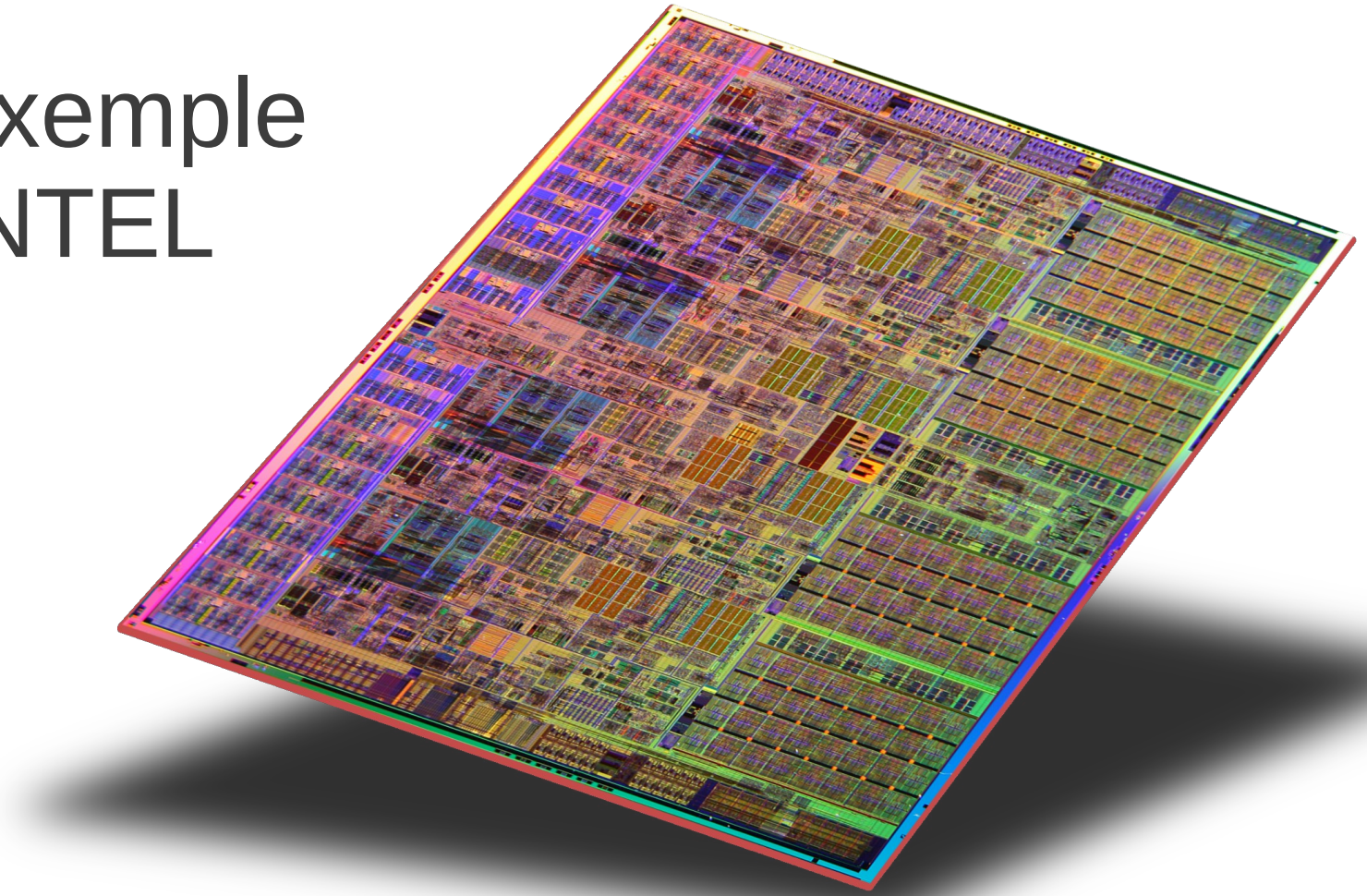
- Pour augmenter la puissance :

  => augmenter le parallélisme



La marche vers le parallélisme

- L'exemple d'INTEL



Intel Core i7

La marche vers le parallélisme

IF (Instruction Fetch)

ID (Instruction Decode)

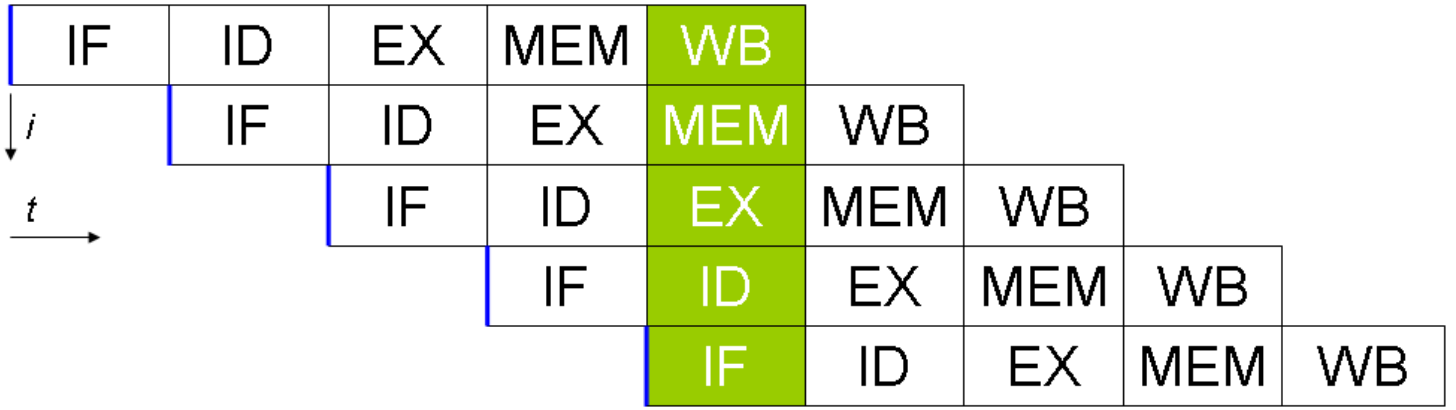
EX (Execute)

MEM (Memory)

WB (Write Back)

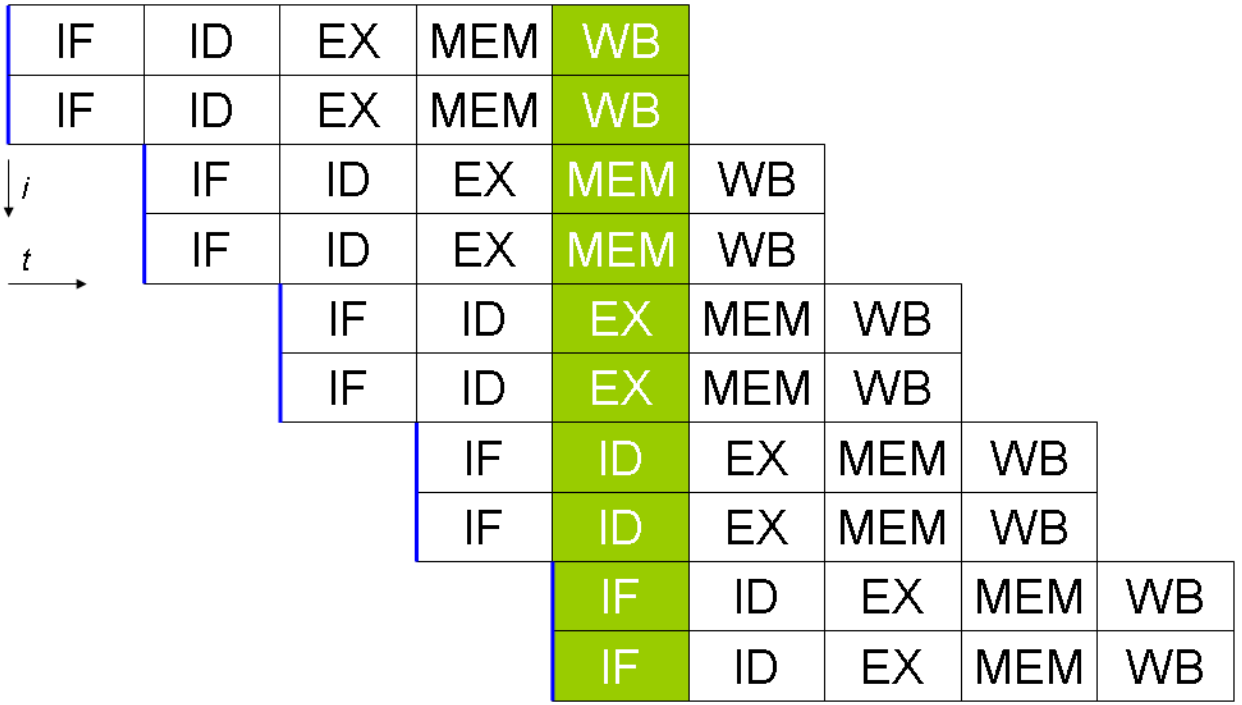
La marche vers le parallélisme

- **i486**
  - pipeline



La marche vers le parallélisme

- i486
- **Pentium**
  - Super-scalaire



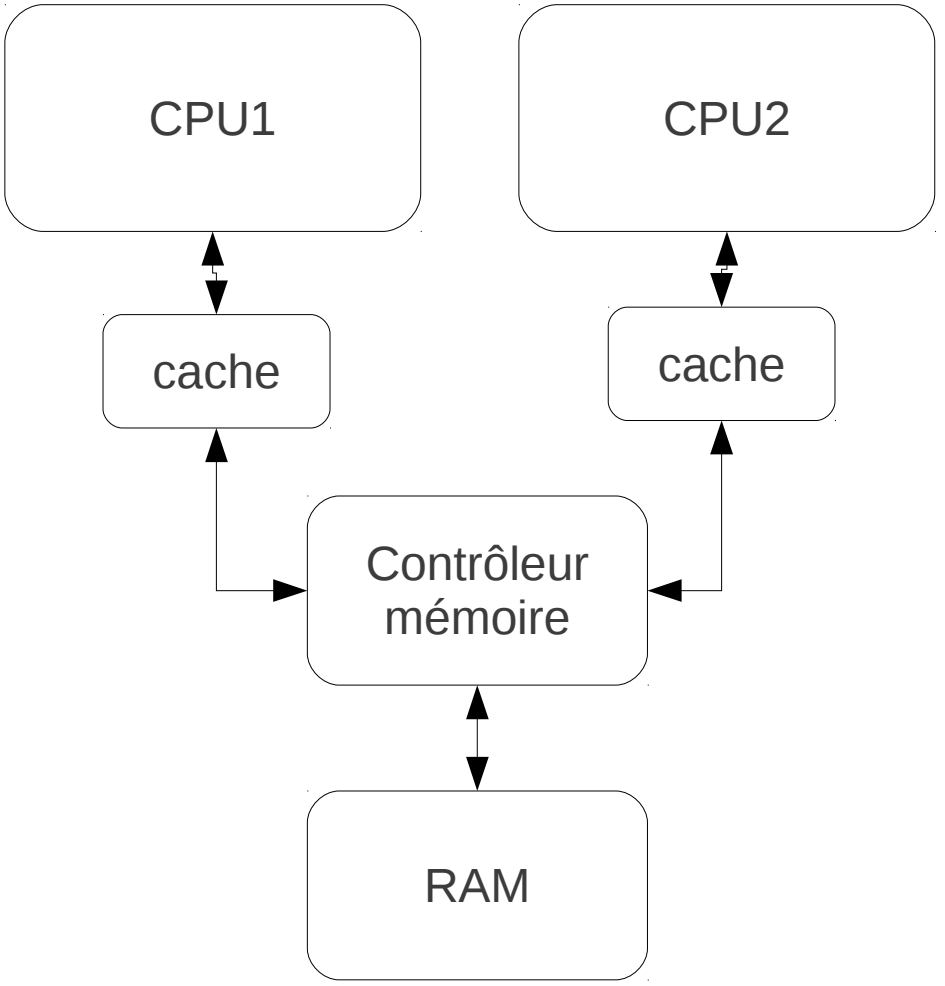
La marche vers le parallélisme

- i486
- Pentium
- **Pentium MMX**
  - Instructions SIMD entiers

```
{                {                {
X1,              X2,              X1 op X2,
Y1,  op          Y2,      =       Y1 op Y2,
Z1,              Z2,              Z1 op Z2,
W1               W2               W1 op W2
}                }                }
```

La marche vers le parallélisme

- i486
- Pentium
- Pentium MMX
- **Pentium PRO**
  - Symetric Multi-processor (SMP)

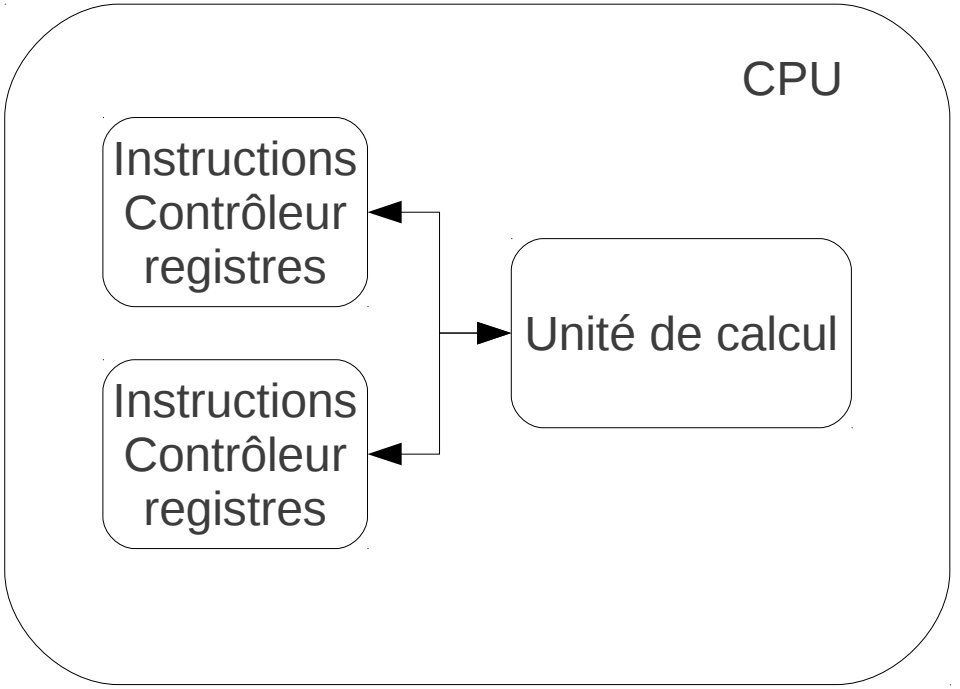


La marche vers le parallélisme

- i486
- Pentium
- Pentium MMX
- Pentium PRO
- **Pentium III**
  - SSE (SIMD 4 flottants 32 bits)

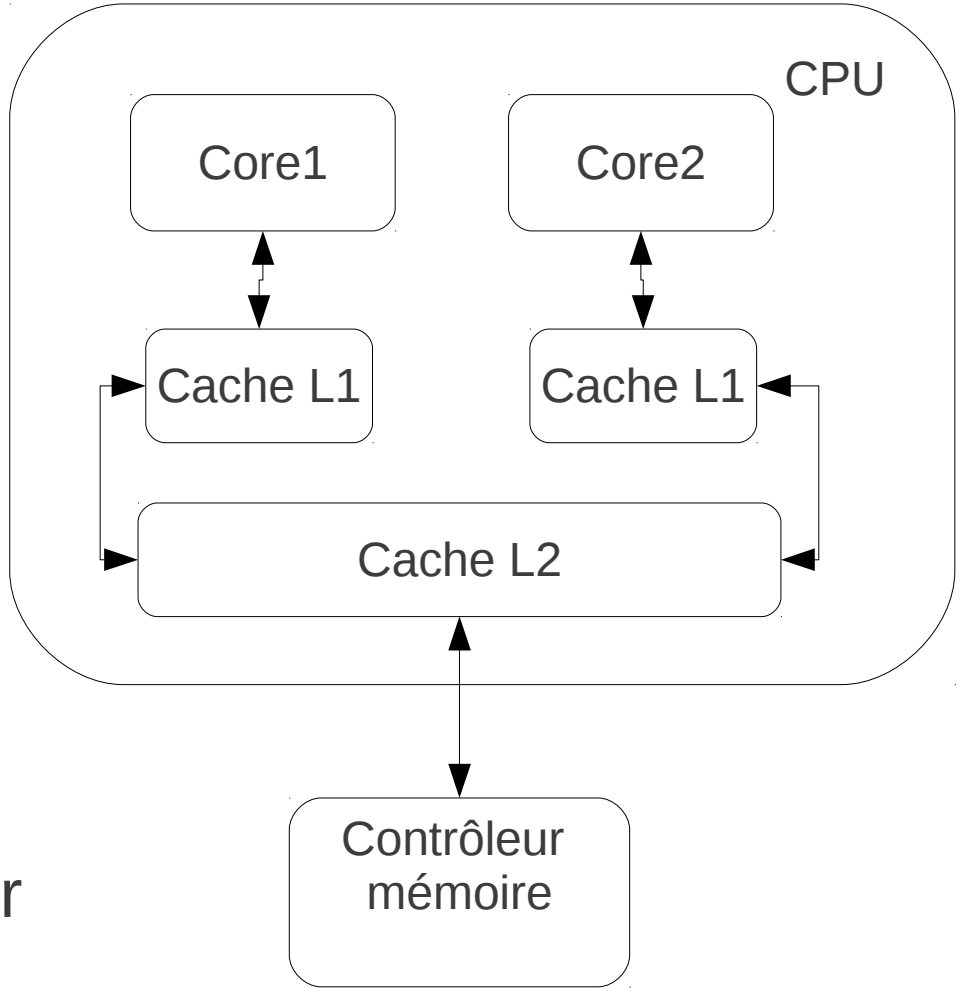
La marche vers le parallélisme

- i486
- Pentium
- Pentium MMX
- Pentium PRO
- Pentium III
- **Pentium 4**
  - Hyper-threading



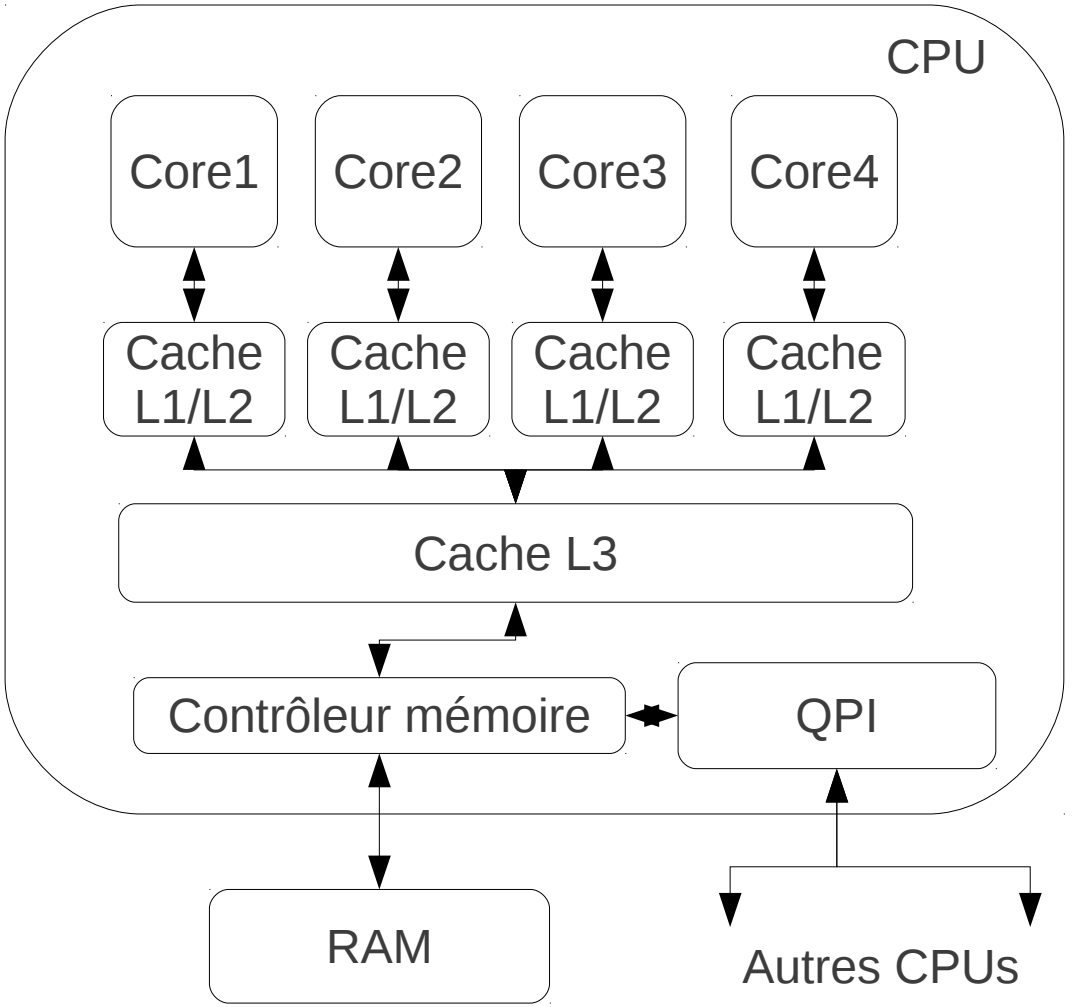
La marche vers le parallélisme

- i486
- Pentium
- Pentium MMX
- Pentium PRO
- Pentium III
- Pentium 4
- **Core 2**
  - Bi-cœur et double bi-cœur
  - Super scalaire 4-voies



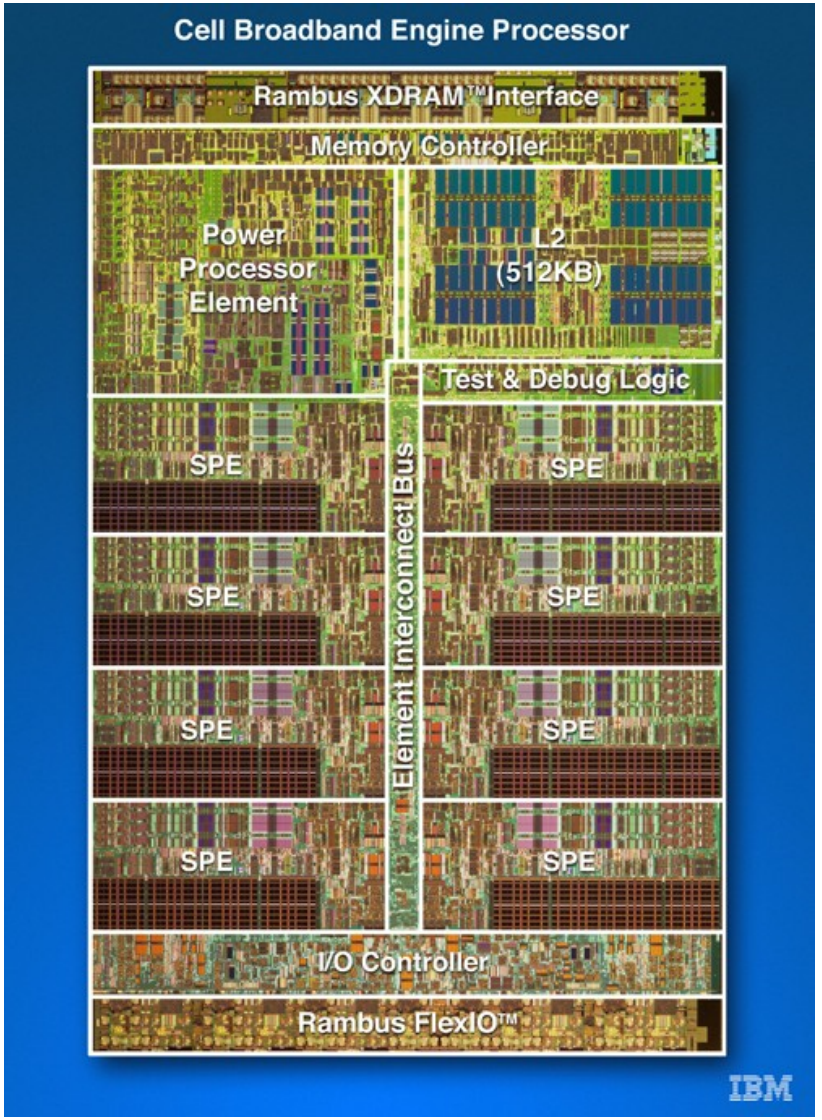
La marche vers le parallélisme

- **Core i7**
  - Vrai quadri-cœur
  - Hyper-threading
    - 8 threads en simultanée
  - QPI : Inter-connection point à point entre CPUs
    - Jusqu'à 4 CPU soit 4x4x2=32 threads en simultanée!



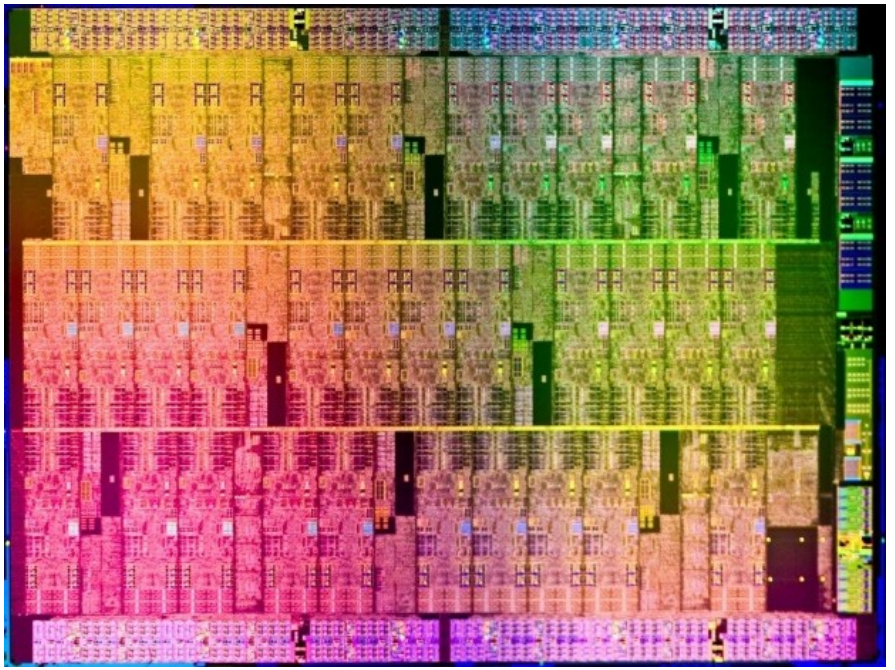
Les CPU Hybrides

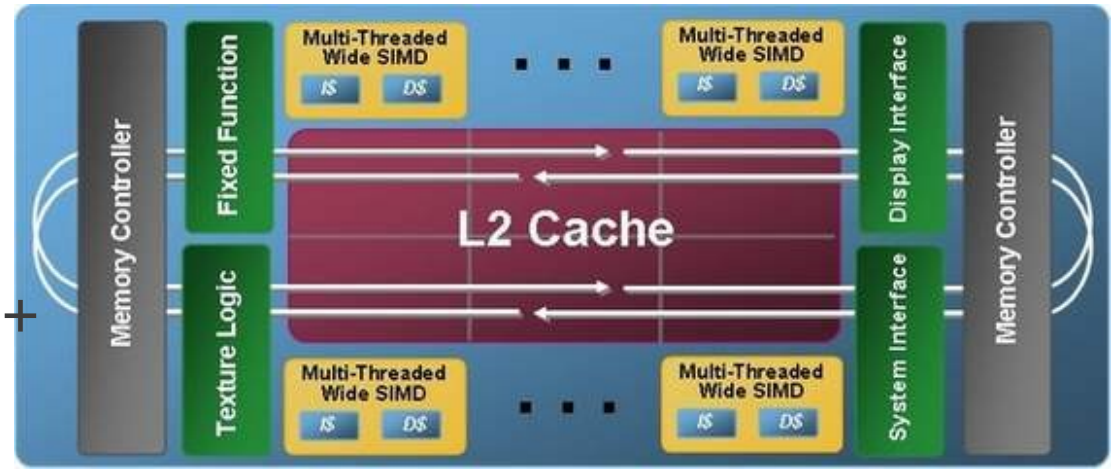
- CellBE (IBM, TOCHIBA, SONY)
  - Un cœur « classique » ~ PowerPC bi-cœur
  - 8 cœurs SPE :
    - SIMD 4 voies
    - Équivalent à SSE ou AltiVec
  - 200 Gflops (2 * 10^11 opérations / sec)



Les CPU Hybrides

- CellBE
- Fusion AMD CPU+GPU
  - Knights Ferry INTEL (ancien Larrabee)
  - Instructions x86
  - SIMD à 16 voies
  - 32 cœurs
  - 2 TFLops
  - Hyper-threading x4
  - Programmable en C/C++ via ICC





1ère Conclusion:

- Exploiter la pleine puissance des processeurs d'aujourd'hui et de demain nécessitera de faire du parallélisme.

Multi-taches

- Pourquoi ?
  - Plusieurs Taches
    - longues
    - Mais autonomes
      - Lancer et attendre
    - Interruptibles ?
  - Réactivité nécessaire
    - Interaction utilisateur
    - Client / serveur
  - Plusieurs unités d'exécution

Multi-taches

- Pourquoi ?
  - Plusieurs Taches
    - longues
    - Mais autonomes
      - Lancer et attendre
    - Interruptibles ?
  - Réactivité nécessaire
    - Interaction utilisateur
    - Client / serveur
  - Plusieurs unités d'exécution

Exemples familiers:

- Chauffer le four, mettre la table, placer le gâteau dans le four , ...
- Le minuteur sonne, le téléphone sonne, ...
- Chéri peux tu répondre, je m'occupe du four.

Multi-taches

- Comment?
  - Sauvegarde du contexte
  - Gestion des ressources partagées
    - Ordonnanceur
      - CPU
    - Ressources Critiques

Multi-taches

- Comment?
  - Sauvegarde du contexte
  - Gestion des ressources partagées
    - Ordonnanceur
      - CPU
    - Ressources Critiques

Exemples familiers:

- Qu'est ce que j'étais en train de faire ?
- Je n'ai que deux bras
  - Que dois je faire en premier?
  - Je sors le gâteau du four, je ne peux pas répondre au téléphone

Contexte d'exécution de la tache

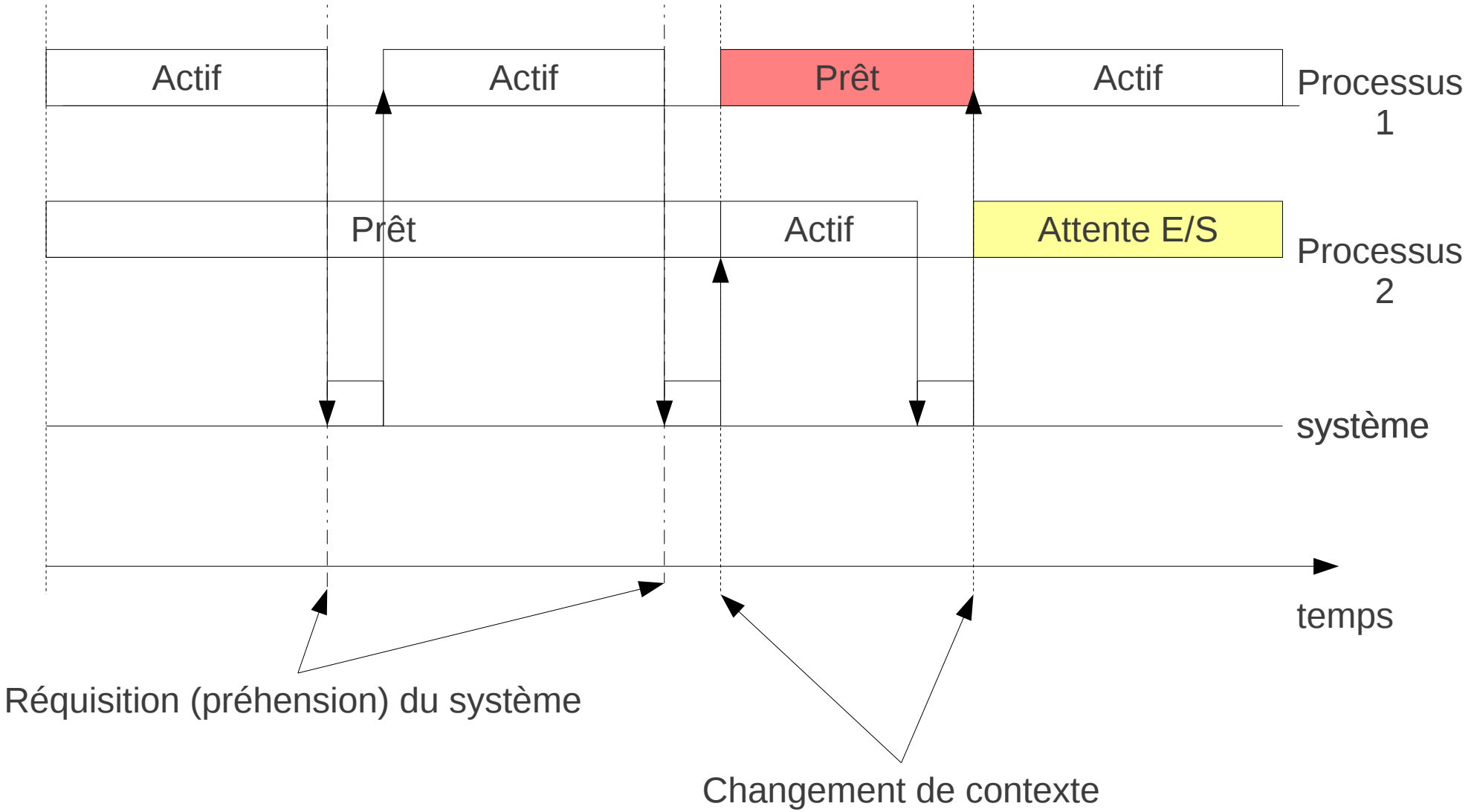
```
int f(int c){
if (c>1)
   return c*f(c-1);
else
   return 1;
}
```

- Piles d'instructions
- Registres
- Compteur d'instruction
- Mémoire utilisée
- Fichiers ouverts

Partage de ressources: le CPU

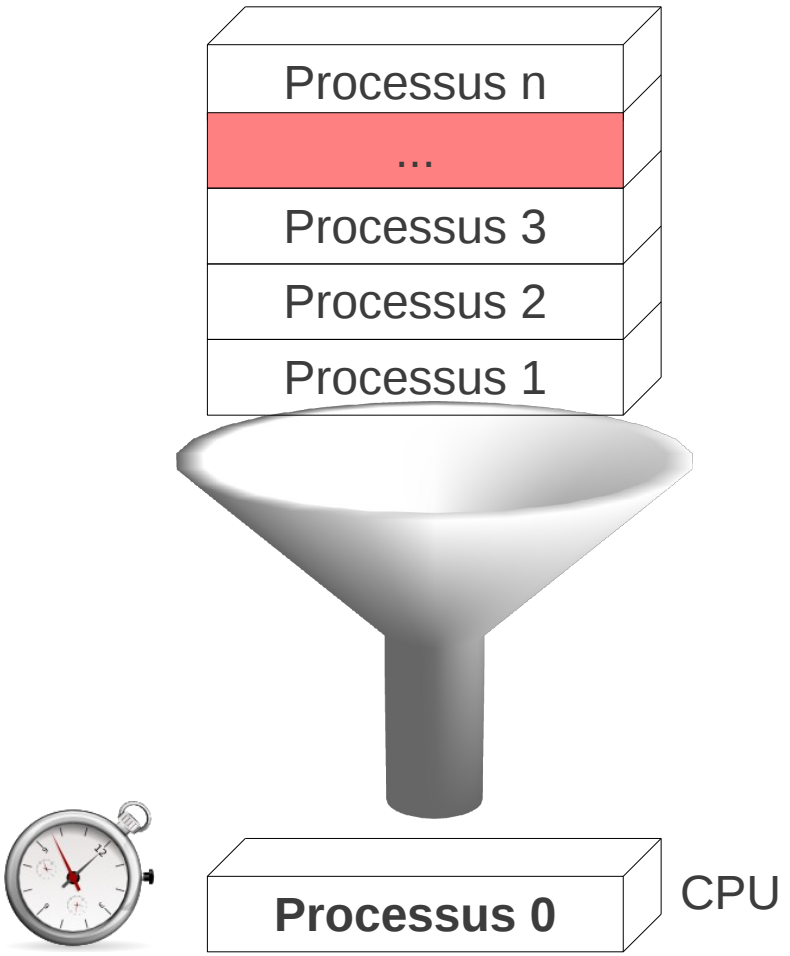
- N processus et M CPU
  - N > M => utilisation d'un **ordonnanceur**
- Ordonnanceur
  - Géré par le noyau d'OS au plus près de la machine
    - Accès entrées/Sorties
    - Interruptions
    - Réquisition (Préhension)
  - Partager au mieux le temps CPU entre les processus
    - Politique de répartition

Partage du CPU



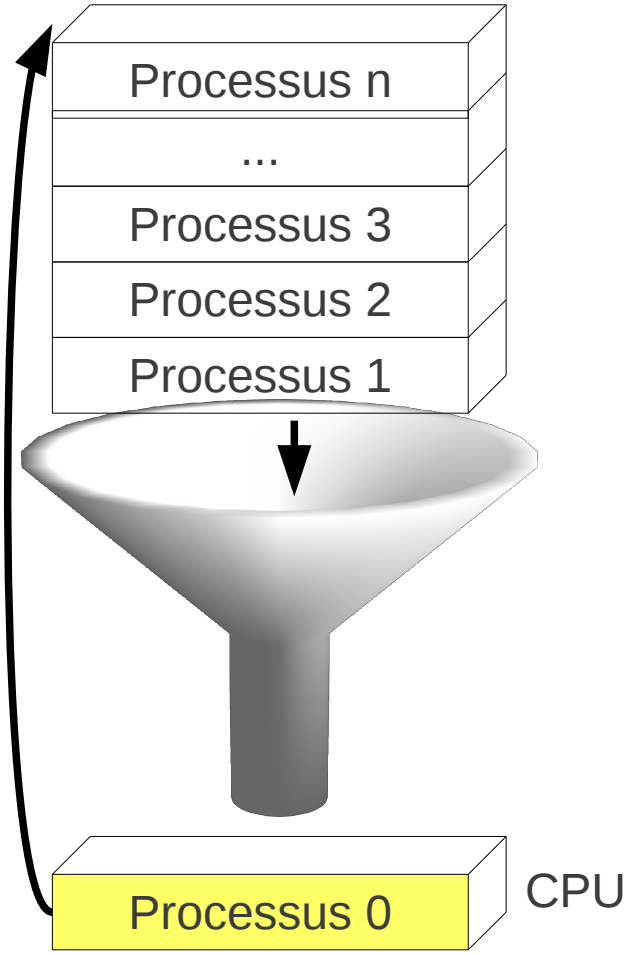
L'ordonnanceur

- Politique de répartition du temps CPU
  - Round-Robin (chaque un son tour)
    - Une queue de processus en attente
    - Un processus actif sur un *quanta* de temps



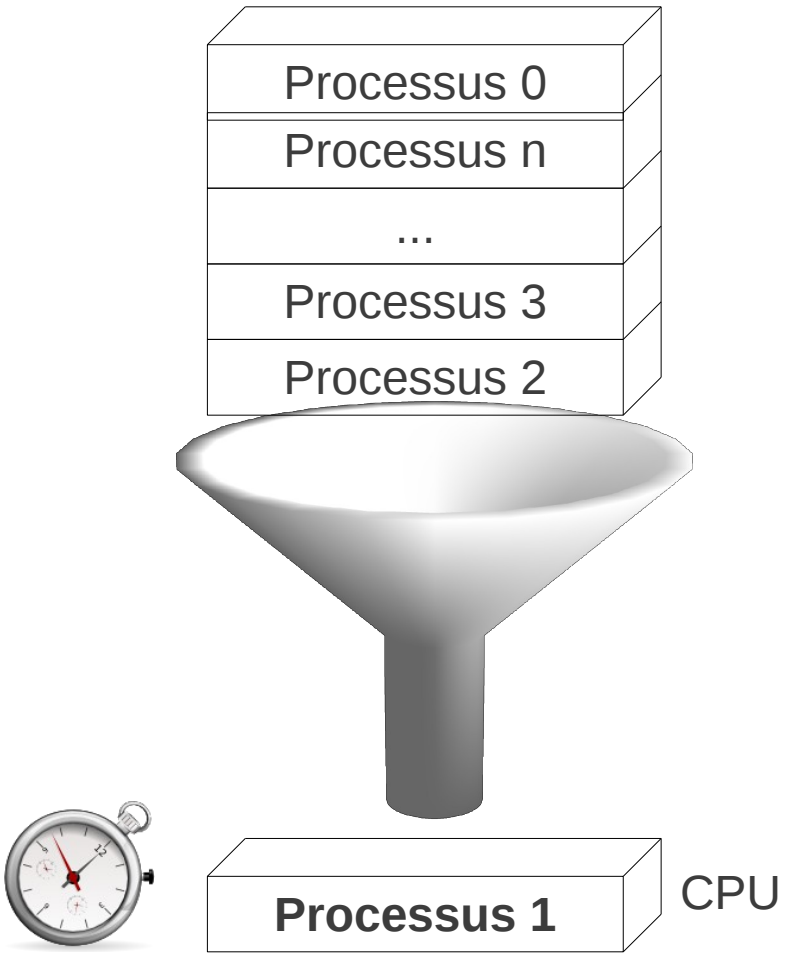
L'ordonnanceur

- Politique de répartition du temps CPU
  - Round-Robin (chaque un son tour)
    - Le processus rend le CPU (ex. accès disque) ou le perd (quanta dépassé)
    - Ordonnanceur change le contexte actif



L'ordonnanceur

- Politique de répartition du temps CPU
  - Rond-Robin (chaque un son tour)
    - Le premier processus de la queue devient actif
    - Le précédent est placé en dernier dans la queue



L'ordonnanceur

- Politique de répartition du temps CPU
  - Round-Robin
    - Taille du quanta = Granularité
    - petit quanta => temps perdu en changements de contextes
    - Gros quanta => réactivité faible

L'ordonnanceur

- Politique de répartition du temps CPU
  - Round-Robin avec Priorités
    - Plusieurs queues avec des probabilités différentes de passer dans le CPU
    - Chaque queue à un quanta de taille différente
      - Forte priorité => petit quanta
      - Faible priorité => grand quanta

L'ordonnanceur

- Politique de répartition du temps CPU
  - Rond-Robin avec Priorités dynamiques
    - Si un processus ne libère pas le CPU avant la fin du quanta il perd de la priorité
    - Si le processus bloque/libère le CPU pour un accès entrée/sortie il gagne de la priorité
    - Utilisé par Windows

L'ordonnanceur

- Politique de répartition du temps CPU
  - Completely Fair Scheduler (ordonnanceur complètement équitable en anglais ou CFS)
    - Utilise un arbre binaire « rouge/noir » pour trier rapidement les processus.
    - Le critère de tri est la différence entre le temps idéal du processus (un cpu par processus) et le temps réellement occupé.
    - Utilisé à partir du noyaux de Linux 2.6.23

Processus lourd et léger

- Processus **lourd** a des ressources indépendantes:
  - Piles d'instructions
  - Registres
  - Compteur d'instructions
  - Mémoire utilisée
  - Fichiers ouverts
  - État du processus

Processus lourd et léger

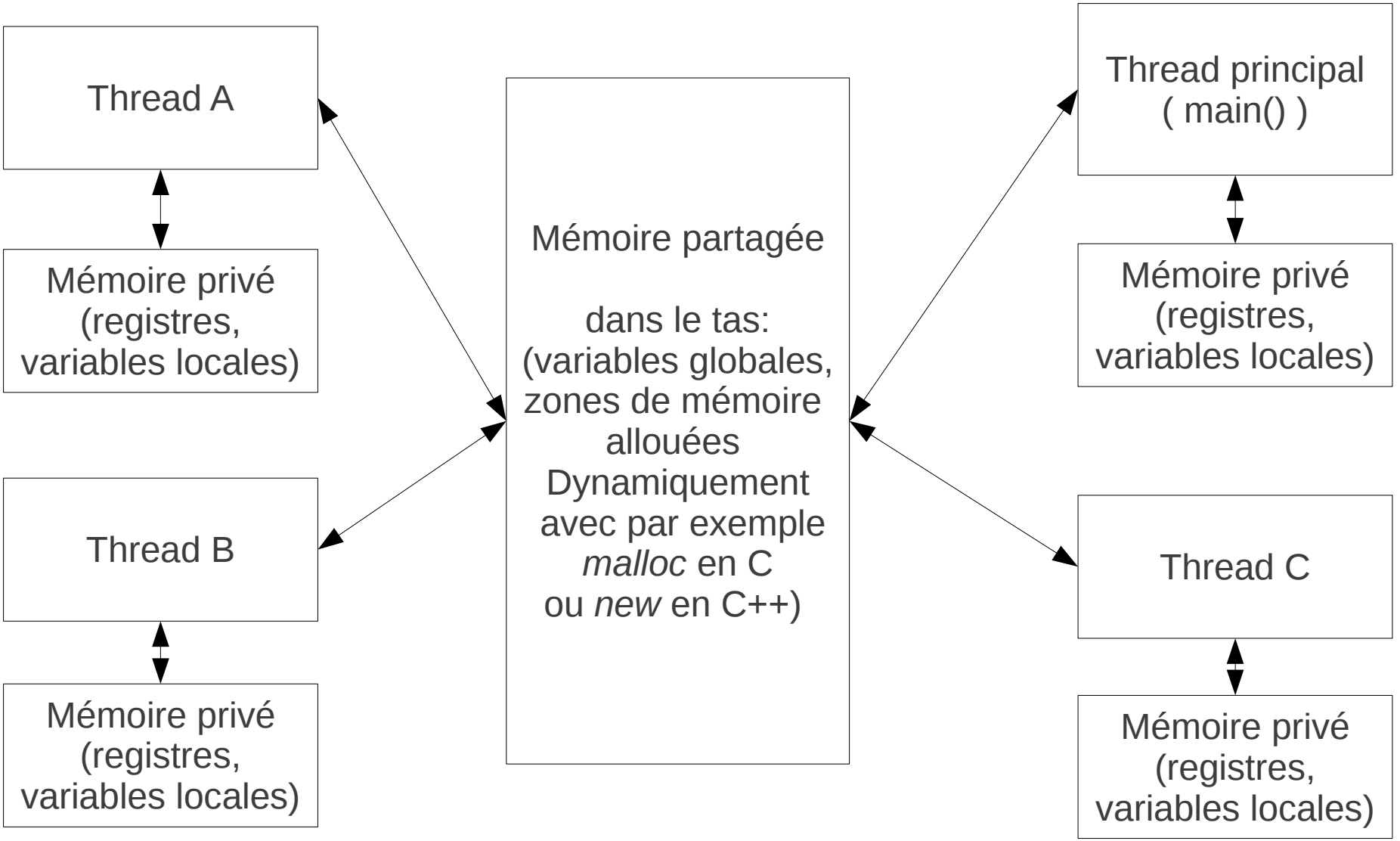
- Processus **lourd** a des ressources indépendantes:
  - Piles d'instructions
  - Registres
  - Compteur d'instructions
  - Mémoire utilisée
  - Fichiers ouverts
  - État du processus

- Processus **léger** a des ressources indépendantes:
  - Piles d'instructions
  - Registres
  - Compteur d'instructions
- Et des **ressources partagées**:
  - Mémoire utilisée
  - Fichiers ouverts
  - État du processus

Processus lourd et léger

- Processus Lourd
  - Indépendance d'exécution
  - Changement de contexte moins rapide
  - Communication **externe** entre processus
- Processus léger *Thread* (fil)
  - Changement de contexte rapide
  - Attaché à un processus lourd « père »
  - Communication via la mémoire partagée
  - Possibilités d'incohérence mémoire

Partage de la mémoire



Multi-taches: API Pthread

- Gestion des processus légers (threads)
- Simple
- Normalisé IEEE POSIX 1003.1c
- Portable
  - Mis à disposition par la plupart des OS (Mac OS X, Linux, SunOS, etc.)
  - Sur MS Windows, 2 options:
    - Services for UNIX (SFU) 3.5
    - Open Source POSIX Threads for Win32

Définition du Thread

Accès à l'API

```
#include<pthread.h>
void* momThread(
                void *arg){
// exemple:
float* a= (float
*)arg;
...
pthread_exit((void*)co
de);
return code;
}
```

Définition du Thread

Fonction contenant le code du thread

```
#include<pthread.h>
void* momThread(
                void *arg){
// exemple:
float* a= (float *)arg;
...
pthread_exit((void*)code);
return code;
}
```

Définition du Thread

Paramètre pointeur générique passé à la création du thread

```
#include<pthread.h>

void* momThread(
                void *arg){

// exemple:

float* a= (float
*)arg;

...

pthread_exit((void*)co
de);

return code;

}
```

Définition du Thread

Il peut être transformé en un autre type (ici en pointeur de flottant)

```
#include<pthread.h>
void* momThread(
                void *arg){
// exemple:
float* a= (float *)arg;
...
pthread_exit((void*)code);
return code;
}
```

Définition du Thread

```
#include<pthread.h>
void* momThread(
                void *arg){
// exemple:
float* a= (float
*)arg;
...
pthread_exit((void*)co
de);
return code;
}
```

Fonction de fin du thread

Définition du Thread

Peut renvoyer une valeur de retour de type pointeur générique

```
#include<pthread.h>
void* momThread(
                void *arg){
// exemple:
float* a= (float *)arg;
...
pthread_exit((void*)code);
return code;
}
```

Création d'un thread

Fonction qui crée et lance le thread

```
float A[100];
int main(){
pthread_t thread;
int
rc=pthread_create(
        & thread,
        NULL,
        monThread,
        (void *)A );
...
}
```

Création d'un thread

Déclaration de l'identifiant du thread

```
float A[100];
int main(){
pthread_t thread;
int
rc=pthread_create(
          & thread,
          NULL,
          monThread,
          (void *)A );
...
}
```

Création d'un thread

remplissage de l'identifiant du thread

```
float A[100];
int main(){
pthread_t thread;
int
rc=pthread_create(
 & thread,
 NULL,
 monThread,
 (void *)A );
...
}
```

Création d'un thread

Passage des attributs du thread (ici valeurs par défaut)

```
float A[100];
int main(){
pthread_t thread;
int
rc=pthread_create(
                & thread,
                NULL,
                monThread,
                (void *)A );
...
}
```

Création d'un thread

Passage de la fonction initiale du thread

```
float A[100];
int main(){
pthread_t thread;
int
rc=pthread_create(
                & thread,
                NULL,
                monThread,
                (void *)A );
...
}
```

Création d'un thread

Passage d'un paramètre au thread (ici un tableau de flottants en mémoire partagée)

```
float A[100];
int main(){
pthread_t thread;
int
rc=pthread_create(
          & thread,
          NULL,
          monThread,
          (void *)A );
...
}
```

Fin et jonction de threads

Fonction permettant d'attendre la fin de l'exécution du thread lancé précédemment

```
int main(){
pthread_t thread;
Pthread_create(
          & thread,
          ... );
...
void *res;
pthread_join(thread,
&res);
...
}
```

Fin et jonction de threads

```
int main(){
pthread_t thread;
Pthread_create(
          & thread,
          ... );
...
void *res;
pthread_join(thread,
&res);
...
}
```

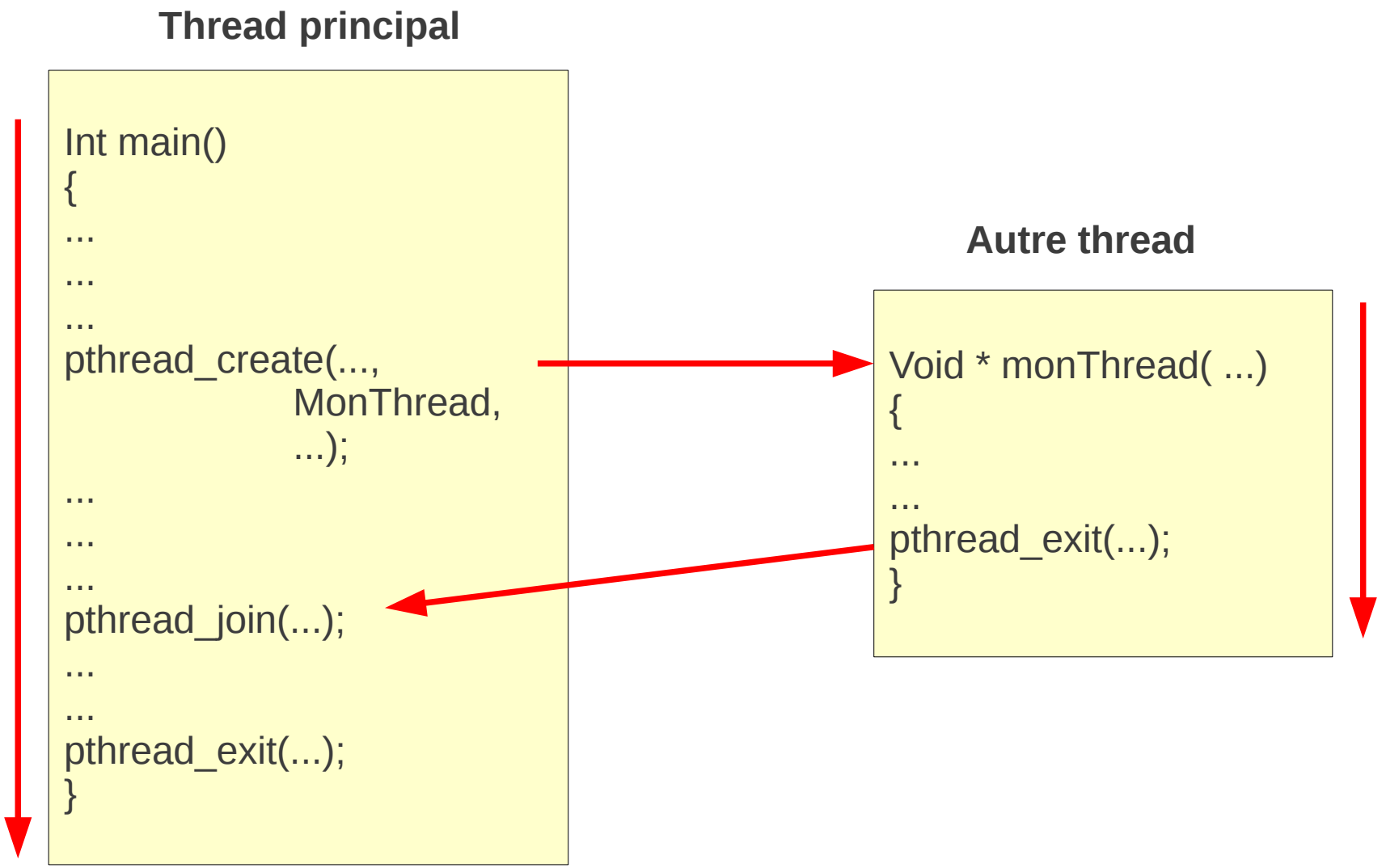
Fonction permettant d'attendre la fin de l'exécution du thread lancé précédemment

Fin et jonction de threads

```
int main(){
pthread_t thread;
Pthread_create(
          & thread,
          ... );
...
void *res;
pthread_join(thread,
&res);
...
}
```

Récupération d'une valeur de retour du thread (peut être remplacé par NULL si elle n'est pas utile)

Création en fin du thread



Mutex

Thread A

```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
Y= X * 2;
X=Y+1;
..
...
pthread_exit(...);
}
```

Thread B

```
Void * ThreadA( ...)
{
...
int Z=0;
Z= X * 2;
X=Z-1;
...
pthread_exit(...);
}
```

X=10 Z=0 Y=0

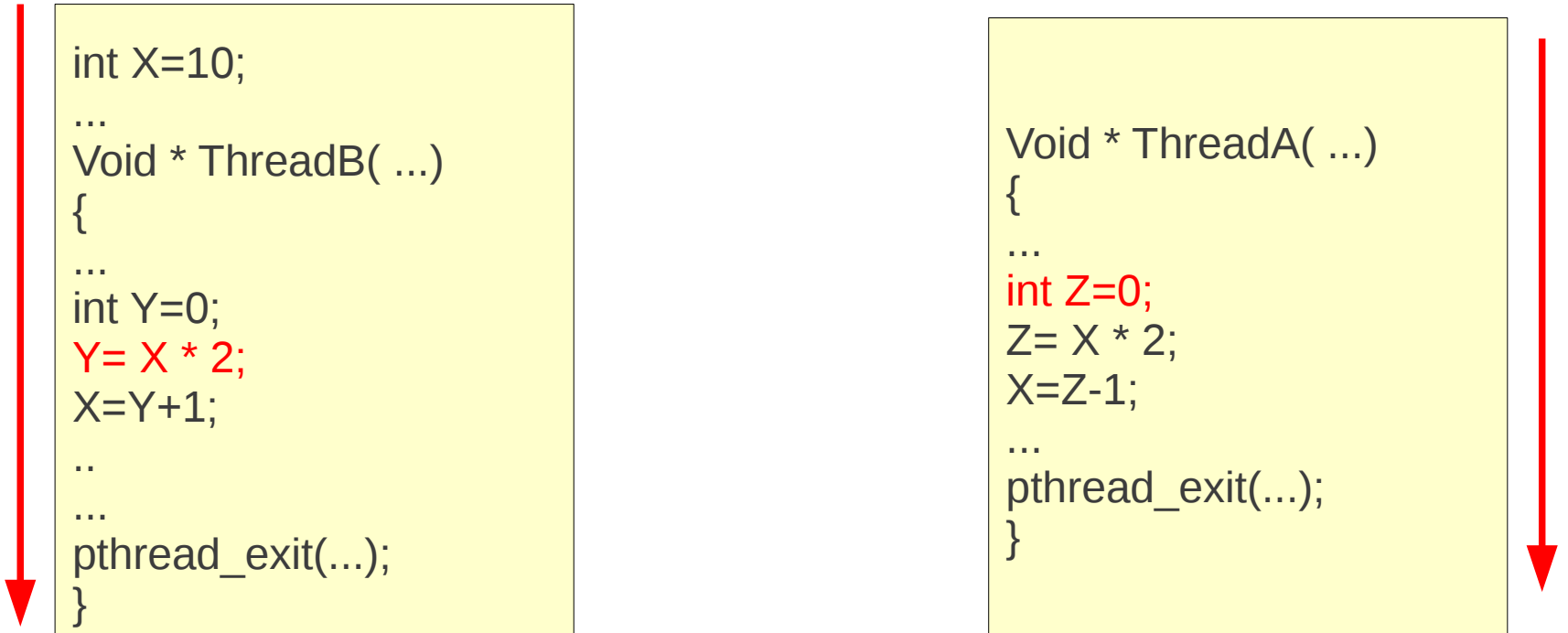
Mutex

Thread A

```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
Y= X * 2;
X=Y+1;
..
...
pthread_exit(...);
}
```

Thread B

```
Void * ThreadA( ...)
{
...
int Z=0;
Z= X * 2;
X=Z-1;
...
pthread_exit(...);
}
```



X=10 Z=0 Y=20

Mutex

Thread A

```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
Y= X * 2;
X=Y+1;
..
...
pthread_exit(...);
}
```

Thread B

```
Void * ThreadA( ...)
{
...
int Z=0;
Z= X * 2;
X=Z-1;
...
pthread_exit(...);
}
```

X=10 Z=20 Y=20

Mutex

Thread A

```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
Y= X * 2;
X=Y+1;
..
...
pthread_exit(...);
}
```

Thread B

```
Void * ThreadA( ...)
{
...
int Z=0;
Z= X * 2;
X=Z-1;
...
pthread_exit(...);
}
```

X=19 Z=20 Y=20

Mutex

Thread A

```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
Y= X * 2;
X=Y+1;
..
...
pthread_exit(...);
}
```

Thread B

```
Void * ThreadA( ...)
{
...
int Z=0;
Z= X * 2;
X=Z-1;
...
pthread_exit(...);
}
```

X=21 Z=20 Y=20

Mutex

- Mutex = Exclusion Mutuelle
  - Éviter les conflits de ressources
  - mettre un verrou
    - Autres threads qui essayent de mettre le verrou sont bloqués

    Libérer le verrou
    - Un autre thread peut alors mettre le verrou

Mutex

- Mutex = Exclusion Mutuelle
  - Éviter les conflits de ressources
  - mettre un verrou
    - Autres threads qui essayent de mettre le verrou sont bloqués

    Libérer le verrou
    - Un autre thread peut alors mettre le verrou
- Exemple familier:
  - Les toilettes
    - Une seule personne à la fois
    - Le verrou ne peut être mis que si les toilettes sont libres
    - Une fois le verrou mis les autres attendent (dehors)
    - Le verrou libéré permet à une autre personne de verrouiller les toilettes

Mutex

- Déclaration
- Initialisation
- Verrouiller
- Déverrouiller
- Test du verrou =

```
pthread_mutex_t a_mutex;

pthread_mutex_init(&a_mutex,
                   NULL);

pthread_mutex_lock(&a_mutex);

pthread_mutex_unlock(&a_mutex);

if( pthread_mutex_trylock(&a_mutex)!
                                    EBUSY )

  /*OK*/
else
  /* Faire autre chose en
  attendant*/
```

Mutex

Thread A

```
int X=10;
...
Void * ThreadB( ...)
{
...
int Y=0;
pthread_mutex_lock(&a_mutex);
Y= X * 2;
X=Y+1;
pthread_mutex_unlock(&a_mutex);
..
...
pthread_exit(...);
}
```

Thread B

```
Void * ThreadA( ...)
{
...
int Z=0;
pthread_mutex_lock(&a_mutex);
Z= X * 2;
X=Z-1;
pthread_mutex_unlock(&a_mutex);
...
pthread_exit(...);
}
```

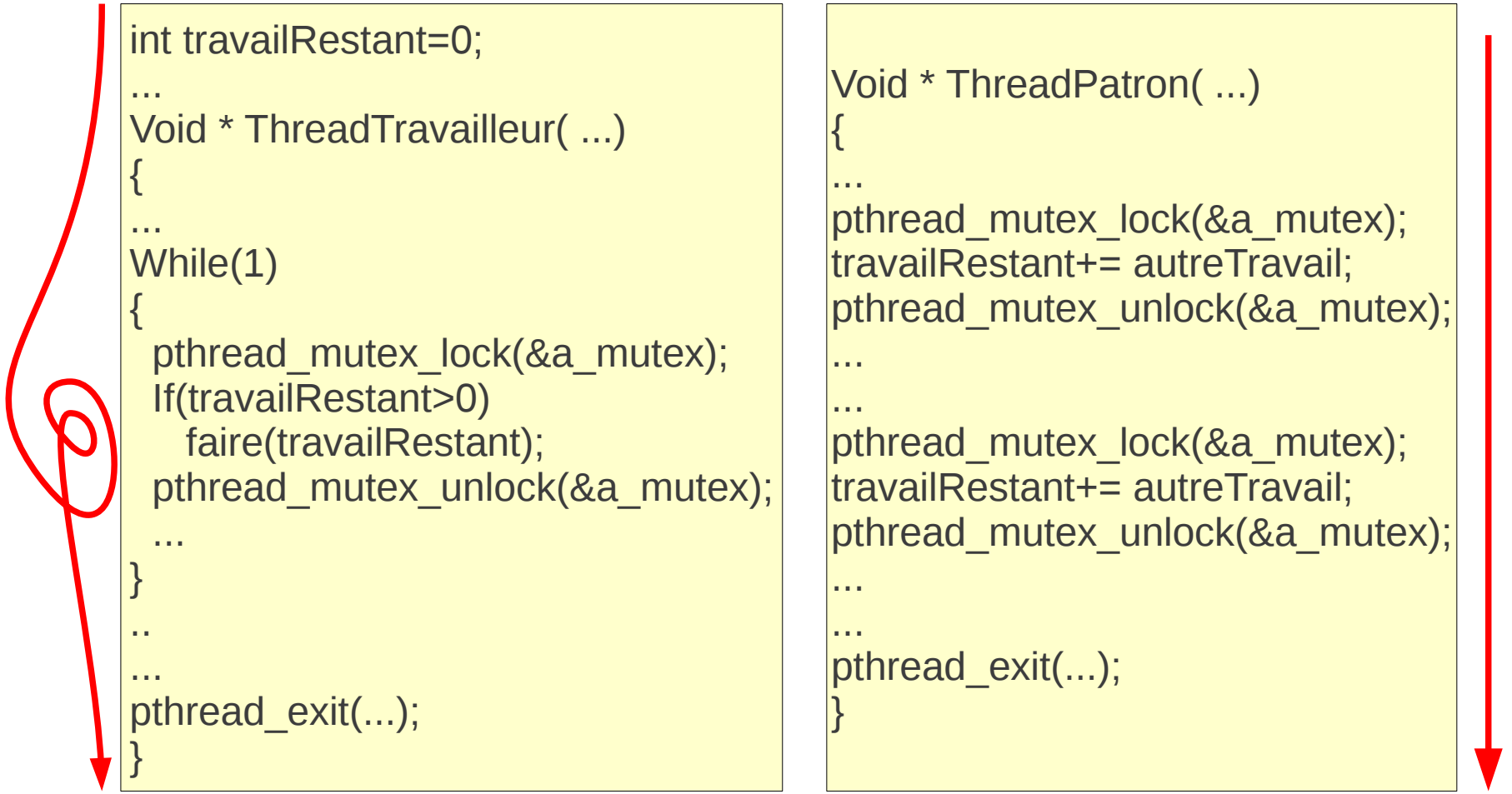
Mutex et condition

Thread Travailleur

```
int travailRestant=0;
...
Void * ThreadTravailleur( ...)
{
...
While(1)
{
  pthread_mutex_lock(&a_mutex);
  If(travailRestant>0)
     faire(travailRestant);
  pthread_mutex_unlock(&a_mutex);
  ...
}
..
...
pthread_exit(...);
}
```

Thread Patron

```
Void * ThreadPatron( ...)
{
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_exit(...);
}
```



Mutex et condition

Thread Travailleur

```
int travailRestant=0;
...
Void * ThreadTravailleur( ...)
{
...
While(1)
{
  pthread_mutex_lock(&a_mutex);
  If(travailRestant>0)
     faire(travailRestant);
  pthread_mutex_unlock(&a_mutex);
  ...
}
..
...
pthread_exit(...);
}
```

100% CPU !!!

Thread Patron

```
Void * ThreadPatron( ...)
{
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_mutex_unlock(&a_mutex);
...
...
pthread_exit(...);
}
```

Mutex et condition

- Déclaration
  - Initialisation
  - Destruction
- Utilisation
  - Mise en attente
    - Libère le mutex
  - Envoi du signal
    - Verrouille le mutex
  - Envoi à tous

```
pthread_cond_t a_cond;
pthread_cond_init (&a_cond,NULL);
pthread_cond_destroy (&a_cond);
pthread_cond_wait (&a_cond, &a_mutex);
pthread_cond_signal (&a_cond);
pthread_cond_broadcast (&a_cond);
```

Mutex et condition

Thread Travailleur

```
int travailRestant=0;
...
Void * ThreadTravailleur( ...)
{
...
While(1)
{
  pthread_mutex_lock(&a_mutex);
  If(0==travailRestant)
    pthread_cond_wait (&a_cond,
                       &a_mutex);
  faire(travailRestant);
  pthread_mutex_unlock(&a_mutex);
  ...
}
...
pthread_exit(...);
}
```

Thread Patron

```
Void * ThreadPatron( ...)
{
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_cond_signal (&a_cond);
pthread_mutex_unlock(&a_mutex);
...
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_cond_signal (&a_cond);
pthread_mutex_unlock(&a_mutex);
...
...
pthread_exit(...);
}
```

Mutex et condition

Thread Travailleur

```
int travailRestant=0;
...
Void * ThreadTravailleur( ...)
{
...
While(1)
{
  pthread_mutex_lock(&a_mutex);
  If(0==travailRestant)
    pthread_cond_wait (&a_cond,
                       &a_mutex);
  faire(travailRestant);
  pthread_mutex_unlock(&a_mutex);
  ...
}
...
pthread_exit(...);
}
```

Thread Patron

```
Void * ThreadPatron( ...)
{
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_cond_signal (&a_cond);
pthread_mutex_unlock(&a_mutex);
...
...
pthread_mutex_lock(&a_mutex);
travailRestant+= autreTravail;
pthread_cond_signal (&a_cond);
pthread_mutex_unlock(&a_mutex);
...
...
pthread_exit(...);
}
```

Références

- Architecture CPUs http://fr.wikipedia.org/wiki/Pipeline_(informatique)
- Comparatif des CPU Intel
  http://www.clubic.com/article-175286-2-intel-core-i7-nehalem-core-2-duo.html
- Completely Fair Scheduler http://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- Introduction au pthreads https://computing.llnl.gov/tutorials/pthreads/
- Open Source POSIX Threads for Win32 http://sourceware.org/pthreads-win32/
- Téléchager Microsoft Windows SFU 3.5
  http://www.microsoft.com/downloads/details.aspx?FamilyID=896C9688-601B-44F1-81A4-02878FF11778&displaylang=en

Plan du cours

- 1ère partie : Multi-taches et processus légers
- 2ème partie : **Machine parallèle et Multi-cœurs**
- 3ème partie : GPUs et CUDA
- 4ème partie : OpenCL
- 5ème partie : Exercice de synthèse

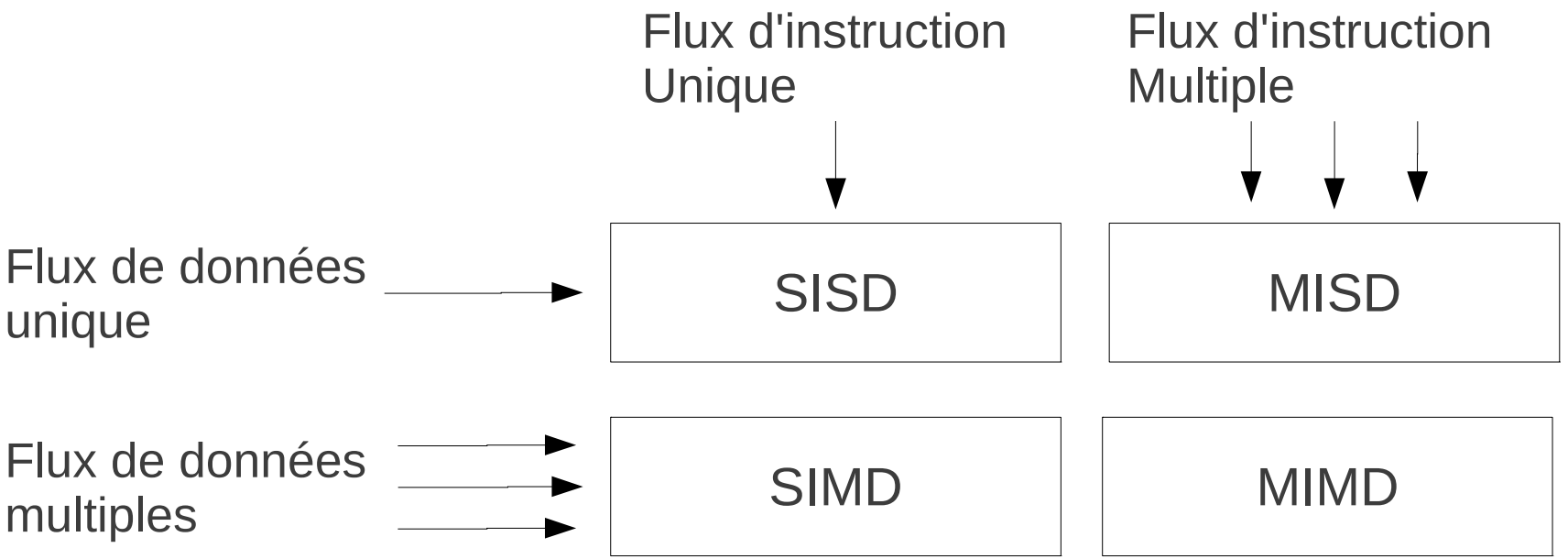
Plan 2ème partie

- Machine Parallèle
  - SIMD
    - SSE
      - Exercice
  - MIMD
    - MPI
  - MISD : Pipelines
- Multi-cœurs
  - OpenMP
    - Exercice

Machine parallèle

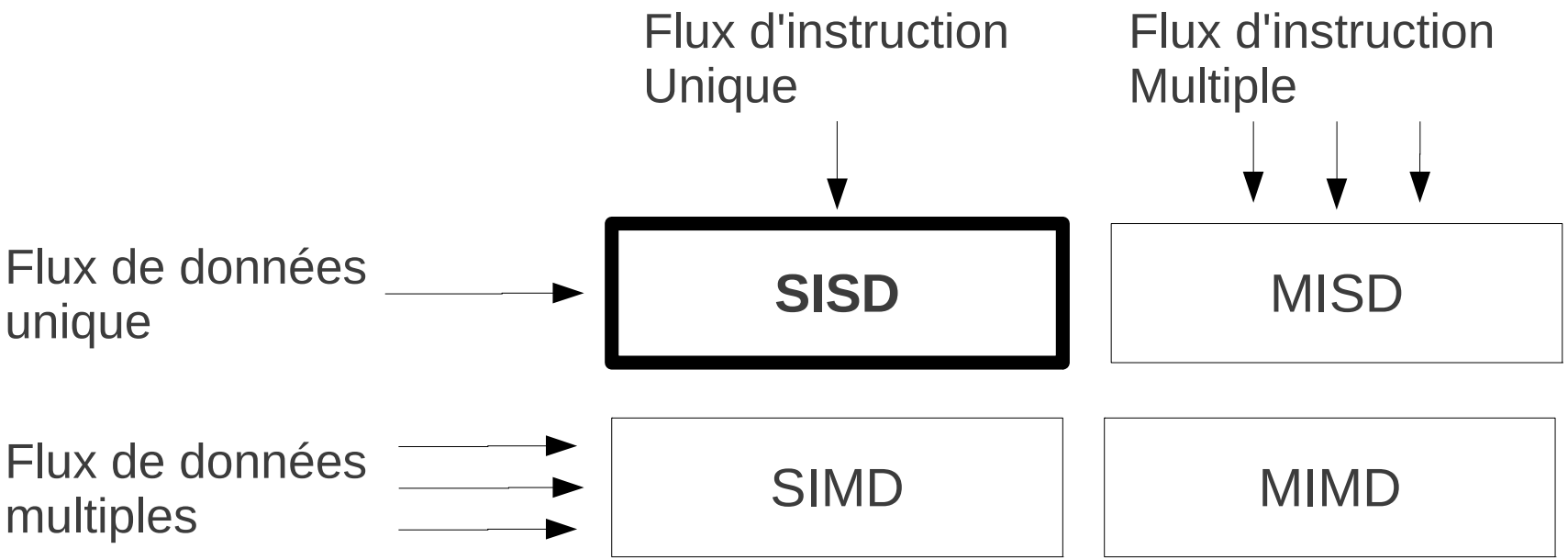
- Définition: Plusieurs unités de traitement utilisées en simultané

Classification des machines parallèles [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

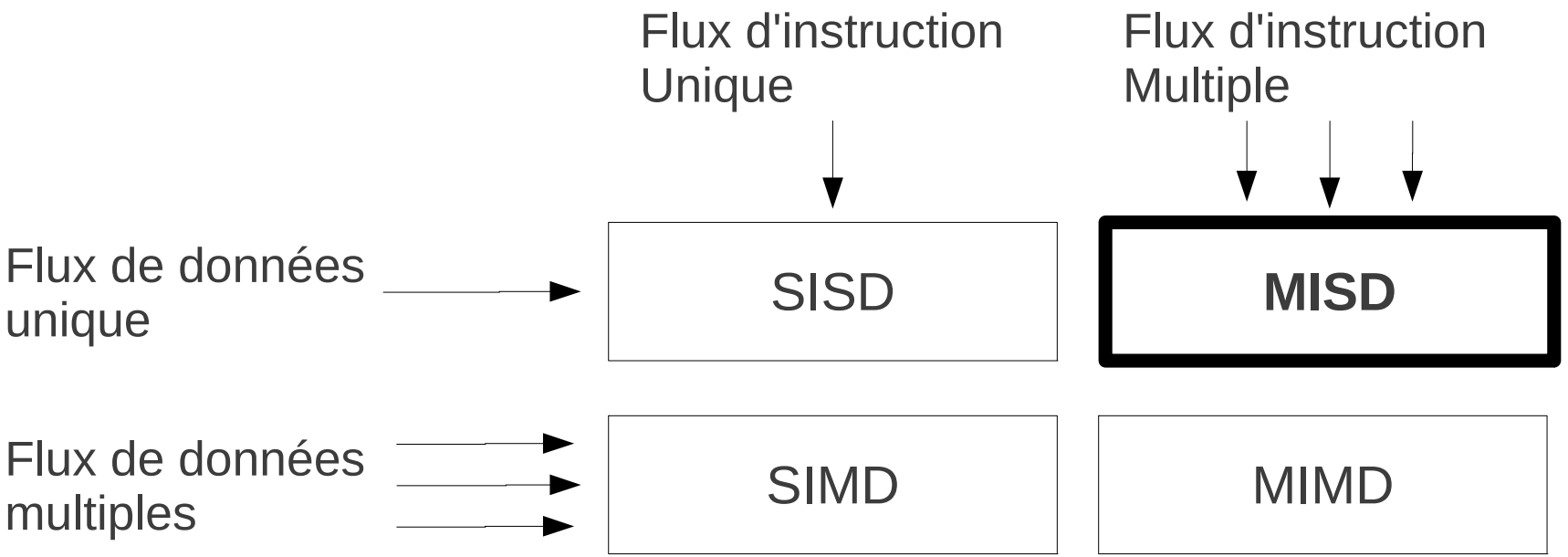
Classification des machines parallèles [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Exemple: Ordinateur séquentiel classique

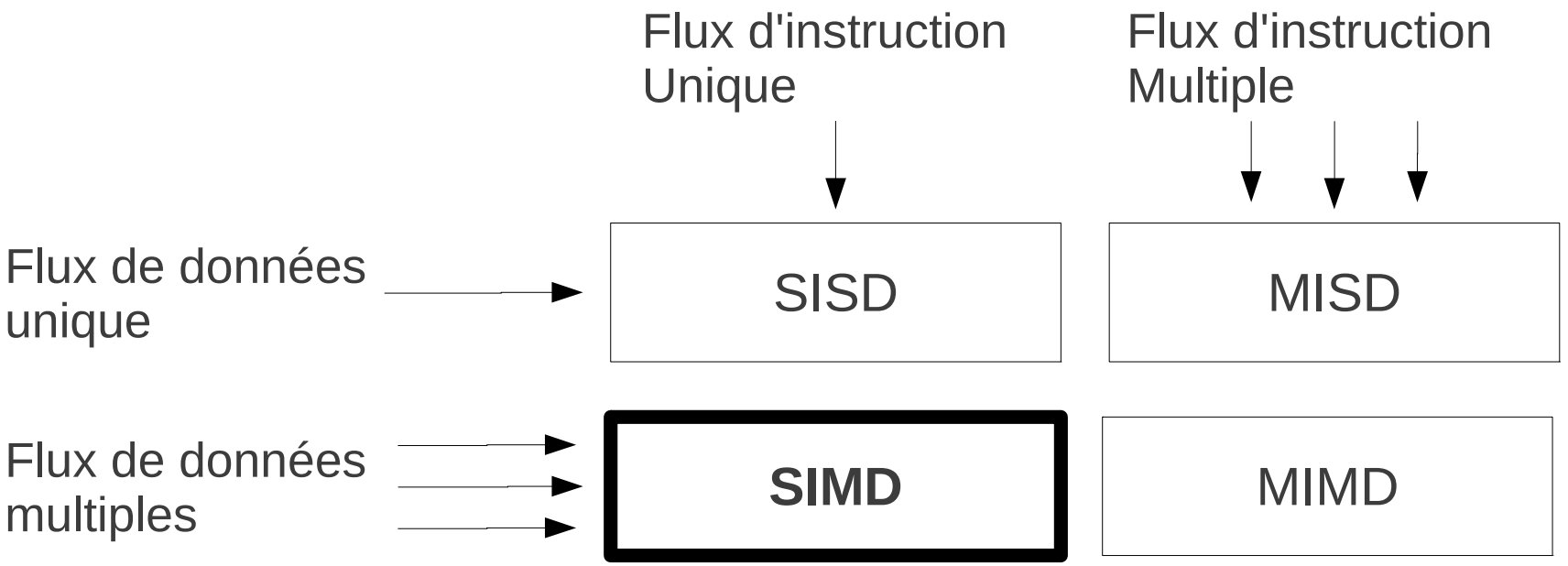
Classification des machines parallèles [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Exemple: Pipeline

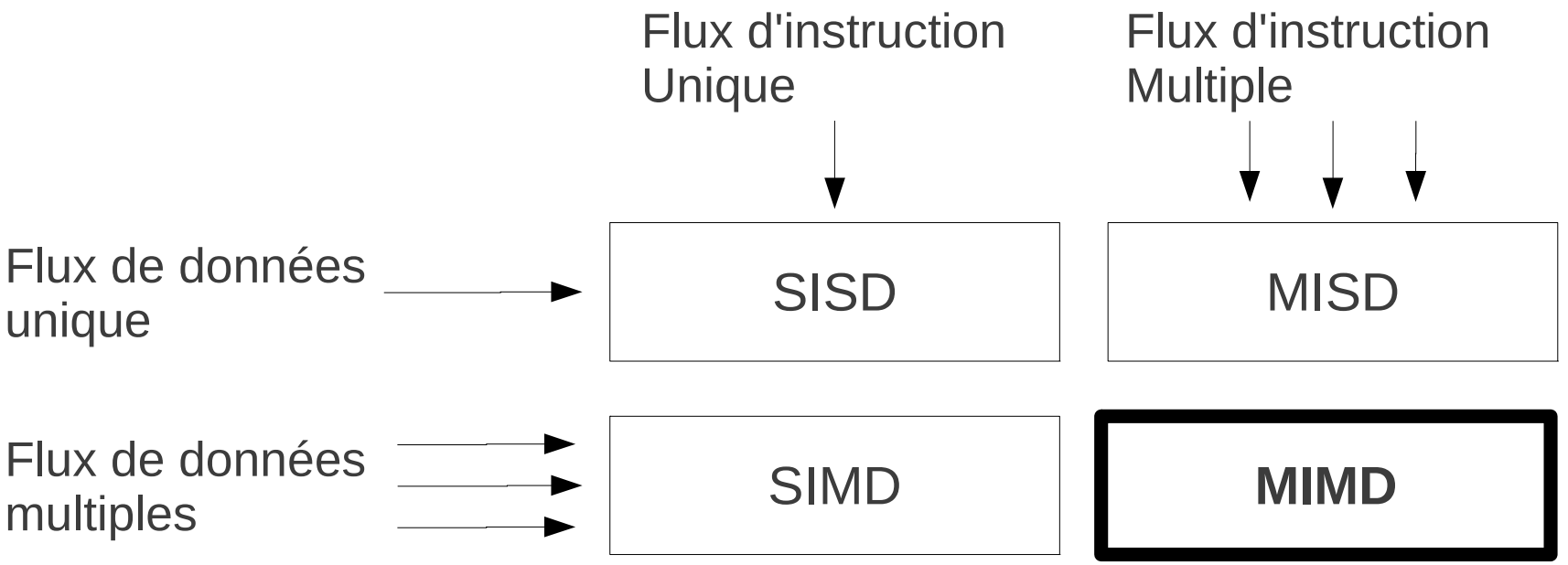
Classification des machines parallèles [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Exemple: SSE, GPU NVIDIA G200

Classification des machines parallèles [Flynn 1966]



S: single
M: Multiple
I: instruction
D: data

Exemple: Cluster PC MPI, GPU AMD RV770, GPU NVIDIA FERMI

SIMD: Single Instruction Multiple Data

- Cas d'étude : Instructions SSE Intel
- SSE (Streaming SIMD Extensions)
  - Plusieurs versions SSE, SSE2, SSE3, SSE4, SSE4a, SSE4.2
    - **SSE2** disponible sur la plupart de CPU x86 actuels d'Intel, d'AMD et de VIA.
  - Instructions SIMD sur registres de 128bits
    - 4 x 32 bits flottants ou entiers
    - 2 x 64 bits doubles ou entiers longs
    - 8 x 16 bits entiers courts

SSE2

- Utilisation en C / C++: `#include<xmmintrin.h>`
- Vecteurs:
  - 4 float 32 bits: `__m128`
  - 4 int 32 bits : `__m128i`
  - 2 float 64 bits : `__m128d`
- Alignement des données sur 16 bit
  - Sur OS 64 bits c'est alignement par défaut
  - Sur OS 32 bits remplacer malloc(...) par :
    - memalign(16, ...) sur GCC
    - _align_malloc(...,16) sur MSVC

SSE flottants 32 bits en C/C++

- Déclaration

```
__m128 a, c, b={0.0f,1.0f,
                2.0f,3.0f};
```

- Affectation
  - 1 float

```
a = _mm_set1_ps(5.01f);
```

  - 4 floats

```
a = _mm_set_ps(0.0f,1.0f,
    2.0f,3.0f);
```

  - À partir d'un tableau

```
float X[1000]={...}; // ~= 250 __m128
a = _mm_load_ps(X+4*i); // i dans [0,250[
```

- Copie du contenu
  - Vers un flottant

```
float x0=((float*)&a)[0];
```

  - Vers un tableau

```
_mm_store_ps(X+4*i,b); // i dans [0,250[
```

SSE flottants 32 bits en C/C++

- Addition
- Soustraction
- Produit
- Division
- Racine

```
__m128 c = _mm_add_ps(a,b);
__m128 c = _mm_sub_ps(a,b);
__m128 c = _mm_mul_ps(a,b);
__m128 c = _mm_div_ps(a,b);
__m128 c = _mm_sqrt_ps(a);
```

```
{                {                {
X1,              X2,              X1 op X2,
Y1,      op      Y2,      =       Y1 op Y2,
Z1,              Z2,              Z1 op Z2,
W1               W2               W1 op W2
}                }                }
```

SSE flottants 32 bits en C++

- En MS Visual Studio 2008 et Intel C++ Compiler (icc)

```
#include<fvec.h>
```

  - Classe vecteurs

```
F32Vec4 vecA, vecB(0.1f,0.2f,
                   0.3f,0.4f);
```

  - Opérateurs:

```
F32Vec4 vecC = vecA + vecB;
```

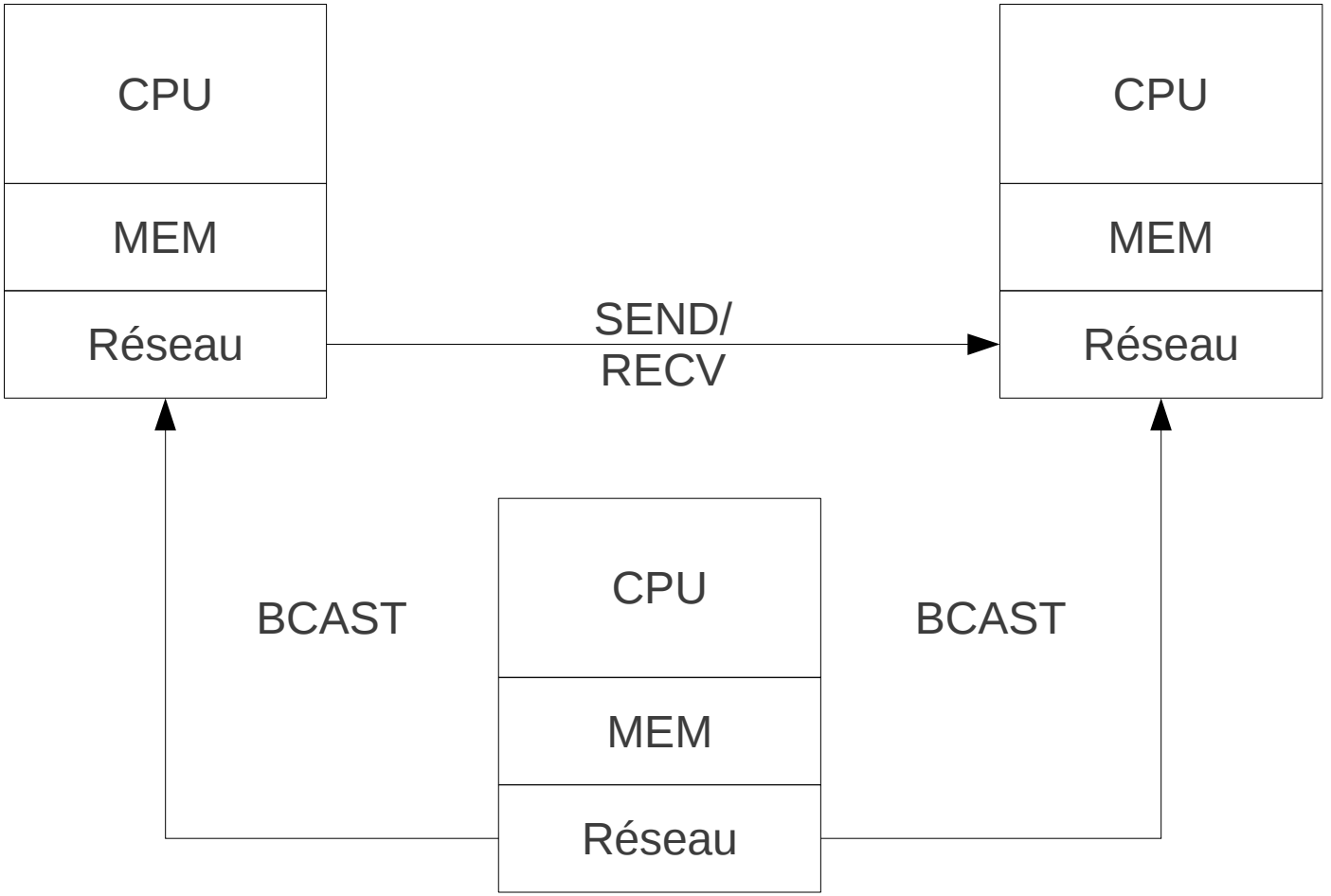
- En g++
  - Opérateurs:

```
_m128 c =  a + b;
```

MIMD : Multiple Instructions Multiple Data

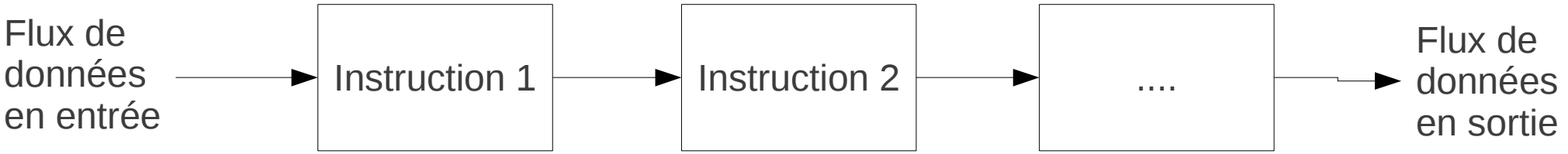
- Processeurs asynchrones
- Mémoire partagée
  - SMP: *Symetric Multi-processors*
    - Exemple : Pthread
- Mémoire répartie
  - MPI *Message Passing Interface*

MPI

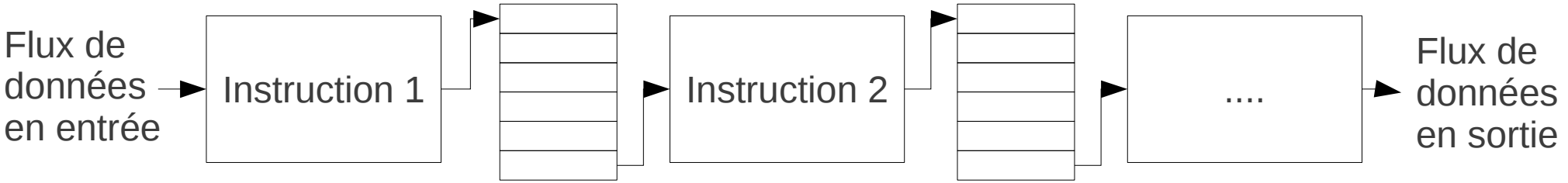

CPU
MEM
Réseau
SEND/
RECV
CPU
MEM
Réseau
BCAST
CPU
MEM
Réseau
BCAST

MISD : Multiple Instruction Single Data

- Filtres et pipelines
  - Synchrone



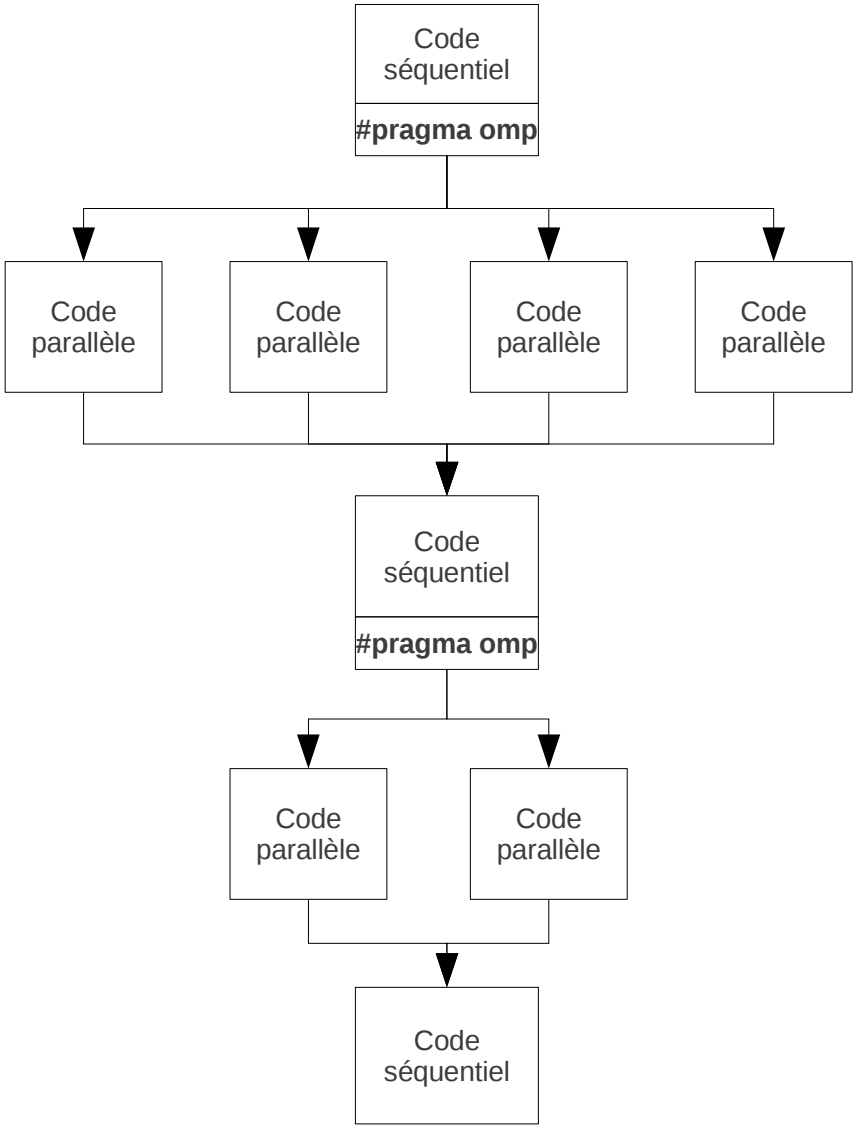
  - Asynchrone (avec queues)



OpenMP (Open Multi-Processing)

- Compilation et librairie
- Ajout des informations de parallélisme au code
  - Définition de zones parallèles à l'intérieur d'un code séquentiel.
  - En C/C++ ajout des directives de compilation
    - `#pragma omp ...`
  - Algorithmes
    - Partage d'instructions
    - Partage de mémoire
    - Réduction

Parallélisme partiel OMP



Exemple

```
void a1(int n, float *a, float *b)
{
     int i;

     for (i=1; i<n; i++)
         b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Exemple

```
void a1(int n, float *a, float *b)
{
     int i;
 #pragma omp parallel for
     for (i=1; i<n; i++)
         b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Exemple

```
void a1(int n, float *a, float *b)
{
      int i;
 #pragma omp parallel for
      for (i=1; i<n; i++)
          b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Zone parallèle avec
- Partage du code
- Partage de **a** et **b**
- Variable **i** privée implicite

Synchronisation

```
#include <math.h>
void a8(int n, int m, float *a, float *b, float
*y, float *z)
{
   int i;
   #pragma omp parallel
   {
     #pragma omp for
       for (i=1; i<n; i++)
          b[i] = (a[i] + a[i-1]) / 2.0;
     #pragma omp for
       for (i=0; i<m; i++)
          y[i] = sqrt(z[i]);
   }
}
```

Synchronisation

```
#include <math.h>
void a8(int n, int m, float *a, float *b, float
*y, float *z)
{
   int i;
   #pragma omp parallel
   {
     #pragma omp for
       for (i=1; i<n; i++)
         b[i] = (a[i] + a[i-1]) / 2.0;
     #pragma omp for
       for (i=0; i<m; i++)
         y[i] = sqrt(z[i]);
   }
}
```

Barrière de synchronisation implicite

Barrière de synchronisation implicite

Synchronisation

```
#include <math.h>
void a8(int n, int m, float *a, float *b, float
*y, float *z)
{
   int i;
   #pragma omp parallel
   {
     #pragma omp for nowait
       for (i=1; i<n; i++)
          b[i] = (a[i] + a[i-1]) / 2.0;
     #pragma omp for nowait
       for (i=0; i<m; i++)
          y[i] = sqrt(z[i]);
   }
}
```

Réduction

```
void a31_1(float *x, int *y, int n)
{
  int i, b;
  float a;
  A = 0.0;
  b = 0;


  for (i=0; i<n; i++) {
    a += x[i];
    b ^= y[i];

  }

}
```

Réduction

```
void a31_1(float *x, int *y, int n)
{
  int i, b;
  float a;
  A = 0.0;
  b = 0;
#pragma omp parallel for private(i) shared(x, y, n) \
  reduction(+:a) reduction(^:b)

  for (i=0; i<n; i++) {
    a += x[i];
    b ^= y[i];

  }

}
```

Indique les variables partagées et les opérateurs associés à la réduction

Réduction

```
void a31_1(float *x, int *y, int n)
{
  int i, b;
  float a;
  A = 0.0;
  b = 0;
#pragma omp parallel for private(i) shared(x, y, n) \
                         reduction(+:a) reduction(^:b)
  for (i=0; i<n; i++) {
    a += x[i];
    b ^= y[i];

  }

}
```

Indique les autres variables partagées

Références

- Classification des machines parallèles http://fr.wikipedia.org/wiki/Taxinomie_de_Flynn
- Les architectures parallèles http://dio.obspm.fr/PDF/archi_mpi.pdf
- Les instruction SSE http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
- Alignement mémoire et performance http://www.gamasutra.com/view/feature/3942/data_alignment_part_1.php?print=1
- Allocation mémoire alignée avec gcc http://www.gnu.org/s/libc/manual/html_node/Aligned-Memory-Blocks.html
- Référence SSE Microsoft MSDN http://msdn.microsoft.com/en-us/library/t467de55(VS.71).aspx
- Spécifications OpenMP 2.5 (supportée par MS Visual Studio 2008) http://www.openmp.org/mp-documents/spec25.pdf

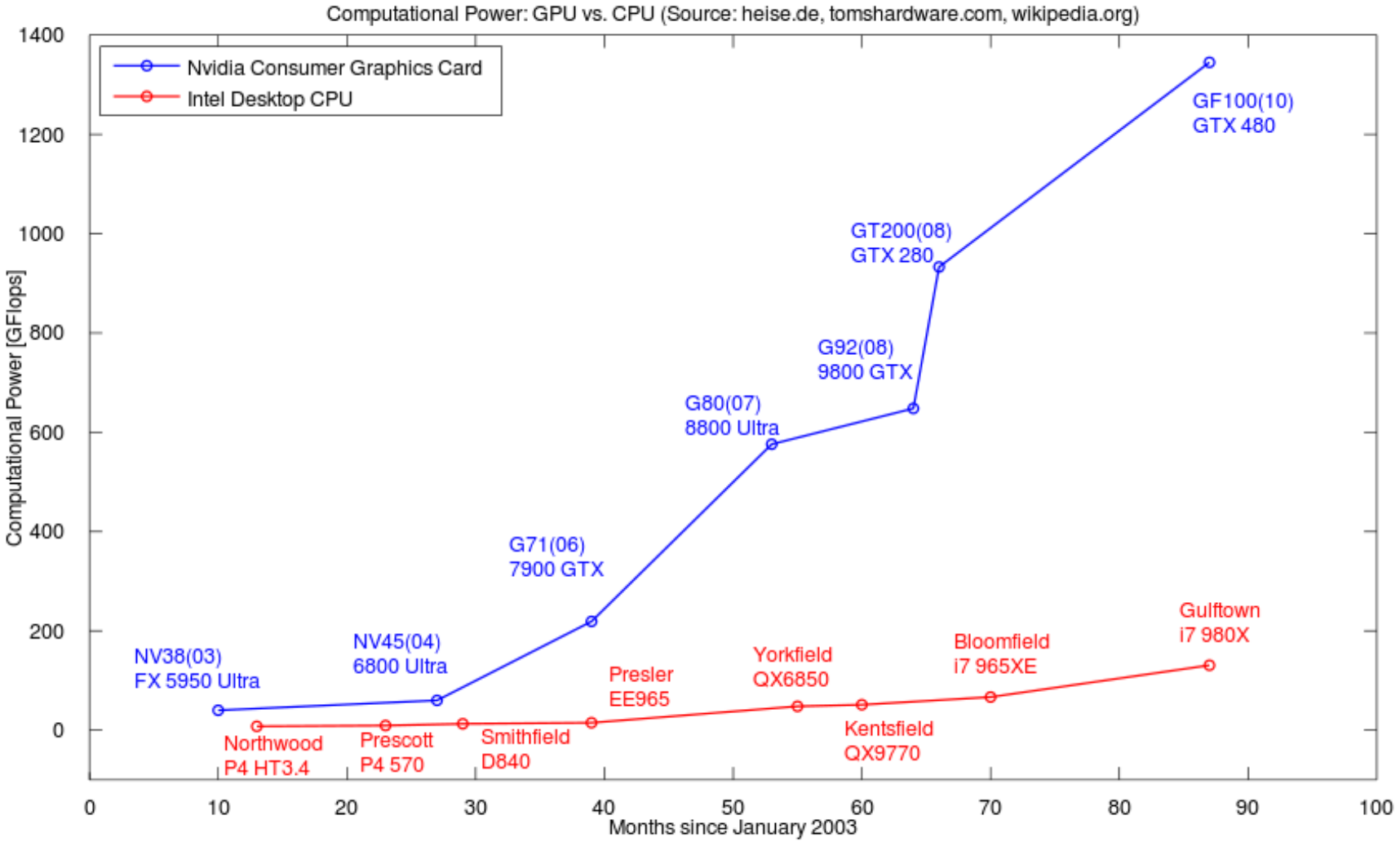
Plan du cours

- 1ère partie : Multi-taches et processus légers
- 2ème partie : Machine parallèle et Multi-cœurs
- **3ème partie : GPUs et CUDA**
- 4ème partie : OpenCL
- 5ème partie : Exercice de synthèse

Plan 3ème partie

- Pourquoi le GPU
- Architecture
  - NVIDIA et ATI
- Langages de programmation
  - Brook+/CAL
  - CUDA/PTX
    - Exercice 0
- Programmer en CUDA
  - Blocks et Grilles
    - Exercice 1
  - Mémoires
  - Threads
    - Exercice CUDA 2

Pourquoi le GPU



http://gpu4vision.icg.tugraz.at/

Pourquoi le GPGPU

- GPUs **massivement parallèles**
- Bande passante interne très importante
- Calcul en virgule flottante 32 bits (IEEE 754, depuis NVIDIA G200)
- Langages de haut et bas niveau pour programmation générique:
  - NVIDIA CUDA / PTX sur GF1X0, G2XX, G9X et G80
  - AMD Brook+ / CAL sur RV870, RV770, R7xx et R600
- Grande série => coût réduits
- Efficacité énergétique: plus de 6 Gflops / watts

Architecture

ATI Radeon HD 5870 (Cypress XT)

- 20 Cœurs SIMD
- 1600 ALUs = 20 x 16 x 5 (VLIW5 ~ 4 ALU +[1 ALU ou 1 SFU])
- 1 Go RAM (GDDR5)
- Bande passante **153,6** GB/s
- **2,72 TFLOPS**

NVIDIA GeForce GTX 480 (GF100)

- 16 = 4 x 4 Streaming Multiproc. (SM)
- 512 ALUs scalaires = 16 x 32 et 64 = 16 x 4 SFU (sin, cos, srqt, etc.)
- 1,5 Go RAM (GDDR5)
- Bande passante **177,4** GB/s
- **1,35 TFLOPS**

Architecture

ATI Radeon HD 6970 (Northern Islands)

- 24 Cœurs SIMD
- 1536 ALUs = 24 x 16 x 4 (VLIW4 ~ [3 ALU ou 1 SFU] + 1 ALU)
- 2 Go RAM (GDDR5)
- Bande passante **176** GB/s
- **2,70 TFLOPS**

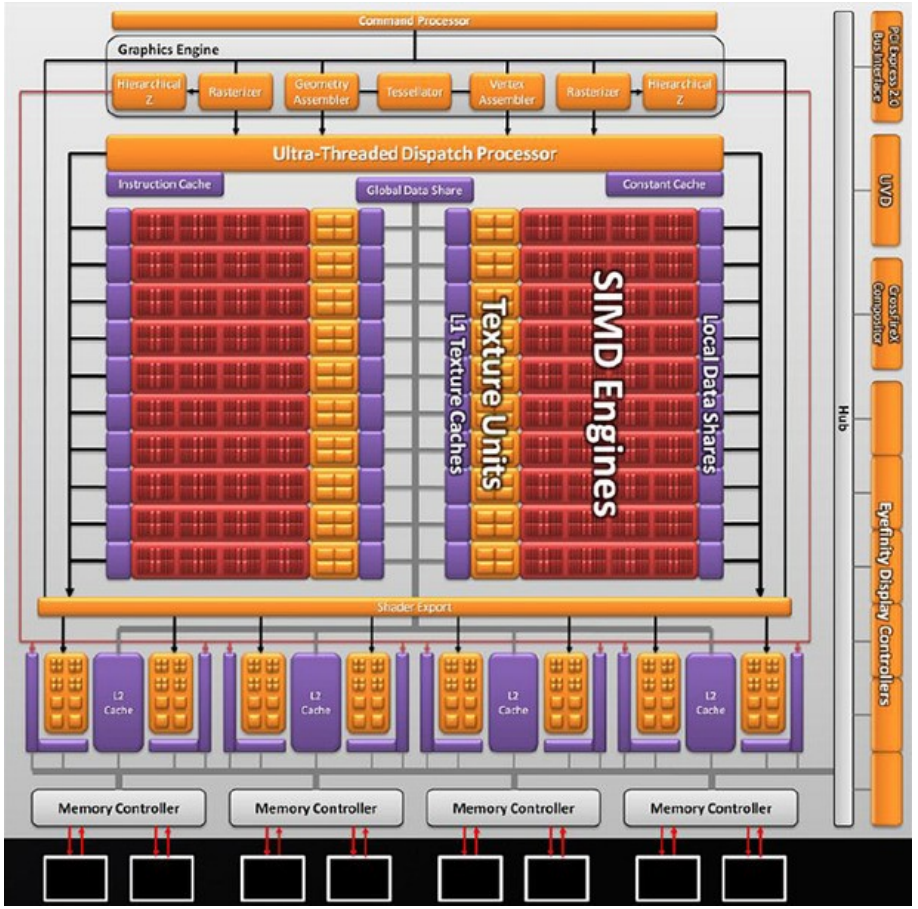
NVIDIA GeForce GTX 580 (GF110)

- 16 = 4 x 4 Streaming Multiproc. (SM)
- 512 ALUs scalaires = 16 x 32 et 64 = 16 x 4 SFU
- Jusqu'à 3 Go RAM (GDDR5)
- Bande passante **192,4** GB/s
- **1,581 TFLOPS**

Architecture

ATI Radeon HD 5870 (Cypress XT)

- 2x10 SIMD Cores



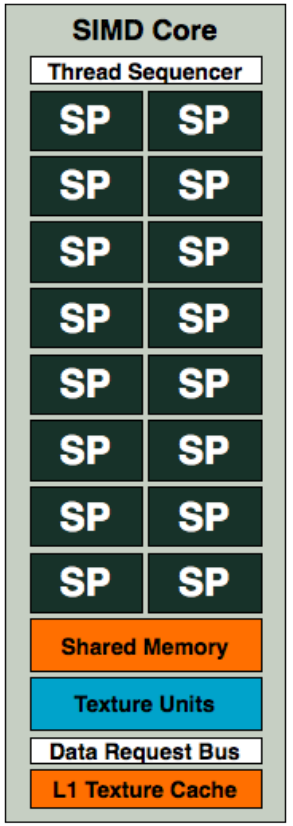
NVIDIA GeForce GTX 480 (GF100)

- 4 GPC : 4 SM + 1 raster unit. chaque

Unités de calcul

ATI Radeon HD 4870 (Cypress XT)

- SP (4 scalaire + 1 SFU)
- SIMD Core:
  - 16 SP
  - M-Thread Ctl.
  - **Memoire partagée**
  - Unités Tex.

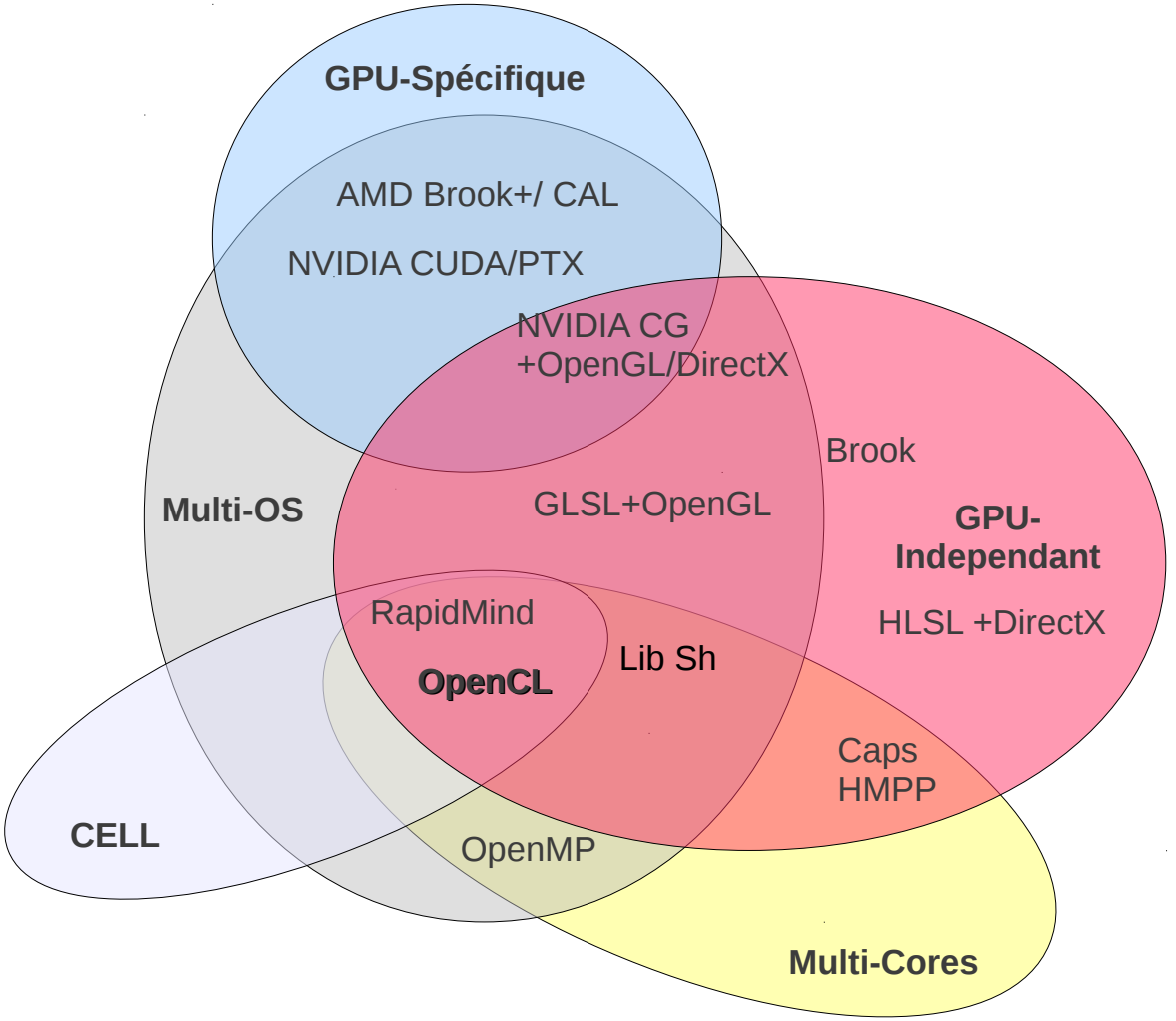


NVIDIA GeForce GTX 480 (GF100)

- SP (1 scalaire)
- Streaming Multiproc (SM)
  - 32 SP
  - 4 SFU
  - M-Theard Ctl.
  - Memoire Partagée

Streaming Multiprocessor (SM)

Langages de programmation



Langages de Programmation (histoire ancienne ?)

AMD :

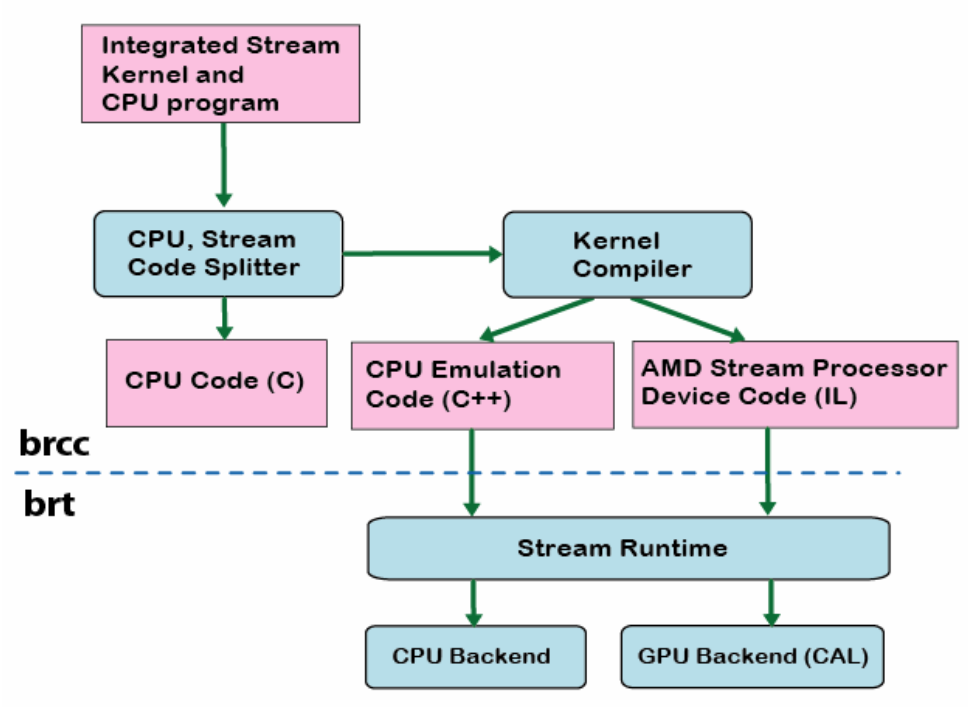
- Brook+
  - Langage de haut niveau
  - Extension du C
  - Origine code de l'université de Stanford
- CAL
  - Langage unifié abstrait bas niveau pour R600 , R7xx et RV870
  - Mais aussi pour CPU multi-cores AMD ou émulation
- SDK et runtime disponible en version 14 pour Windows XP 64 et 32 bits et linux.

NVIDIA:

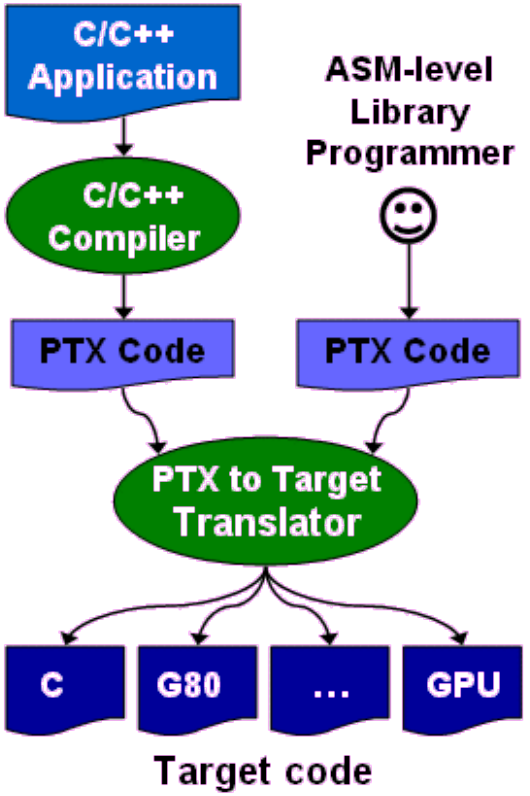
- CUDA
  - Langage de haut niveau
  - Extensions au C/C++
- PTX
  - Langage unifié abstrait bas niveau pour G8x, G9x, G200 et GF100
  - Émulation sur CPU possible
- SDK et runtime disponible en version 3.2 sur Windows XP et Linux en 64 et 32 bits et en version 2.3 sur Mac OS X

Production du programme

AMD :



NVIDIA:



Brook+

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float output_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, output_c);
    ...
}
```

Brook+

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float output_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, output_c);
    ...
}
```

Noyau à exécuter dans le GPU

Brook+

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float output_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, output_c);
    ...
}
```

Références aux matrices de données à traiter par le GPU

Brook+

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float output_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, output_c);
    ...
}
```

Envoi de données vers le GPU

Brook+

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float output_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, output_c);
    ...
}
```

exécution du noyau dans le GPU

Brook+

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
int main(int argc, char** argv)
{
    int i, j;
    float a<10, 10>;
    float b<10, 10>;
    float c<10, 10>;
    float input_a[10][10];
    float input_b[10][10];
    float output_c[10][10];

    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    sum(a, b, c);
    streamWrite(c, output_c);
    ...
}
```

Retour du résultat vers le CPU

CUDA

```
__global__ void sum(float* A, float *B,
                    float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
...
    cudaMemcpy(a,input_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b,input_b, mem_size), cudaMemcpyHostToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size,cudaMemcpyDeviceToHost);
...
}
```

CUDA

```
__global__ void sum(float* A, float *B,
                    float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
...
    cudaMemcpy(a,input_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b,input_b, mem_size), cudaMemcpyHostToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size,cudaMemcpyDeviceToHost);
...
}
```

Noyau à exécuter dans le GPU

CUDA

```
__global__ void sum(float* A, float *B,
                    float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
...
    cudaMemcpy(a,input_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b,input_b, mem_size), cudaMemcpyHostToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size,cudaMemcpyDeviceToHost);
...
}
```

Allocation des matrices de données à traiter par le GPU

CUDA

```
__global__ void sum(float* A, float *B,
                    float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
...
    cudaMemcpy(a,input_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b,input_b, mem_size), cudaMemcpyHostToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size,cudaMemcpyDeviceToHost);
...
}
```

Envoi de données vers le GPU

CUDA

```
__global__ void sum(float* A, float *B,
                    float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
...
    cudaMemcpy(a,input_a, mem_size, cudaMemcpyHostToDevice);
    cudaMemcpy(b,input_b, mem_size, cudaMemcpyHostToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size,cudaMemcpyDeviceToHost);
...
}
```

exécution du noyau dans le GPU

CUDA

```
__global__ void sum(float* A, float *B,
                    float* C, int width)
{
    unsigned int idx = threadIdx.y * width + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
int main()
{
...
    unsigned int mem_size=10*10*sizeof(float);
    float *a; cudaMalloc((void**)&a, mem_size);
    float *b; cudaMalloc((void**)&b, mem_size);
    float *c; cudaMalloc((void**)&c, mem_size);
    float input_a[100]; // size = 10*10
    float input_b[100];
    float output_c[100];
...
    cudaMemcpy(a,input_a, mem_size, cudaMemcpyHo    oDevice);
    cudaMemcpy(b,input_b, mem_size), cudaMemcpy    stToDevice);
    dim3 dimBlock(10, 10);
    sum<<<1, dimBlock>>>(a, b, c, width);
    cudaMemcpy(output_c, c, mem_size,cudaMemcpyDeviceToHost);
...
}
```

Retour du résultat vers le CPU

Langages de Programmation Unification : OpenCL

```
// OpenCL Kernel Code

__kernel void

vectorAdd(__global const float * a,

          __global const float * b,

          __global float * c)

{

  // Vector element index

  int nIndex = get_global_id(0);

  c[nIndex] = a[nIndex] + b[nIndex];

}
```

Langages de Programmation Unification : OpenCL

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
0, 0, 0);
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
nContextDescriptorSize, aDevices, 0);
```

Langages de Programmation Unification : OpenCL

```
// create a command queue for first device the context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices[0], 0, 0);
// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
                                     sProgramSource, 0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0) ;
// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "vectorAdd", 0);
```

Langages de Programmation Unification : OpenCL

```
// allocate host vectors
float * A = new float[n];
float * B = new float[n];
float * C = new float[n] ;
// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             n * sizeof(cl_float),
                             A,
                             0) ;
hDeviceMemB = clCreateBuffer(hContext,
                             CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                             n * sizeof(cl_float),
                             B,
                             0) ;
hDeviceMemC = clCreateBuffer(hContext,
                             CL_MEM_WRITE_ONLY,
                             n * sizeof(cl_float),
                             0, 0);
```

Langages de Programmation Unification : OpenCL

```
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hDeviceMemC);
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                       &n, 0, 0, 0, 0);
// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, 0,
                    n * sizeof(cl_float),C, 0, 0, 0);
```

Langages de Programmation Unification : OpenCL

```
delete[] A;

delete[] B;

delete[] C;

clReleaseMemObj(hDeviceMemA);

clReleaseMemObj(hDeviceMemB);

clReleaseMemObj(hDeviceMemC);
```

Langages de Programmation Unification : OpenCL

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
0, 0, 0);
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
nContextDescriptorSize, aDevices, 0);
```

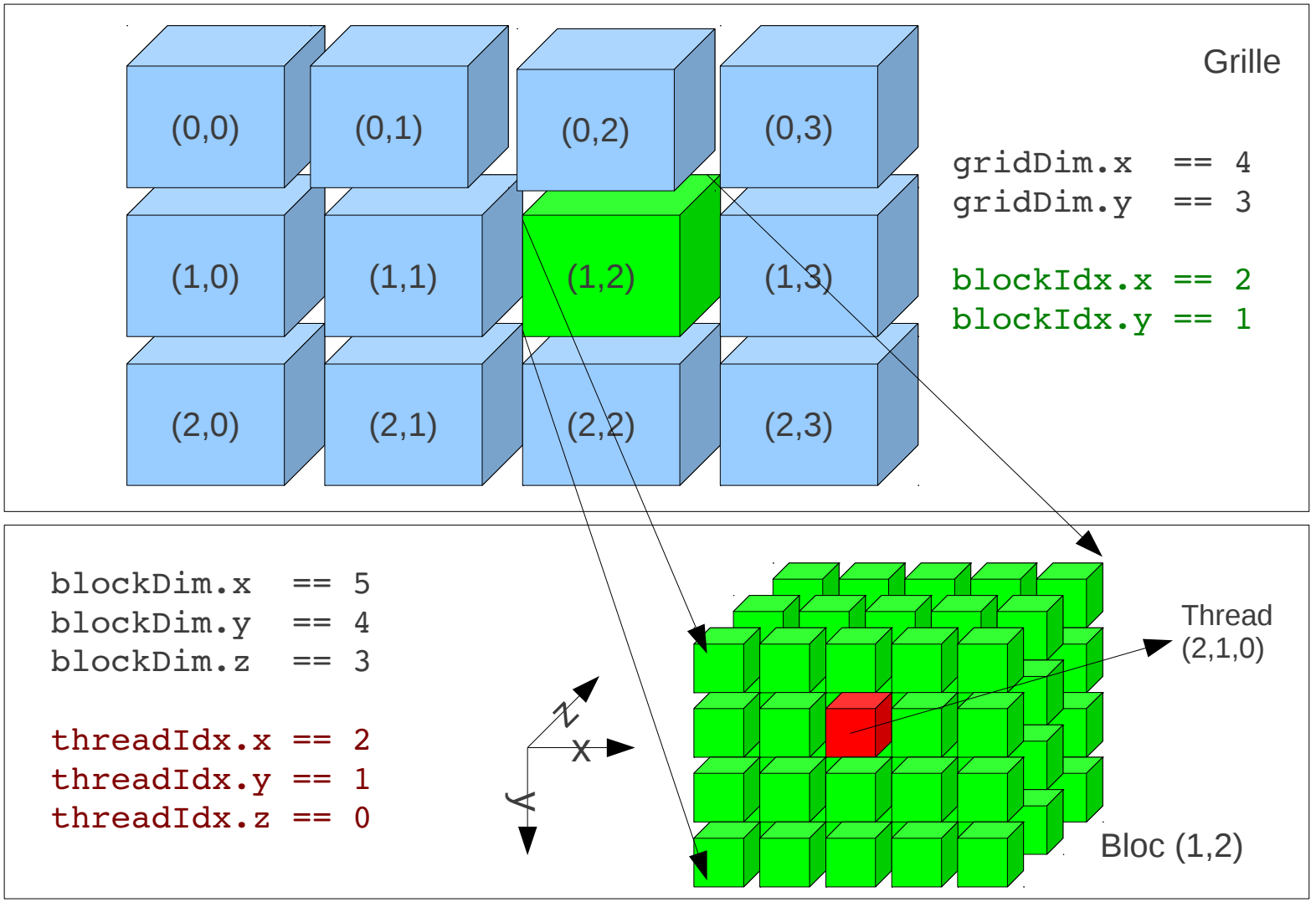
Programmer en CUDA

- Types vectoriels:
  - {char, short, uint, int, float, long, ulong}{1-4}

exemple: float4 data[10];
data[1].x=0.1; data[1].y=0.1; data[1].z=0.1; data[1].w=0.2;

- Fonctions mathématiques:
  - sqrtf(x), sinf(x), log10f(x), etc.
  - Versions rapides : __sinf(x), __log10f(x), etc. (pas de __sqrt(x) !)
- Branchements:
  - if(), for(; ;), while(), etc.
  - Possibilité de dérouler automatiquement des boucles (taille connue et faible)
- Mesure du temps: clock()
- Vote (>G9x) : int __all(int predicate); int __any(int predicate) ;
- Atomic (>G84) : int atomicAdd(int* address, int val);

Grilles et blocs



Grilles et blocs

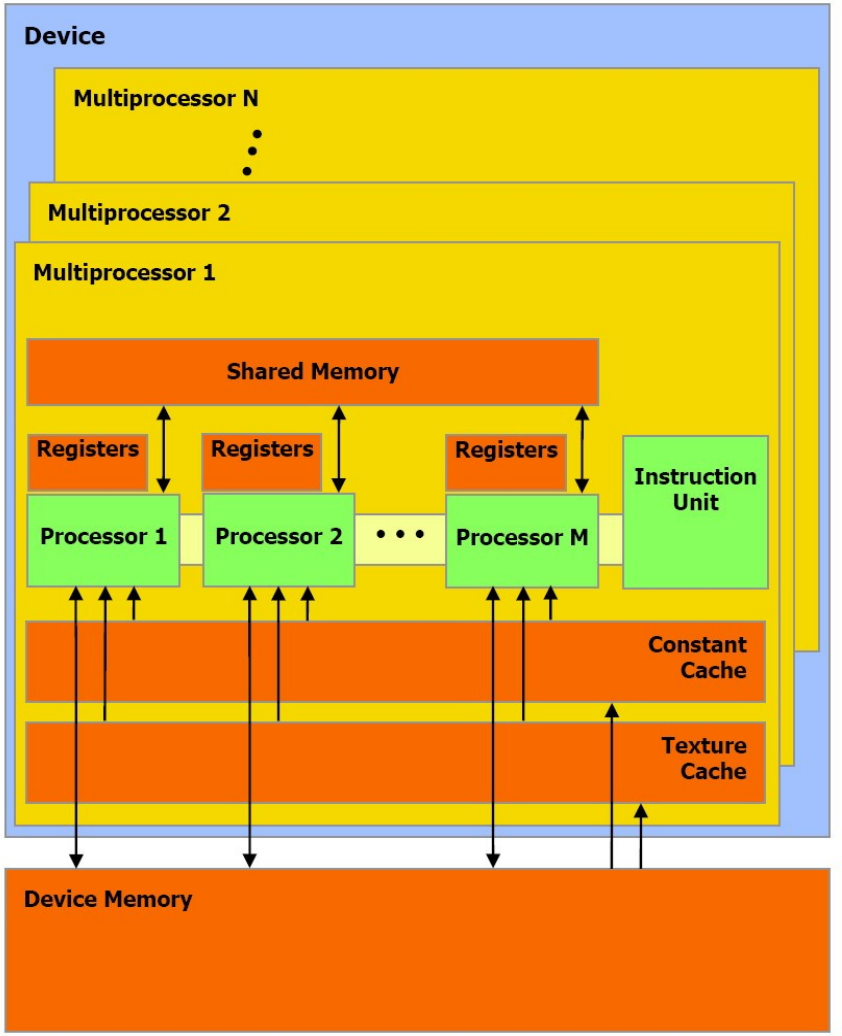
```
//  paramètres d'exécution
dim3 grid(size_x / BLOCK_DIM, size_y / BLOCK_DIM, 1);
dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);

// appel au noyau « myKernel »
myKernel<<< grid, threads >>>(device_out_data, device_in_data, size_x, size_y);
```

Grilles et blocs

```
// déclaration du noyau « myKernel »
__global__ void myKernel(float *odata, float* idata, int width, int
height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
   if (xIndex < width && yIndex < height)
   {
      unsigned int index_in  = xIndex + width * yIndex;
      unsigned int index_out = yIndex + height * xIndex;
      odata[index_out] = idata[index_in];
   }
}
```

Mémoire

- Pour chaque Multiprocesseur (SM):
  - Une mémoire partagée (lec./écrit.)
  - Un cache texture (lecture)
  - Un cache de constantes (lecture)
  - Un cache d'instructions
  - 8 processeurs scalaires (ALUs)
- Pour chaque processeur (ALU):
  - Jeux de registres (lec./écrit.)
- Mémoire partagée presque aussi rapide que les registres
- Accès direct à la DRAM de la carte (*global memory*)



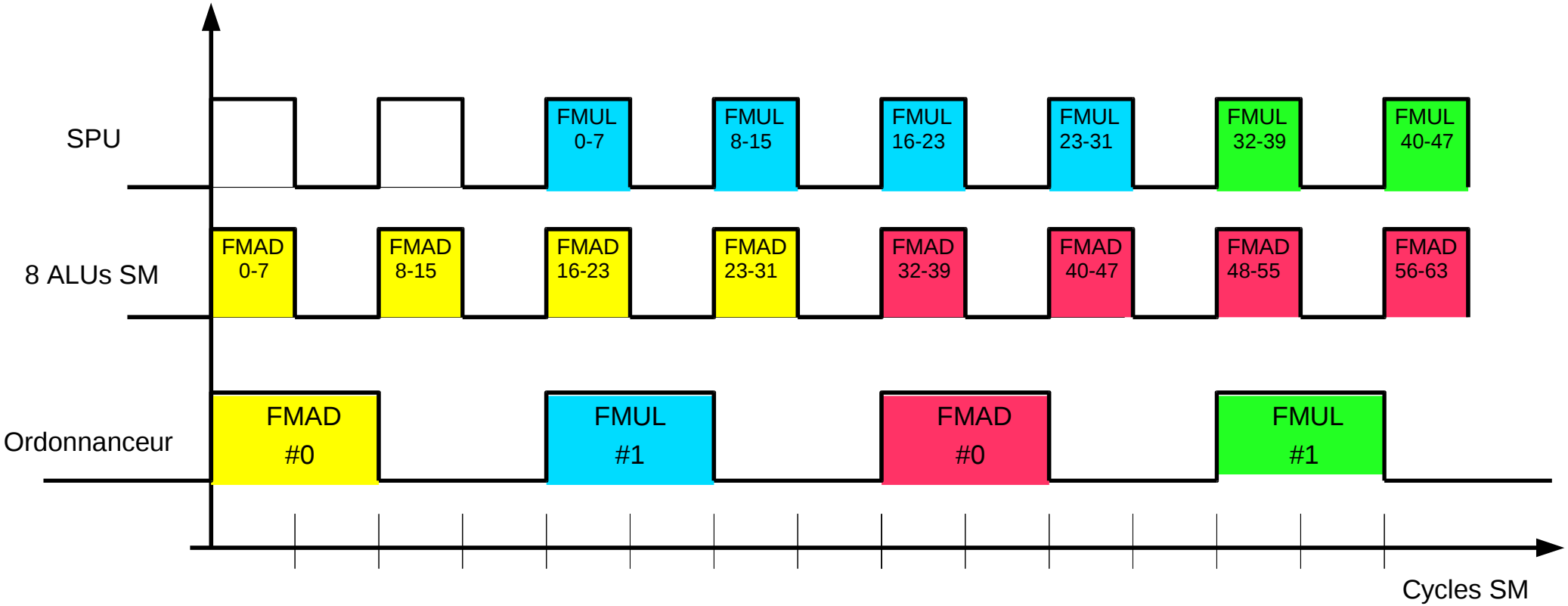
Mémoire

```
__shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
__constant__ float3 kColorMetric = { 1.0f, 1.0f,
1.0f };
texture<unsigned char, 2> tex;

// allocation dans la mémoire de la carte (device)
float* device_data;
cudaMalloc( (void**) &device_data, mem_size);
// liberation de la mémoire de la carte
cudaFree(device_data);
```

Mémoire

```
// copie de la mémoire du CPU (host)
// à celle de la carte (device)
cudaMemcpy( device_data, host_data, mem_size,
            cudaMemcpyHostToDevice);

// copie de la carte (device)
// à celle de la mémoire du CPU (host)
cudaMemcpy( host_data, device_data, mem_size,
            cudaMemcpyDeviceToHost);
```

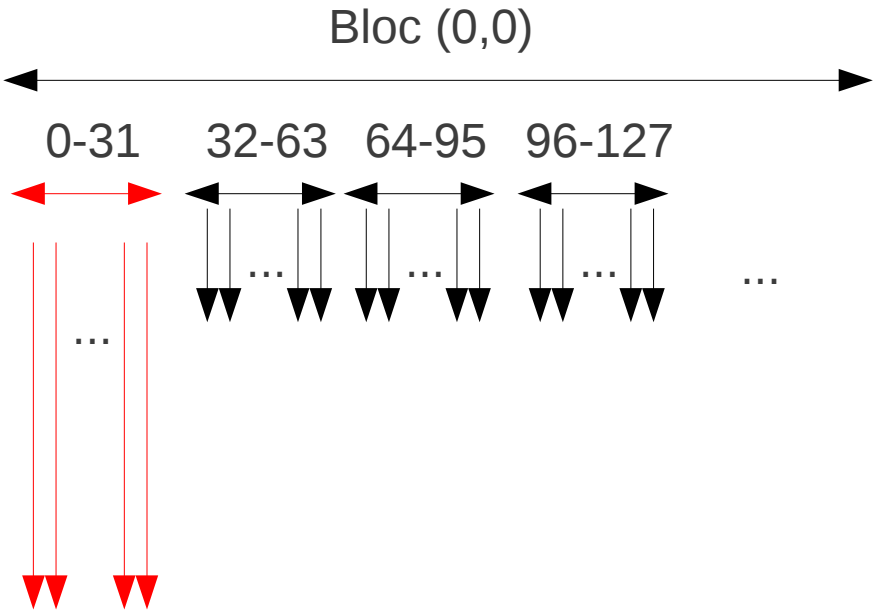
Warps



4x8=32 threads
par WARP

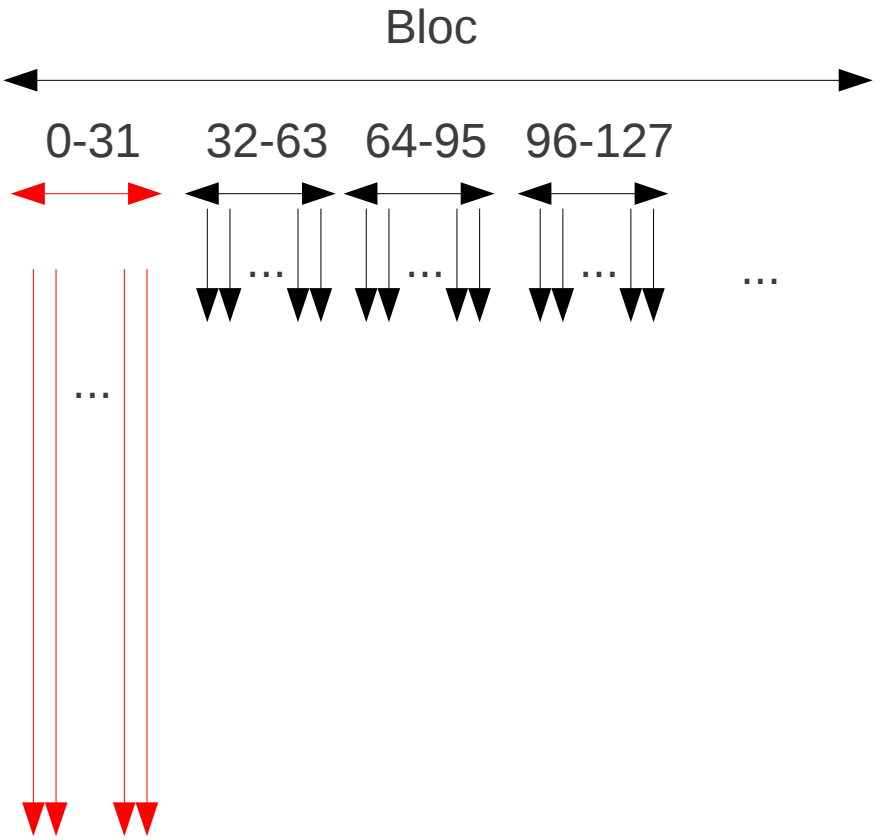
Threads

```
 global myKernel(float
*A,...)
{
   ...
   float a= x*(x+0.5f)-1 ;
   ...
   A[g_index]=a;
   ...
}
```



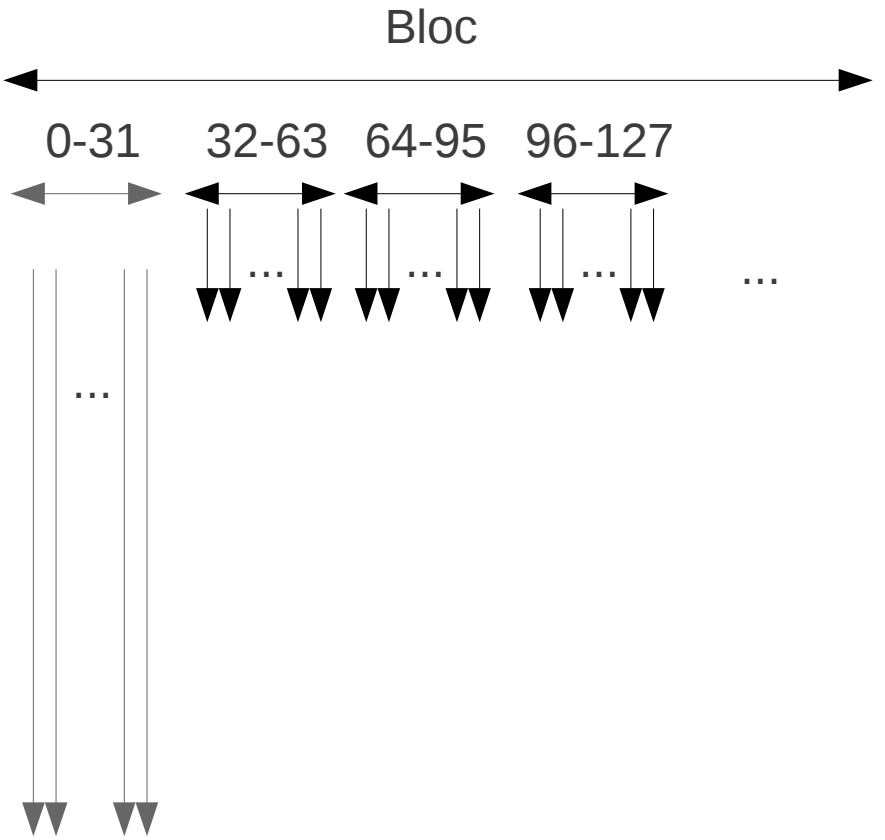
Threads

```
 global myKernel(float
*A,...)
{
   ...
   float y= x*(x+0.5f)-1 ;
   ...
   A[g_index]=a;
   ...
}
```



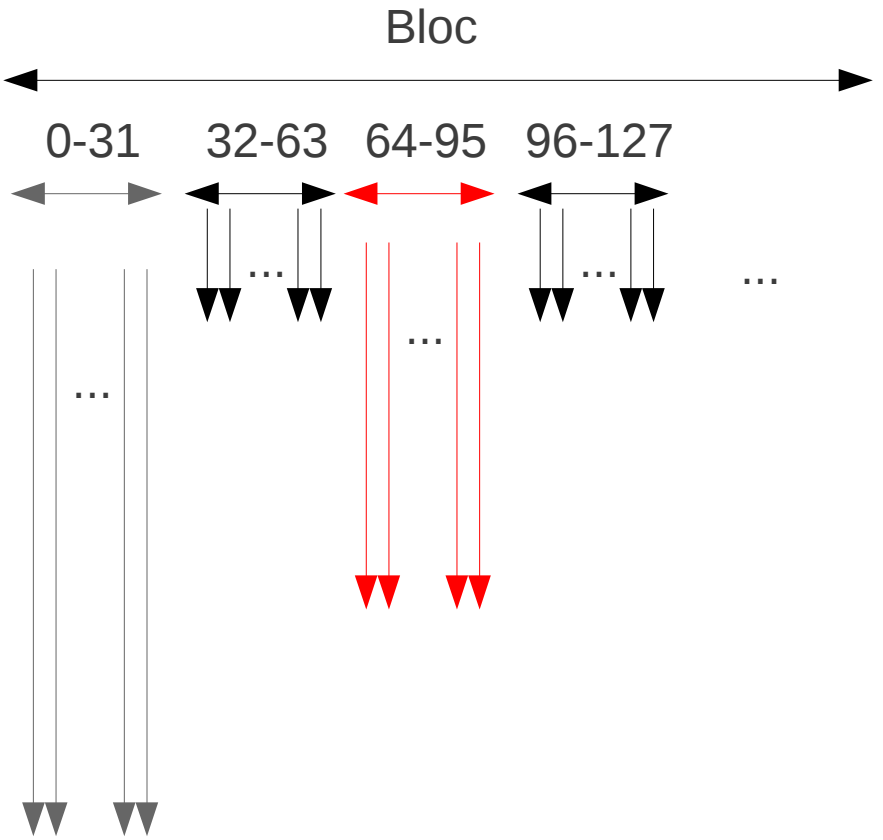
Threads

```
 global myKernel(float
*A,...)
{
   ...
   float y= x*(x+0.5f)-1 ;
   ...
   A[g_index]=a;
   ...
}
```



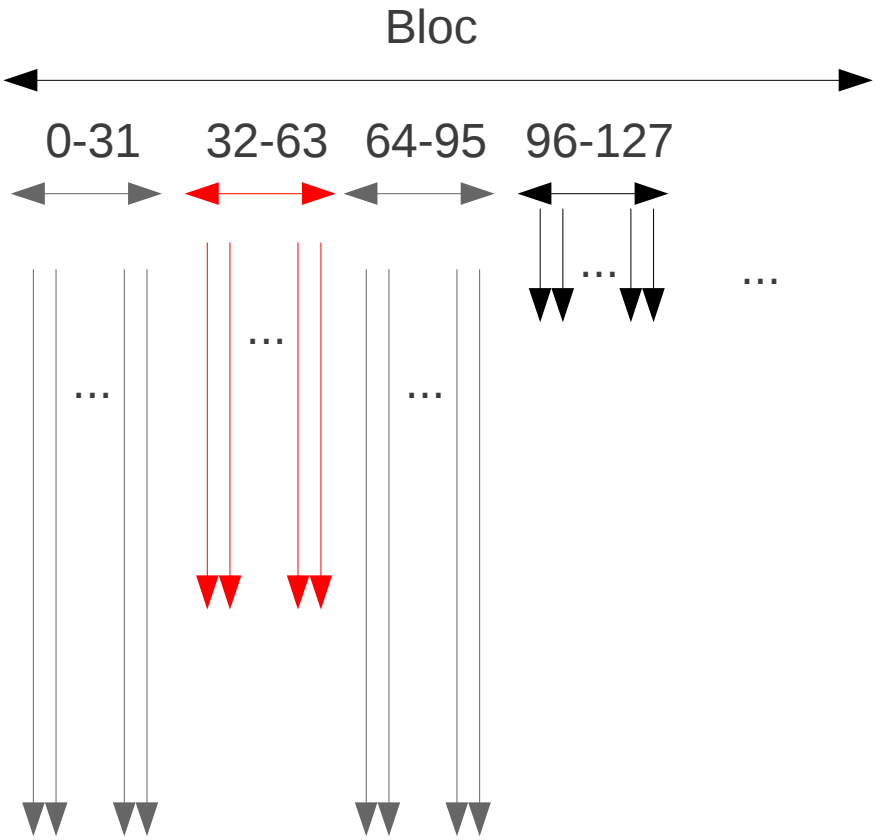
Threads

```
 global myKernel(float
*A,...)
{
   ...
   float y= x*(x+0.5f)-1 ;
   ...
   A[g_index]=a;
   ...
}
```



Threads

```
 global myKernel(float
*A,...)
{
   ...
   float y= x*(x+0.5f)-1 ;
   ...
   A[g_index]=a;
   ...
}
```



Threads

```
__global__ void myKernel(float *odata, float* idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    if((xIndex < width) && (yIndex < height))
    {
    unsigned int index_in = yIndex * width + xIndex;
    // copie dans la mémoire partage
    block[threadIdx.y][threadIdx.x] = idata[index_in];
    }
    __syncthreads();
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    if((xIndex < height) && (yIndex < width))
    {
    unsigned int index_out = yIndex * height + xIndex;
    // traitement
    odata[index_out] = block[threadIdx.x][threadIdx.y];
    }
}
```

Références

- Étude Carte Nvidia GTX 480 et GTX 470 http://techreport.com/articles.x/18682
- Architecture GPU NVIDIA GF100 http://techreport.com/articles.x/17670
- Architecture GPU AMD RV870 http://www.bit-tech.net/hardware/graphics/2009/09/30/ati-radeon-hd-5870-architecture-analysis/
- AMD's Radeon HD 6970 & Radeon HD 6950: Paving The Future For AMD http://www.anandtech.com/show/4061/amds-radeon-hd-6970-radeon-hd-6950
- Nvidia OpenCL jump start guide http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf

Plan du cours

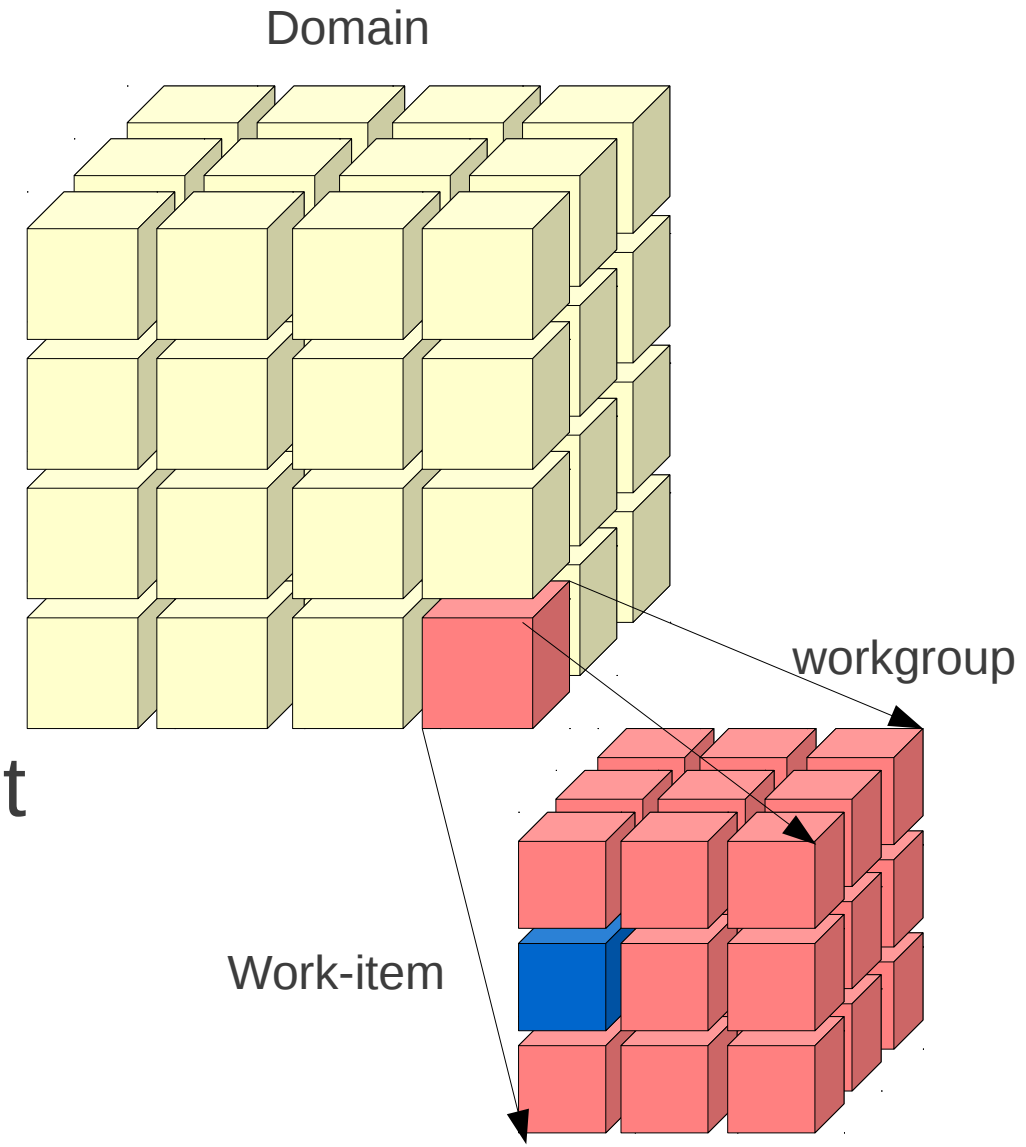
- 1ère partie : Multi-taches et processus légers
- 2ème partie : Machine parallèle et Multi-cœurs
- 3ème partie : GPUs et CUDA
- **4ème partie : OpenCL**
- 5ème partie : Exercice de synthèse

OpenCL

- Standard ouvert promu à l'origine par Apple
  - Géré par KHRONOS Group, un consortium d'entreprises : AMD, Intel, Apple, NVIDIA, ARM, IBM...
- API commune en C et noyaux en C99
  - Exécutables en parallèle sur plusieurs CPUs,GPUs ou CELL/BE
  - Gestion des contextes multiples
  - Gestion explicite du transfert mémoire
  - Exécution synchrone ou asynchrone

Concepts d'OpenCL

- Domaine N-dimension du traitement *global dimensions*
  - Décomposition en groupes *workgroups*
- Décomposition de chaque groupe en éléments de traitement *work-items*
  - *Local dimensions*

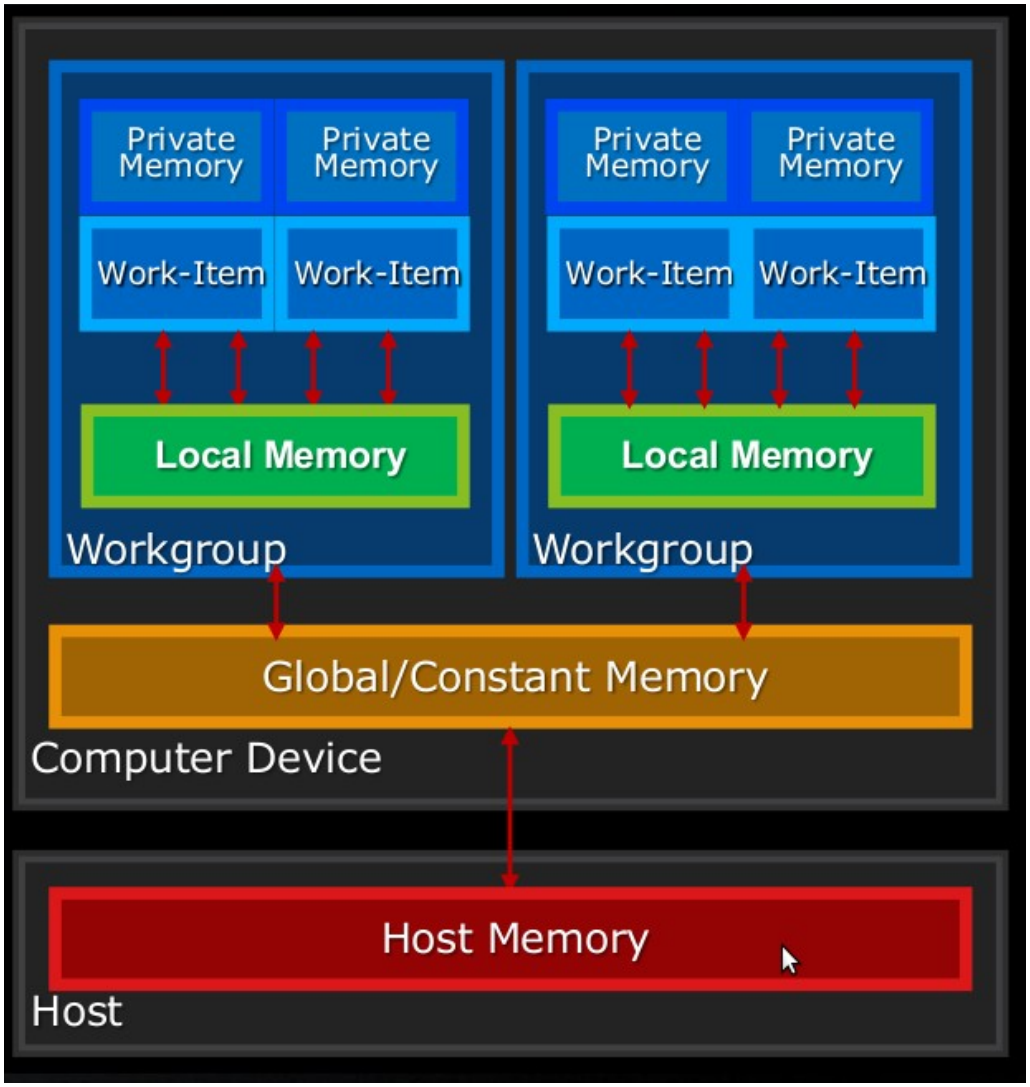


Objets OpenCL

- *contexte* : collection de plusieurs dispositif de calcul (*devices*)
  - GPUs
  - CPUs (*cores* + SSE)
  - CELL BE
- *program* : Collection de noyaux *kernels*
- Mémoires :
  - *Buffers* :1D
  - *Images* : 2D ou 3D
- Une *queue* associée à chaque *device* des ordres :
  - d'exécution de noyaux
  - copie de mémoires
- *events* : synchronisation des exécutions entre *devices*

Modèle de mémoire

- Mémoire privée (*private memory*) à chaque work-item
- Mémoire partagée (*local memory*) entre *work-items* d'un même *workgroup*
- Mémoire globale du *device*
- Mémoire du hôte



Noyaux OpenCL

```
__kernel void transpose(
          __global float *a_t, __global float *a,
          unsigned a_width, unsigned a_height,
          __local float *a_local)
        {
            int base_idx_a   =
              get_group_id(0) * get_local_size(0) +
              get_group_id(1) * a_width;
            int base_idx_a_t =
              get_group_id(1) * get_local_size(1) +
              get_group_id(0) * a_height;

          int glob_idx_a   = base_idx_a + get_local_id(0) + a_width * get_local_id(1);
          int glob_idx_a_t = base_idx_a_t + get_local_id(0) + a_height * get_local_id(1);

          a_local[get_local_id(1)*get_local_size(1)+get_local_id(0)] = a[glob_idx_a];

          barrier(CLK_LOCAL_MEM_FENCE);

          a_t[glob_idx_a_t] = a_local[get_local_id(0)*get_local_size(0)+get_local_id(1)];
        }
```

STDCL - Standard Compute Layer Interface

- Simplification de l'API en C
- #include <stdcl.h>
  - Contextes prédéfinis par défaut
  - Chargeur dynamique de programmes
  - Gestion des mémoires à la malloc/free
  - Gestion de noyaux simplifiée
  - Synchronisation Host/Devices
- OpenSource Licence LGPL3
- http://www.browndeertechnology.com/coprthr_stdcl.htm

Exemple

```
__kernel void
add(__global const float * a,
__global const float * b,
__global float * c)
{
  // Vector element index
  int nIndex = get_global_id(0);
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

Exemple

```
#include <stdcl.h>
#define SIZE 1024
int main()
{
   CONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   void* program = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel kernel = clsym(cp, program, "add", CLLD_NOW);
   float* A = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* B = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* C = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   ...
   clndrange_t ndr = clndrange_init1d(0,SIZE,64);
   clmsync(cp,0,A,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,B,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clarg_set_global(cp,k_addvec,0,A);
   clarg_set_global(cp,k_addvec,1,B);
   clarg_set_global(cp,k_addvec,2,C);
   clfork(cp, 0, kernel,&ndr, CL_EVENT_NOWAIT);
   clmsync(cp, 0,C,CL_MEM_HOST|CL_EVENT_NOWAIT);
   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT|CL_EVENT_RELEASE);
   ...
   if (A) clfree(A);
   if (B) clfree(B);
   if (C) clfree(C);
   clclose(cp,program);
}
```

Exemple

```
#include <stdcl.h>
#define SIZE 1024
int main()
{
   CONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   void* program = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel kernel = clsym(cp, program, "add", CLLD_NOW);
   float* A = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* B = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* C = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   ...
   clndrange_t ndr = clndrange_init1d(0,SIZE,64);
   clmsync(cp,0,A,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,B,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clarg_set_global(cp,k_addvec,0,A);
   clarg_set_global(cp,k_addvec,1,B);
   clarg_set_global(cp,k_addvec,2,C);
   clfork(cp, 0, kernel,&ndr, CL_EVENT_NOWAIT);
   clmsync(cp, 0,C,CL_MEM_HOST|CL_EVENT_NOWAIT);
   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT|CL_EVENT_RELEASE);
   ...
   if (A) clfree(A);
   if (B) clfree(B);
   if (C) clfree(C);
   clclose(cp,program);
}
```

Declaration de mémoires comme tableaux C

Exemple

```
#include <stdcl.h>
#define SIZE 1024
int main()
{
   CONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   void* program = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel kernel = clsym(cp, program, "add", CLLD_NOW);
   float* A = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* B = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* C = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   ...
   clndrange_t ndr = clndrange_init1d(0,SIZE,64);
   clmsync(cp,0,A,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,B,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clarg_set_global(cp,k_addvec,0,A);
   clarg_set_global(cp,k_addvec,1,B);
   clarg_set_global(cp,k_addvec,2,C);
   clfork(cp, 0, kernel,&ndr, CL_EVENT_NOWAIT);
   clmsync(cp, 0,C,CL_MEM_HOST|CL_EVENT_NOWAIT);
   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT|CL_EVENT_RELEASE);
   ...
   if (A) clfree(A);
   if (B) clfree(B);
   if (C) clfree(C);
   clclose(cp,program);
}
```

Copie dans la carte graphique

Exemple

```
#include <stdcl.h>
#define SIZE 1024
int main()
{
   CONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   void* program = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel kernel = clsym(cp, program, "add", CLLD_NOW);
   float* A = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* B = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* C = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   ...
   clndrange_t ndr = clndrange_init1d(0,SIZE,64);
   clmsync(cp,0,A,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,B,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clarg_set_global(cp,k_addvec,0,A);
   clarg_set_global(cp,k_addvec,1,B);
   clarg_set_global(cp,k_addvec,2,C);
   clfork(cp, 0, kernel,&ndr, CL_EVENT_NOWAIT);
   clmsync(cp, 0,C,CL_MEM_HOST|CL_EVENT_NOWAIT);
   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT|CL_EVENT_RELEASE);
   ...
   if (A) clfree(A);
   if (B) clfree(B);
   if (C) clfree(C);
   clclose(cp,program);
}
```

Appel au noyau

Exemple

```
#include <stdcl.h>
#define SIZE 1024
int main()
{
   CONTEXT* cp = (stdgpu)? stdgpu : stdcpu;
   void* program = clopen(cp, "add_vec.cl",CLLD_NOW);
   cl_kernel kernel = clsym(cp, program, "add", CLLD_NOW);
   float* A = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* B = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   float* C = (float*)clmalloc(cp,SIZE*sizeof(float),0);
   ...
   clndrange_t ndr = clndrange_init1d(0,SIZE,64);
   clmsync(cp,0,A,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(cp,0,B,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clarg_set_global(cp,k_addvec,0,A);
   clarg_set_global(cp,k_addvec,1,B);
   clarg_set_global(cp,k_addvec,2,C);
   clfork(cp, 0, kernel,&ndr, CL_EVENT_NOWAIT);
   clmsync(cp, 0,C,CL_MEM_HOST|CL_EVENT_NOWAIT);
   clwait(cp,0,CL_MEM_EVENT|CL_KERNEL_EVENT|CL_EVENT_RELEASE);
   ...
   if (A) clfree(A);
   if (B) clfree(B);
   if (C) clfree(C);
   clclose(cp,program);
}
```

Retour du résultat dans la mémoire

Contextes prédéfinis par défaut

- **`stddev`** : tout les *devices* OpenCL disponibles
- **`stdcpu`** : tout les CPU multi-cœurs
- **`stdgpu`** : tout les GPUs
- **`stdrpu`** : tout les processeurs reconfigurables

Chargeur dynamique de programmes

```
program = clopen(stdgpu,
"myProgram.cl",CLLD_NOW);

kernel = clsym(stdgpu, program,
"myKernel", CLLD_NOW);

...

clclose( stdgpu, program);
```

Gestion de la mémoire

```
int n= 1000 ;

float* A = (float*)

      clmalloc(stdgpu,n*sizeof(float),0);

A[50] = 1.0f;

unsigned int devnum=0 ;

clmsync(stdgpu, devnum, A ,CL_MEM_DEVICE|
CL_EVENT_NOWAIT);

...

clmsync(stdgpu, devnum, A ,CL_MEM_HOST|
CL_EVENT_WAIT |CL_EVENT_RELEASE);

...

clfree(A);
```

Gestion des noyaux

- Domaines

```
clndrange_t domain = clndrange_init2d( 0, 512, 4,
0,2048,16);
```

- Arguments du noyau

```
cl_float4 v={0.0f,0.0f,1.0f,0.0f} ;
clarg_set( stdgpu, kernel, 0, v);
clarg_set_global( stdgpu, kernel, 1, A );
clarg_set_local( stdgpu, kernel, 2, 32* sizeof(float) );
```

- Appel au noyau

```
clfork( stdgpu, devnum, kernel, &domain, CL_EVENT_WAIT);
clfork( stdgpu, devnum, kernel, &domain, CL_EVENT_NOWAIT |
CL_EVENT_RELEASE);
```

Synchronisation

- Vider la queue (appel non bloquant)

```
clflush( stdgpu, devnum, 0);
```

- Appel bloquant :

```
clwait( stdgpu, devnum,
CL_KERNEL_EVENT|CL_MEM_EVENT|
CL_EVENT_RELEASE );
```

PyOpenCL

- Interface d'OpenCL pour le langage Python
- Simplification de l'interface mais garde la totalité de possibilités
- Ajout de fonctions de haut-niveau :
  - Map/ Reduce
  - Générateur de nombres aléatoires
  - FFT via pyfft
- OpenSource licence type MIT
- http://documen.tician.de/pyopencl/

Exemple

```
import pyopencl as cl
import numpy
import numpy.linalg as la
a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()
print la.norm(a_plus_b - (a+b))
```

Références

- Site officiel OpenCL http://www.khronos.org/opencl/
- Welcome to PyOpenCL's documentation! http://documen.tician.de/pyopencl/
- Introduction to OpenCL http://developer.amd.com/zones/OpenCLZone/Events/assets/IntroductionToOpenCL-final.pdf
- OpenCL Programming in Detail http://developer.amd.com/zones/OpenCLZone/Events/assets/OpenCLProginDetailDRichiev3.pdf
- STDCL. The STandarD Compute Layer library provides a simplified programming interface for OpenCL™ http://www.browndeertechnology.com/coprthr_stdcl.htm

Plan du cours

- 1ère partie : Multi-taches et processus légers
- 2ème partie : Machine parallèle et Multi-cœurs
- 3ème partie : GPUs et CUDA
- 4ème partie : OpenCL
- **5ème partie : Exercice de synthèse**

Plan 5ème partie

- Optimisation
  - CPU
  - GPU
- Problème de N corps (travail par équipes)
  - Optimisation CPU
    - Registres
    - SSE
    - OMP
  - Optimisation GPU
    - Registres
    - Mémoire partagée
    - Dérouler les boucles

Optimisation: conditions nécessaires

- D'abord avoir un code qui marche bien!
  - Procédure de vérification / Test
  - Code de référence
- Critère de performance
- Avoir un scénario d'utilisation réaliste
  - Banc d'essai
  - Analyse de la performance (*profiling*)
    - Identifier le code critique

Le CPU

- Débit mémoire vs. Débit instructions
  - transfert CPU / RAM à 8 Go/s
    - 1 flottant 32 bits = 4 o => $2*10^9$ flottants transférés/s
  - Calcul CPU
    - SSE flottants 32 bits 4 opérations par cycle @ 3 Ghz => $4*3 = 12*10^9$ flottants traités par seconde = 12 Gflops
    - 4 cœurs => 4*12 = 48 Gflops

=> application optimale : au minimum 24 Opérations / flottant transféré

Le GPU

- Débit mémoire vs. Débit instructions
  - transfert GPU / G-RAM à 115 Go/s
    - 1 flottant 32 bits = 4 o => 29 *10^9 flottants transférés/s
  - Calcul GPU
    - 994 Gflops

  => application optimale : au minimum 34 Opérations / flottant transféré
- Mais si transfert GPU / RAM PC à 5 Go/s

  => application optimale : au minimum **795** Opérations / flottant transféré

GPU

- Vitesse GPU / Vitesse mémoire globale :
  - Dérouler les boucles => plus de traitements « utiles » par thread
  - Plus de threads par block => cacher la latence de la mémoire globale
  - Mais limité par la taille du jeu de registres
- Utiliser la mémoire partagée:
  - Aussi rapide que les registres ~ 1 cycle / 400 cycles pour mémoire globale
  - Changer les algorithmes pour partager les données
  - Mais limité par la taille de la mémoire partagée
- limiter la divergence des threads
  - contrôler les « if »

Exemple d'optimisation

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x
Kernel 6: completely unrolled	0.381 ms	43.996 GB/s	1.41x	21.16x
Kernel 7: multiple elements per thread	0.268 ms	62.671 GB/s	1.42x	30.04x

Kernel 7 on 16M elements: 72 GB/s!

http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf

Exercice: Problème à N corps

- Application
  - Astronomie
  - Physique des matérieaux
  - Simulation mécanique
  - Simulation de foules

Définition

- Particules $n \gg 1$
- Calcul de l'interaction pour chaque particule

$$i \in [0, n-1]$$

$$F(i) = \sum_{j=0}^{j=n-1} f(x_i, x_j)$$

- Avec (ici pour simplifier) :

$$f(x_i, x_j) = x_i - x_j$$

Première version CPU

- ```
  inline float f(float xi, float xj)
  { return xi – xj; }
  ```

- ```
  void calcul( float* F, float* X, unsigned int len)
  {
     F[i]=0.0f;
     for( unsigned int i = 0; i < len; i++)
        for( unsigned int j = 0; j < len; ++j)
           F[i] = F[i] + f(X[i],X[j]);
  }
  ```

Expérimentations

- Nombre de Particules
  - n=1024000
- GPU :
  - NVIDIA GTX 280
- CPU :
  - bi-quadcore Intel Xeon CPU X5482 à 3.20GHz (2x4 cœurs).
- Mesure du temps :
  - Inclus le temps de transferts des données et de retour du résultat entre la mémoire centrale du PC et la carte graphique.

Résultats 1 cœur CPU

```
void computeF( float* F,
float*  X, const unsigned int
len)
{
 for(unsigned int i = 0; i <
     len; i++)
 {
  float f_i= 0.0f;
  for(unsigned int j = 0; j <
      len; ++j)
    f_i = f_i + X[i]-X[j];
  F[i] = f_i;
 }
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
1 cœur CPU	19749,4	1061,9	

Résultats 1 cœur CPU + SSE

```
void computeF_SSE( float*  F, float*
X, const unsigned int len)
{
  for( unsigned int i = 0 ; i < len;
i+=4)
  {
  __m128 f_i, x_i;
  f_i=_mm_set_ps1(0.0f);
  x_i=_mm_load_ps(X+i);
  for( unsigned int j = 0;j < len;++j)
  {
      __m128 x_j=_mm_set_ps1(X[j]);
      f_i = f_i + x_i - x_j;
  }
  _mm_store_ps((F+i), f_i );
  }
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
1 cœur CPU	19749,4	1061,9	
1 cœur CPU + SSE	4934,6	4249,9	

Résultats 8 cœurs CPU OMP+ SSE

```
void computeF_SSE( float*  F, float*  X,
const unsigned int len)
{
  for( unsigned int i = 0 ; i < len;
i+=4)
  {
    __m128 f_i, x_i;
    f_i=_mm_set_ps1(0.0f);
    x_i=_mm_load_ps(X+i);
#pragma omp parallel for reduction(+:f_i)
shared(X)
    for( unsigned int j = 0;j < len;++j)
    {
      __m128 x_j=_mm_set_ps1(X[j]);
        f_i = f_i + x_i - x_j;
    }
    _mm_store_ps((F+i), f_i );
  }
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
1 cœur CPU	19749,4	1061,9	
1 cœur CPU + SSE	4934,6	4249,9	
8 cœur CPU OMP + SSE	**698,8**	30010,8	

Résultats Version 0 CUDA

```
__global__ void Cal_F_v0
( float * X, unsigned int
n, float * F)
{
 unsigned int i =
blockDim.x * blockIdx.x +
threadIdx.x ;
 F[i] = 0.0f ;
 for(int j = 0; j < n; j++)
   F[i] = F[i] +
f(X[i],X[j]) ;
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
OMPx8 + SSE	**698,8**	30010,8	
Bloc 32	943,3	22232,1	
Bloc 64	**847,3**	24751,0	
Bloc 128	854,7	24536,7	
Bloc 256	854,0	24556,8	
Bloc 512	862,6	24312,0	

Résultats Version 0 CUDA

```
__global__ void Cal_F_v0
( float * X, unsigned int
n, float * F)
{
 unsigned int i =
blockDim.x * blockIdx.x +
threadIdx.x ;
 F[i] = 0.0f ;
 for(int j = 0; j < n; j++)
   F[i] = F[i] +
f(X[i],X[j]) ;
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
OMPx8 + SSE	**698,8**	30010,8	
Bloc 32	943,3	22232,1	133392,9
Bloc 64	847,3	24751,0	**148506,5**
Bloc 128	854,7	24536,7	147220,7
Bloc 256	854,0	24556,8	147341,4
Bloc 512	862,6	24312,0	145872,4

Résultats Version 1 CUDA

```
__global__ void Cal_F_v0
( float * X, unsigned int n,
float * F)
{
 unsigned int i =  blockDim.x
* blockIdx.x + threadIdx.x ;
 float f_i= 0.0f ;
 float x_i= X[i];
 for(int j = 0; j < n; j++)
   f_i = f_i + f(x_i,X[j]) ;
 F[i]=f_i;
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
OMPx8 + SSE	698,8	30010,8	
Bloc 32	568,6	36882,73	73766,9
Bloc 64	337,8	62082,65	124167,7
Bloc 128	284,2	73791,41	147585,7
Bloc 256	283,4	73999,72	148002,3
Bloc 512	**282,4**	74261,76	**148526,4**

Utilisation de la mémoire version 1

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

X_i

Pour i=0

Utilisation de la mémoire version 1

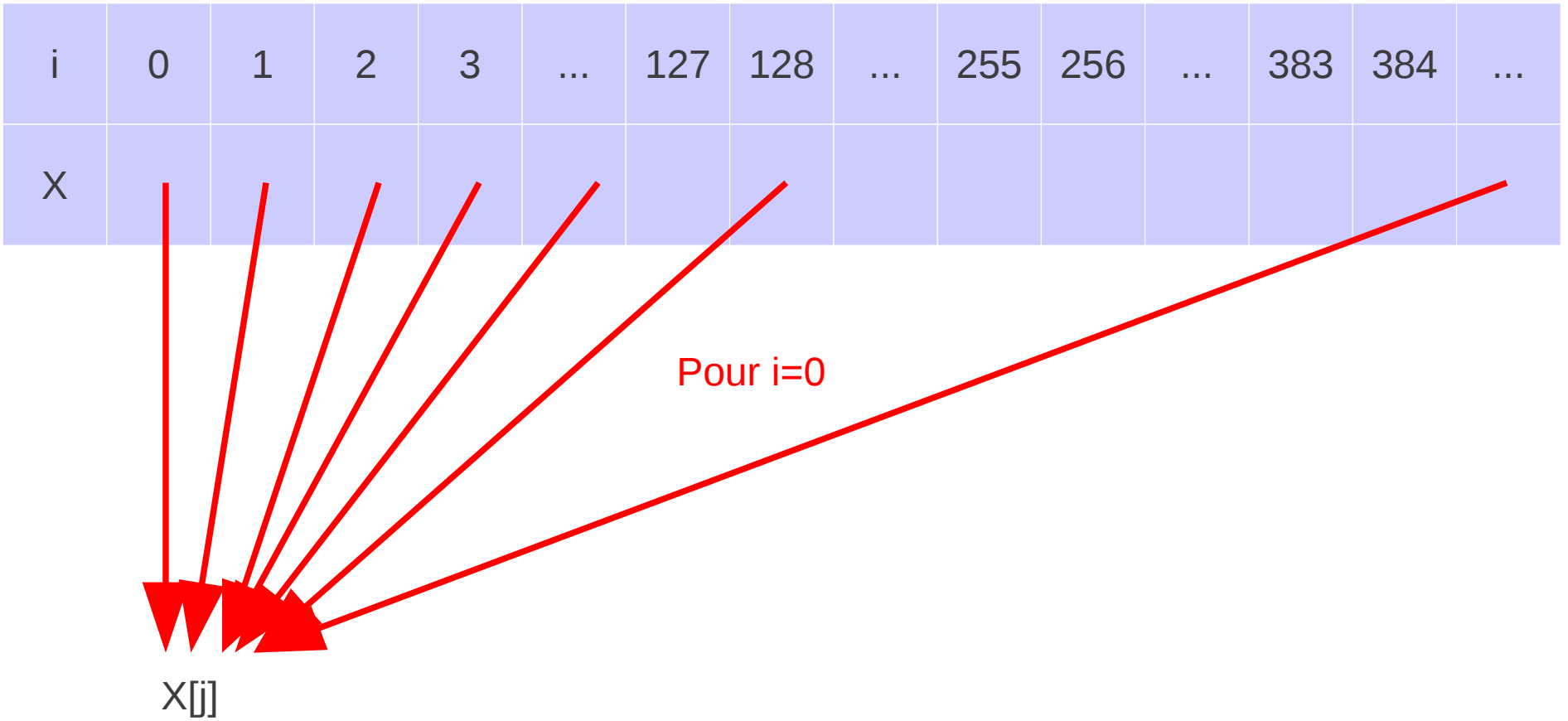
i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

Accès en lecture mémoire globale

X_i

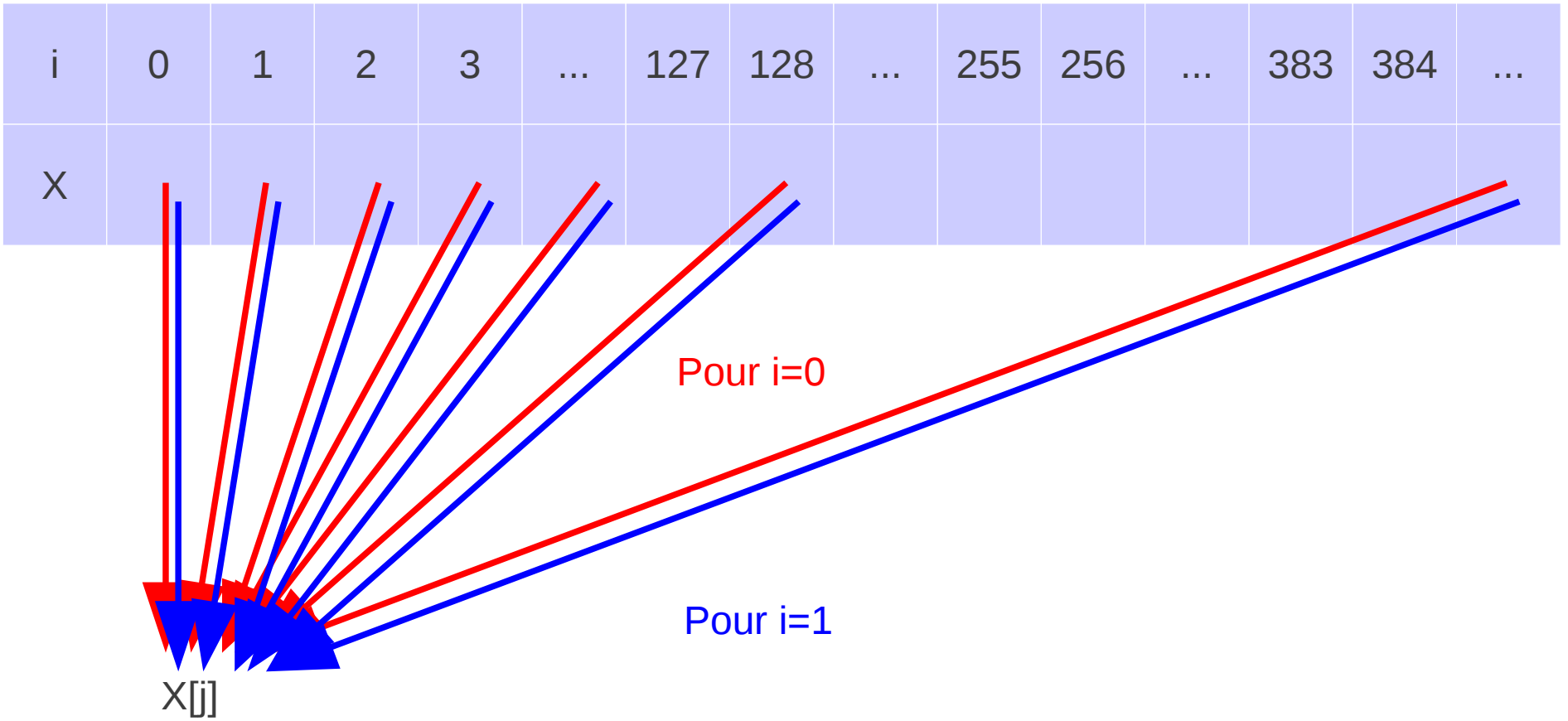
Pour i=0

Utilisation de la mémoire version 1



Puis : pour J dans [0,n[

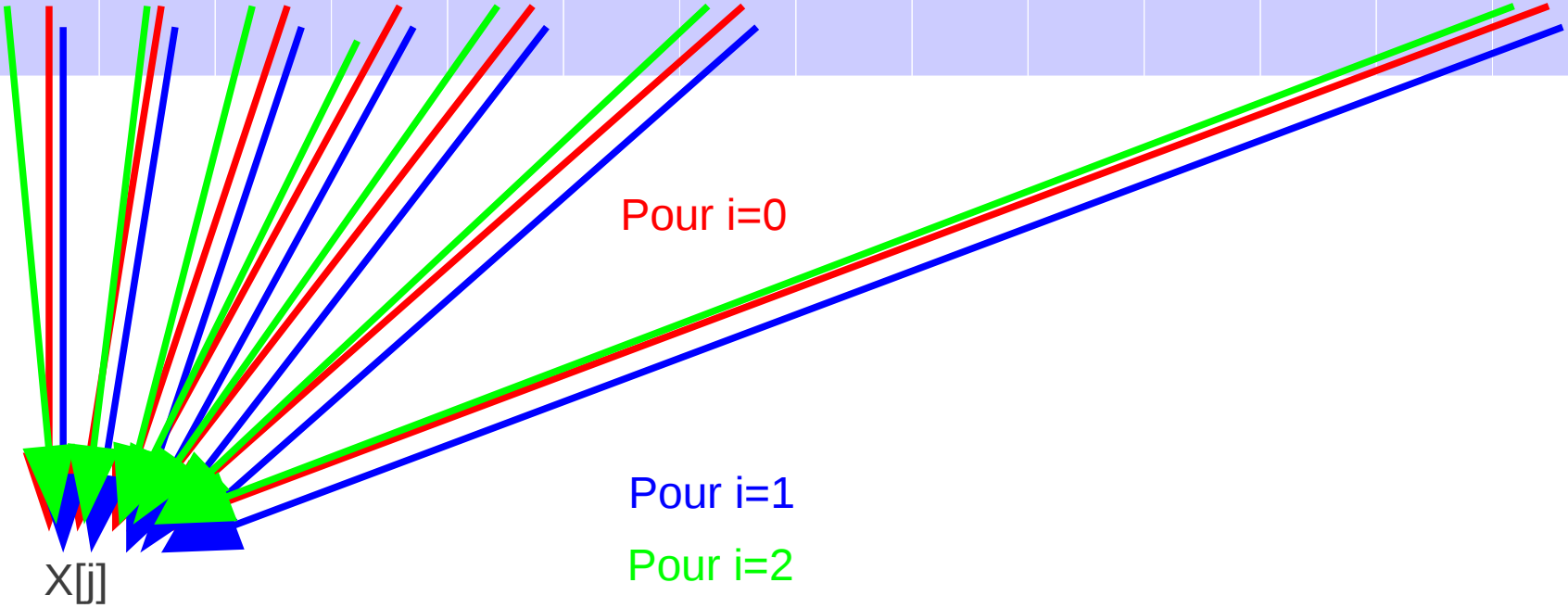
Utilisation de la mémoire version 1



Puis : pour J dans [0,n[

Utilisation de la mémoire version 1

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														



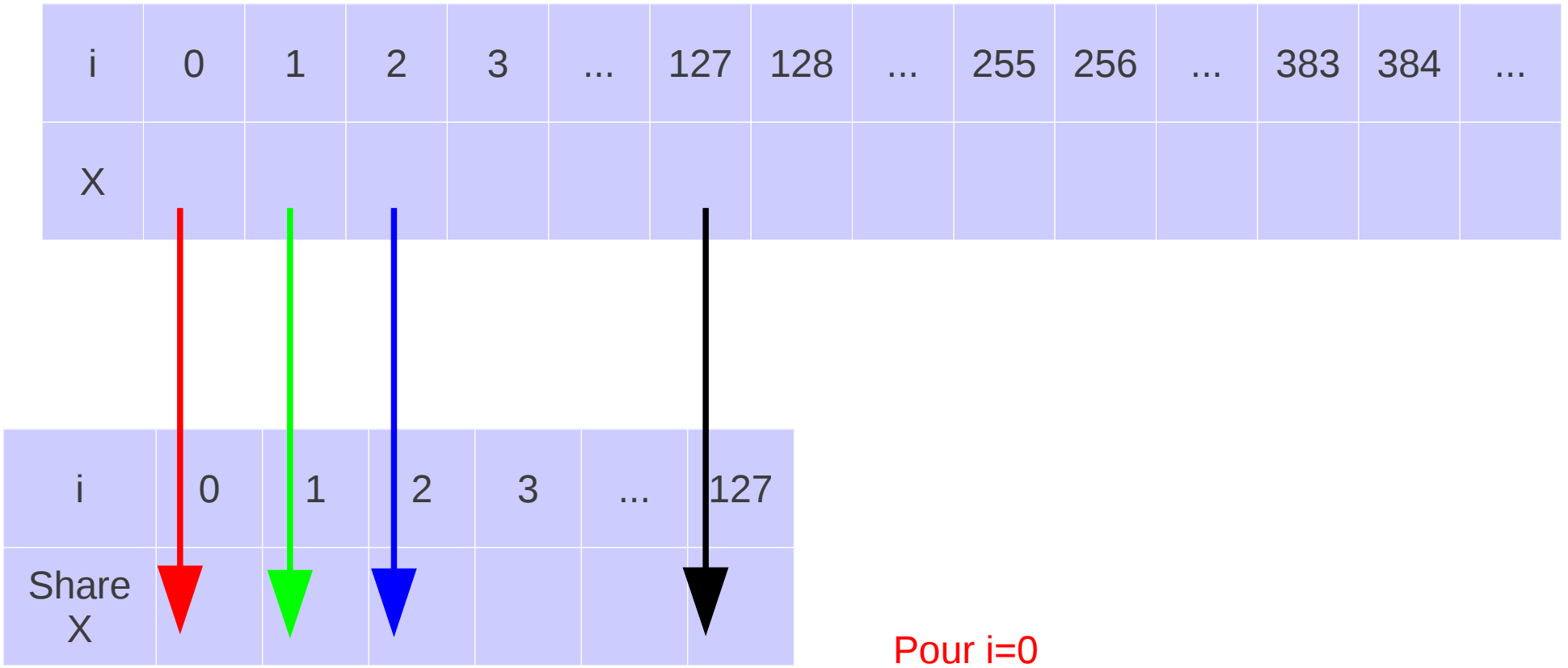
Puis : pour J dans [0,n[

Utilisation de la mémoire partagée

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

i	0	1	2	3	...	127
Share X						

Utilisation de la mémoire partagée



Pour i=0

Pour i=1

Pour i=2

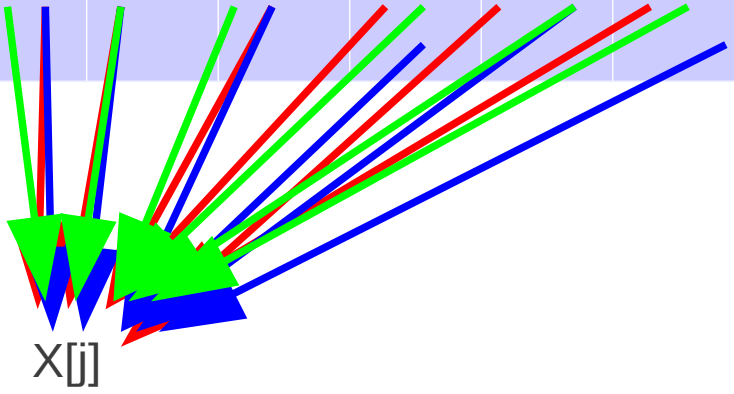
...

Pour i=127

Utilisation de la mémoire partagée

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

i	0	1	2	3	...	127
Share X						



Pour i=0

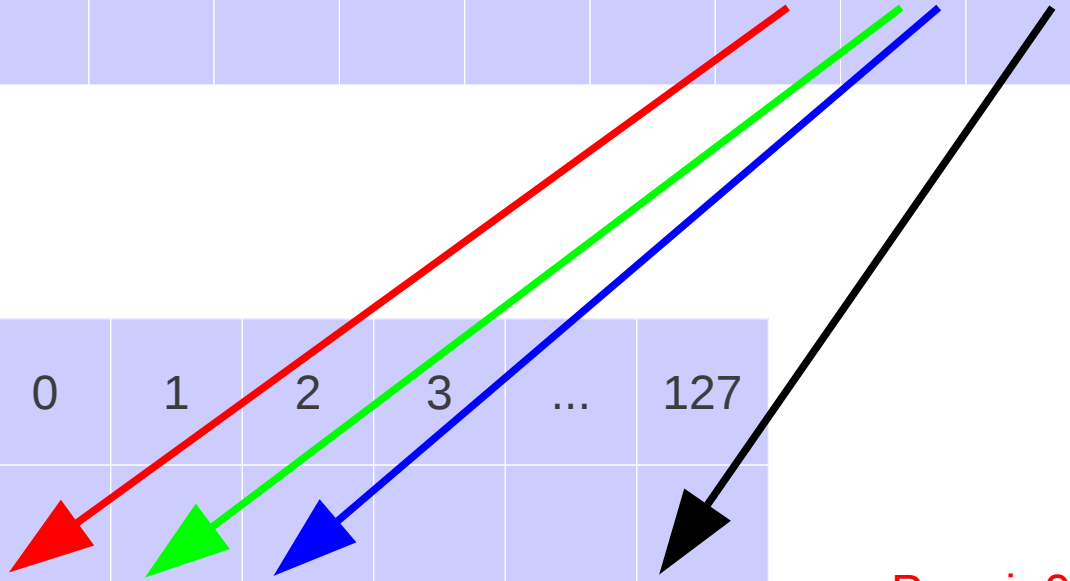
Pour i=1

Pour i=2

Puis : pour J dans [0,**127**]

Utilisation de la mémoire partagée

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														



i	0	1	2	3	...	127
Share X						

Pour i=0

Pour i=1

Pour i=2

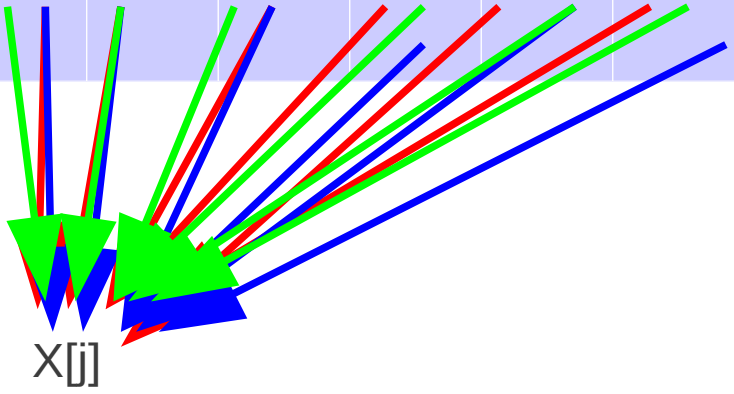
...

Pour i=127

Utilisation de la mémoire partagée

i	0	1	2	3	...	127	128	...	255	256	...	383	384	...
X														

i	0	1	2	3	...	127
Share X						



Pour i=0

Pour i=1

Pour i=2

Puis : pour J dans [**128**,**255**]

Résultats Version 2 CUDA

```
__global__ void Cal_F_v2 ( float * X,
unsigned int n, float * F)
{
 extern  __shared__  float shareX[];
 unsigned int i = blockDim.x * blockIdx.x
+ threadIdx.x ;
 float f_i= 0.0f ;
 float x_i= X[i];
 for(unsigned int k = 0; k < n;
k+=blockDim.x)
 {
  shareX[threadIdx.x]=X[k+threadIdx.x];
  __syncthreads();
  for(unsigned int j = 0; j < blockDim.x;
j++)
    f_i = f_i + f(x_i,shareX[j]) ;
  __syncthreads();
 }
 F[i]=f_i;
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
OMPx8 + SSE	698,8	30010,8	
Bloc 32	204,1	102751,2	6422,1
Bloc 64	193,3	108492,1	3390,4
Bloc 128	**188,1**	111491,3	1742,1
Bloc 256	193,8	108212,2	845,4
Bloc 512	193,3	108492,1	423,8

Déroulement de boucles

CODE CUDA

```
for(unsigned int j = 0; j < blockDim.x; j++)
  f_i = f_i + f(x_i,shareX[j]) ;
```

CODE PTX généré

```
$Lt_0_14:
.loc  16  73  0
ld.shared.f32  %f4, [%rd12+0];
shareX+0x0
sub.f32  %f5, %f1, %f4;
add.f32  %f2, %f2, %f5;
add.u32  %r11, %r11, 1;
add.u64  %rd12, %rd12, 4;
setp.ne.u32  %p3, %r11, %r1;
@%p3 bra $Lt_0_14;
```

Déroulement de boucles

CODE CUDA

```
for(unsigned int j = 0; j < blockDim.x; j++)
  f_i = f_i + f(x_i,shareX[j]) ;
```

CODE PTX généré

```
$Lt_0_14:
.loc  16  73  0
ld.shared.f32  %f4, [%rd12+0];
shareX+0x0
sub.f32  %f5, %f1, %f4;
add.f32  %f2, %f2, %f5;
add.u32  %r11, %r11, 1;
add.u64  %rd12, %rd12, 4;
setp.ne.u32  %p3, %r11, %r1;
@%p3 bra $Lt_0_14;
```

Opérations « productives » de l'algorithme

Opérations «nécessaires» au maintien de la boucle mais « improductives »

Déroulement de boucles

CODE CUDA version 2 bis

```
for(unsigned int j = 0; j < blockDim.x; j+=2)
{
    f_i = f_i + f(x_i,shareX[j]) ;
    f_i = f_i + f(x_i,shareX[j+1]) ;
}
```

Doublement du nombre d'opérations « productives »

Même nombre d'opérations «nécessaires» au maintien de la boucle

CODE PTX généré

```
$Lt_2_14:
.loc  16  128 0
ld.shared.f32  %f4, [%rd12+0];
sub.f32  %f5, %f1, %f4;
add.f32  %f2, %f2, %f5;
.loc  16  129 0
ld.shared.f32  %f6, [%rd12+4];
shareX+0x0
sub.f32  %f7, %f1, %f6;
add.f32  %f2, %f2, %f7;
add.u32  %r16, %r16, 2;
add.u64  %rd12, %rd12, 8;
setp.lt.u32  %p3, %r16, %r1;
@%p3 bra $Lt_2_14;
```

Résultats Version 2 bis CUDA

```
__global__ void Cal_F_v2bis ( float * X,
unsigned int n, float * F)
{
 extern  __shared__  float shareX[];
 unsigned int i = blockDim.x * blockIdx.x
+ threadIdx.x ;
 float f_i= 0.0f ;
 float x_i= X[i];
 for(unsigned int k = 0; k < n;
k+=blockDim.x)
 {
  shareX[threadIdx.x]=X[k+threadIdx.x];
  __syncthreads();
  for(unsigned int j = 0; j < blockDim.x;
j+=2)
    f_i = f_i + f(x_i,shareX[j]) ;
    f_i = f_i + f(x_i,shareX[j+1]) ;
  __syncthreads();
 }
 F[i]=f_i;
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
OMPx8 + SSE	698,8	30010,8	
Bloc 128 Ver. 2	188,1	111491,3	1742,1
Bloc 128 Ver. 2 bis	**138,0**	151956,5	2374,4

Résultats Version 3 CUDA

(déroulement automatique de la boucle)

```
__global__ void Cal_F_v3 ( float * X,
unsigned int n, float * F)
{
 extern  __shared__  float shareX[];
 unsigned int i = blockDim.x * blockIdx.x
+ threadIdx.x ;
 float f_i= 0.0f ;
 float x_i= X[i];
 for(unsigned int k = 0; k < n;
k+=blockDim.x)
 {
  shareX[threadIdx.x]=X[k+threadIdx.x];
  __syncthreads();
  #pragma unroll 128
  for(unsigned int j = 0; j < blockDim.x;
j++)
    f_i = f_i + f(x_i,shareX[j]) ;
  __syncthreads();
 }
 F[i]=f_i;
}
```

n=102400	Temps en *ms*	*Calcul Mflops*	*Débit Mo/s*
OMPx8 + SSE	698,8	30010,8	
Bloc 128 Ver. 2	188,1	111491,3	1742,1
Bloc 128 Ver. 2 bis	138,0	151956,5	2374,4
Bloc 128 Ver. 3	**90,6**	231473,7	3616,8