Architecture

Bertrand Granado
Enseignant-Chercheur

ETIS / ENSEA
Mel : Bertrand.Granado@ensea.fr

Automne 2009

Why ?

La question

- Pourquoi faire du parallélisme ?

Mauvaise question

- Le monde est parallèle, la bonne question est de savoir pourquoi nous n'avons pas déjà exploité électroniquement ce parallélisme !

Qu'est-ce que le parallélisme ?

Traitement logiciel traditionnel

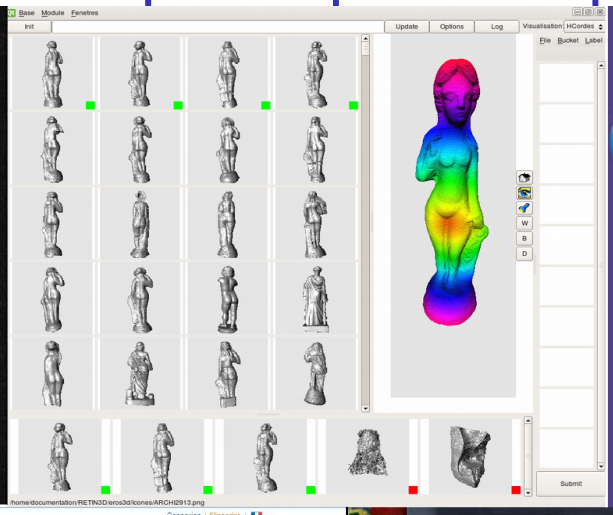
- Traitement série :
  - exécution sur un unique ordinateur doté d'un unique processeur
  - spécifiation d'une application sous forme d'une série d'instructions discrètes
  - les instruction sont exécutées les unes à la suite des autres
  - une seule instruction est exécutée à un instant donné.

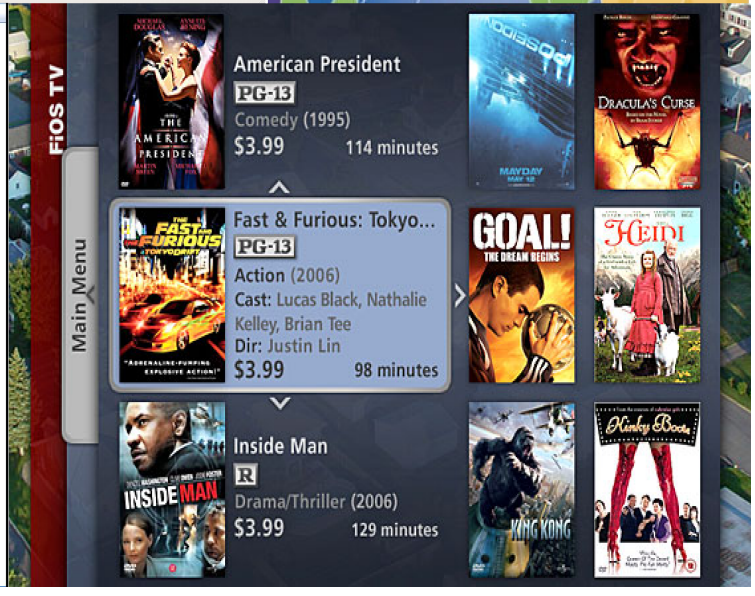
Qu'est-ce que le parallélisme ?

Traitement logiciel parallèle

- En première approche, le traitement parallèle est l'utilisation simultanée de plusieurs ressources de calcul pour traiter une application traditionnelle :
  - exécution sur plusieurs ordinateurs qui peuvent contenir plusieurs processeurs
  - spécification de l'application en parties pouvant être résolue en concurrence
  - les instruction de chaque parties pouvant s'exécuter simultanément sur différents processeurs

Le parallélisme pour quelles applications ?





Le parallélisme : pour quoi faire ?

Efficacité

L'idée de base est de multiplier les ressources matérielles (les processeurs de calcul) pour diviser le temps d'exécution d'autant. Le facteur de gain linéaire (on va N fois plus vite si on utilise N processeurs) n'est pas facile à maintenir en pratique dès que le nombre de processeurs devient grand.

Efficacité

- Question : Si 1 TGV met 2 heures pour faire Paris-Lyon combien mettent 2 TGV en parallèle ?
- Question : Si 1 TGV met 2 heures pour transporter 400 personnes combien mettent 2 TGV en parallèle ?

Le parallélisme : pour quoi faire ?

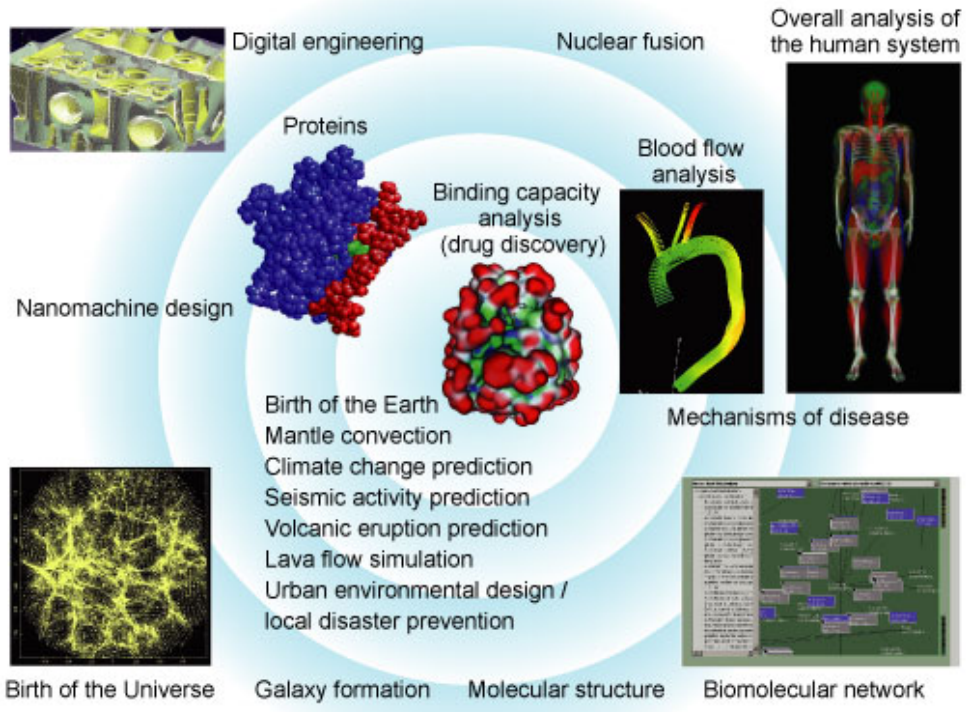
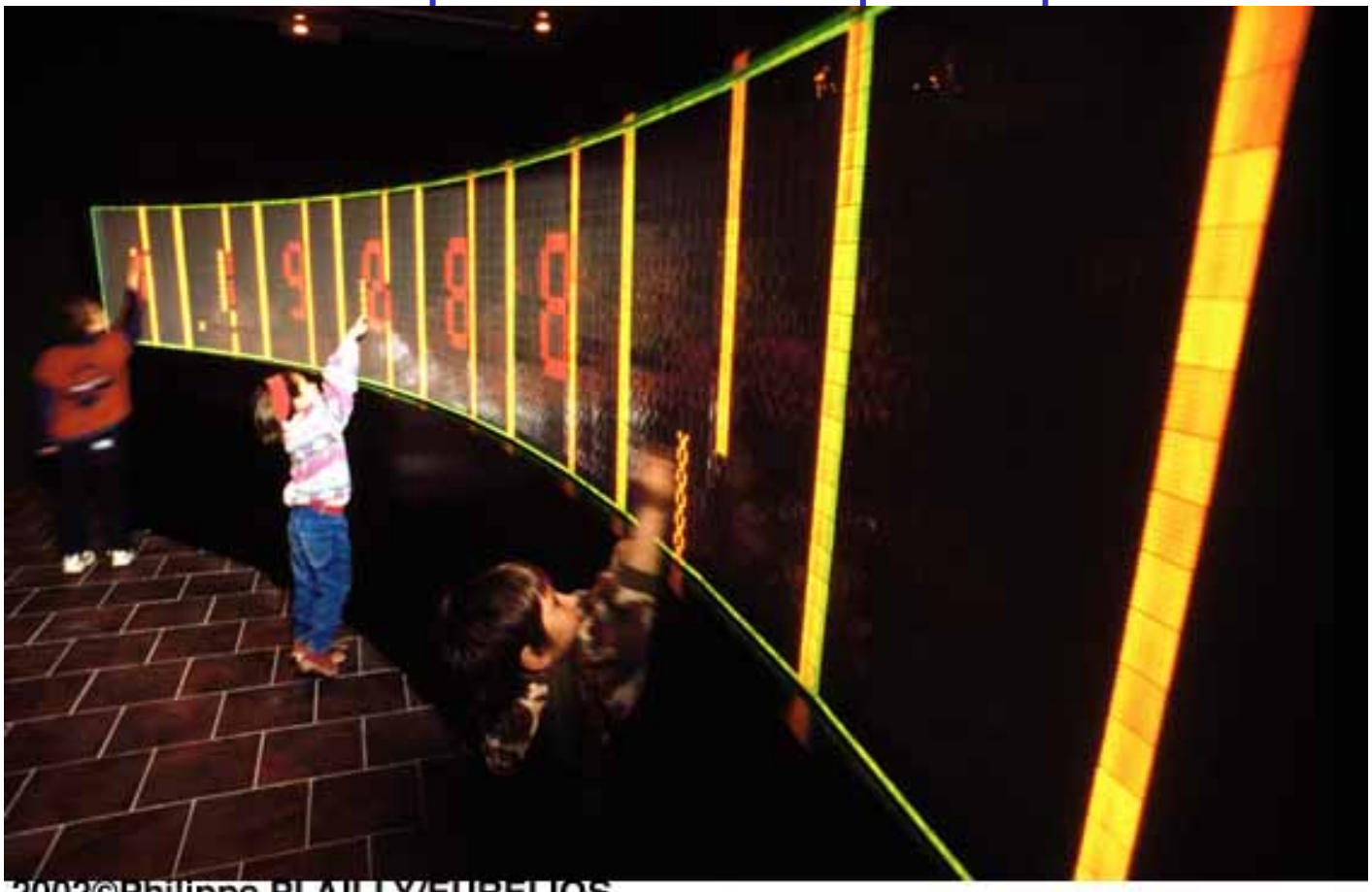


Figure 1: Possible fields of application for a 10-petaflop computer running a "multiphysic" simulation that can deal simultaneously with different physical phenomena on multiple levels from the micro to the macro scale.

Le parallélisme : pour quoi faire ?



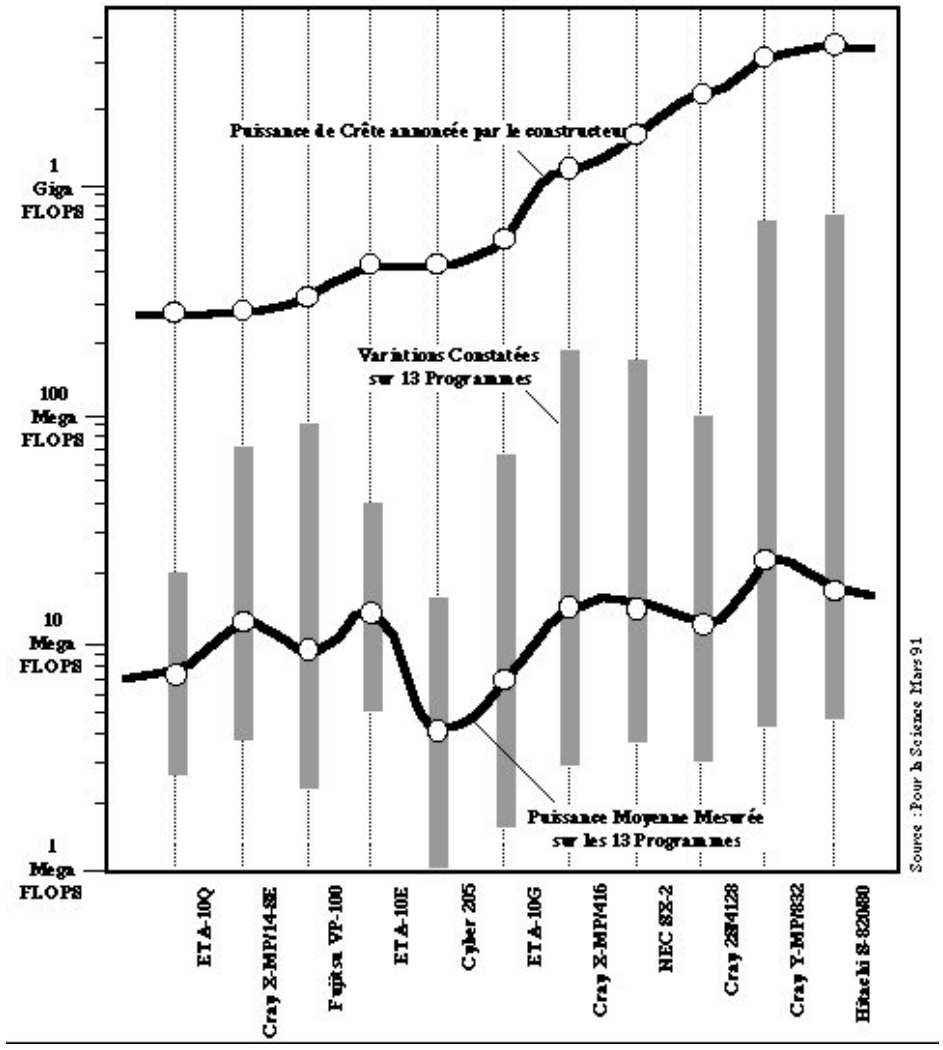
2002©Philippe PLAILLY/EURELIOS
Cybernetique/ Sainte Croix / Villa Reuge/ Exposition Biowall/ Experience "Counter"/Auto réparation moléculaire

Exploitation du parallélisme

Différentes phases

1. exhibition (mise en évidence du caratère parallèle de l'application)
2. extraction (utilisation de modèles de calcul pour extraire le parallélisme)
3. exécution (utilisation d'outils logiciels et matériels de mise en oeuvre de modèles de calcul)

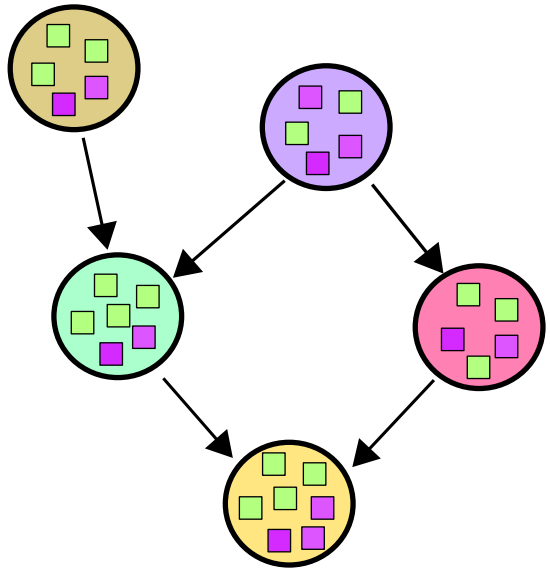
Exploitation du parallélisme

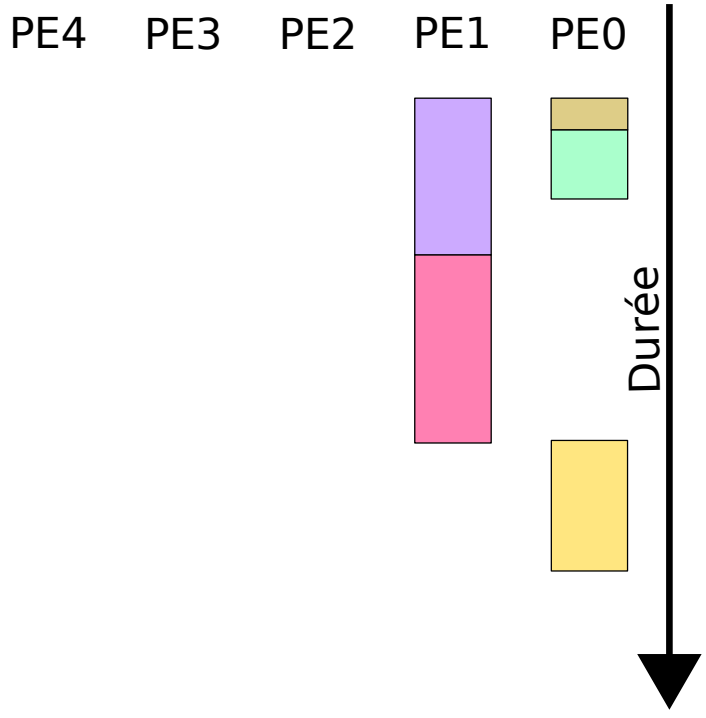


Les sources du parallélisme

- Il existe trois différentes sources de parallélismes
  - le Contrôle
  - le Flux
  - les Données

Parallélisme de Contrôle





Parallélisme de Contrôle

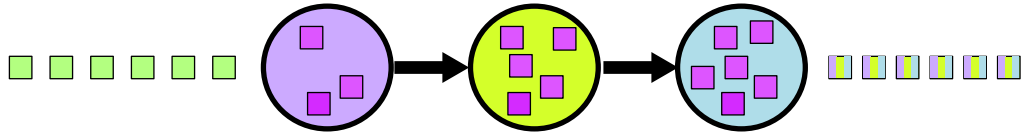
Les dépendances

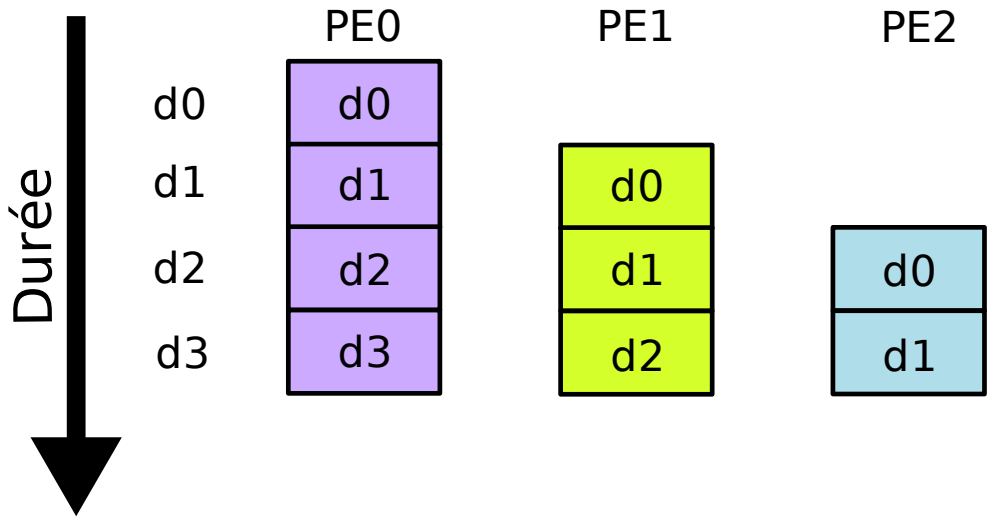
- Dépendance de contrôle de séquence : correspond au séquencement dans un algorithme classique,
- Dépendance de contrôle de communication : lorsqu'une action envoie des informations à une autre action.

Parallélisme de Contrôle

- Extraction d'attributs d'une région
- Taille d'une région
- Texture d'une région

Parallélisme de Flux





Parallélisme de Flux

Sources du parallélisme de flux

1. Données de type vectoriel placées en mémoire. Très forte avec le cas du parallélisme de données. La différence réside dans le placement spatio-temporel des données par rapport aux actions.
2. Données de type scalaire provenant d'un dispositif d'entrée. Environnement temp-réel. Le Flux de données continu peut être considéré comme une source infinie.

Parallélisme de Flux

Gain obtenu

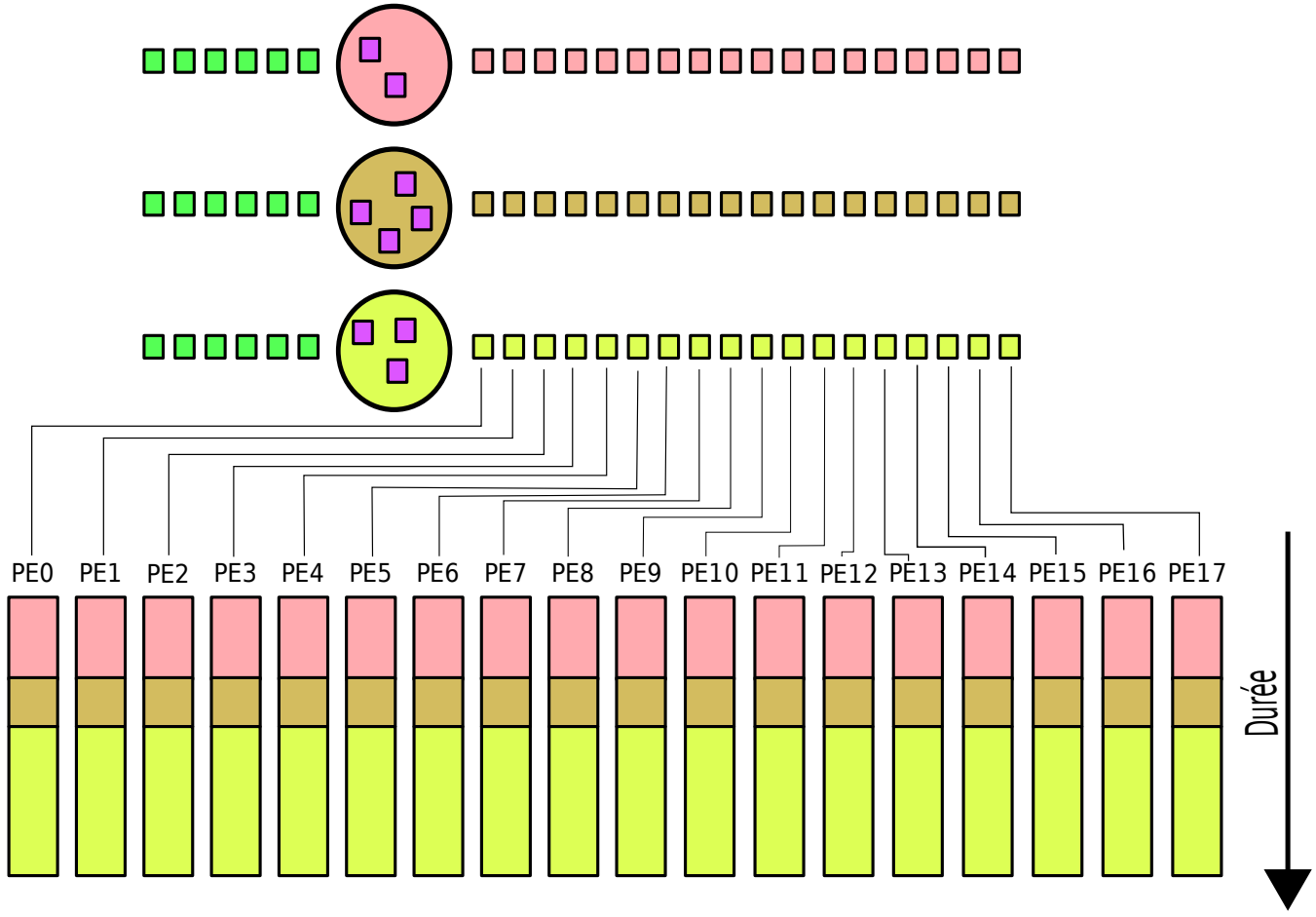
Le gain obtenu est linéaire par rapport au nombre de PEs en mode de croisière ; au début, tous les PEs ne sont pas occupés tant que la première donnée n'a pas traversé tout le pipe-line ; il en est de même en fin de flux.

Si le flux présente de fréquentes discontinuités les phases transitoires de début et de fin peuvent diminuer sérieusement le gain.

Parallélisme de Flux

- Chaîne de traitement d'image

Parallélisme de Données



Parallélisme de Données

Calcul Vectoriel

Le programmeur raisonne comme pour un programme séquentiel dont les actions portent sur des données vectorielles

Stop : Scalaire !

Les applications qui manipulent des données vectorielles utilisent aussi des scalaires !
Si une tache ne travaille que sur une donnée, alors un seul PE travaille sur le scalaire pendant que les N-1 PEs restants sont inactifs.
Efficacité de la machine parallèle grandement diminuée d'autant plus que le nombre de PEs est grand.

Parallélisme de Données

- Binarisation par seuillage par LUT
- Convolution pour filtrage (Effet de Bord)
- Majorité d'algorithme de Traitement d'Image Bas Niveau

Présence des Parallélismes

- Parallélisme de Contrôle : 3
- Parallélisme de Flux : 30
- Parallélisme de Données : 1 000 0000

Présence des Parallélismes

Contrôle

Le Parallélisme de Contrôle est lié à la largeur de l'expression fonctionnelle du programme source : cela revient à dire qu'il est lié au nombre des arguments dans une Fonction : de nombreuses mesures faites sur de grosses applications fonctionnelles écrites en Lisp font apparaître un taux moyen de 1,7 arguments. Des mesures faites sur des programmes écrits en Fortran montrent que le parallélisme de contrôle est de deux ou trois en moyenne.

Présence des Parallélismes

Flux

Le Parallélisme de Flux est lié à la profondeur de l'expression fonctionnelle du programme source : cela revient à dire qu'il est lié au nombre des imbrications statiques des fonctions dans un programme : on peut considérer que pour les grosses applications il est compris entre 30 et 100. Pour les applications de contrôle/commande, il est plus proche de la dizaine seulement et pour les applications de traitement du signal ou image le nombre d'étages (c'est-à-dire de filtres) est de quelques unités.

Présence des Parallélismes

Données

Le Parallélisme de Données est lié aux nombre des éléments dans les données vectorielles : en moyenne, on traite des matrices pleines de l'ordre de 1000x1000 éléments. Mais, les applications qui manipulent des vecteurs utilisent aussi des scalaires : en moyenne, on compte une opération vectorielle pour quatre opérations scalaires ! Cela fait très vite tomber l'efficacité.

Présence des Parallélismes

Exemple

Soit 3 vecteurs A,B de 1000 éléments et C de 100 éléments. Et soit le traitem

```
        Pour chaque élément des vecteur A et B faire
                element(A)=element(A)*element(A)
                element(B)=element(B)*element(B)
                temp=element(A) + element(B)
                Si temp > 3,14 alors
                        element(C) = element(A) / 10
                Sinon
                        element(C) = temps
                Fin Si
        Fin Pour
Fin
```

Limitation des sources de parallélisme

Contrôle

- Dépendances Temporelles : Elle sont liées au séquencement et à la communication entre les actions.
- Dépendances Spatiales : Elles se produisent lorsqu'on associe des ressources matérielles aux composants logiques de l'application. S'il n'y a pas assez de ressources (ALUs, mémoire, Liens de communication, ...) un phénomène de Contention se produit : il faut multiplexer.
- Overhead de gestion des grains : Lorsqu'on diminue la taille des grains de parallélisme le temps de gestion (activation, suspension, transmission d'un évènement ...) a tendance à devenir de plus en plus grand relativement au travail effectué par les grains.

Limitation des sources de parallélisme

Flux

- Transitoires : Lors de la phase de démarrage, tous les PEs ne travaillent pas tant qu'ils n'ont pas encore été alimentés ; cette phase dure N cycles si le nombre d'étages de la machine est N. Le même phénomène se produit lors de l'arrêt. Si on utilise le parallélisme de Flux pour exploiter des données en mode pipe-line (machines Vectorielles), alors on a une transitoire pour chaque donnée : la perte de parallélisme introduite par les transitoires de départ et d'arrêt est inversement proportionnelle à la taille de la donnée.
- Branchements : Dans une machine organisée en pipe-line, les branchements provoquent des ruptures du flux qui, à leur tour, nécessitent des transitoires pour redémarrer.

Limitation des sources de parallélisme

Données

- Vecteurs : Si la taille de la machine parallèle support est plus petite que la taille des données vectorielles utilisées, il faut les couper en tranches : cela produit un effet de multiplexage auquel il faut ajouter le temps de sa gestion.
- Scalaires : Les applications qui utilisent des données vectorielles utilisent aussi des données scalaires. Pendant qu'elle traite un scalaire, un seul PE de la machine travaille !
- Opérations de diffusion et de réduction : Les opérations sur les données vectorielles ne sont pas toutes purement parallèles (a-notations) il existe aussi des opérations qui mettent en jeu un scalaire vers un vecteur (diffusion) ou un vecteur vers scalaire (b-réduction). Dans ces cas, le nombre d'opérations passe de 1 à N en logarithme (resp. de N à 1) et le nombre de PEs inactifs s'accroît (resp. décroît).

Grain et Degré

Grain

Le Grain de parallélisme est défini comme la taille moyenne des actions mesurées en

- nombre d'instructions exécutées par tache (1, 10, 1000 ...)
- nombre de mots mémoire utilisés (1, 10, 1000 ...)

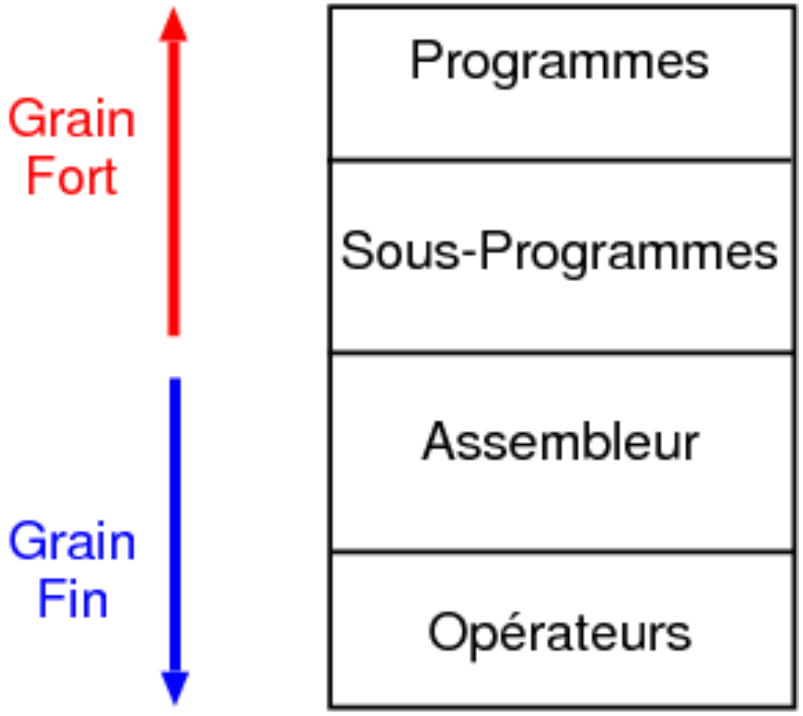
Degré

Le Degré de parallélisme est défini comme le nombre d'actions dans l'application mesuré en :

- Statique : nombre total d'actions dans l'application,
- Dynamique : nombre moyen d'actions exécutées en parallèle lors de l'exécution de l'application.

Grain du Parallélisme

- Définition du parallélisme en terme de grosseur de tâches exécutées.

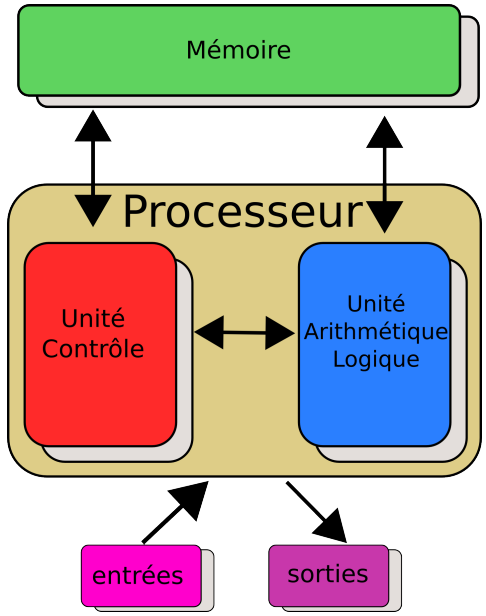


Classification du Parallélisme

- Classification de Flynn

	Flux d'instruction	
Flux de Donnée	SISD (Von Neumann)	SIMD (Processeurs Elementaire)
	MISD (Pipeline ?)	MIMD (Grappe d'ordinateurs)

SISD



Le modèle historique

- Appelé ainsi après que John von Neumann, un mathématicien hongrois, est édicté les besoins électroniques généraux pour réaliser un ordinateur en 1945.
- Depuis lors, extérieurement, tous les ordinateurs ont respecté ce schéma

SISD

Univac

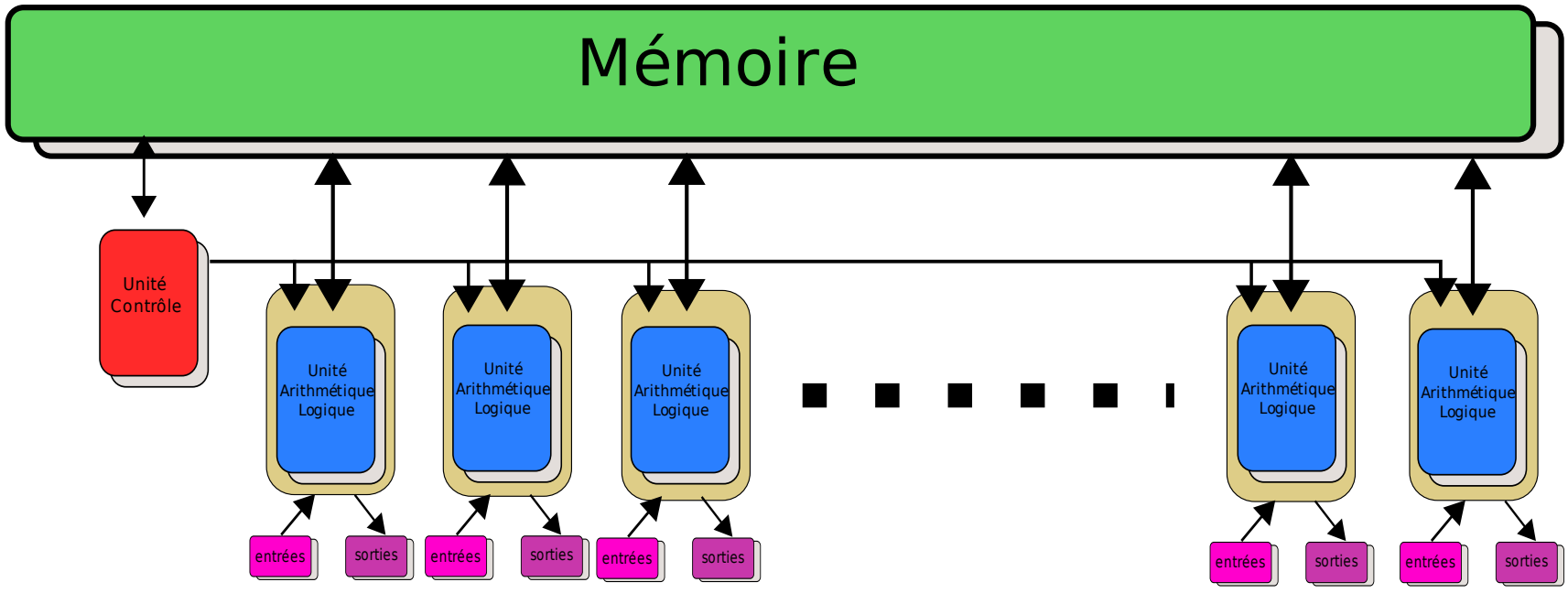
IBM 360

CDC 7600

PDP 1

iBook

SIMD


Mémoire
Unité Contrôle
Unité Arithmétique Logique
Unité Arithmétique Logique
Unité Arithmétique Logique
Unité Arithmétique Logique
Unité Arithmétique Logique
entrées
sorties
entrées
sorties
entrées
sorties
entrées
sorties
entrées
sorties

SIMD

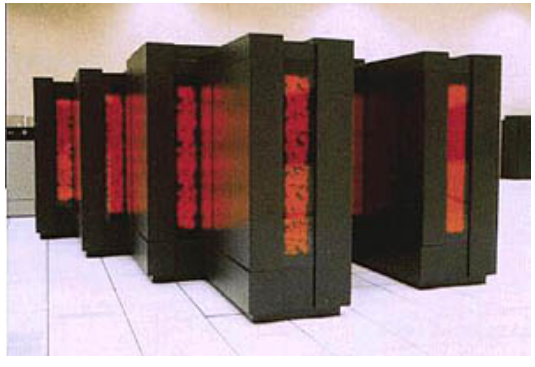
ILLIAC IV

MasPar

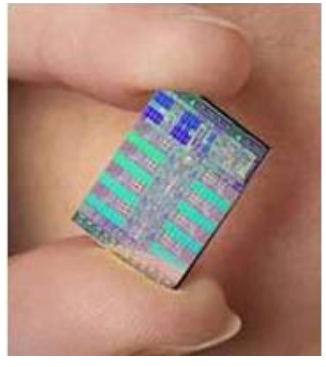
Cray X-MP

Cray Y-MP

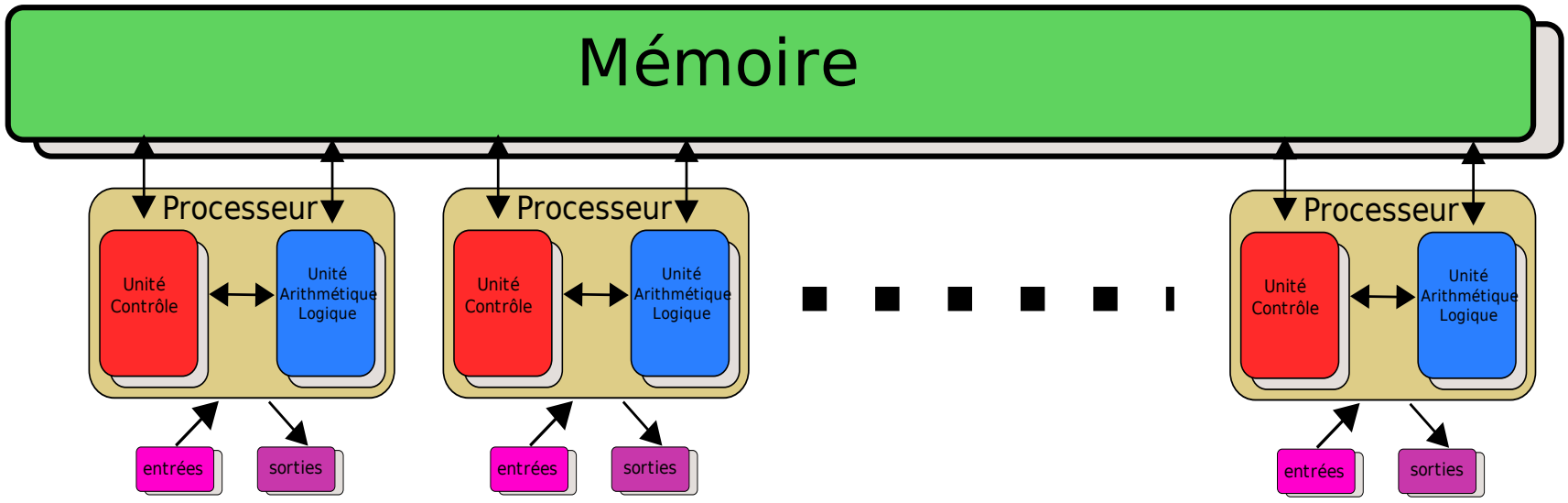
CM-2



Cell



MIMD


Mémoire
Processeur
Unité Contrôle
Unité Arithmétique Logique
entrées
sorties
Processeur
Unité Contrôle
Unité Arithmétique Logique
entrées
sorties
Processeur
Unité Contrôle
Unité Arithmétique Logique
entrées
sorties

MIMD

IBM Power5

Alpha Server

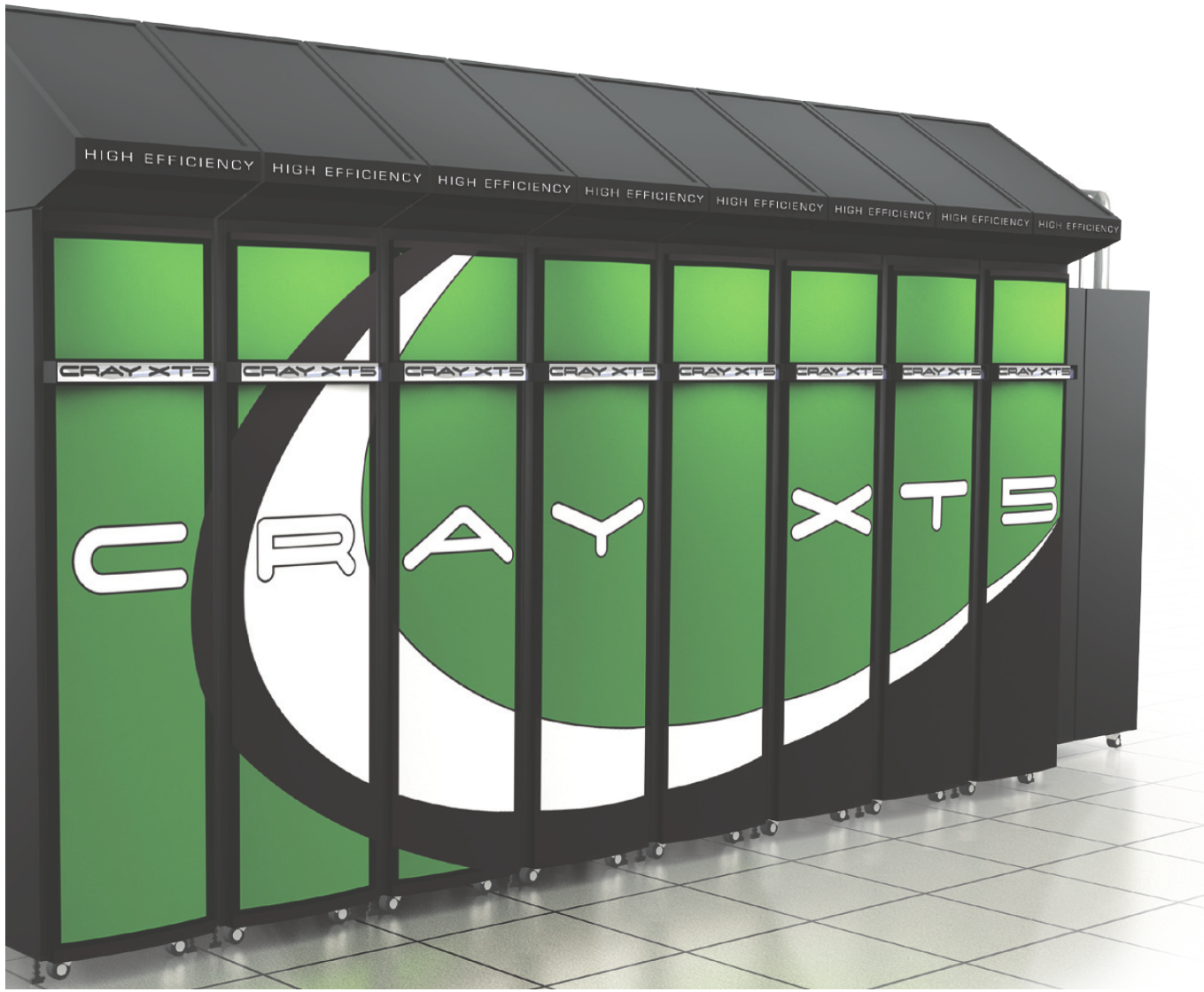
Intel IA32

AMD Opteron

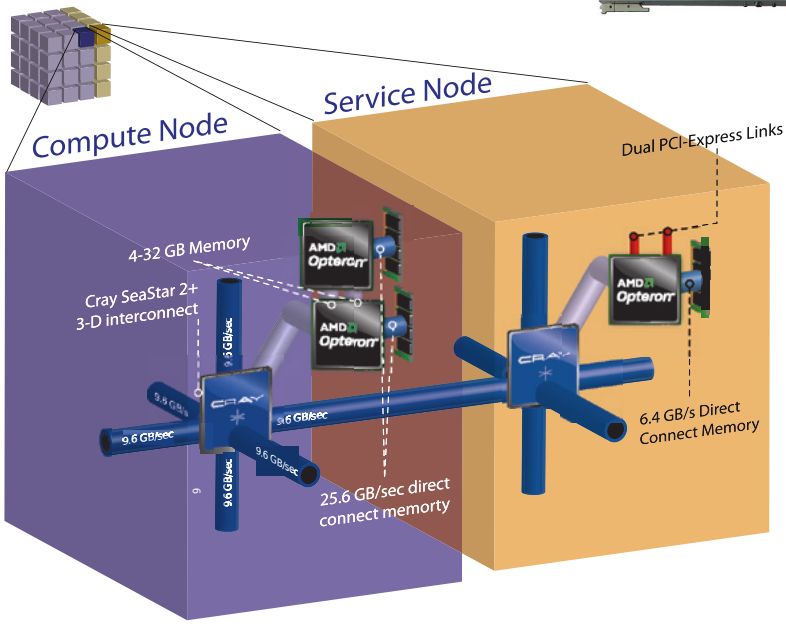
Cray XT3

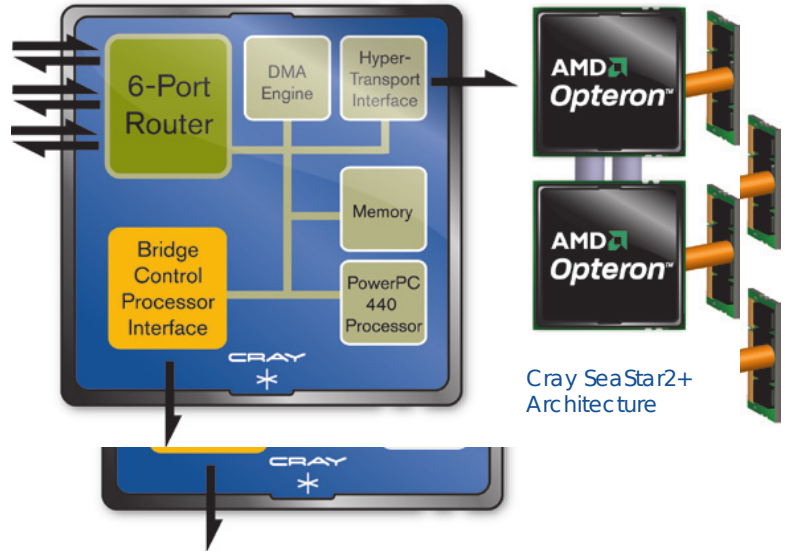
IBM BG/L

CrayXT5



CrayXT5


Compute Node
Service Node
Dual PCI-Express Links
4-32 GB Memory
Cray SeaStar 2+
3-D interconnect
AMD Opteron
9.6 GB/sec
6.4 GB/s Direct
Connect Memory
25.6 GB/sec direct
connect memorty


6-Port
Router
DMA
Engine
Hyper-
Transport
Interface
Memory
Bridge
Control
Processor
Interface
PowerPC
440
Processor
CRAY
AMD
Opteron
Cray SeaStar2+
Architecture

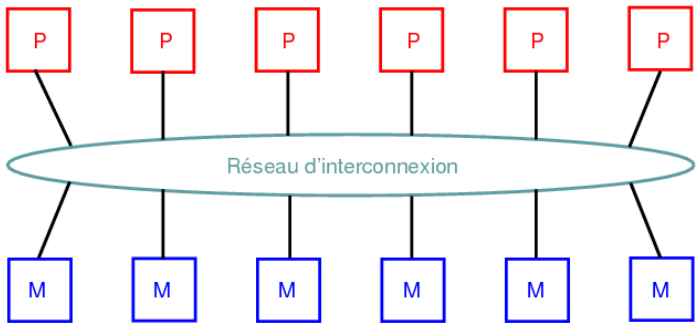
MIMD

CrayXT5

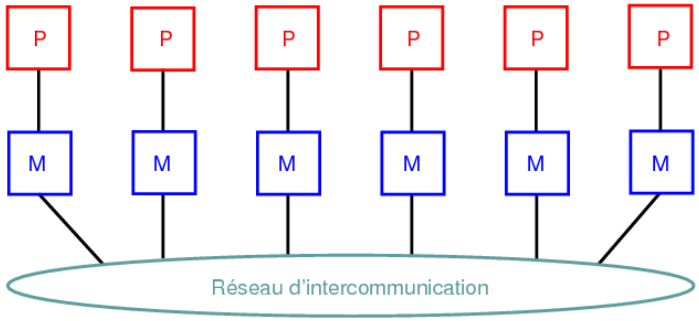
- Quatre et six-coeur 64-bit AMD Opteron jusqu'à 192 par "cabinet"
- 7 à 12 Téraflops par "cabinet"
- 25,6 GB/sec par noeud de calcul
- Interconnexion en tore 3D
- 32 à 42.7 kW (32,7 à 43,6 kVA) par "cabinet"
- National Center for Computational Sciences (Oak Ridge) : 37544 processeurs soit 224 256 coeurs de calcul
  - 1,759e+06 GigaFlops soit 1,7 PétaFlops

Parallélisme - Mémoire

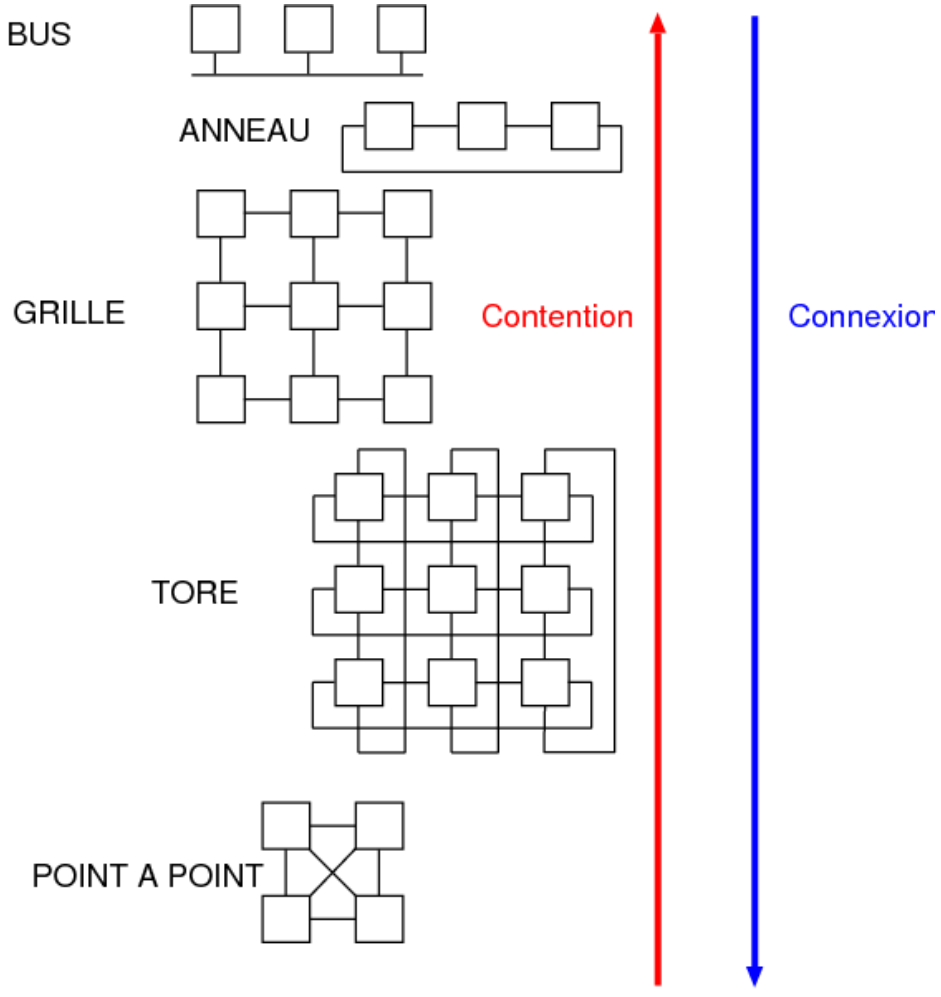
- Mémoire Partagée
  - Réseau d'interconnexion
  -



- Mémoire Distribuée
  - Réseau de Communication
  -



Parallélisme - Réseaux de liaison



Machine à mémoire partagée

Caractéristiques Générales

- tous les processeurs partagent le même espace d'adressage.
- les processeurs sont indépendants mais partagent les mêmes ressources mémoires
- tout changement effectué sur une donnée mémoire par un processeur est visible de tous les autres.
- Il y a deux classe de machines à mémoire partagée : UMA (Uniform Memory Access) et NUMA (Non Uniform Memory Access).

Machine à mémoire partagée

UMA

- Mémoire partagée physiquement et logiquement
- Les représentant les plus courant sont les machines SMP (Symmetric Multiprocessor)
- Processeurs identiques
- Temps et type d'accès identique à la mémoire pour tous les processeurs
- Parfois appelé CC-UMA (Cache Coherent UMA). Cache coherent indique que si un processeur met à jour une donnée dans la mémoire partagées tous les autres processeurs sont aux courant de cette mise à jour, notamment ceux qui possèdent des copies locale de la donnée dans leur mémoire cache. La cohérence est assuré au niveau matériel.

Machine à mémoire partagée

NUMA

- Mémoire distribuée physiquement et partagée logiquement
- Réalisé en connectant deux ou plus SMP (par exemple)
- Un SMP peut directement accéder à la mémoire d'un autre SMP.
- Tous les processeurs n'ont pas le même temps d'accès à toutes les mémoires.
- Les accès à travers le lien entre SMP sont plus lents.
- Si il est maintenu une cohérence de cache, alors on parle aussi de CC-NUMA (Cache Coherent NUMA)

Machine à mémoire partagée

Avantages

- L'adressage global offre un modèle de calcul, basé sur les threads, simple et accessible.
- Le partage de données entre taches est rapide et uniforme du à la proxiité de la mémoire par rapport aux processeurs.

Machine à mémoire partagée

Inconvénients

- Problème de mise à l'échelle lié au goulet d'étranglement que représente la connexion à la mémoire. Ajouter des processeurs augment le traffic vers la mémoire partagée, et pour les CC-UMA et les CC-NUMA vers les mémoires caches des processeurs.
- C'est la responsabilité du programmeur que de s'assurer, à travers des primitives de synchronisation (sémaphore par exemple) de l'accès correct aux données de la mémoire.
- Dispendieux : il devient compliqué et cher de produire des systèmes à mémoire partagées dès lors que l'on augmente le nombre de processeurs.

Machine à mémoire distribuée

Caractéristiques Générales

- Nécessite un réseau de communication inter-processeur
- Les processeurs ont leur propre mémoire locale. Les adresses mémoire d'un processeur ne recouvrent pas celles d'un autre. Il n'y a pas d'espace d'adressage global.
- Puisque chaque processeur a sa propre mémoire locale, il opère indépendamment des autres processeurs. Les changements effectués sur ses données locales n'ont pas d'effet sur les données d'autres processeur. Il n'existe pas de notion de cohérence de cache.

Machine à mémoire distribuée

Caractéristiques Générales

- Quand un processeur a besoin d'accéder à une donnée d'un autre processeur, le programmeur doit explicitement définir quand et comment cette donnée est communiquée. La synchronisation entre tache est de la responsabilité du programmeur.
- Le réseau utilisé pour communiquer peut être aussi simple que l'Ethernet et aussi complexe qu'un Hypercube.

Machine à mémoire distribuée

Avantages

- La mémoire est proportionnelle au nombre de processeurs. Augmenter le nombre de processeur augmente automatiquement la mémoire du même rapport
- Chaque processeur peut rapidement accéder à sa propre mémoire sans interférence et sans maintenir une cohérence de cache
- Le coût de réalisation peut-être bas, il peut être utilisé des composants sur étagère et un réseau existant.

Machine à mémoire distribuée

Inconvénients

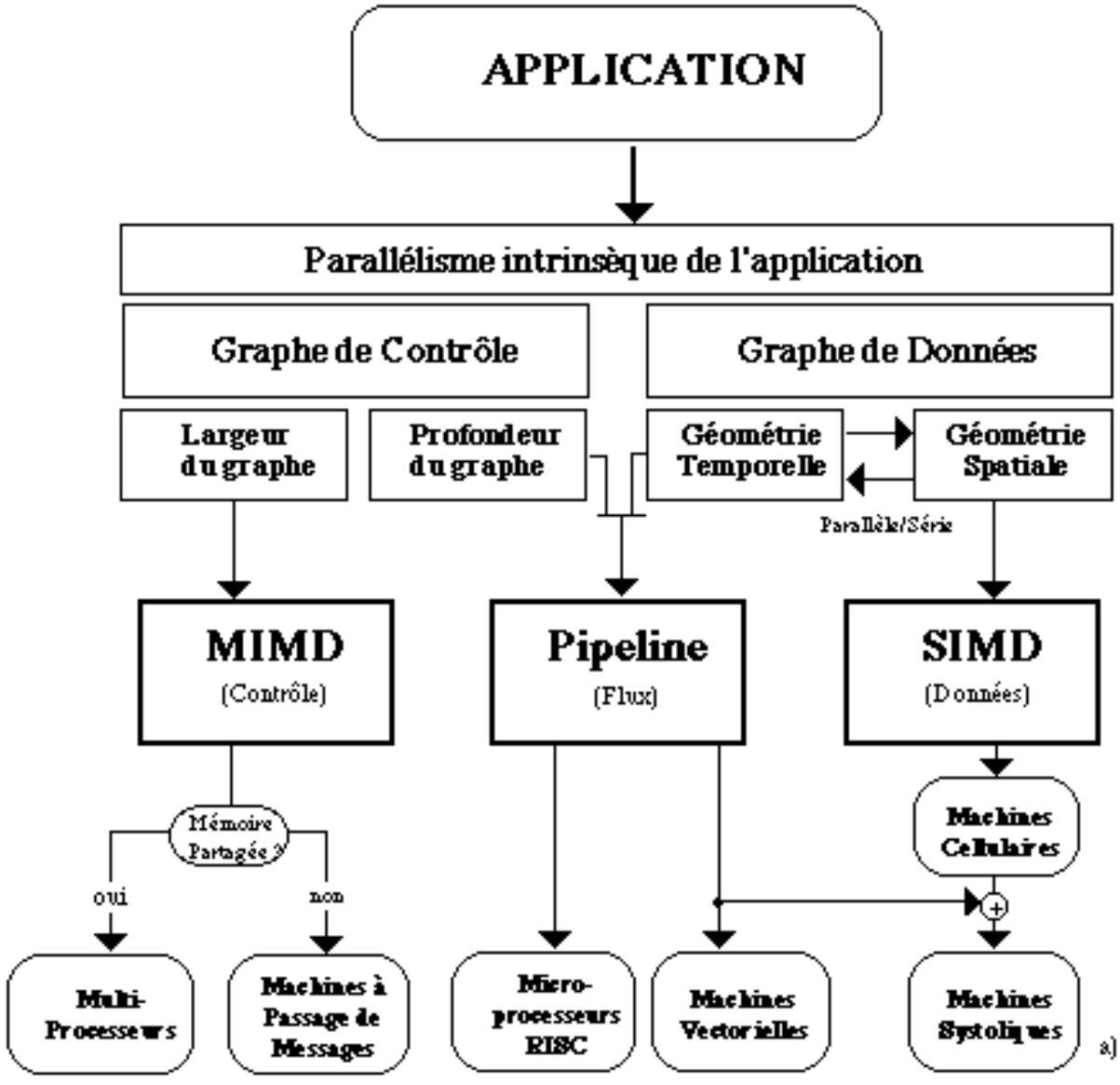
- Le parallélisme est explicite, le programmeur est responsable de tous les détails associés aux communications inter-processeurs.
- Il peut-être difficile de transformer des structures de données définies pour utiliser une mémoire globale.
- Les accès mémoire sont non-uniformes en temps.

Machine hybrides mémoire distribuée/mémoire partagée

L'union fait la force !

- Les superordinateurs dans le monde emploient les deux techniques (CrayXT5)
- Il existe des noeuds de type SMP
- Les noeuds SMP sont distribués. Les processeur d'un SMP ne connaissent que les données mémoire de leur SMP pas celles des autres SMP.
- Des communications sont requises pour communiquer les données d'un SMP à l'autre
- Les projections laissent penser que ce type d'architecture prévaudra encore dans un futur proche
- Le meilleur des deux mondes !

Classification par sources



Modèle de Programmation

Il existe plusieurs modèle de programmation parallèle :

- par mémoire partagée
- Threads
- par passage de message
- langage Data Parallel

Modèle de programmation parallèle

Différents Modèles

- Calcul Concurrents (Threads, MPI, PVM, ...)
  - Exploitation du parallélisme de contrôle
  - Le contrôle est parallèle incompatible avec une version séquentielle de l'application
- Langages parallèles (OpenMP, CUDA, ...)
  - Exploitation du parallélisme de données
  - Parallélisation du calcul sur les données souvent compatible avec une version séquentielle de l'application (vecteur vs scalaire)
  - Possibilité de parallélisation automatique ou quasi-automatique d'un code séquentiel

Spécification du parallélisme

- de manière explicite : demande au programmeur de spécifier les tâches à accomplir en parallèle
- de manière implicite : prétend que cela revient à la machine.

Mémoire Partagée

- Les tâche partagent un espace commun d'adressage
- Les lectures et écritures sont asynchrones
- L'utilisation de mécanismes de verrouillage sont nécessaire pour accèder à la mémoire partagée
- Un des avantages de ce modèle du point de vue programmeur est l'absence de communication de données entre taches, simplifiant ainsi la mise au point des programmes.
- Un des désavantages est la difficulté à comprendre et gérer la localité des données.
- Gérer la localité des données, pour avoir un cache efficace, est dur à analyser
- Provoque des rafraichissement de cache et augmente le traffic du bus

Threads

- Un processus peut avoir de multiples exécution concurrentes.
- Voici une analogie simple pour comprendre le concept :
  - Le programme principal a.out est ordonnancé et acquiert toutes les ressources nécessaires à l'exécution de l'application.
  - a.out réalise des travaux séquentiels, puis crée un certain nombre de tache (les threads) qui peuvent s'exécuter en concurrence.
  - Chaque thread possède ses propres données locales, mais aussi a accès à toutes les ressources du programme principal a.out.
  - Il y a donc un gain du à la non-réplication des ressources du programme principal.
  - Chaque thread bénéficie aussi d'une vue globale sur la mémoire puisqu'il partage l'espace mémoire du programme principal a.out.

Threads

- Un thread peut-être décrit comme une sous-routine du programme principal
  - Peu de changement par rapport à une programmation séquentielle !
- N'importe quel thread peut exécuter n'importe que fonction en même temps que n'importe quel autre thread.
- La communication entre thread s'effectue à travers la mémoire principale.
- Il est nécessaire d'avoir des constructions de synchronisation pour éviter l'accès en écriture simultané à une variable identique.
- Les thread naissent et meurent, mais le programme qui les a engendré perdure fournissant les ressources partagées nécessaires à l'aboutissement de l'application.
- Les threads sont généralement exécuté sur des architectures à mémoire partagée

Support de cours Java

Programmation concurrente et Distribuée

H. Mounier

Université Paris sud

2004/2005

Threads

Références :

- The Java Language Specification, J. Gosling, B. Joy et G. Steele
- Java Threads, S. Oaks et J. Wong
- Concurrent Programming in Java. Design Principles and Patterns, D. Lea

Modèle de threads de Java

- Threads et processus
- États d'une thread
- Priorités
- Synchronisation

Threads et processus

- Multi-tâches de threads : moins de surcharge que multi-tâches de processus
- Processus :
  - Un ensemble de ressources ; par ex. sous UNIX, segment de code, de données utilisateur, de données systèmes (répertoire de travail, descripteurs de fichiers, identificateurs de l'utilisateur ayant lancé le processus, du groupe dont il fait partie, infos. sur l'emplacment des données en mémoire, . . .)

Threads et processus ⋙

- Une unité d'exécution (avec, par ex. sous UNIX, des informations nécessaires à l'ordonnanceur, la valeur des registres, le compteur d'instructions, la pile d'exécution, des informations relatives aux signaux)

Threads et processus ⋙

- Thread : on ne retient que l'aspect d'unité d'exécution.
- Contrairement aux processus, les threads sont légères :
  - elles partagent le même espace d'adressage,
  - elles existent au sein du même processus (lourd),
  - la communication inter-thread occasionne peu de surcharge,
  - le passage contextuel (context switching) d'une thread à une autre est peu coûteux.

Threads et processus ⋙

- Le multi-tâches de processus n'est pas sous le contrôle de l'environnement d'exécution java.
- Par contre, il y a un mécanisme interne de gestion multi-threads.
- Le peu de surcharge occasionné par le système de multi-threads de Java est spécialement intéressant pour les applications distribuées.
- Par ex., un serveur multi-tâches (avec une tâche par client à servir)

Etats d'une thread

- Dans un environnement mono-thread, lorsqu'une thread se bloque (voit son exécution suspendue) en attente d'une ressource, le programme tout entier s'arrête.
  Ceci n'est plus le cas avec un environnement multi-threads
- Une thread peut être :
  - en train de s'exécuter (running) ;
  - prête à s'exécuter (ready to run), dès que la CPU est disponible ;
  - suspendue (suspended), ce qui stoppe temporairement son activité ;

Etats d'une thread ⋙

- poursuivre l'exécution (resumed), là où elle a été suspendue ;
- bloquée (blocked) en attendant une ressource.

Une thread peut se terminer à tout moment, ce qui arrête son exécution immédiatement. Une fois terminée, une thread ne peut être remise en route

Priorités d'une thread

- Une thread peut
  - soit volontairement laisser le contrôle.
    Ceci est réalisé en passant le contrôle explicitement, dormant ou bloquant en E/S. La thread prête à s'exécuter et de plus haute priorité est alors lancée.
  - soit être supplantée par une autre de plus haute priorité.
    Si c'est systématiquement le cas, on parle de multi-tâches préemptif.

Priorités d'une thread ⋙

- Dans le cas où 2 threads de même priorité veulent s'exécuter, le résultat est dépendant du système d'exploitation (de son algorithme d'ordonnancement).
- Sous Windows 95, le même quantum de temps est tour à tour alloué aux threads de même priorité.
- Sous Solaris 2.x, des threads de même priorité doivent explicitement passer le contrôle à leurs pairs ...

Synchronisation

- Synchronisation inter-threads *via* des moniteurs. Notion définie par C.A.R Hoare.
- Moniteur : mini boîte ne pouvant contenir qu'une thread. Une fois qu'une thread y est entrée, les autres doivent attendre qu'elle en sorte.
- Pas de classe "`Monitor`". Chaque objet a son propre moniteur implicite dans lequel on entre automatiquement lorsqu'une méthode synchronisée de (l'instance de) l'objet est appelée.

Synchronisation ⋙

- Lorsqu'une thread est dans une méthode synchronisée, aucune autre thread ne peut appeler de méthode synchronisée de la même instance de l'objet.
- Permet un code clair et compact, la synchronisation étant construite dans le langage.
- Il est également possible pour 2 threads de communiquer par un mécanisme simple et souple.

Création et utilitaires

- Création d'une thread
- Méthodes utilitaires

Classe Thread et interface Runnable

- Système de multi-threads : autour de la classe `Thread` et de l'interface `Runnable`.
- Liste des méthodes les plus courantes :

`methode()`	But
`getName()`	Obtenir le nom d'une thread
`getPriority()`	Obtenir la priorité d'une thread
`isAlive()`	Déterminer si une thread est toujours en cours d'exécution
`join()`	Attendre la terminaison d'une thread
`resume()`	Poursuivre l'exécution d'une thread suspendue

Classe `Thread` et interface `Runnable` ⋙

`run()`	Point d'entrée (d'exécution) de la thread
`sleep()`	Suspendre la thread pour une période de temps donnée
`start()`	Débuter une thread en appelant sa méthode `run()`
`suspend()`	Suspendre une thread

Classe `Thread` et interface `Runnable` ⋙

Création d'une thread

- Deux possibilités :
  - Implanter l'interface `Runnable`,
  - Créer une sous-classe de `Thread`
- Pour implanter `Runnable`, il y a seulement besoin d'implanter la méthode `run()` :

```
public abstract void run()
```

- Dans `run()`, on place les instructions de la thread à lancer.

Création d'une thread : implanter `Runnable`

- Ayant créé une classe implantant `Runnable`, on créé une instance de `Thread` dans cette classe :

  ```
  Thread(Runnable objThread, String nomThread)
  ```

- La thread ne débute pas tant qu'on n'appelle pas `start()`.
- En fait `start()` effectue un appel à `run()`. C'est un cadre logiciel.

  ```
  synchronized void start()
  ```

Création d'une thread : exemple avec Runnable

```
class NouvelleActivite implements Runnable {
    Thread th;

    NouvelleActivite() {
        th = new Thread(this, "Activite demo");
        System.out.println("Activite enfant : " + th);
        th.start();                    // Debuter l'activite
    }

    // Point d'entree de la nouvelle activite
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
```

Création d'une thread : exemple avec Runnable

```
                System.out.println("Activite enfant : " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Enfant interrompu.");
        }
        System.out.println("Sortie de l'activite enfant");
    }// run()
}// class NouvelleActivite

class ActiviteDemo {
    public static void main(String args[]) {
```

Création d'une thread : exemple avec Runnable

```
new NouvelleActivite();

try {
    for(int i = 5; i > 0; i--) {
        System.out.println("Activite parente : " + i);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Parent interrompu.);
}
System.out.println("Sortie de l'activite parent");
```

Création d'une thread : exemple avec `Runnable`

```
    }// run()
}// class Activite
```

Création d'une thread : hériter de Thread

- Deuxième possibilité : créer une sous-classe de `Thread`, qui redéfinit la méthode `run()`. Même exemple :

```
class NouvelleActivite extends Thread {

    NouvelleActivite() {
        super("Activite demo");
        System.out.println("Activite enfant : " + this);
        start();                    // Debuter l'activite
    }

    // Point d'entree de la nouvelle activite
    public void run() {
```

Création d'une thread : hériter de Thread ⋙

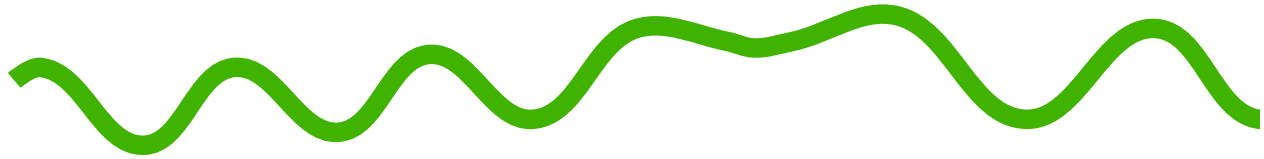
```
            try {
                for(int i = 5; i > 0; i--) {
                    System.out.println("Activite enfant : " + i);
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {
                System.out.println("Enfant interrompu.);
            }
            System.out.println("Sortie de l'activite enfant");
        }// run()
    }// class NouvelleActivite

    class ActiviteDemo {
        public static void main(String args[]) {
```

Création d'une thread : hériter de Thread ⋙

```
            new NouvelleActivite();

            try {
                for(int i = 5; i > 0; i--) {
                    System.out.println("Activite parente : " + i);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                System.out.println("Parent interrompu.);
            }
            System.out.println("Sortie de l'activite parent");
        }// run()
    }// class Activite
```

Passage de Messages

- L'application est composée d'un certain nombre de taches qui possède leurs propres données durant l'exécution du programme.
- Les taches peuvent être soit sur la même machine physique, soit à travers un réseau sur un nombre artbitraire de machine
- Les données et les communications s'effectue par des message de réception et d'envoi (Recv et Send)
- les fonctions de transfert sont pairées : à un Recv doit correspondre un Send.



Introduction à MPI – Message Passing Interface

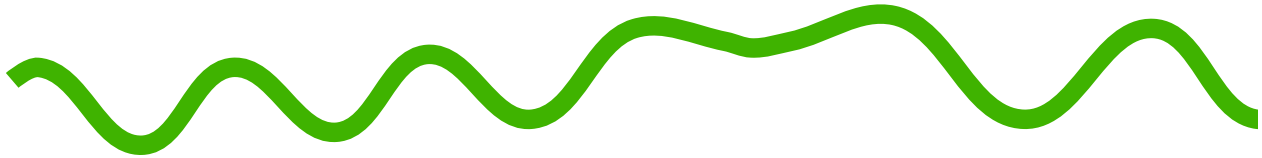
Outils pour le calcul scientifique à haute performance
École doctorale sciences pour l'ingénieur
mai 2001

Philippe MARQUET

phm@lifl.fr

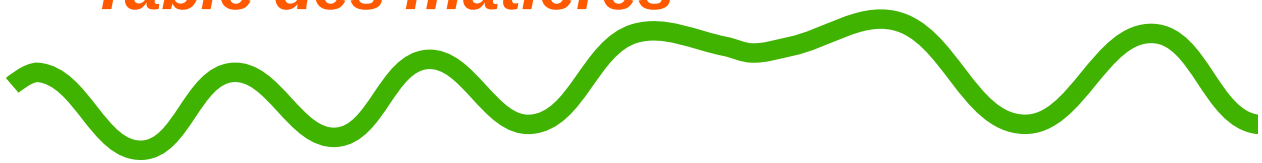
Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille

MPI

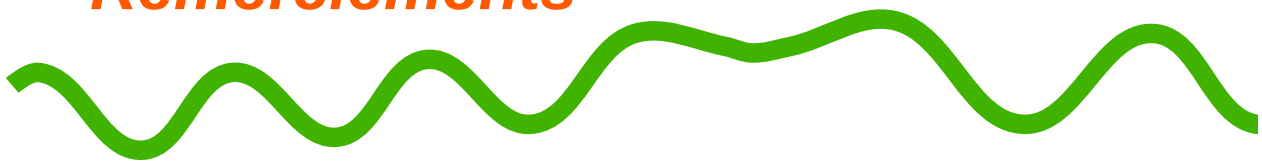


- Ce cours est diffusé sous la licence GNU Free Documentation License, `http://www.gnu.org/copyleft/fdl.html`
- La dernière version de ce cours est accessible à partir de `http://www.lifl.fr/west/courses/cshp/`
- `$Id: mpi.tex,v 1.11 2002/04/29 07:32:58 marquet Exp $`

Table des matières



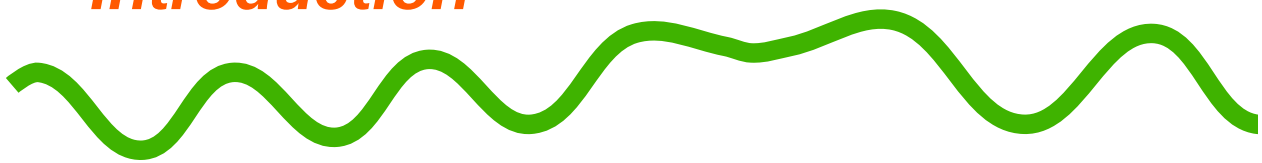
Remerciements



Cette présentation de MPI est essentiellement basée sur

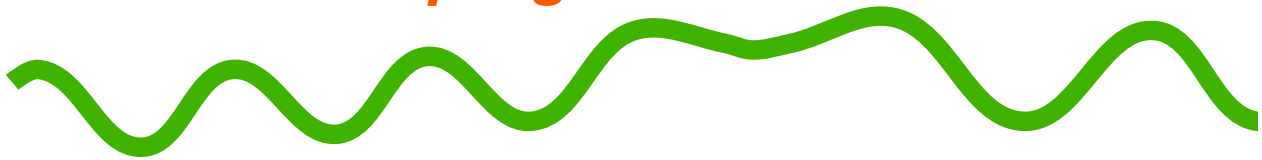
- *A User's Guide to MPI*
  Peter S. PACHERO
  University of San Francisco
  `ftp://math.usfca.edu/pub/MPI/`
- *Designing & Building Parallel Programs*
  Chapitre *Message Passing Interface*
  Ian FORSTER
  Argonne National Laboratory
  `http://www.mcs.anl.gov/dbpp`
  `http://www.mcs.anl.gov/dbpp/web-tours/mpi.html`

Introduction

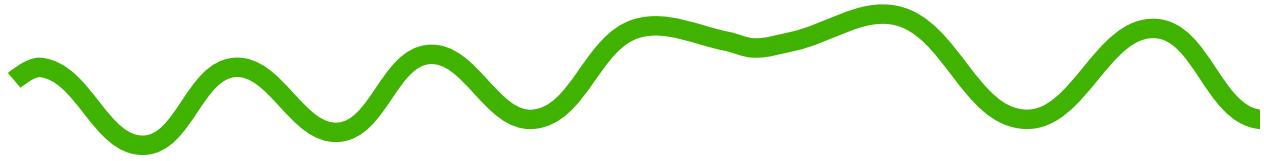


- MPI : *Message-Passing Interface*
  - Bibliothèque de fonctions utilisables depuis C, Fortran, C++
  - Exploitation des machines multi-processeurs par passage de messages
  - Conçue en 1993–94 $\longrightarrow$ standard
- Cette présentation :
  - Grandes lignes de MPI
  - Exemples simples
  - Utilisation de MPI depuis C

Modèle de programmation

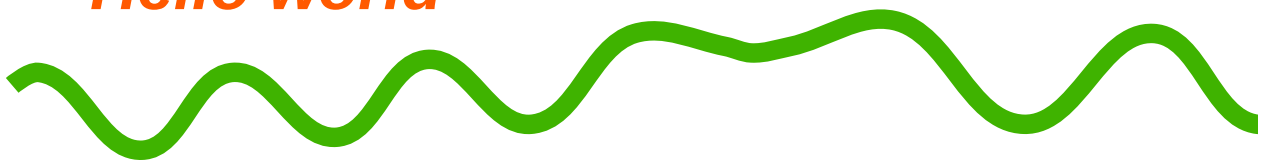


- Modèle de programmation de MPI
  - parallélisme de tâches
  - communication par passage de messages
- Même programme exécuté au lancement de l'application par un ensemble de processus
  - SPMD *Single Program Multiple Data*
  - M-SPMD *Multiple-SPMD*
- Un seul programme = un seul code source, celui d'un processus

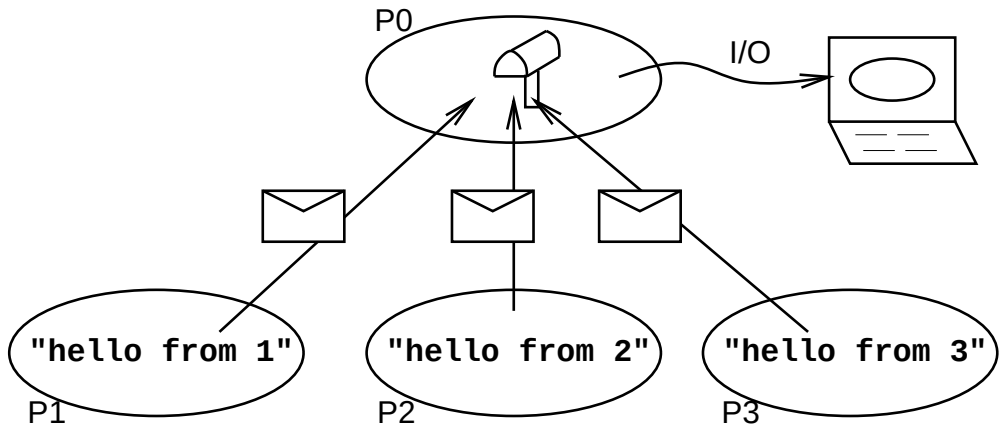


Hello world

Hello world

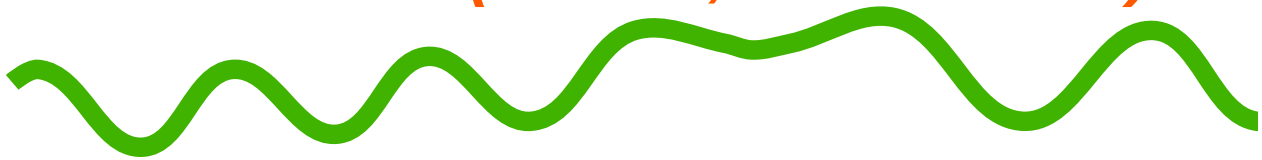


- p processus : P_0 à P_{p-1}
- Les processus $P_{i,i>0}$ envoient un message (chaîne de caractères) à P_0
- P_0 recoit $p-1$ messages et les affiche



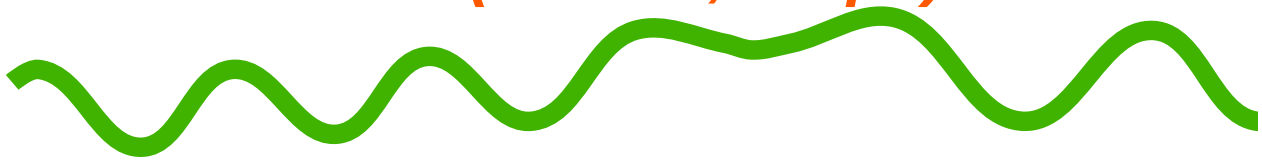
- Programmation SPMD (*Single Program Multiple Data*) : Suis-je le P_0 ?

Hello World (code C, initialisation)



```
#include <stdio.h>
#include "mpi.h"

main(int argc, char *argv[]) {
    int my_rank;        /* Rang du processus */
    int p;              /* Nombre de processus */
    int source;         /* Rang de l'emetteur */
    int dest;           /* Rang du recepteur */
    int tag = 50;       /* Tag des messages */
    char message[100];  /* Allocation du message */
    MPI_Status status;  /* Valeur de retour pour le recepteur

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

Hello World (code C, corps)



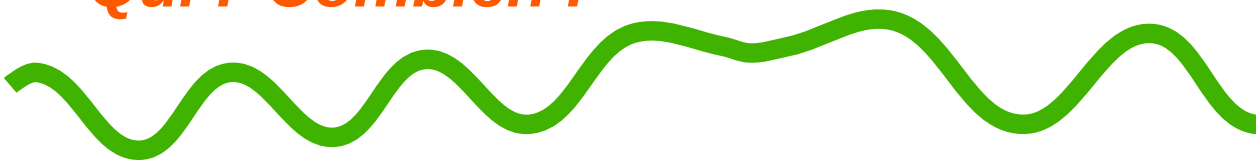
```
    if (my_rank != 0) {
        /* Creation du message */
        sprintf(message, "Hello from process %d!", my_rank);
        dest = 0;
        /* strlen + 1 => le '\0' final est envoye */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
            tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize();
} /* main */
```

Structure d'un programme MPI

- Utilisation de la bibliothèque MPI :
  - Enrollement dans MPI
  - Quitter MPI proprement

```
...
#include "mpi.h"
...
main (int argc, char *argv []) {
   ...
   MPI_Init (&argc, &argv) ;
   ...
   MPI_Finalize () ;
   ...
}
```

Qui ? Combien ?



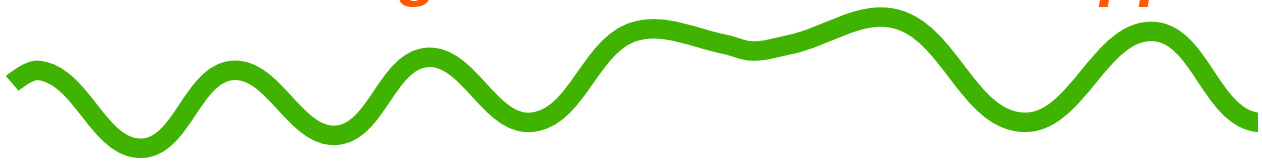
- Qui suis-je ?

```
int MPI_Comm_rank (MPI_Comm comm, int *rank) ;
```

  - Communicateur : collection de processus pouvant communiquer
  - Communicateur `MPI_COMM_WORLD` predéfini : tous les processus
- Combien sommes-nous ?

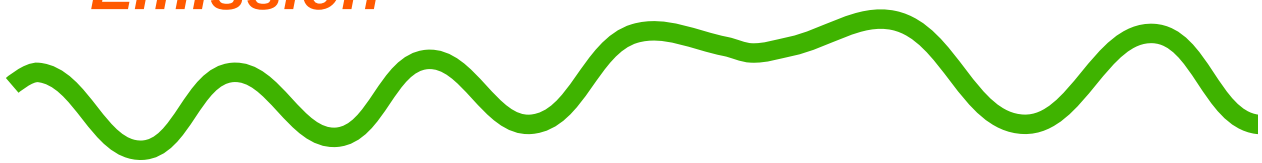
```
int MPI_Comm_size (MPI_Comm comm, int *size) ;
```

Un message = données + enveloppe



- Fonctions de base d'émission (`MPI_Send ()`) et de réception (`MPI_Recv ()`) de messages
- Enveloppe : informations nécessaires
  - Rang du receveur (pour une émission)
  - Rang de l'émetteur (pour une réception)
  - Un tag (`int`)
    $\Rightarrow$ Distinguer les messages d'un même couple émetteur/receveur
  - Un communicateur (`MPI_Comm`)
    $\Rightarrow$ Distinguer les couples de processus

Émission



- Émission d'un message
  (standard, bloquant ou non selon l'implantation)

```
int MPI_Send (void *message, int count,
              MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm) ;
```

- Message =
  - enveloppe
  - données :
    - bloc de mémoire d'adresse `message`
    - de `count` valeurs
    - de type `datatype`
- Correspondance des `MPI_Datatype` avec les types du langage (C ou Fortran)

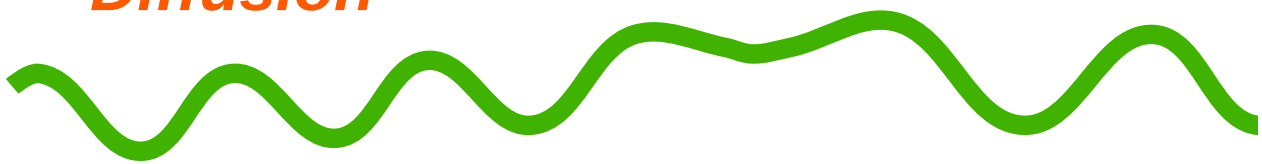
Type MPI_Datatype

- Correspondance des `MPI_Datatype` avec les types du C :

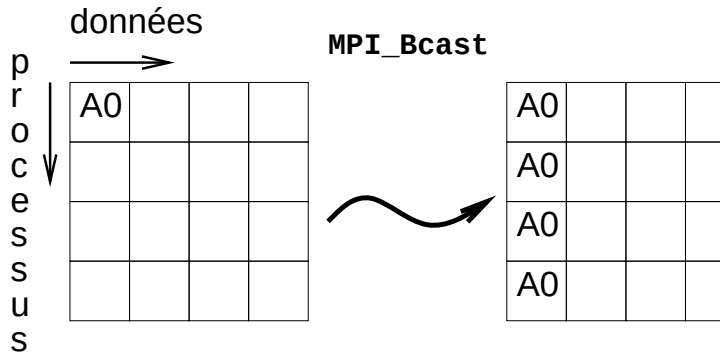
`MPI_CHAR`	`signed char`	`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`	`MPI_LONG`	`signed long int`
…			
`MPI_FLOAT`	`float`	`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`	`MPI_PACKED`	`<< struct >>`

- Types particuliers :
  - `MPI_BYTE` Pas de conversion
  - `MPI_PACKED` Types construits (*cf. infra*)

Diffusion



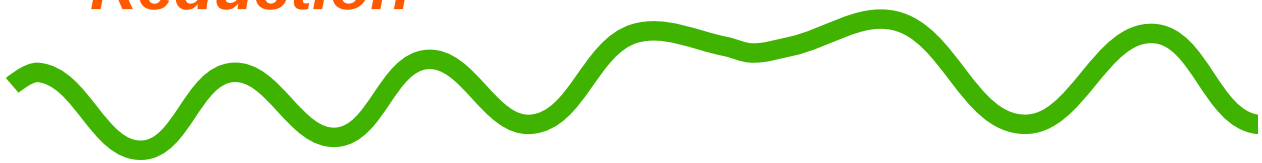
- Un même processus envoie une même valeur à tous les autres processus (d'un communicateur)




```
int MPI_Bcast (void *message, int count, MPI_Datatype datatyp
               int root, MPI_Comm comm) ;
```

- Appel par tous les processus du communicateur `comm` avec la même valeur de `root`, `count`, `datatype`
- La valeur détenue par le processus `root` sera émise et rangée chez chacun des autres processus
- ⇝ Réécriture de `Get_data ()`

Réduction



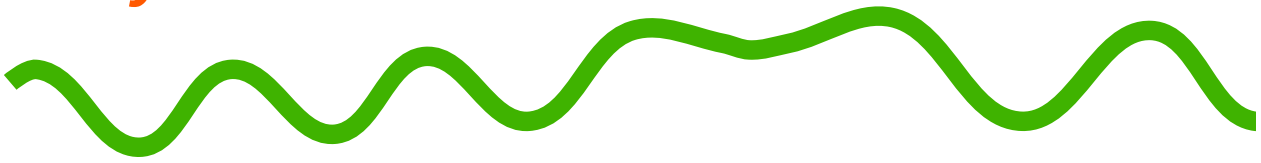
- Collecte par un processus d'un ensemble de valeurs détenues par tous les processus
- Réduction de cette valeur
- Fonction SPMD

```
int MPI_Reduce (void *operand, void *result, int count,
                MPI_Datatype datatype, MPI_Op op,
                int root, MPI_Comm comm) ;
```

- Appel par tous les processus du communicateur `comm` avec une même valeur de `count`, `datatype`, `op`
- Opérations binaires prédéfinies par MPI (`MPI_MAX`, `MPI_SUM`...)
- (Possibilité de définir de nouvelles opérations)
- Le processus `root` détient le résulat
- ⇝ Réécriture de la collecte globale du résulat

Autres communications collectives

Synchronisation



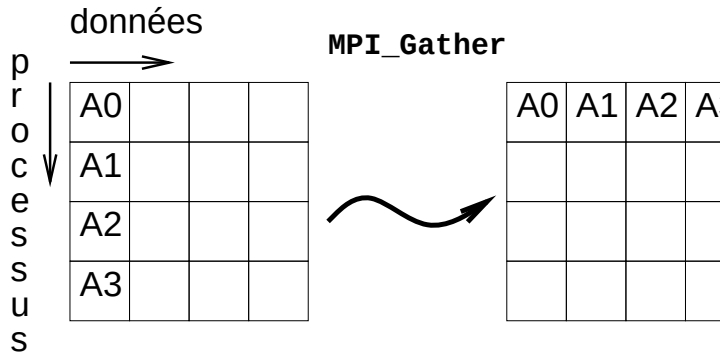
- Synchronisation ou rendez-vous
- Pas d'échange d'informations
- Tous les processus sont assurés que tous ont ralliés le *point de synchronisation*

```
int MPI_Barrier (MPI_Comm comm) ;
```

Autres communications collectives

Rassemblement

- « All to one »
- Mise bout à bout des messages de chacun des processus

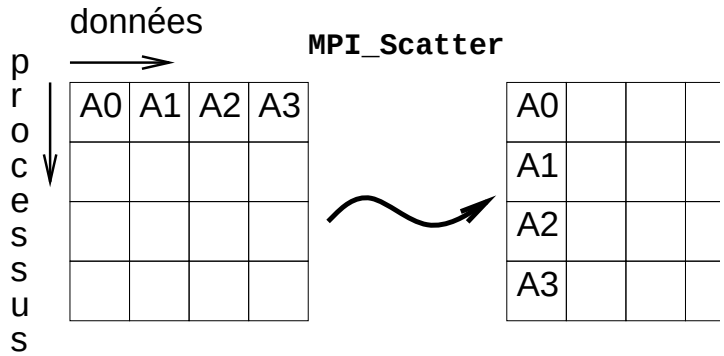



```
int MPI_Gather (void *send_buf, int send_count,
                MPI_Datatype send_type,
                void *recv_buf, int recv_count,
                MPI_Datatype recv_type,
                int root, MPI_Comm comm) ;
```

Autres communications collectives

Distribution personnalisée

- « One to all »
- Distribution d'un message personnalisé aux autres processus




```
int MPI_Scatter (void *send_buf, int send_count,
                 MPI_Datatype send_type,
                 void *recv_buf, int recv_count,
                 MPI_Datatype recv_type,
                 int root, MPI_Comm comm) ;
```

Langage Data Parallel

- Le parallélisme de données est le plus important
- Il s'agit de répéter la même opération sur un jeu de données organisées dans une structure commune tel qu'une matrice ou un vecteur.
- Un ensemble de taches travaillement collectivement sur la même structure de donnée
- Toutefois, chaque tache travaille sur sa propre partition de données
- Les taches réalise la même opération sur leur partition
- Sur des architectures à mémoire partagée toutes les tâches ont accès à la structure globalisée
- Sur des architectures à mémoire distribuée il est nécessaire de répartir la structure dans les différentes mémoires.

Langage Data Parallel

NESL

```
function QUICKSORT(S) =
if (#S <= 1) then S
else
  let a   = S[rand(#S)];
      S_1 = {e in S | e < a};
      S_2 = {e in S | e == a};
      S_3 = {e in S | e > a};
      R   =  {QUICKSORT(v): v in [S_1, S_3]};
  in R[0] ++ S_2 ++ R[1];
```

- *e in S*$|e < a$ peut se lire ”en parallèle pour chaque e dans S sélectionner tous les e qui sont plus petits que a”
- de même *QUICKSORT*(v) : *v in* $[S_1, S_3]$ peut se lire ”en parallèle pour chaque v dans la séquence [S_1, S_3] faire QUICKSORT(v).”

Automatisation et Parallélisation manuelle

- Concevoir et développer des programmes parallèle ont souvent été des processus manuels
- Le programmeur est responsable de l'exhibition et de l'extraction du parallélisme
- Le développement de code parallèle peut-être chronophage et complexe.
- Depuis plusieurs années des outils de conversion de programme séquentiel en programme parallèle existe
- Les plus connus sont les compilateur vectorisant et le pré-processeurs

Automatisation et Parallélisation manuelle

- Un compilateur vectorisant fonctionne suivant plusieurs modes :
  - Totalement automatique (Utopique ?)
    - Le compilateur analyse le code et identifie les sources de parallélismes
    - L'analyse doit aussi mesurer les inhibiteur de parallélisme et vérifier que le code vectorisé augmentera les performances
    - Les boucles sont les parties du codes les plus souvent vectorisées
  - Orienté par le programmeur Programmer Directed
    - en utilisant des directives de compilation ou des drapeaux, le programmeur indique explicitement où se situe le parallélisme exploitable
    - peut-être ajouté à une parallélisation plus automatique pour de meilleures performances

Automatisation et Parallélisation manuelle

- intéressant pour les néophytes du parallélisme
- Mais des résultats incorrects peuvent être produits.
- Il y a moins de flexibilité qu'une parallélisation manuelle.
- Limité à un sous-ensemble du code.

Programme parallélisable ?

refléxion

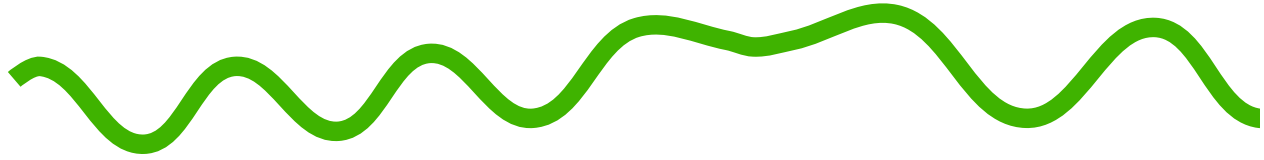
Avant de développer une solution parallèle pour la résolution d'un problème, il est nécessaire de déterminer si ce problème peut ou non être parallélisé

énergie minimale

Calculer les conformations de milliers de molécules et trouver l'énergie minimale ?
La conformation de chaque molécule peut être calculée indépendamment des autres, la recherche du minimum peut aussi être parallélisé.

Fibonacci

$F(k+2) = F(k+1) + F(k)$
Impossible à paralléliser



Introduction à OpenMP

Outils pour le calcul scientifique à haute performance
École doctorale sciences pour l'ingénieur
mai 2001

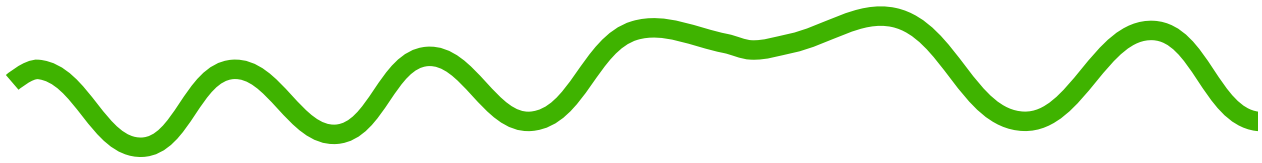
Pierre BOULET

Pierre.Boulet@lifl.fr

Laboratoire d'informatique fondamentale de Lille

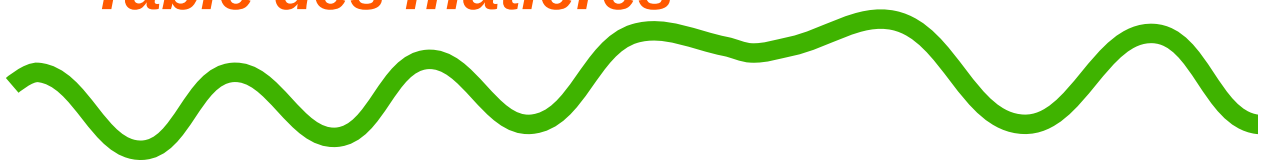
Université des sciences et technologies de Lille

OpenMP

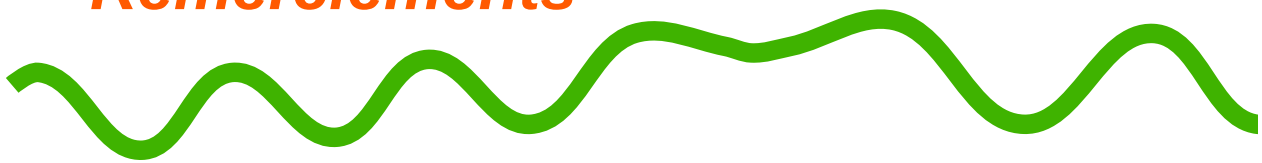


- Ce cours est diffusé sous la licence GNU Free Documentation License, `http://www.gnu.org/copyleft/fdl.html`
- La dernière version de ce cours est accessible à partir de `http://www.lifl.fr/west/courses/cshp/`
- `$Id: openmp.tex,v 1.6 2002/03/18 07:18:21 marquet Exp $`

Table des matières



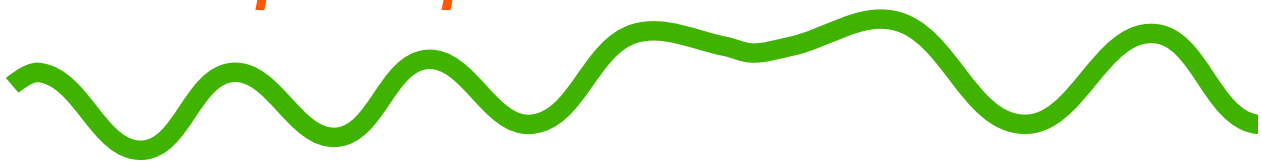
Remerciements



Cette présentation d'OpenMP est essentiellement basée sur

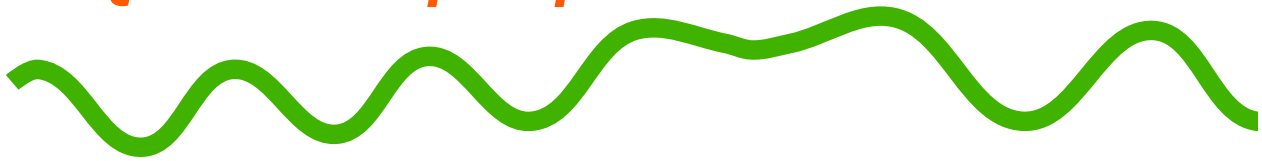
- *Tutoriel OpenMP à Supercomputing'99*
  OpenMP Architecture Review Board
  Tim Mattson, Rudolf Eigenmann
  `http://www.openmp.org/presentations/index.cgi?sc99_tutoria`
- *OpenMP Workshop*
  Isabel Loebich
  `http://www.hlrs.de/news-events/events/1999/openmp/openmp/`
- Cours OpenMP
  IDRIS
  `http://www.idris.fr/data/cours/parallel/openmp/choix_doc.html`

Pourquoi OpenMP ?

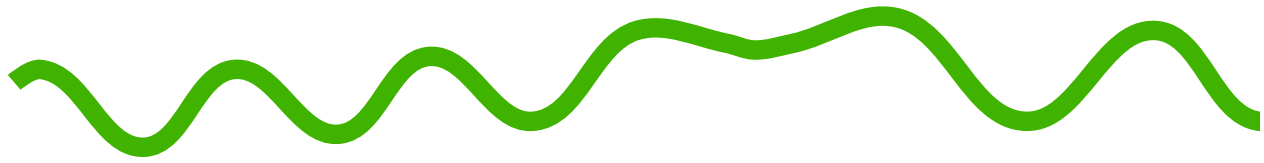


- les difficultés perçues de la programmation parallèle dépassent les avantages
  - diminuer les difficultés perçues
  - langage de haut niveau
- standardiser les pratiques de programmation à mémoire partagée
  - 15 ans de développement non concerté
  - recherche de la portabilité
- standard industriel contrôlé par l'OpenMP Architecture Review Board
  - organismes de recherche
  - constructeurs de supercalculateurs
  - fournisseurs de logiciels
  - sociétés de services

Qu'est-ce qu'OpenMP ?



- API de programmation pour les applications multithreadées
  - directives de compilation
    - introduites par `!$OMP` en Fortran
    - pragmas en C : `#pragma omp`
  - bibliothèque de fonctions
  - variables d'environnement
- langages supportés :
  - Fortran
    - version 1.0 : octobre 1997
    - version 1.1 : novembre 1999
    - version 2.0 : novembre 2001
  - C/C++
    - version 1.0 : octobre 1998

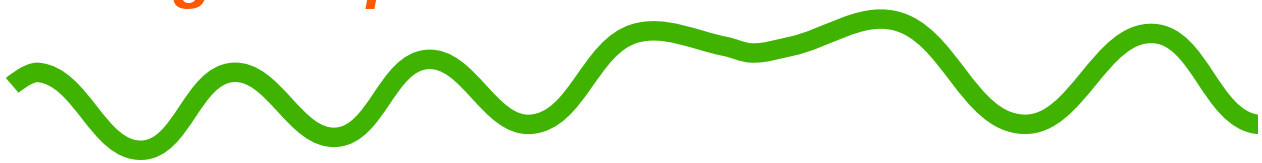


Modèle d'exécution

Modèle d'exécution

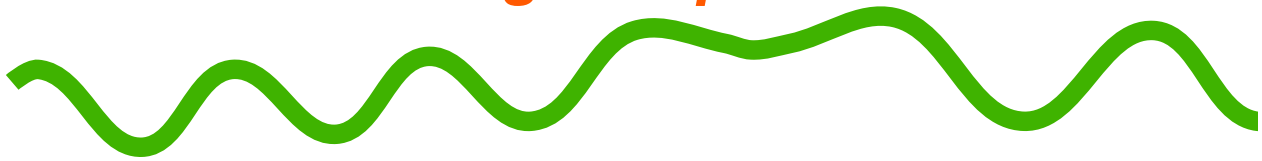
- langage de haut niveau
  - communications implicites
- modèle fork/join
  - un processus (partage de ressources)
  - contenant plusieurs processus légers (exécutions concurrentes)
  - succession de régions séquentielles et de régions parallèles
  - une tâche maître crée les autres
- parallélisme de contrôle
  - répartition des tâches
  - données partagées

Régions parallèles



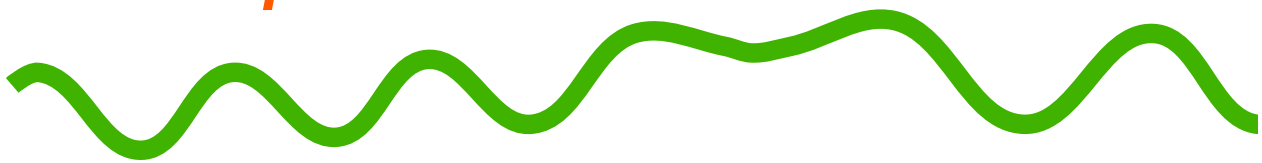
- directive `omp parallel`
  - création de processus légers concurrents
  - aucune garantie du nombre de processeurs
  - réutilisation des processus légers dans plusieurs régions parallèles successives
  - exécution redondante du code sauf utilisation de directives de partag de travail
- terminaison par `omp end parallel`
  - synchronisation en fin de région parallèle
  - le maître continue l'exécution séquentielle
- combien de processus légers ?
  - déterminé par l'environnement d'exécution
  - numérotation à partir de 0 (maître)
  - on peut récupérer le numéro par `omp_get_thread_num()`

Portée des régions parallèles



- portée lexicale
  - le texte entre `omp parallel` et `omp end parallel`
  - doit être un bloc structuré (points d'entrée et de sortie uniques)
- portée dynamique
  - portée lexicale + toutes les fonctions et procédures appelées au sein de cette portée lexicale
- directives orphelines
  - partage du travail en dehors de la portée statique d'une région parallèle
  - exécution séquentielle ou parallèle en fonction du contexte d'exécution
  - permet une parallélisation incrémentale
- imbrication de régions parallèles
  - permise mais
  - le compilateur peut séquentialiser la région interne

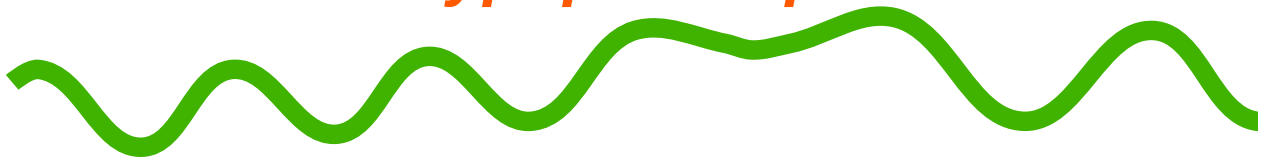
Exemples



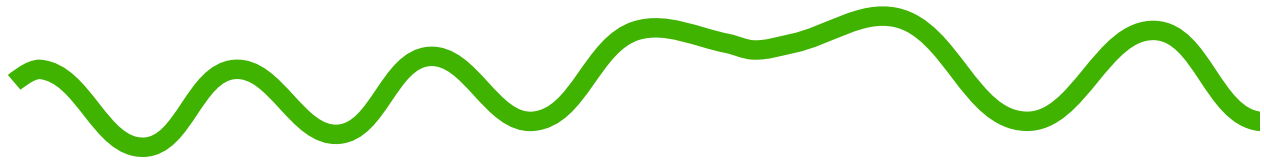
```
double A[1000];
#pragma omp parallel
{
  int id;
  id==omp_get_thread_num();
  foo(id,A);
}
printf("Fin.\n");
```

```
!$OMP PARALLEL
ID=OMP_GET_THREAD_NUM()
PRINT *, 'Hello', ID
PRINT *, 'world', ID
!$OMP END PARALLEL
```

Utilisation typique d'OpenMP

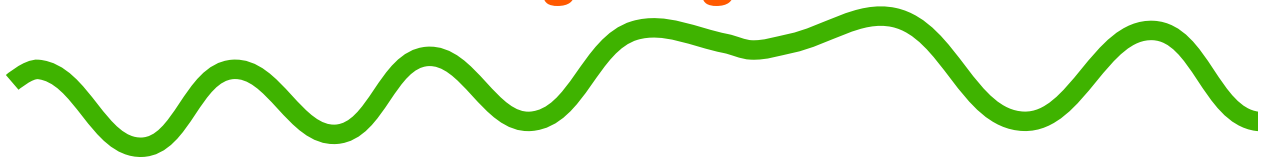


- utilisation classique : parallélisation de boucles
  - trouver les boucles les plus coûteuses du programme
  - répartir les itérations entre des processus légers
- comment ces processus légers interagissent ils ?
  - mémoire partagée
- partage non intentionnel de données peut rendre l'exécution non déterministe
  - le résultat du programme dépend de l'ordonnancement des processus légers
- comment éviter ce non déterminisme ?
  - utiliser des synchronisations pour éviter les conflits de données
- synchronisations coûtent cher, donc
  - changer le stockage des données pour minimiser les besoins de synchronisation



Partage du travail

Parallélisme à gros grain

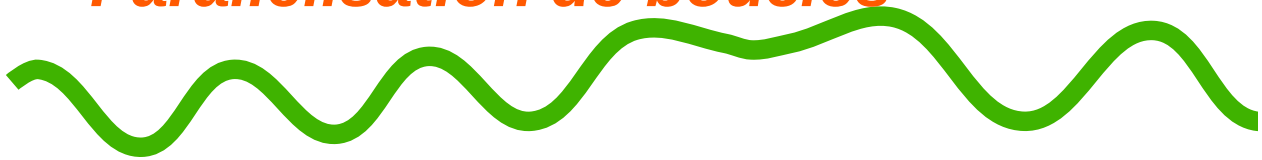


```
#pragma omp sections
{
  calcul_X();
#pragma omp section
  calcul_Y();
#pragma omp section
  calcul_Z();
}
```

```
!$OMP SECTIONS
!$OMP SECTION
CALL calcul_X()
!$OMP SECTION
CALL calcul_Y()
!$OMP SECTION
CALL calcul_Z()
!$OMP END SECTIONS
```

- chaque processus léger exécute un bloc structuré différent
- non extensible
- peut être combinée avec `omp parallel` : `omp parallel sections`

Parallélisation de boucles



```
#pragma omp for
for (i=0; i<n; i++){
  foo(i);
}
```

```
!$OMP DO
do i=0, n
  call foo(i)
end do
!$OMP END DO
```

- chaque processus léger n'exécute qu'une partie des itérations
- seule la boucle suivant immédiatement la directive est parallèle
- peut être combinée avec `omp parallel` : `omp parallel do`

Attributs des données

- données partagées : clause `shared`
  - visibles par tous les processus légers
  - par défaut les variables globales statiques
  - s'applique aux directives de régions parallèles
- données privées : clause `private`
  - répliquées sur chaque processus léger
  - indéfinies à l'entrée de la région parallèle
  - valeurs non transmises à la région séquentielle qui suit
  - par défaut les variables locales et les variables globales dynamiques
  - s'applique aux directives de régions parallèles et de partage du trava

Réductions

- clause `reduction(op : liste)`
  - les variables dans la `liste` doivent être partagées dans la région englobante
  - à l'intérieur de la région :
    - initialisation par l'élément neutre
    - traitement sur la copie locale
    - réduction à la sortie
-
```
#pragma omp parallel for reduction(+:r) private(t)
for (i=0; i<1000; i++){
  t=foo(i);
  r=r+t
}
```

Parallélisme - Gain et Efficacité

- Le gain exprime l'accélération il est égal à

$$G = \frac{Temps_{séquentiel}}{Temps_{parallèle}}$$

- L'efficacité exprime l'utilisation effective des ressources disponibles elle est égale à

$$E = \frac{G}{Nombre_{ressources}}$$

Parallélisme - Loi d'Amdhal

- Question : Si j'ai 100 processeurs vais-je 100 fois plus vite qu'avec 1 seul ?
- Réponse : Non, il existe toujours une partie séquentielle dans le programme qu'on ne peut paralléliser.
- Exemple
  - Un programme de 20 instructions de durée 1 cycle chaque
  - 30% d'instructions séquentielles
  - durée du programme avec 1 processeur : 20 cycles
  - durée idéale avec 20 processeurs : 1 cycle
  - durée réelle avec 20 processeurs : 7 cycles
  - $G = \frac{20}{7} = 2,85$
  - $E = \frac{2,85}{20} = 0,142$

Parallélisme - Loi d'Amdhal

Accélération

$$Acc = \frac{1}{(1-P) + \frac{P}{N}}$$

- P taux de code parallèle
- N nombre de processeurs

Parallélisme - Loi d'Amdhal

Accélération

$$T = S + P$$

$$T(N) = S + \frac{P}{N}$$

$$G = \frac{T}{T(N)}$$

$$G = \frac{S + P}{S + \frac{P}{N}}$$

$$G = \frac{(1 - P) + P}{(1 - P) + \frac{P}{N}}$$

$$G = \frac{1}{(1 - P) + \frac{P}{N}}$$

Parallélisme - Quelques définitions de mesures

- Mesure de la puissance de calcul délivrée par l'architecture
- Unité de mesure : Nombre d'opération par Seconde (Méga OPération par Seconde : MOPS)
- Puissance crête : Eldorado
- Puissance effective : puissance délivrée pour l'algorithme
- Benchmark :
  - Algorithme ou ensemble d'algorithme représentatifs d'un problème
  - BLAS, BLAS2 ...
  - SPECINT, SPECFP

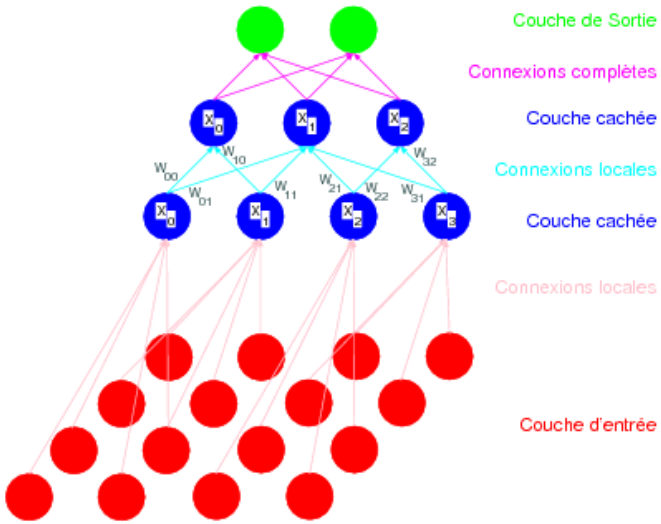
Les Réseaux Neuronaux

Définition

Un réseau de neurones est un outil informatique non plus programmé mais entrainé, c'est-à-dire qu'il construit la fonction qu'il réalise à partir d'exemples. Il existe deux phases pour ce type d'algorithmique, une phase dite de reconnaissance correspondant au fonctionnement souhaité du réseau. L'autre phase existant dans les réseaux de neurones est la phase d'apprentissage où le réseau règle ses paramètres internes afin de construire la fonction qu'il réalise.

Perceptrons multi-couches (PMC)

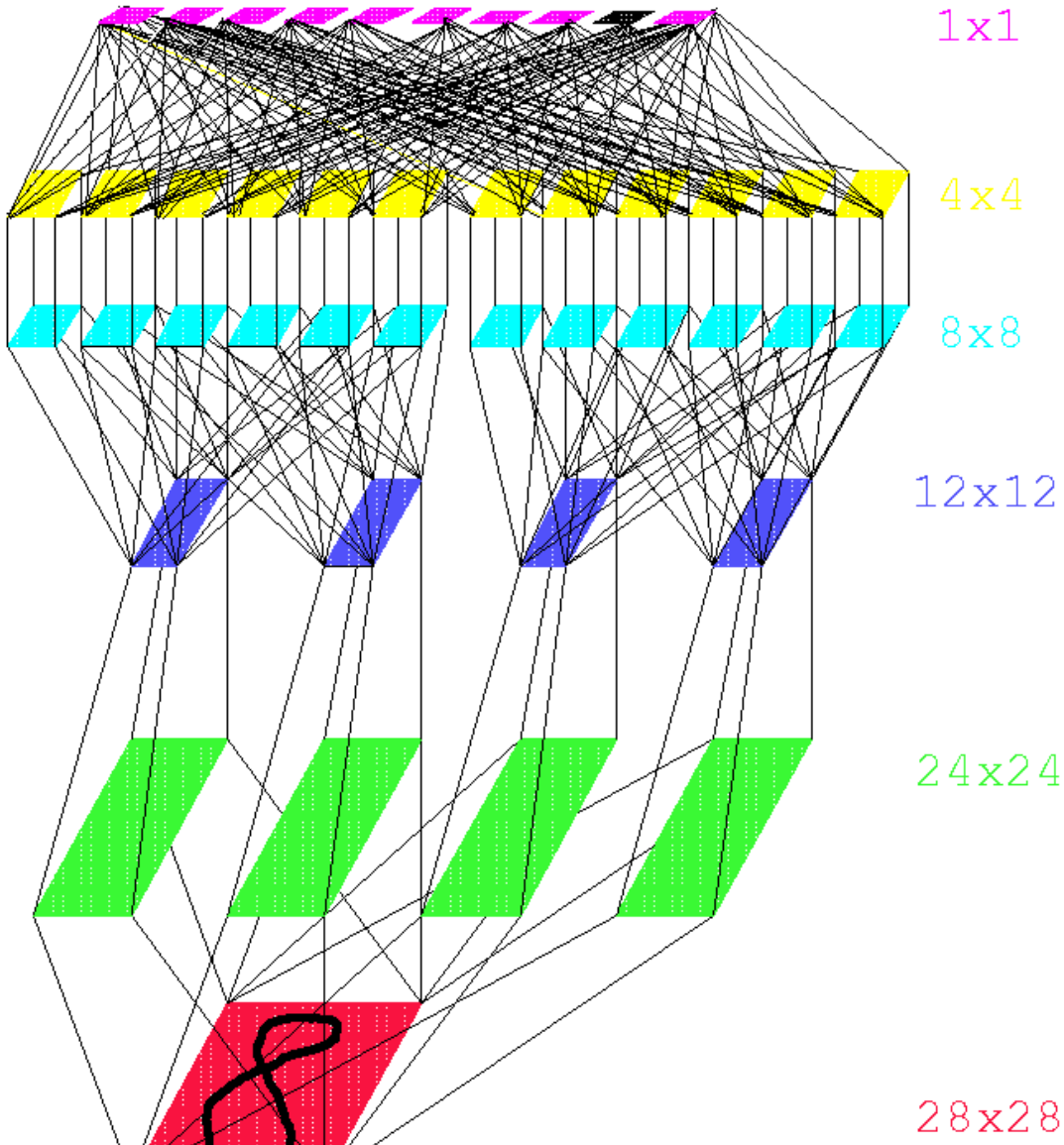
- Structure d'un réseau



- Primitives de calcul pour la dynamique d'un neurone
  - Potentiel Post-Synaptique
    $V_i = g\left(X_{j(j\in E_i)}\right) = \sum_{j\in E_i} X_j * W_{ij}$
  - Etat des neurones
    $X_i = f(V_i) = m\frac{1-e^{-\lambda V_i}}{1+e^{-\lambda V_i}}$

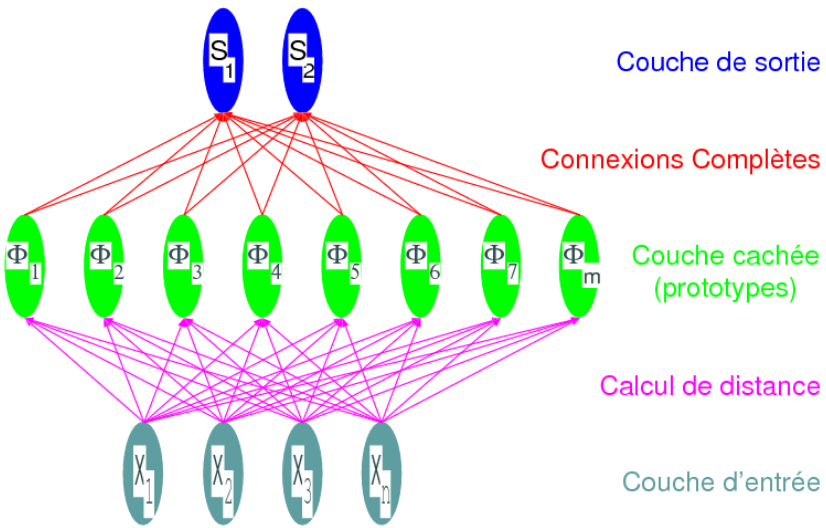
Reconnaissance de Caractère

- Réseau de neurone pour la reconnaissance de Caractères : LeNet



Réseaux à fonction à base radiale (RBF)

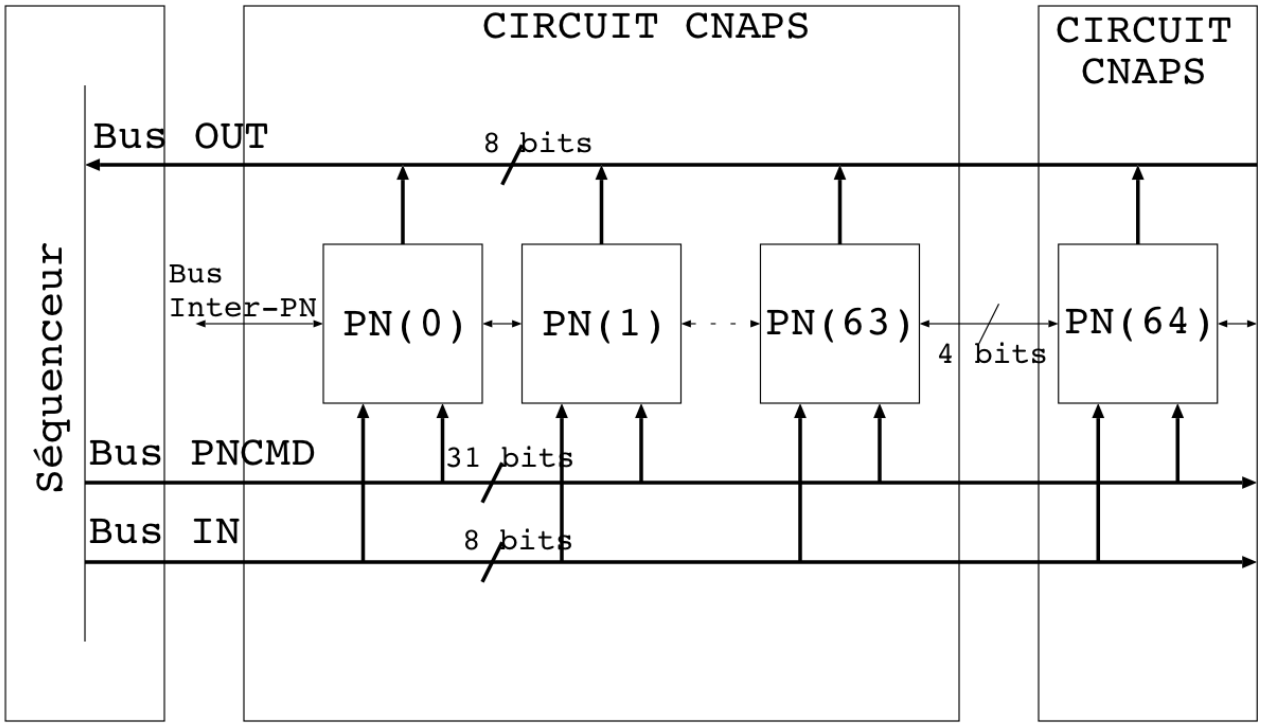
- Structure d'un réseau



- Primitives de calcul pour la dynamique d'un neurone
  - Calcul de distance
    - Manhattan : $V_i = \sum_{j \in E_i} |W_{ji} - X_j|$
    - Euclide : $V_i = \sum_{j \in E_i} (W_{ji} - X_j)^2$
    - Mahalanobis : $V_i = \sum_{j \in E_i} (W_{ji} - X_j)^t \Sigma_i^{-1} (W_{ji} - X_j)$
  - Etat des neurones cachés $\Phi_i = f(V_i) = e^{\frac{-V_i}{\lambda}}$
  - Etat des neurones sortie $S_i = \sum_{j \in E_i} W_{ij} * \Phi_i$

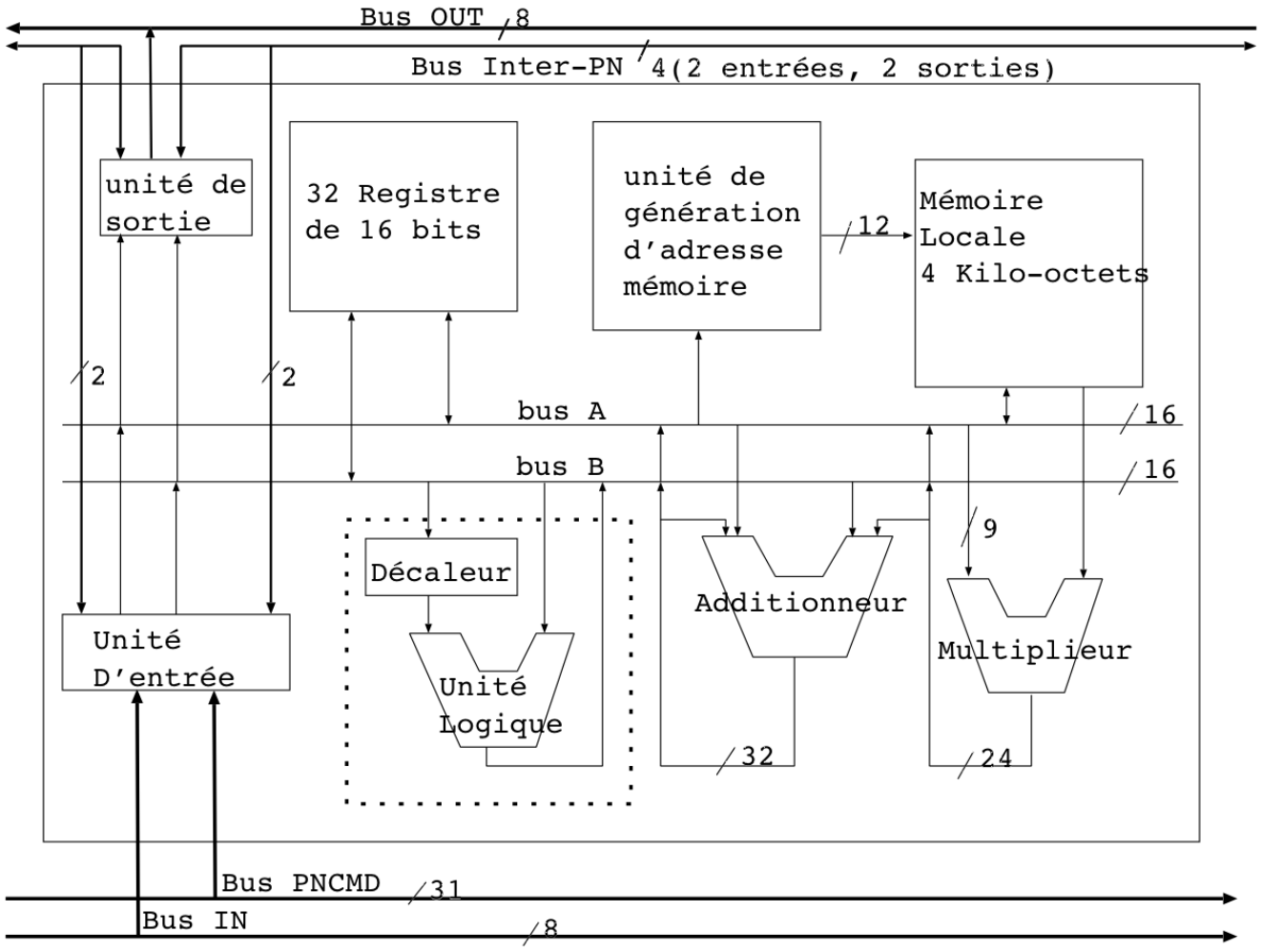
CNAPS

- Architecture Générale

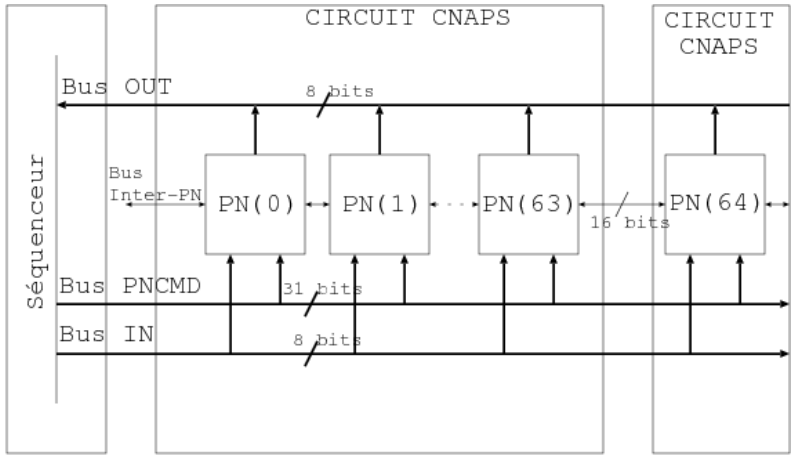


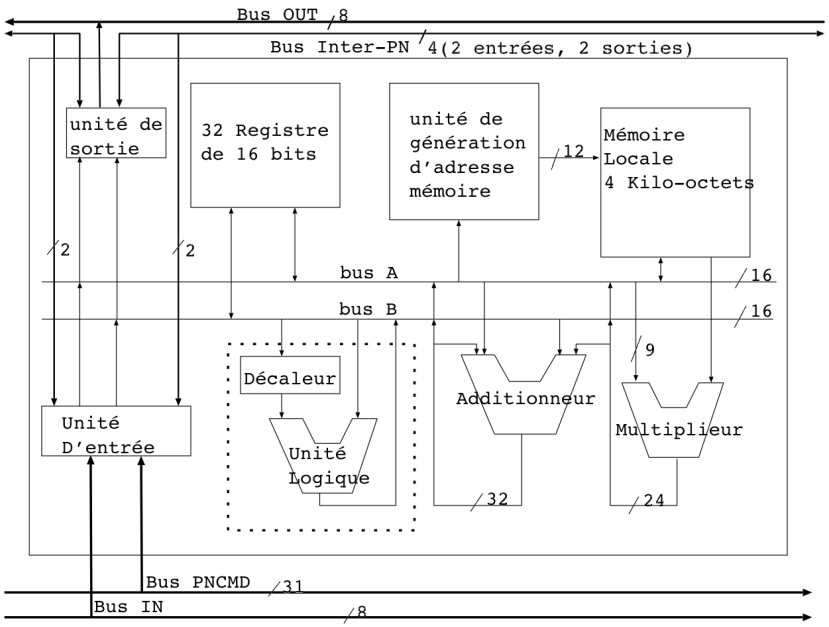
CNAPS

- Architecture Processeur



CNAPS - Améliorations de l'architecture de CNAPS





- Augmentation de la fréquence de fonctionnement
- Augmentation du nombre de processeur
- Extensions de l'architecture de CNAPS
  - Possibilité de décrémenter des adresses
  - Augmentation à 8 bits de la largeur de l'anneau

CNAPS - Prédiction des performances de CNAPS amélioré pour LENET

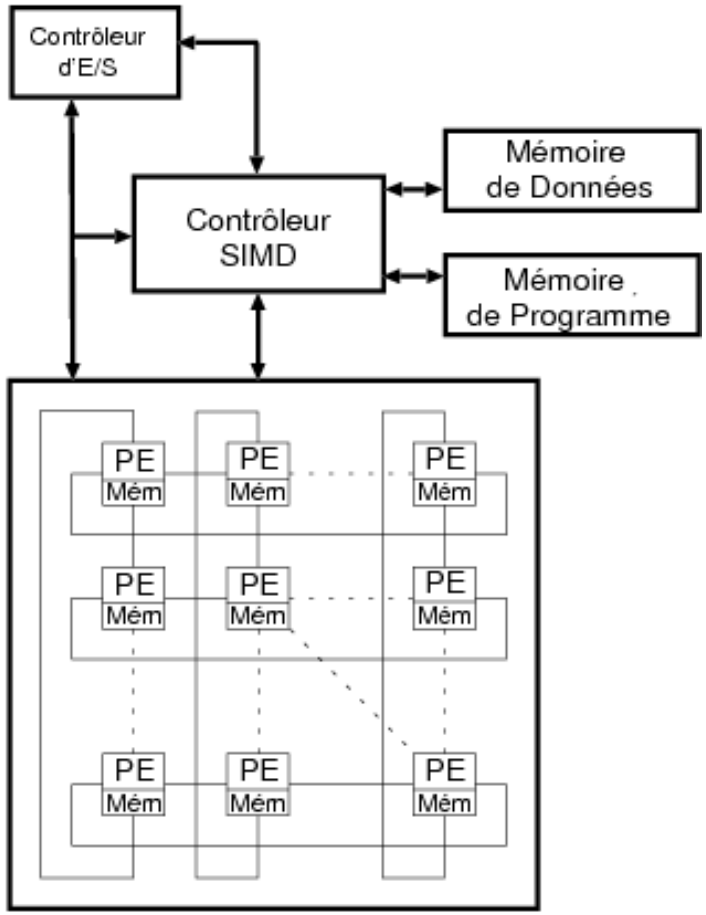
Architecture	**Temps (ms)**	**Vitesse MOPS**	**Gain %**
CNAPS 128 20 Mhz	2,57	78,4	
CNAPS 128 25 Mhz	1,83	110,1	40
CNAPS 512 20 Mhz	1,75	115,1	47
E-CNAPS 128 20 Mhz	1,92	104,9	34
E-CNAPS 512 25 Mhz	1,28	157,4	100
ANNA	1,20	167,9	
Pentium II 266 MHz	2,1	95	

Reconnaissance de Caractère

- Jocelyn Cloutier et Al. - *Université de Montréal - Canada*
- Virtual Image Processor
- Matrice de 2x2 Altera EPF81500 - 16 000 portes
- Mémoire Externe

Reconnaissance de Caractère

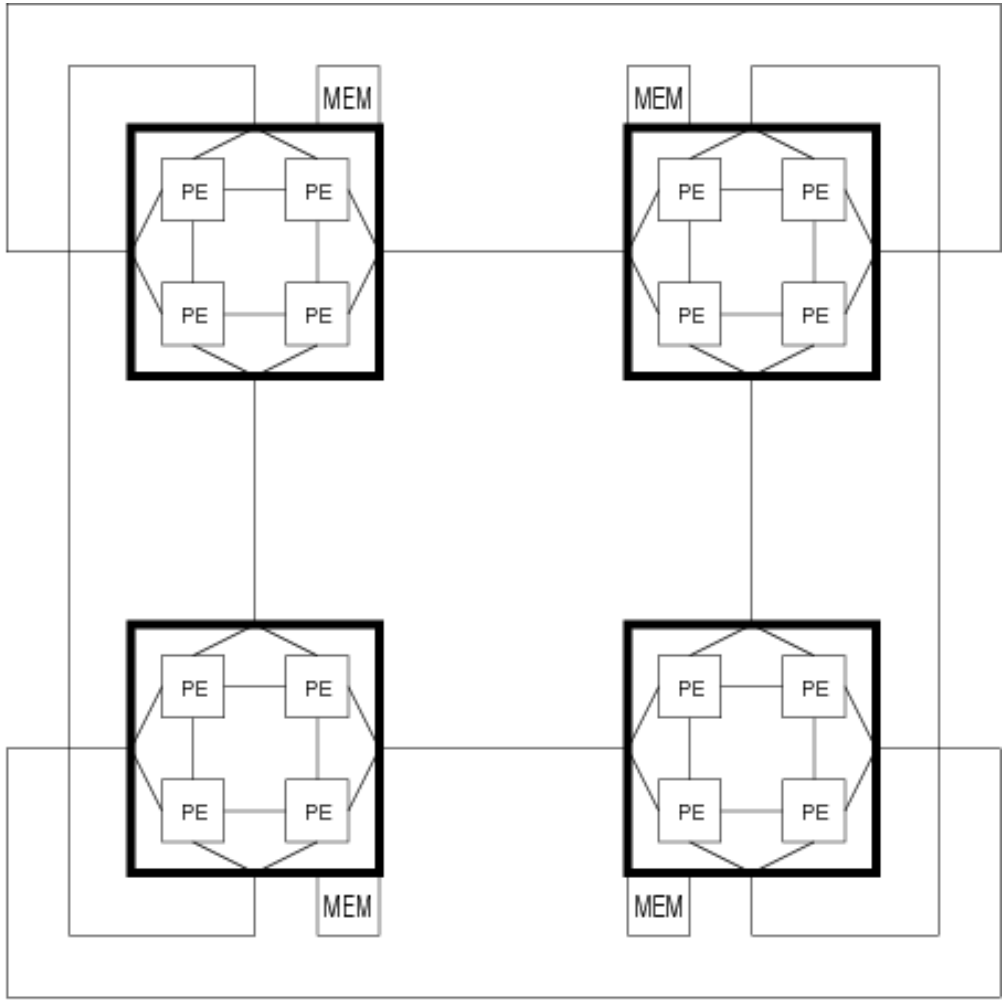
- Architecture



Matrice de Processeurs Elémentaires

Reconnaissance de Caractère

- Mise en Oeuvre



Reconnaissance de Caractère

- Convolution avec un motif - Corrélation

  $z_{i,j} = \begin{cases} 1 & \text{si } y_{i,j} > t \\ 0 & \text{sinon} \end{cases}$ $\quad y_{i,j} = \sum_{k=1}^{N} \sum_{n=1}^{M} f(x_{i+k,j+n}, w_{k,n})$

  où $x_{i,j}$ est le pixel originel, $w_{k,n}$ est le pixel du motif, N et M sont la hauteur et la largeur du motif, et f une fonction booléenne.

- Résultats

Taille Motif	VIP	Pentium 90 MHz
4 x 4	1,5 ms	114 ms
8 x 8	6 ms	238 ms
16 x 16	24 ms	961 ms

- 16 x 16 $\longrightarrow$ 216 ms sur un Pentium 400 MHz

Reconnaissance de Caractère - Réseaux de Neurones

- Performance

ANNA	VIP	CNAPS
1,02 ms	0,75 ms	3 ms

Les Benchmarks : Spec 2000

- Mesure de durée ou de débit de programmes tests
- Choix de programmes tests représentatifs
- Résultats : valeurs moyennée des temps d'exécution

Les Benchmarks : Spec 2000

CINT2000 (Integer Component of SPEC CPU2000)

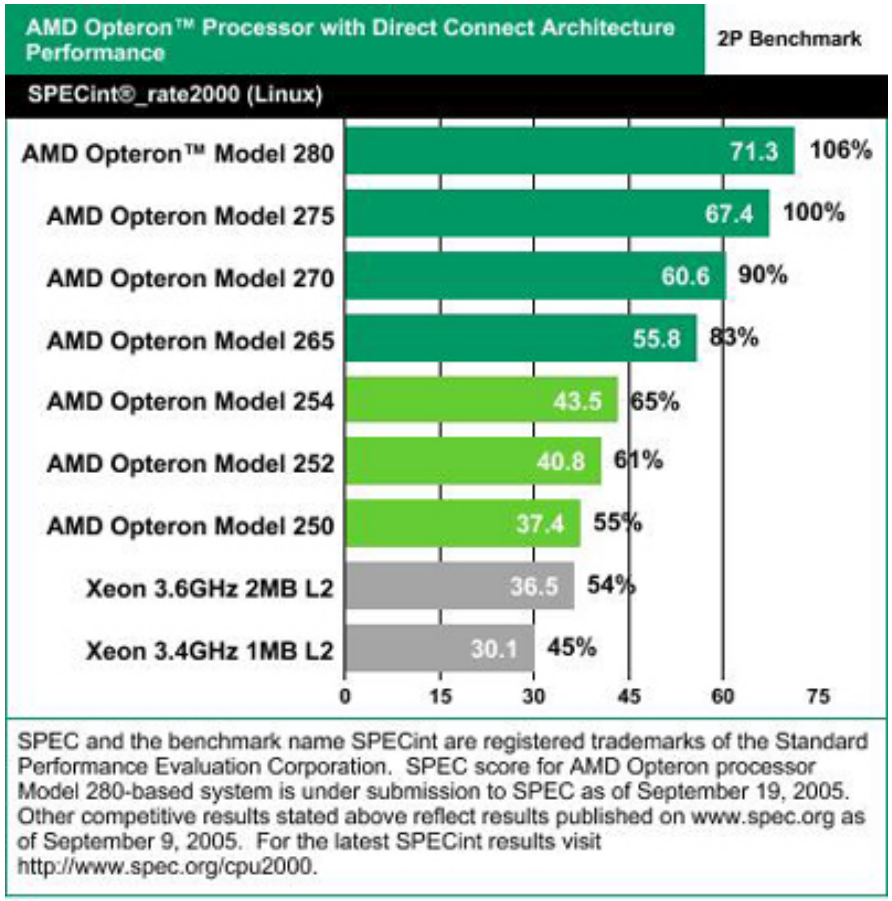
Benchmark	Language	Category
164.gzip	C	Compression
175.vpr	C	FPGA Circuit Placement and Routing
176.gcc	C	C Programming Language Compiler
181.mcf	C	Combinatorial Optimization
186.crafty	C	Game Playing: Chess
197.parser	C	Word Processing
252.eon	C++	Computer Visualization
253.perlbmk	C	PERL Programming Language
254.gap	C	Group Theory, Interpreter
255.vortex	C	Object-oriented Database
256.bzip2	C	Compression
300.twolf	C	Place and Route Simulator

Les Benchmarks : Spec 2000

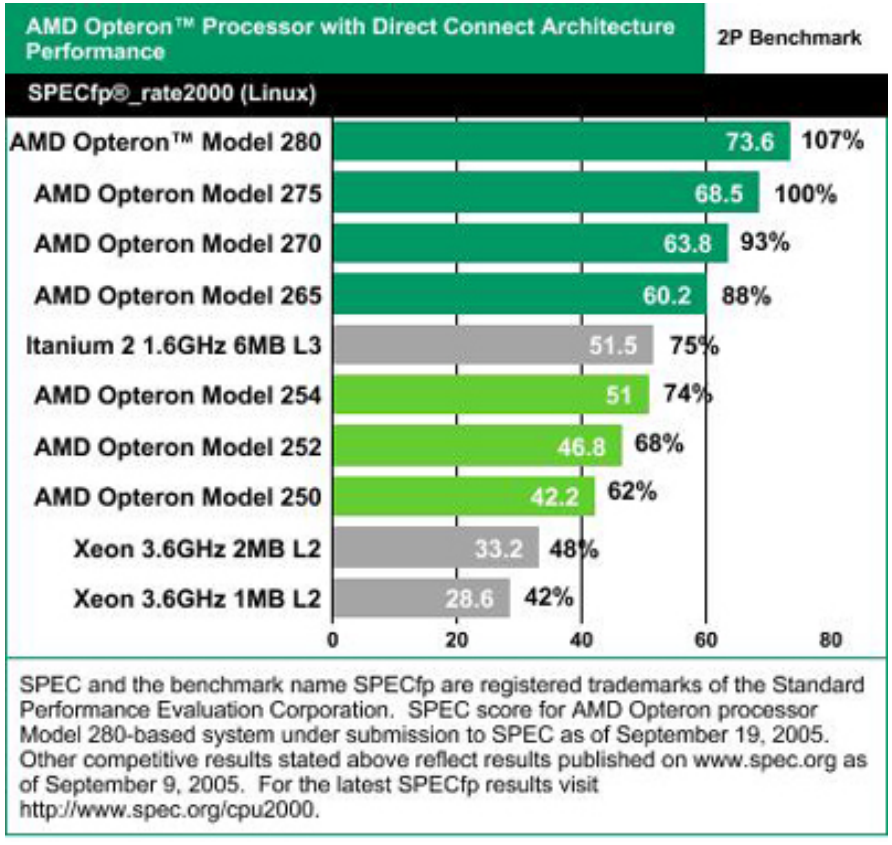
CFP2000 (Floating Point Component of SPEC CPU2000)

Benchmark	Language	Category
168.wupwise	Fortran 77	Physics / Quantum Chromodynamics
171.swim	Fortran 77	Shallow Water Modeling
172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field
173.applu	Fortran 77	Parabolic / Elliptic Partial Differential Equations
177.mesa	C	3-D Graphics Library
178.galgel	Fortran 90	Computational Fluid Dynamics
179.art	C	Image Recognition / Neural Networks
183.equake	C	Seismic Wave Propagation Simulation
187.facerec	Fortran 90	Image Processing: Face Recognition
188.ammp	C	Computational Chemistry
189.lucas	Fortran 90	Number Theory / Primality Testing
191.fma3d	Fortran 90	Finite-element Crash Simulation
200.sixtrack	Fortran 77	High Energy Nuclear Physics Accelerator Design
301.apsi	Fortran 77	Meteorology: Pollutant Distribution

Specint 2000



Specfp 2000


AMD Opteron™ Processor with Direct Connect Architecture Performance
2P Benchmark
SPECfp®_rate2000 (Linux)
AMD Opteron™ Model 280
73.6
107%
AMD Opteron Model 275
68.5
100%
AMD Opteron Model 270
63.8
93%
AMD Opteron Model 265
60.2
88%
Itanium 2 1.6GHz 6MB L3
51.5
75%
AMD Opteron Model 254
51
74%
AMD Opteron Model 252
46.8
68%
AMD Opteron Model 250
42.2
62%
Xeon 3.6GHz 2MB L2
33.2
48%
Xeon 3.6GHz 1MB L2
28.6
42%
0
20
40
60
80
SPEC and the benchmark name SPECfp are registered trademarks of the Standard Performance Evaluation Corporation. SPEC score for AMD Opteron processor Model 280-based system under submission to SPEC as of September 19, 2005. Other competitive results stated above reflect results published on www.spec.org as of September 9, 2005. For the latest SPECfp results visit http://www.spec.org/cpu2000.

Modèle Analytique

Méthode

Développement d'une méthodologie basée sur la détermination d'un modèle analytique des primitives de calcul des réseaux de neurones. A l'aide de ce modèle une prédiction du temps d'exécution est réalisée mesurant ainsi les performances de la machine électronique. Ces primitives sont les règles opératoires définissant l'algorithmique associée au modèle neuronal

Modèle Analytique

Présentation de la méthodologie : 4 étapes

1. détermination des primitives du modèle neuronal : elle revient à chercher les primitives essentielles, c'est-à-dire les plus petits calculs permettant d'identifier la classe du modèle neuronal.
2. détermination du schéma de placement des données en mémoire sur l'architecture cible.
3. détermination du modèle analytique de l'architecture cible pour le modèle neuronal : cette étape permet d'extraire une fonction mathématique indiquant la durée d'exécution des primitives déterminée à l'étape 1 en fonction de divers paramètres.
4. algorithme de prédiction du temps de simulation : cette étape est la prédiction proprement dite et découle des trois étapes précèdentes. Nous prédisons le temps de simulation d'un réseau de neurones à partir d'un graphe de précèdence des primitives.

Algorithme de Prédiction

Prédiction

La construction du graphe de précèdence est défini par les règles suivantes :

1. Pour chaque instance d'une primitive de calcul un nœud dans le graphe est créé. Une valeur est attribuée à ce nœud, elle est égale à la durée d'exécution de la primitive qu'il représente. Cette durée est estimée grâce au modèle analytique extrait à l'étape 3 de la méthodologie.
2. Des arcs sont créés entre les nœuds du graphe, indiquant ainsi les interdépendances entre nœuds.

Modèle Analytique de CNAPS

Moyen de communication	Temps en nombre de cycles
Connexions complètes	
Bus à diffusion	$\alpha + \gamma*NPPI + \varepsilon*NPPI*NPPT + \theta*PI*NPPI*NPPT$
Bus Inter-Processeurs	$\alpha + \varepsilon*NPPI + \theta*NPPI*(P-1) + \psi*NPPI*NPPT + \phi*NPPI*NPPT*PI$
Connexions Locales	
Bus à diffusion	$\alpha + \gamma*NPPI+ \varepsilon*NPPI*PI + \theta*NPPT + \psi*NPPT*L$
Bus Inter-Processeurs	$\alpha + \gamma*NPPT + \phi*D_1 + \varepsilon*D_2 + \theta*NPPT*L_2 + \psi*NPPT*L_2*L_1$
Mise à jour Etats	
	$\alpha + \beta*NPPT$

- **Variables (architecture du système électronique et structure des couches)**
  P : nombre de processeurs physiques de la machine, L : localité des connexions (cas 1-D), L_1 et L_2 : localité des connexions (cas 2-D), NPPI : neurones par processeur couche initiale, NPPT : neurones par processeur couche terminale, PI : processeurs utilisés pour la couche initiale,D_1 : distance maximum de communication via l'anneau par la gauche, D_2 : distance maximum de communication via l'anneau par la droite
- **Paramètres de l'architecture**
  $\alpha\beta\gamma\varepsilon\theta\psi\phi$

Modèle Analytique

- Travaux de L. Gaborit
- Utilisation du modèle de Hockney et Jesshope
- 2 paramètres pour modéliser les performances d'une architecture électronique
- r_∞ est la puissance infinie ou puissance asymptotique.
- $n_{1/2}$ est le volume de donnée nécessaire pour obtenir la moitié de r_∞

Modèle analytique

- Modèle Original pour Machine Vectorielle

$$T(n) = \frac{n + n_{1/2}}{r_\infty}$$

- Equation d'une droite de pente $\frac{1}{r_\infty}$ et d'ordonnée à l'origine $\frac{n_{1/2}}{r_\infty}$
- Modèle Généralisé

$$T(n) = \frac{\omega(n) + \omega(n_{1/2})}{r_\infty}$$

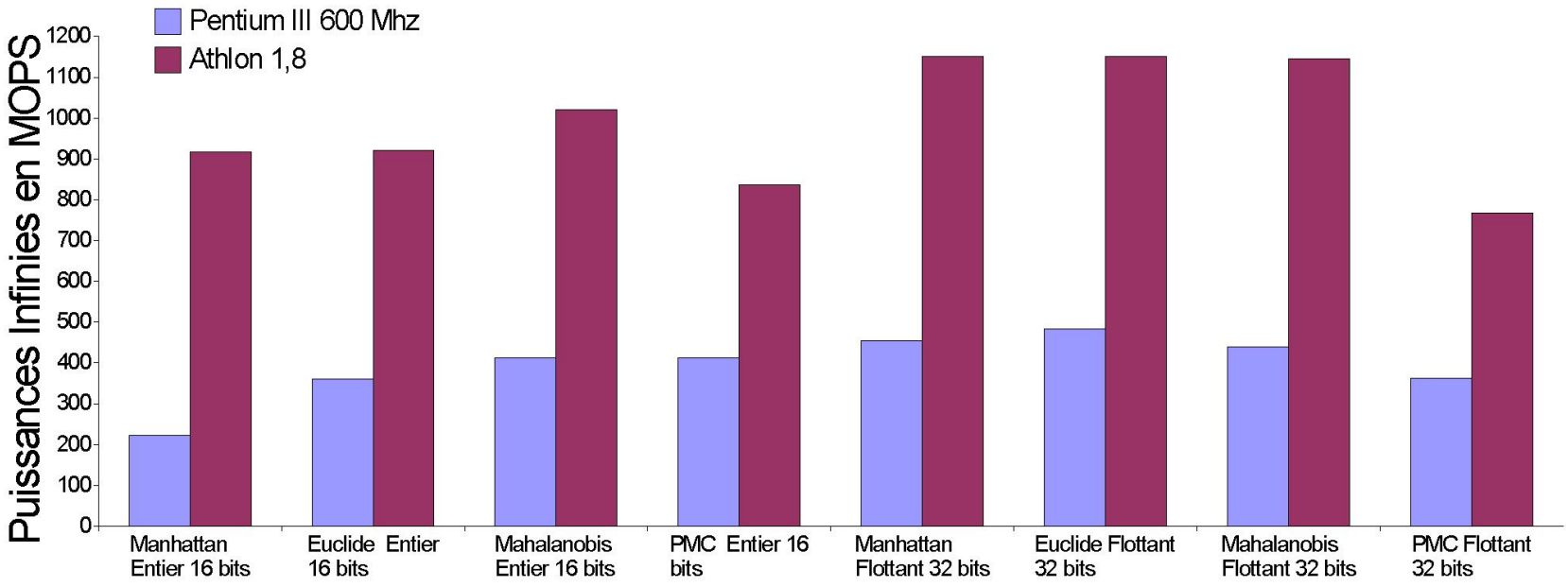
, $\omega(n)$ où $p * n + q$ est le nombre d'opérations arithmétiques réalisées.

- Où bien

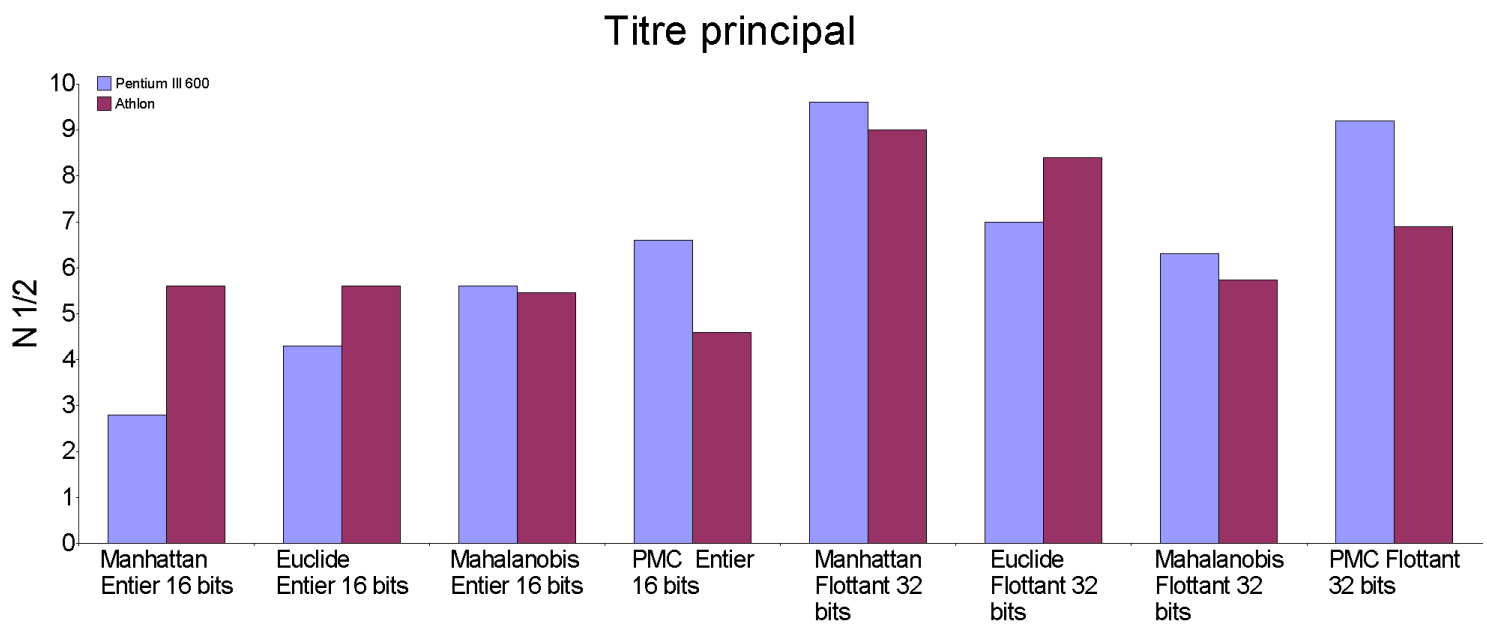
$$T(n) = \frac{\omega(n) + \omega(n_{1/2})}{r_\infty}$$

, espace T, ω

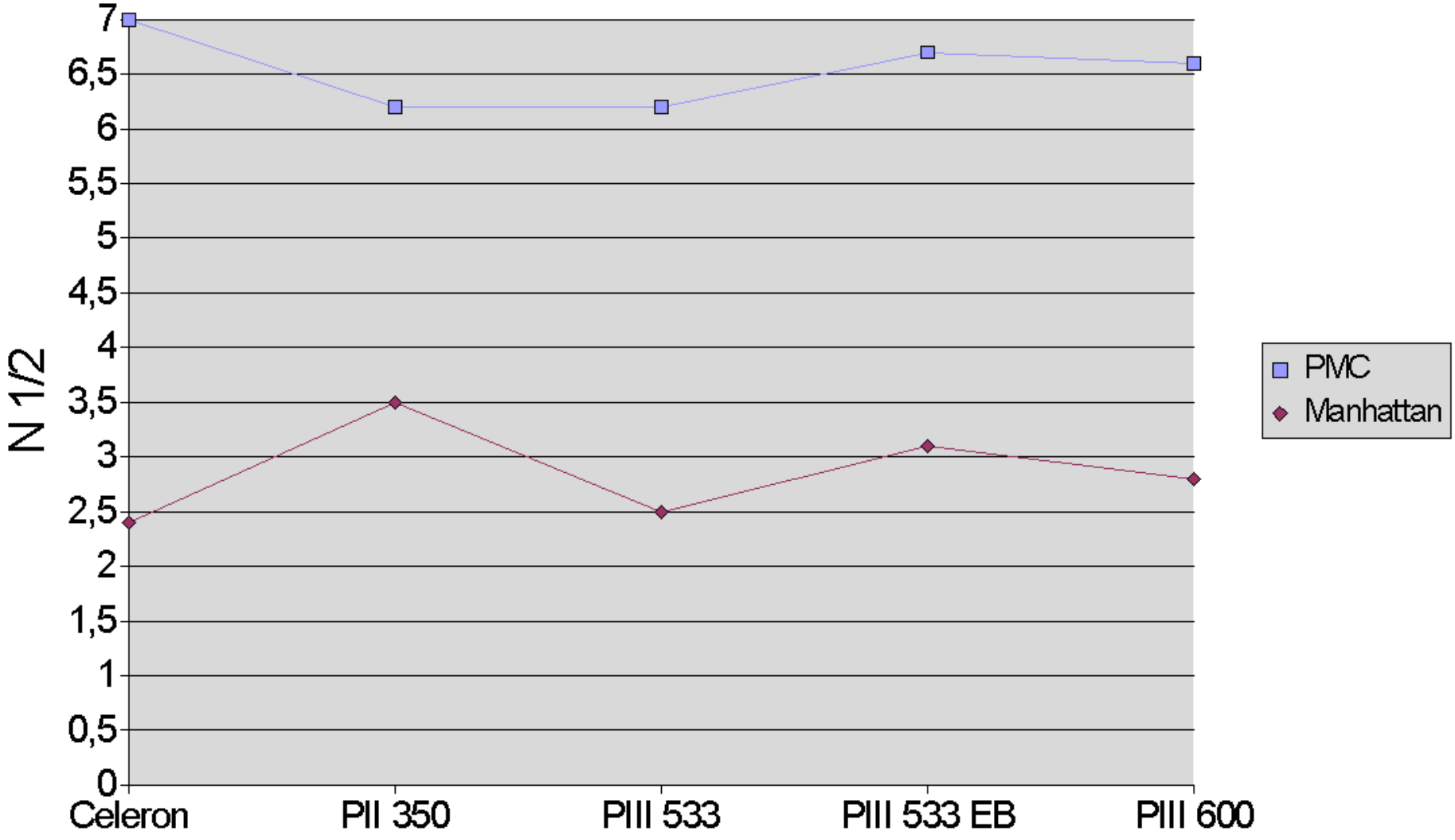
Modèle Analytique



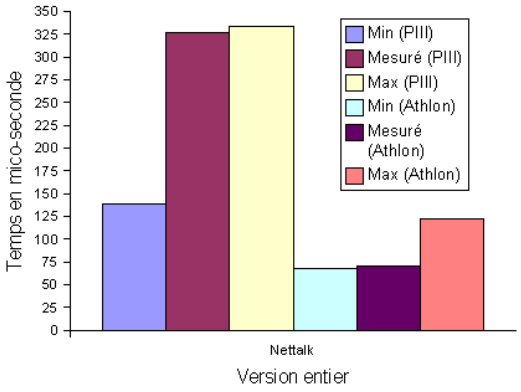
Modèle Analytique

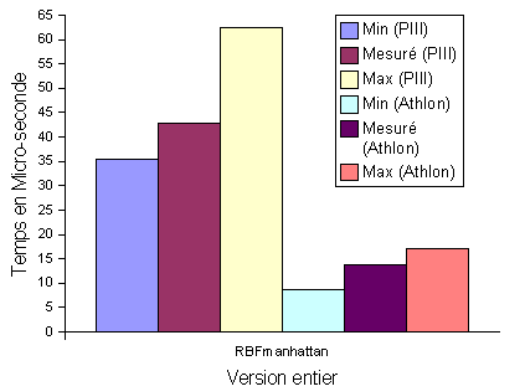


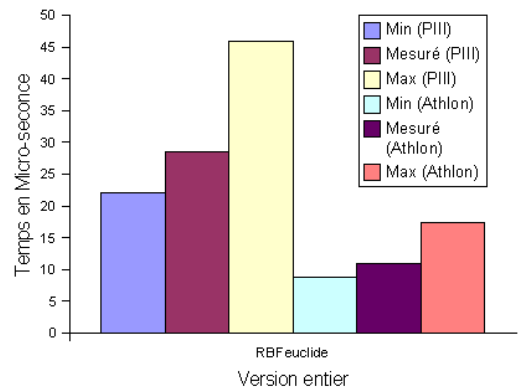
Modèles Analytiques

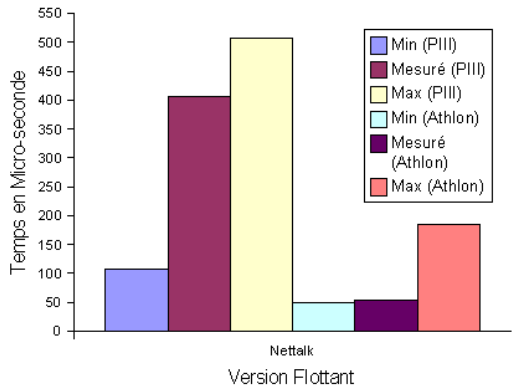


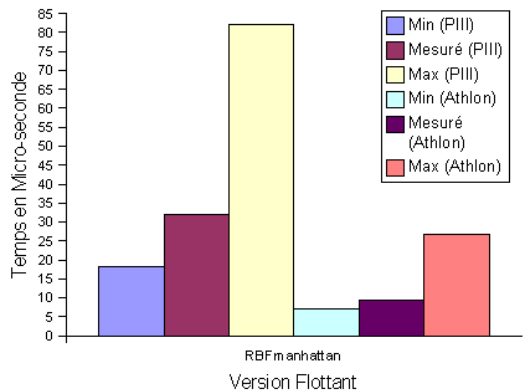
Modèle Analytique

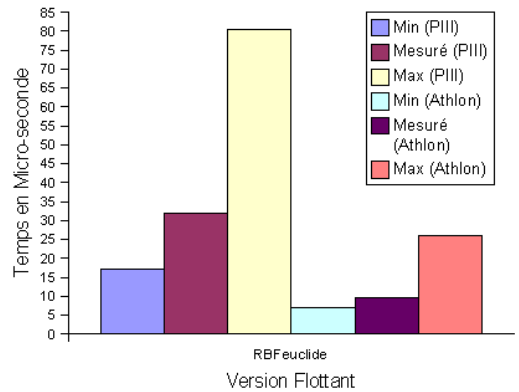


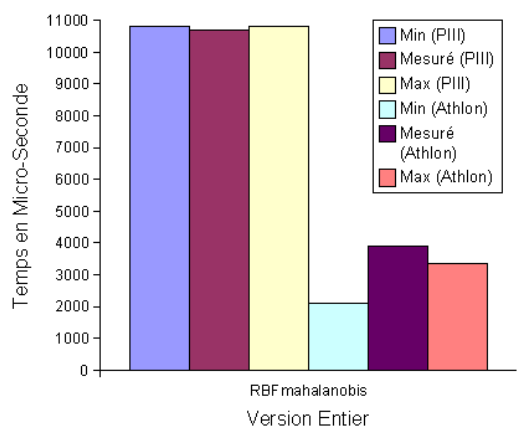


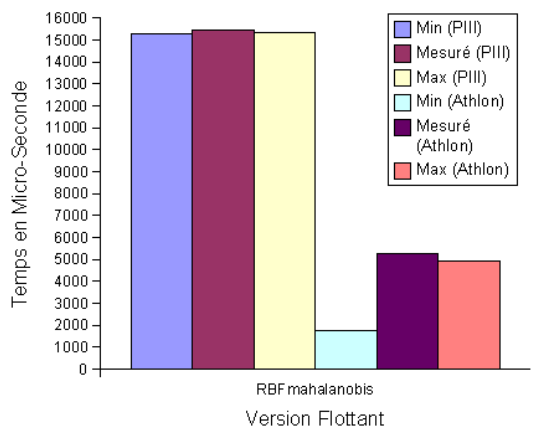












Profil d'exécution

```
#include <stdio.h>
int deuxchiffre(int nb)
{
        int bool;
                if ((nb >= 1) && (nb <100))
                                bool = 1;
                        else
                               bool = 0;
                        return bool;}

long long int factoriel(int n)
{
        long long int i,fact;
        fact = 1;
        for(i=1;i<n+1;i++)
                        fact = fact*i;
        return fact;}

int main(int argc, char **argv)
{
        int i;
        long long int n;
        for(i=0;i<10000;i++)
                for(n=1;n<100;n++)
        {
                if(deuxchiffre(n))
                  fprintf(stderr,"Factoriel = %ld \n",factoriel(n));
                else
                 printf("erreur \n"); }}
```

Gprof

```
gcc -pg -o facto facto.c
facto
gprof facto
```

Gprof

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 92.62      0.56     0.56   990000   570.71   570.71  factoriel
  7.38      0.61     0.04                             main
  0.00      0.61     0.00   990000     0.00     0.00  deuxchiffre
```

Gprof

```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.04    0.56                 main [1]
                0.56    0.00  990000/990000      factoriel [2]
                0.00    0.00  990000/990000      deuxchiffre [3]
-----------------------------------------------
                0.56    0.00  990000/990000      main [1]
[2]     92.6    0.56    0.00  990000         factoriel [2]
-----------------------------------------------
                0.00    0.00  990000/990000      main [1]
[3]      0.0    0.00    0.00  990000         deuxchiffre [3]
-----------------------------------------------
```

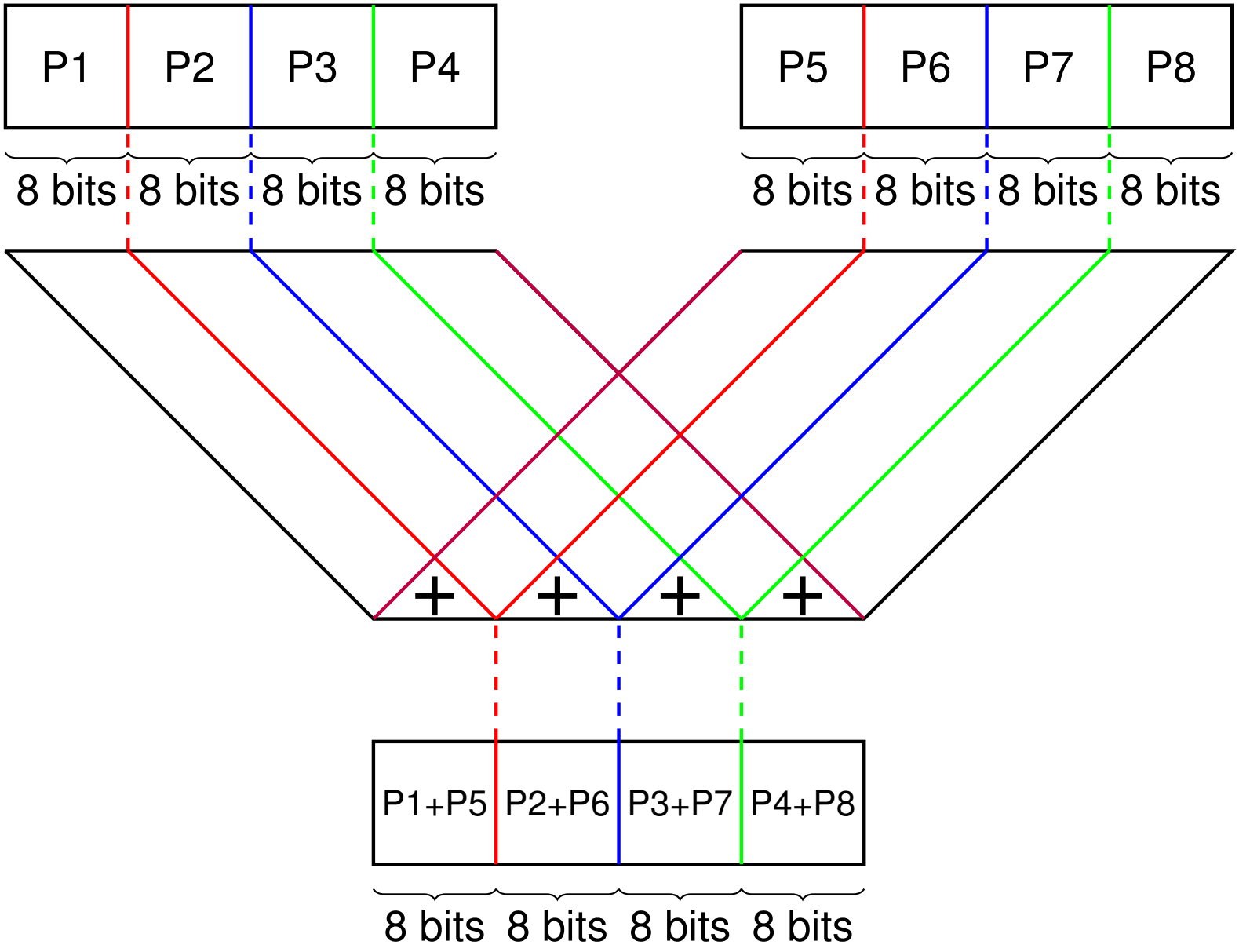
SIMD : Les SWAR

- SWAR : SIMD Within A Register
- Extension des Processeurs généralistes
- Idée : diviser des registres pour faire des calculs en parallèle
- Exemple : Un registre 64 bits peut être vu comme
  - 1 donnée sur 64 bits
  - 2 données sur 32 bits
  - 4 données sur 16 bits
  - 8 données sur 8 bits
- Réalisation d'opérateurs à opérandes à taille variable
- Introduction de mécanisme de saturation

Les SWAR

- Tous les constructeurs ont leur SWAR
  - VIS (SUN - 1995), MMX (Intel - 1997), MAX (HP - 1997)
  - Alpha (Compaq - 1997), MIPS (CGI - 1996) - Altivec (Motorola - 1998)
- Besoin de nouveaux outils
  - Compilateur
  - Type de données

Unité SWAR - Principe



Les SWAR - MMX

- Changement Majeur apporté au Pentium
- Pentium II = Pentium Pro
- 57 nouvelles instructions
- 8 Registres MM0 à MM7 64 bits
- Mappage sur registres flottants
- Calcul Entier
- Tout résultat est stocké dans un registres
- Pas de positionnement de drapeau

Les SWAR - MMX

- Nouveaux types de données
  - Packed Byte (PB) : 8 octets par registre
  - Packed Word (PW) : 4 mots de 16 bits par registre
  - Packed Double Word (PDW) : 3 mots de 32 bits par registres
  - Quad Word (QW) : 1 mot de 64 bit par registre

Les SWAR - MMX

- Instructions
  - Transfert mémoire : QW ou PDW
  - Arithmétique : Addition - Soustraction, PB - PW - PDW, signé ou non
  - Arithmétique : Multiplication, PW, signé
  - Arithmétique : MultiplicationAccumulation - PW - signé
  - Comparaison : Egalité - Supériorité, PB - PW - PDW
  - Conversion
  - Logiques : Et - Ou - OuExclusif - NonEt, QW
  - Logiques : Décalages Logiques, PW - PDW - QW
  - Logiques : Décalages Arithmétiques, PW - PDW
  - EMMX : Instruction de réinitialisation pour les flottants

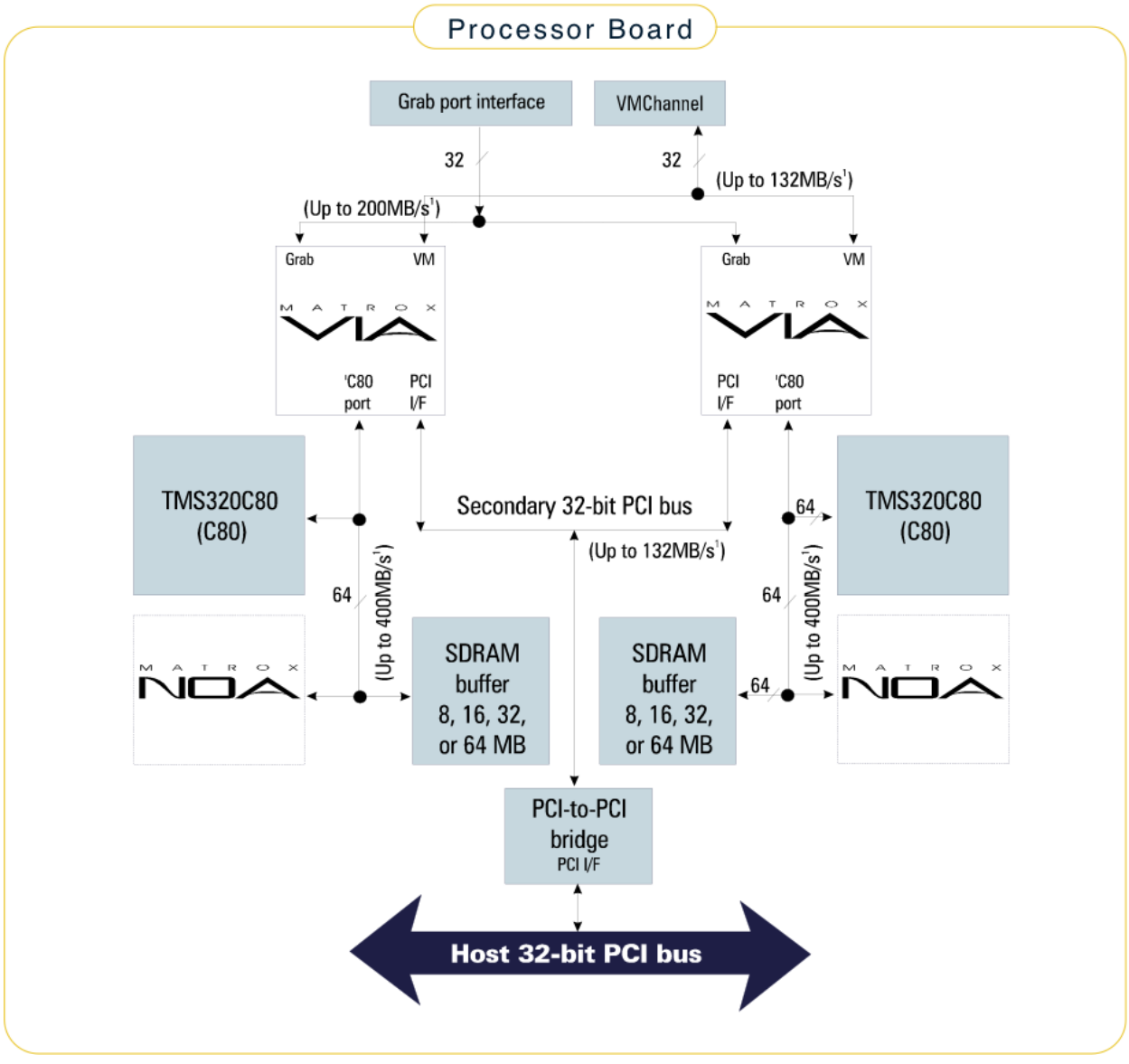
Les SWAR - MMX2 = SSE

- Pentium III = Pentium II + SSE
- Registre XMM0 à XMM7 sur 128 bits
- Physiquement distinct
- 70 nouvelles instructions
- Extension de MMX pour les flottants
- 4 données par registres
- Possibilité de faire du scalaire en ne considérant que la dernière paire
- Contrôle de Cachabilité

Les SWAR - SSE2

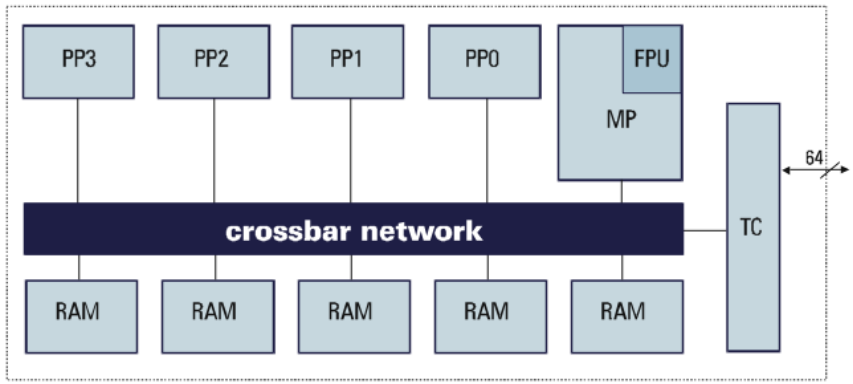
- Introduit dans Pentium IV
- Pentium IV $\neq$ Pentium III + SSE2
- 2 nouveaux types de données
  - Flottants 64 bits
  - Entier 128 bits
- Augmentation de la possibilité du contrôle de cache

Genesis


Processor Board
Grab port interface
VMChannel
32
32
(Up to 132MB/s¹)
(Up to 200MB/s¹)
Grab
VM
Grab
VM
MATROX VIA
MATROX VIA
'C80 port
PCI I/F
PCI I/F
'C80 port
TMS320C80 (C80)
Secondary 32-bit PCI bus
64
TMS320C80 (C80)
(Up to 132MB/s¹)
64
(Up to 400MB/s¹)
64
(Up to 400MB/s¹)
MATROX NOA
SDRAM buffer 8, 16, 32, or 64 MB
SDRAM buffer 8, 16, 32, or 64 MB
64
MATROX NOA
PCI-to-PCI bridge
PCI I/F
Host 32-bit PCI bus

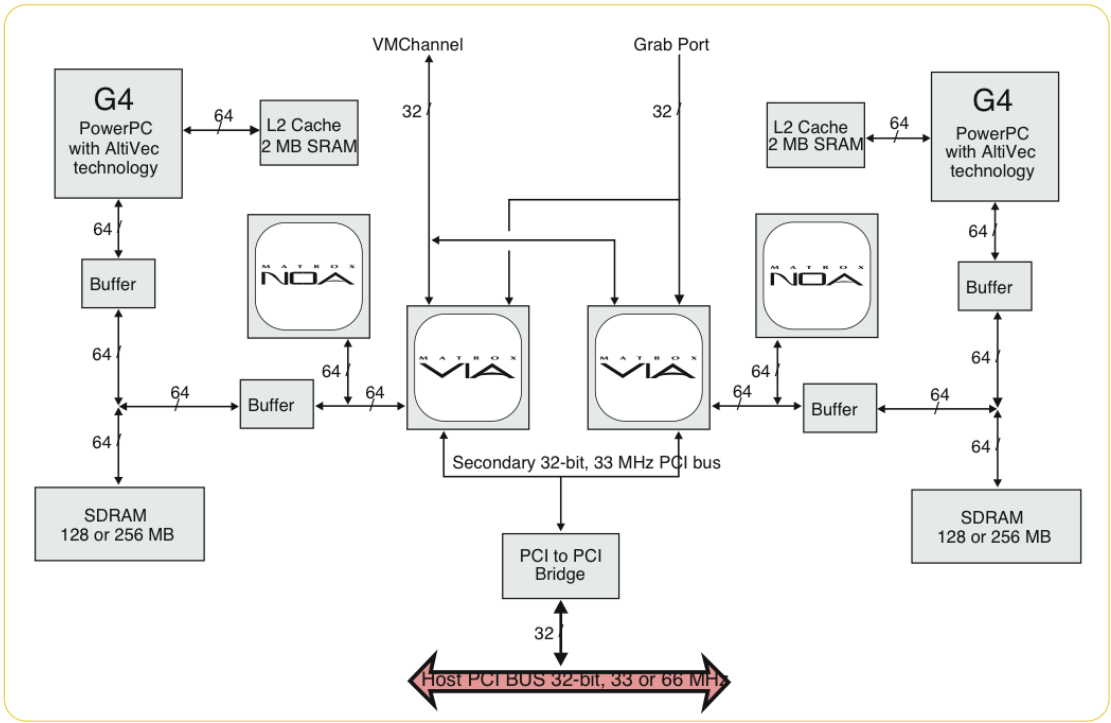
Genesis

TMS320C80

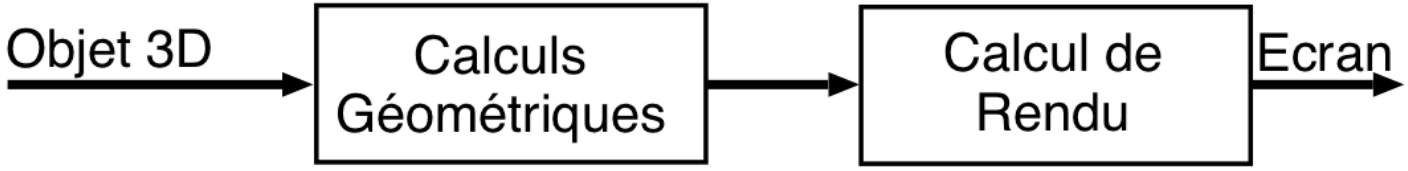


PP0-3 : Parallel processors 0-3 (advanced DSP, 32-bit integer units)
MP : Master processor (32-bit RISC processor with an IEEE-754 FPU)
FPU : Floating-point unit
TC : Transfer controller (transfers data between external and internal memory)
RAM : On-chip memory
Crossbar Network: High-speed bus switching network between processors and RAM

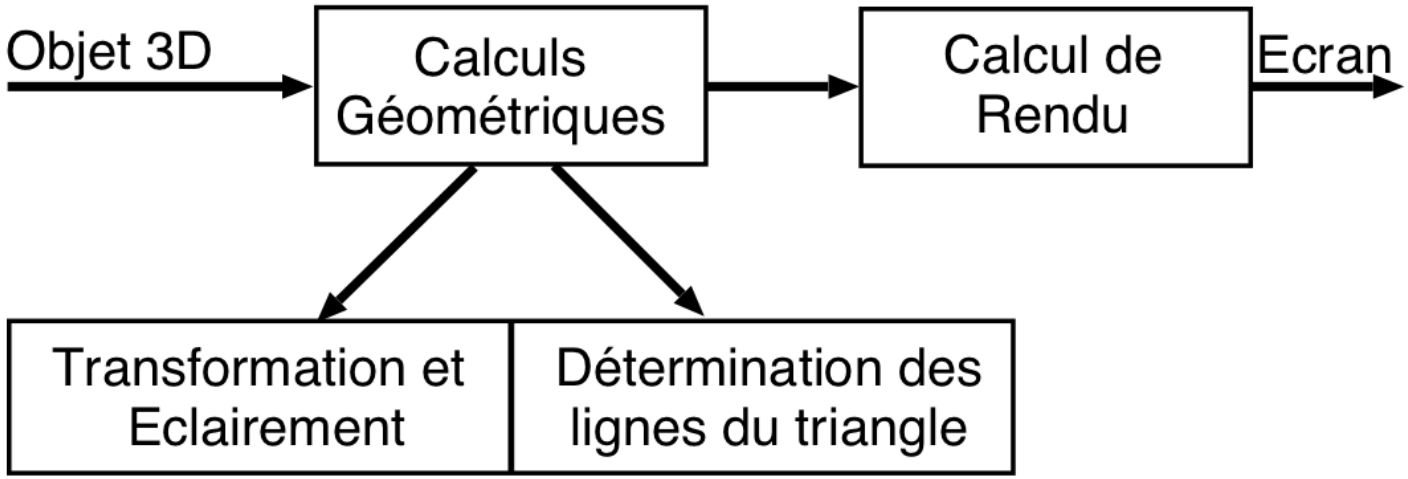
Genesis Plus



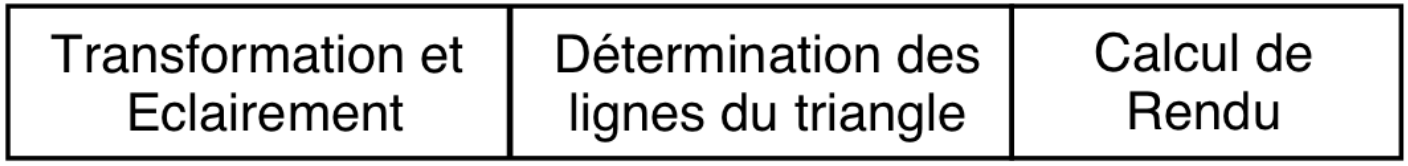
Cartes 3D



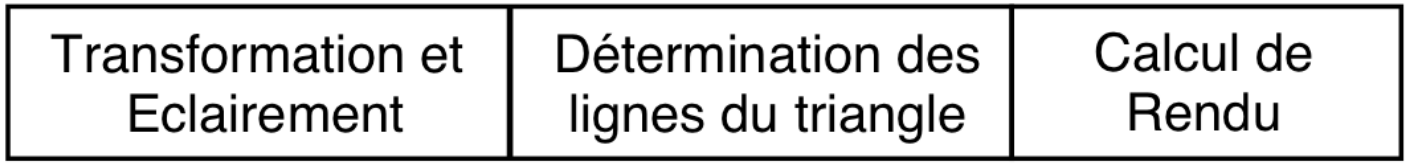
Cartes 3D


Objet 3D
Calculs Géométriques
Calcul de Rendu
Ecran
Transformation et Eclairement
Détermination des lignes du triangle

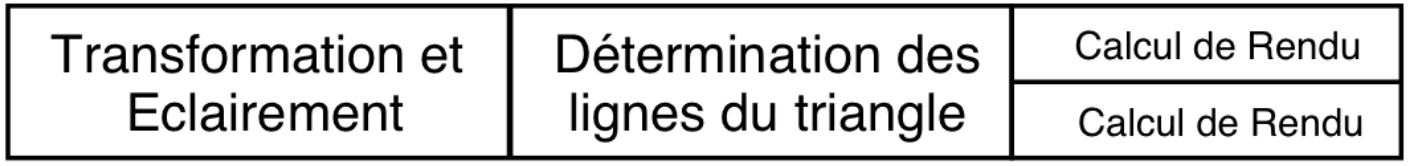
Cartes 3D



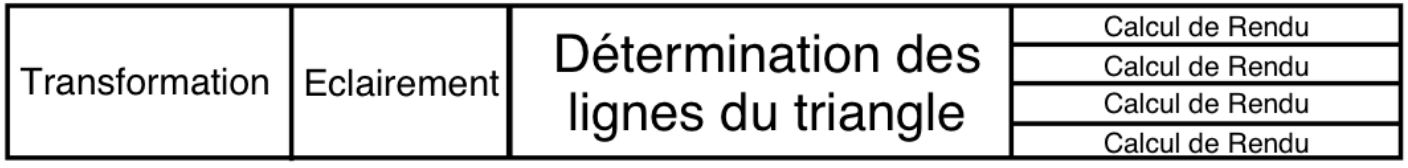
ATI Rage II, S3 Virge



ATI Rage Pro, Nvidia Riva 128 ZX, Matrox G200

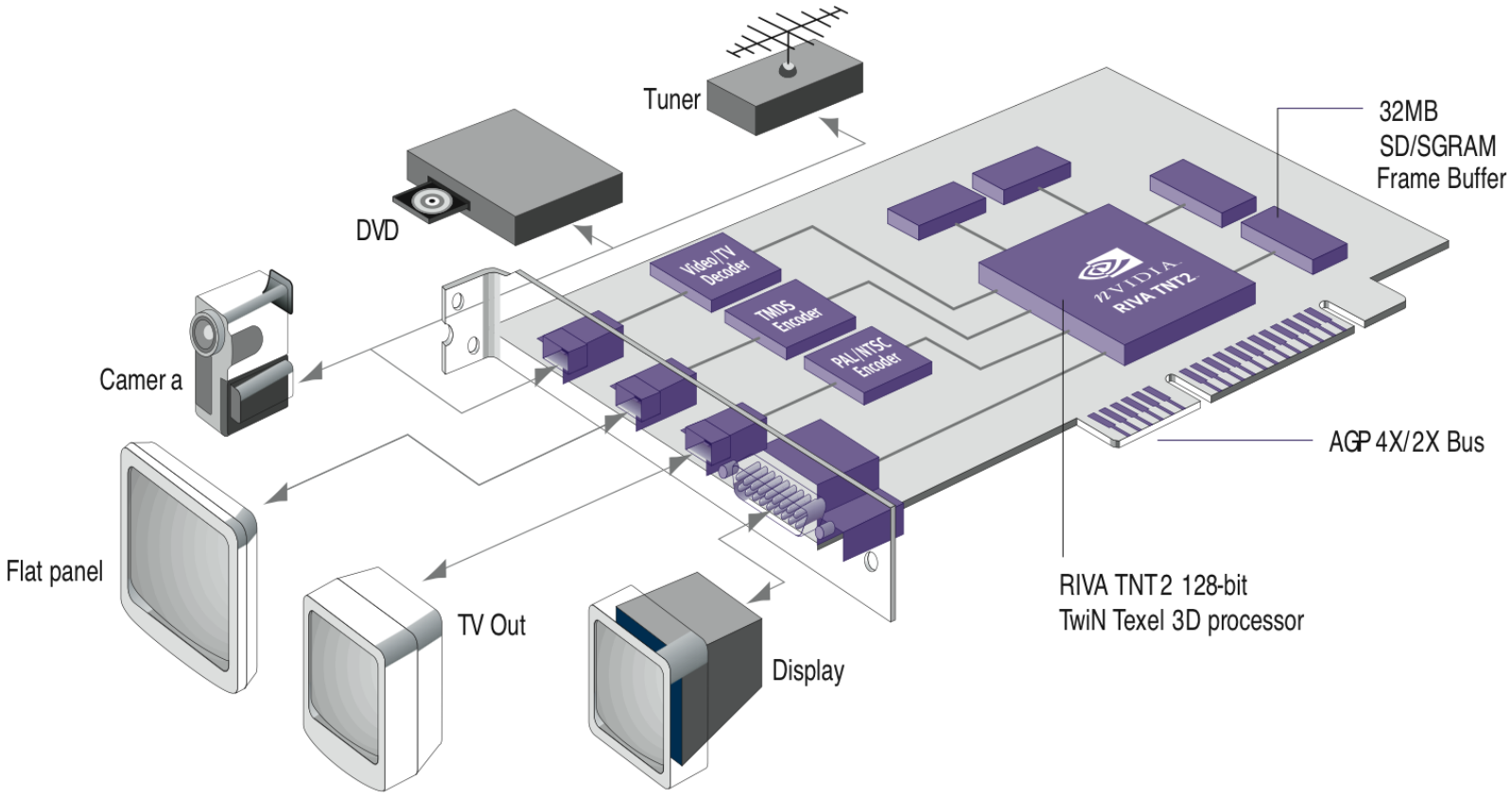


Nvidia TNT2, ATI Rage 128, S3 Savage4, 3Dfx VooDoo3

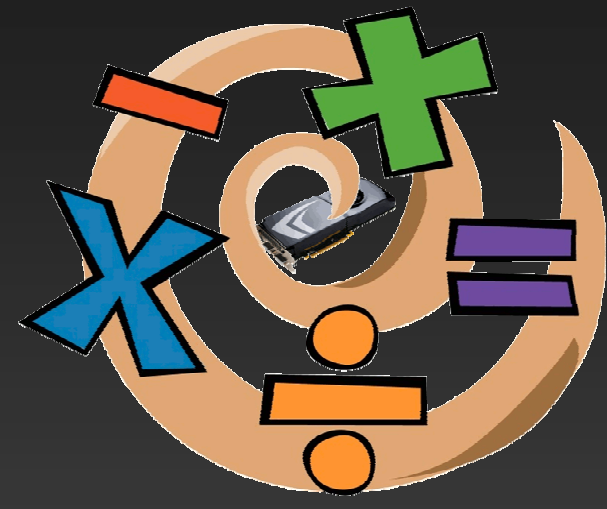


Demain ?

Cartes 3D


Tuner
DVD
Camer a
Flat panel
TV Out
Display
Video/TV Decoder
TMDS Encoder
PAL/NTSC Encoder
NVIDIA
RIVA TNT2
32MB
SD/SGRAM
Frame Buffer
AGP 4X/2X Bus
RIVA TNT2 128-bit
TwiN Texel 3D processor

Prospectives autour des processeurs graphiques

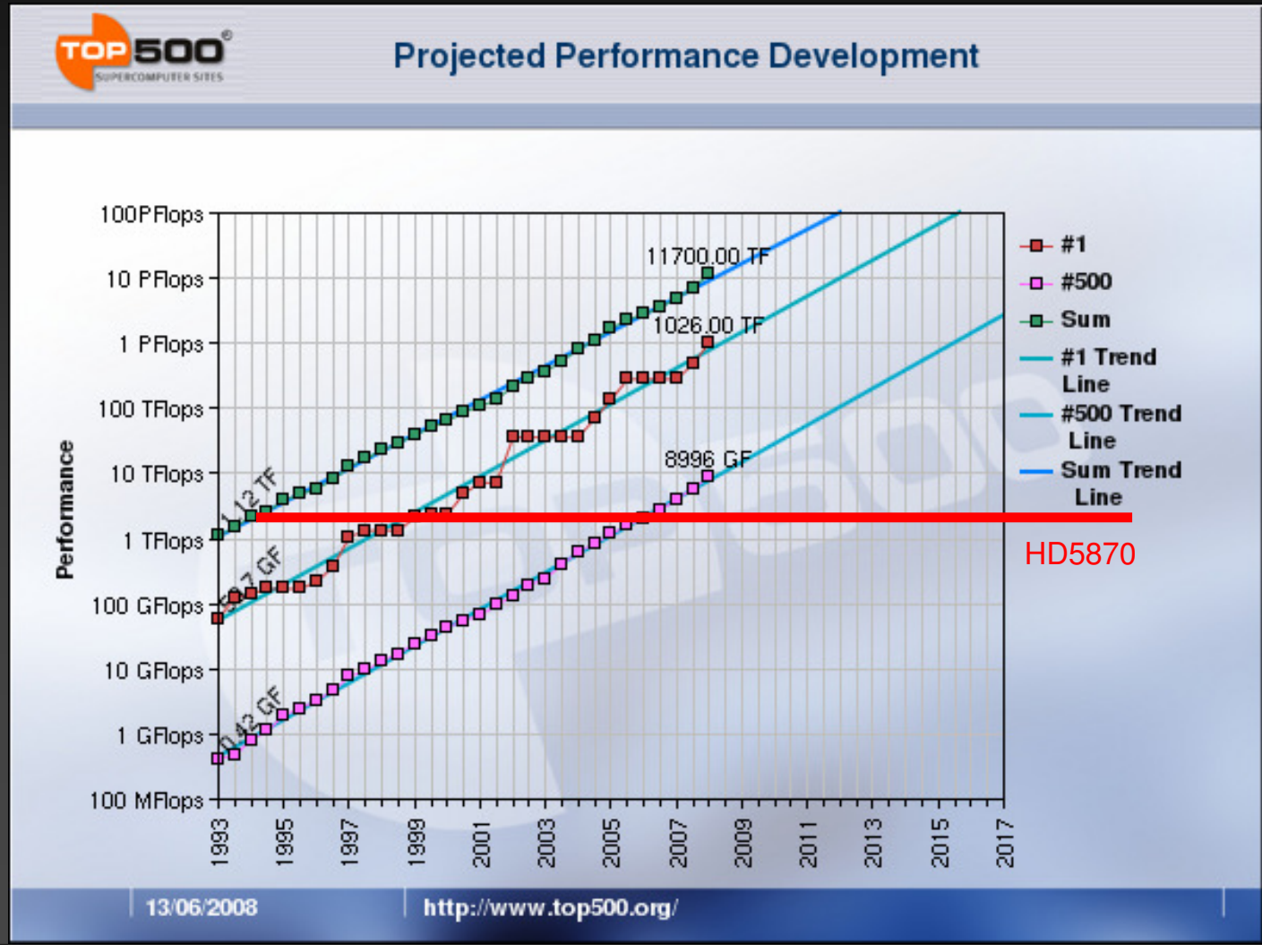


David Defour

Laboratoire ELIAUS,
Université de Perpignan

1. Puissance de calcul

Plus on pédale moins fort, moins on avance plus vite



2. Précision

Ça casse pas des briques à un canard

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Conclusion : Généralisation de la DP, IEEE 754-2008

Source : http://www.canardpc.com/news-39869-la_nouvelle_puce_nvidia_annoncee_ou_la_revolution_des_fermi.html

3. Modèle hiérarchique

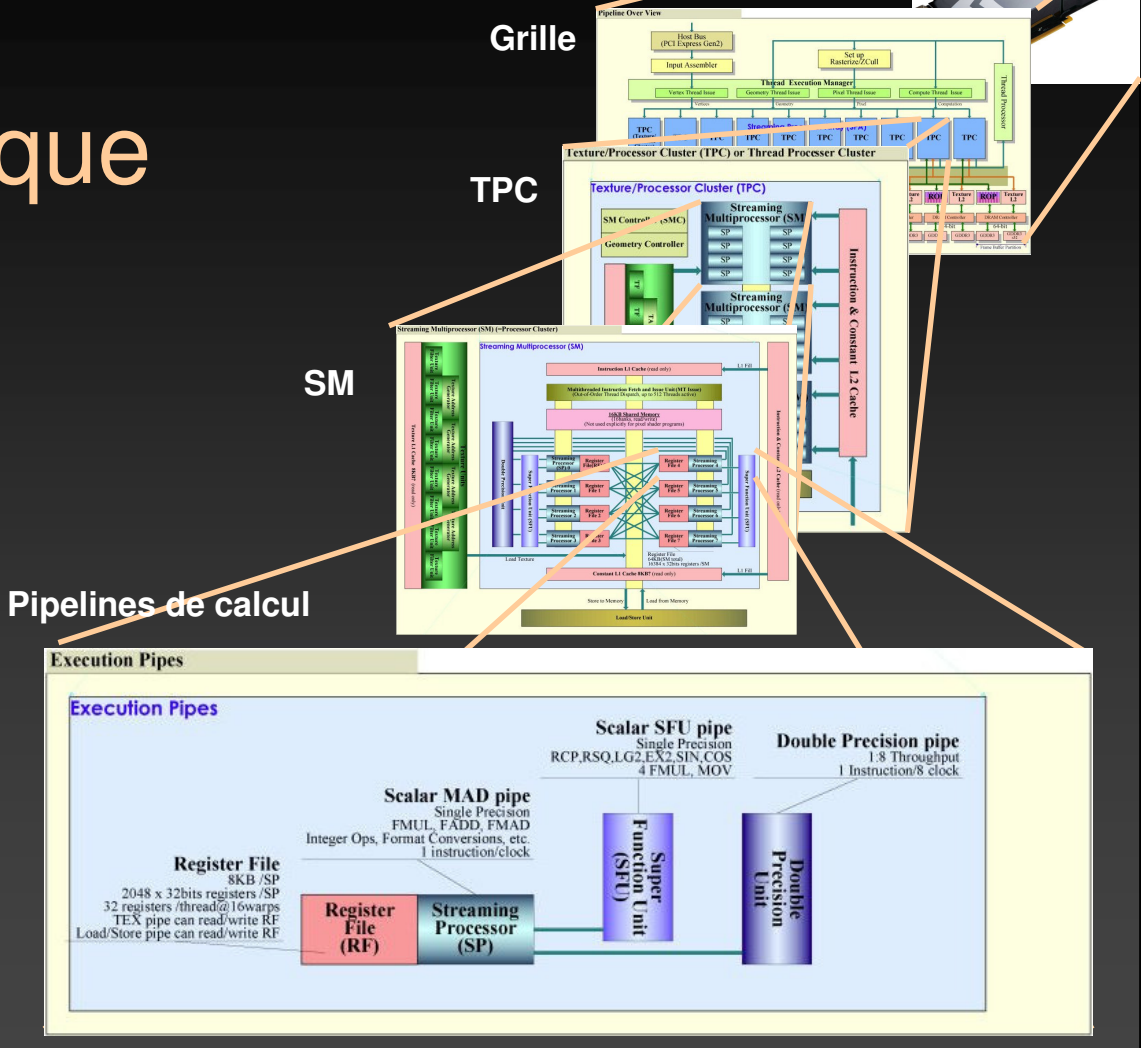
GPU

Grille

TPC

SM

Pipelines de calcul



Conclusion : Loi de Moore appliqué au GPU / Multicoeur =
de cœur x 2 tous les 18 mois

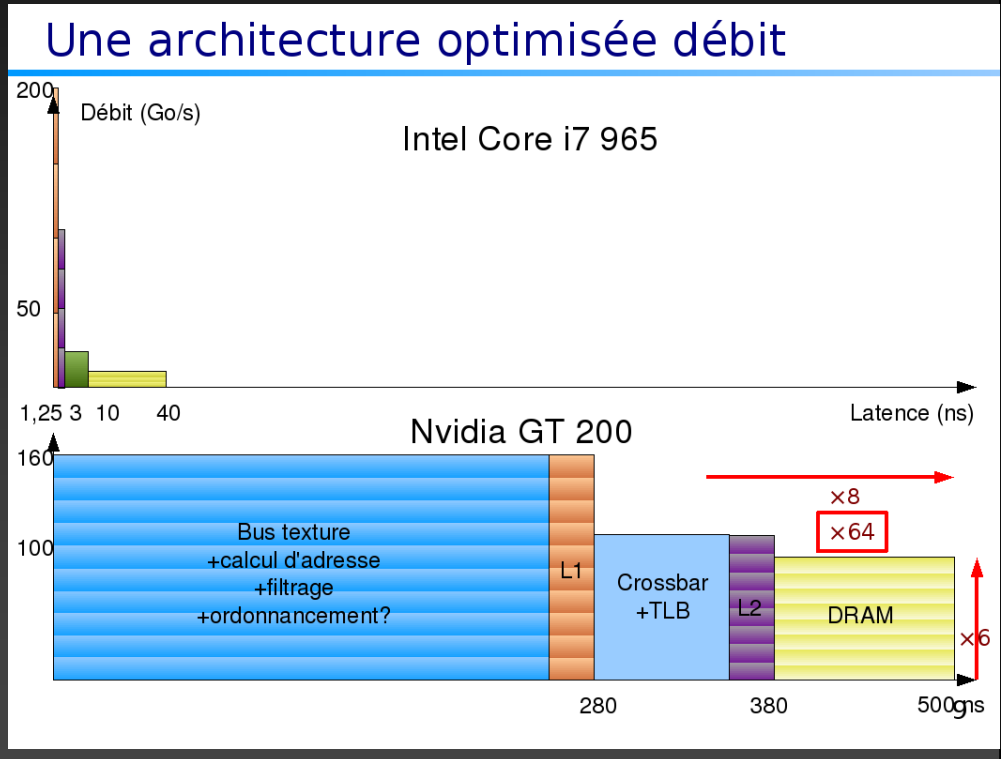
Source : http://pc.watch.impress.co.jp/docs/2008/0702/

4. Unités d'exécution

- Hier
  - Pipeline**S** graphique**S** hyper spécialisés
- Aujourd'hui
  - Pipeline**S** unifié**S** avec encore quelques unités spécialisées (MAD, FMA, texturage, ROP, filtrage, …)
- Demain
  - Quel ratio entre chaque type de pipeline ?

5. Bande passante

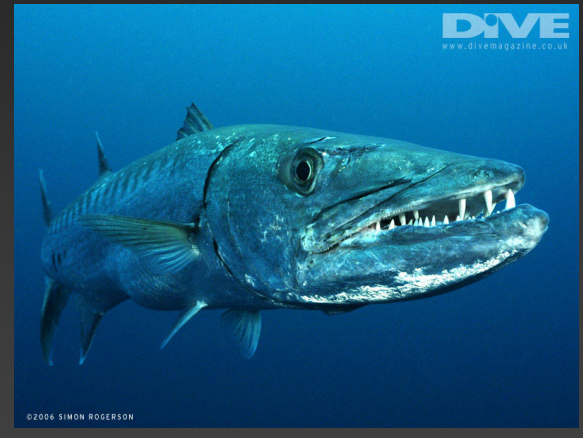
- Data //
  - Peu de communication
  - Accès direct à la mémoire
- Matériel
  - Focalisé sur le débit
  - Hiérarchie mémoire ‘inversée’ + scratchpad
- Demain
  - Dépend du marché cible



Source : Sylvain Collange©

9. Debuggage

- Hier :
  - Pas possible
- Aujourd'hui :
  - Decuda, GPUsim, Barra, compteurs
- Demain :
  - Intégration dans gdb



10. Consommation

- Hier :
  - Pas de problème
- Aujourd'hui :
  - GPU(GTX280): 4 SP GFLOP / Watt
  - CPU(core i7 960): 0.8 SP GFLOP / Watt
- Demain :
  - Power gating,…
  - Green HPC (*Green500*)

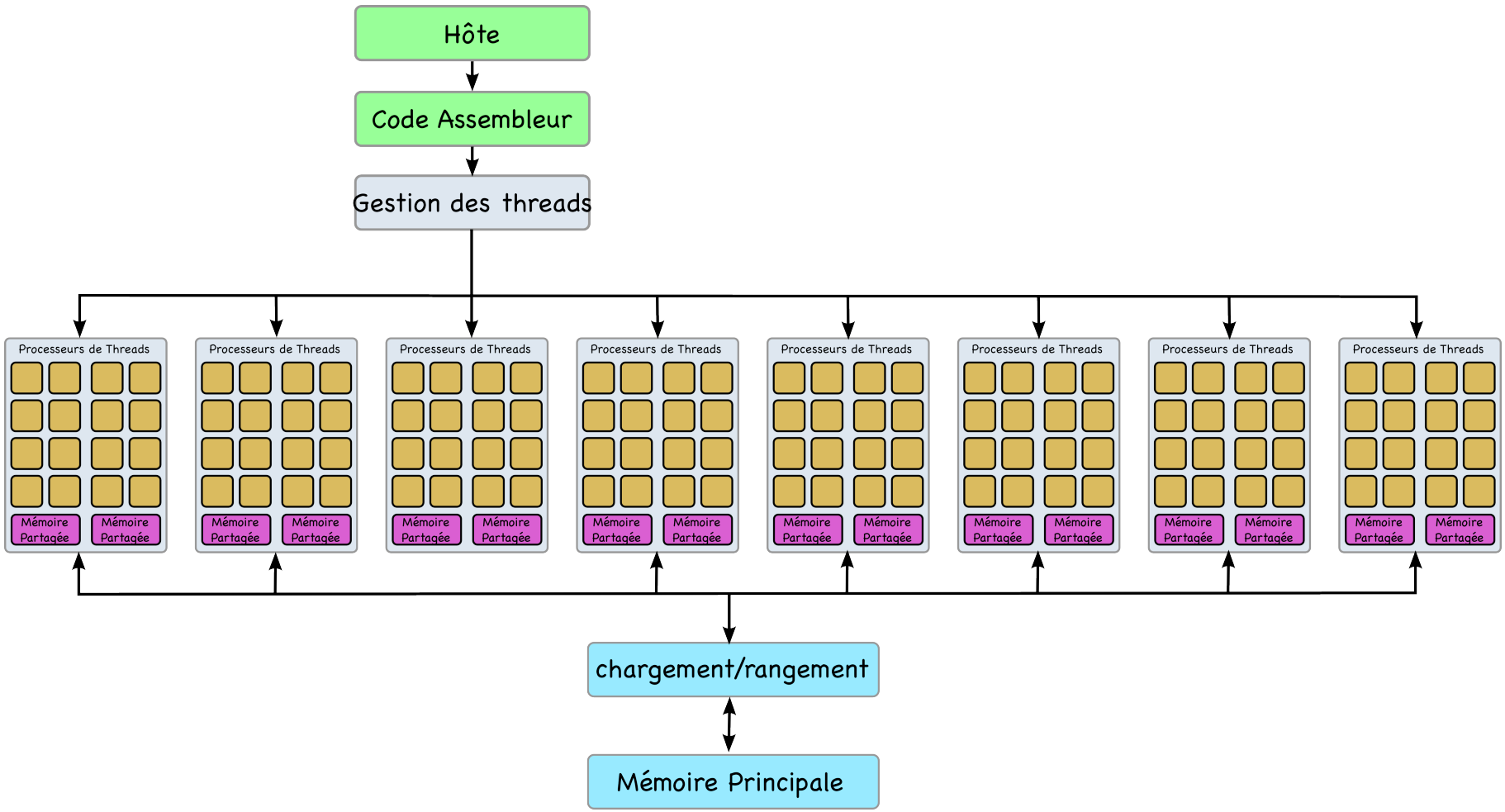
"Power Consuption of GPUs from a Software Perspective", S. Collange, D. Defour, and A. Tisserand, in *ICCS 2009*, vol. 5544 of *LNCS*, pp. 922-931, Springer, 2009.

Consommation : comparaison

Family Chip	Xeon 5160	Cell PS3	NVIDIA 9600 GT	AMD AMD 9270	Virtex5 LX330T
Techno	65 nm	65 nm	65 nm	55 nm	65 nm
Clock	3 GHz	3.2 GHz	1.625 GHz	750 MHz	200 MHz
DP GFlops	12	10.5	-	240	11.4
SP GFlops	24	204.8	312	1200	33
Power (W)	80	135	59–96	160–220	20–30

« Performance Comparison of Single-Precision SPICE Model-Evaluation on FPGA, GPU, Cell, and multi-core Processors », Nachiket Kapre, Andre DeHon, in FPL'2009

G80

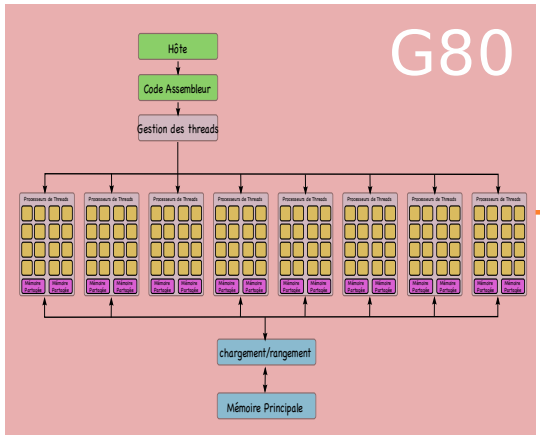

Hôte
Code Assembleur
Gestion des threads
Processeurs de Threads
Mémoire Partagée
chargement/rangement
Mémoire Principale

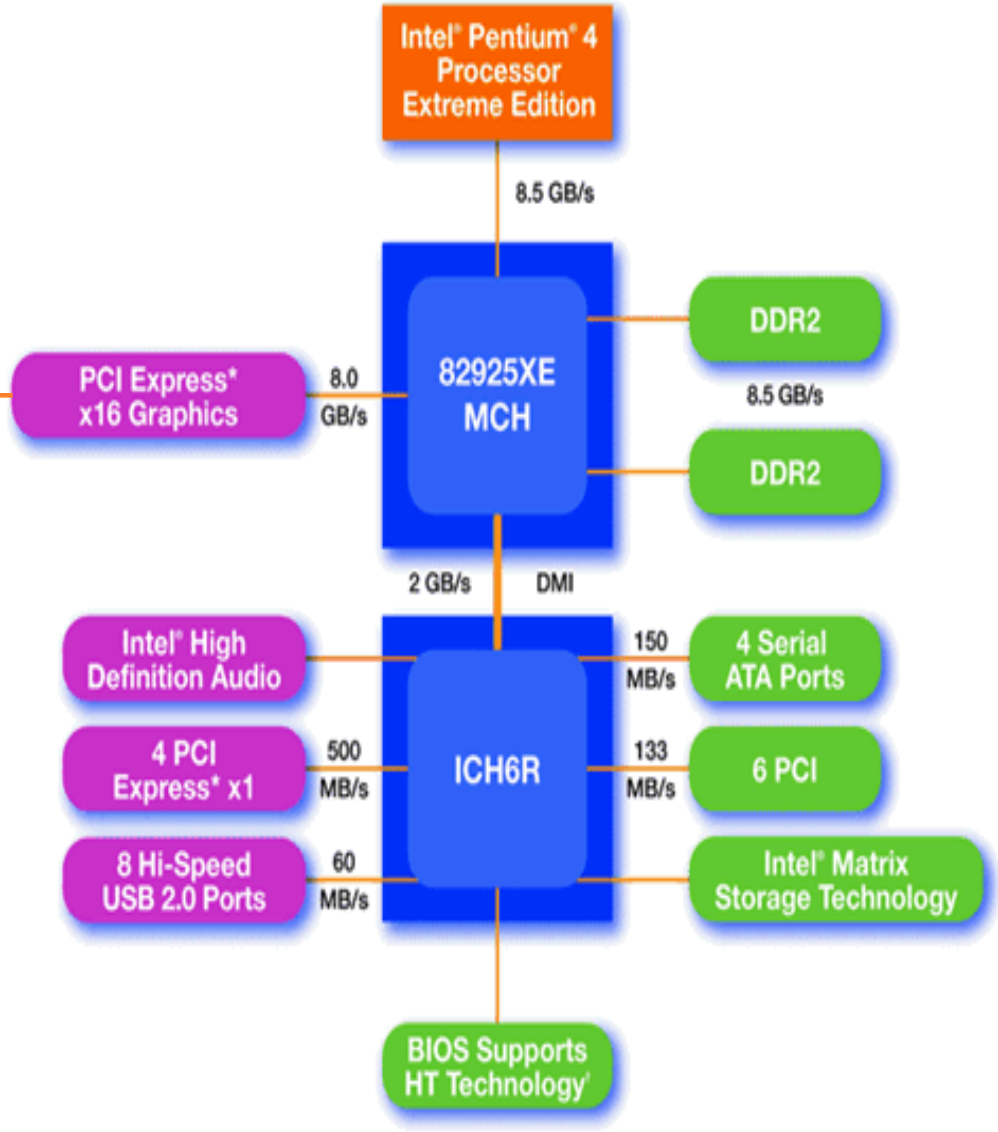
CUDA

”Compute Unified Device Architecture”

- Modèle type langage de haut niveau général
- Lancement en tâches de fond de threads GPU
- GPU = co-processeur massivement parallèle exécutant des threads
- Driver pour le chargement des programmes sur le GPU
- Driver autonome optimisé pour le calcul
- Données partagées avec les buffer objects OpenGL
- Des garanties de débit maximum de téléchargement
- Gestion explicite de la mémoire GPU

GPU : un co-processeur





Principes CUDA

Trois principes fondateurs de CUDA

- Une hiérarchie de groupes de Threads
- Des mémoires partagées
- Une barrière de synchronisation

Principes CUDA

- Mode de programmation SPMD
- C for CUDA étend le langage C en permettant au programmeur de définir de fonctions C, appelées Noyau (Kernel), qui quand elles sont appelées sont exécutées *N* fois par *N* différents threads.

Principes CUDA

Kernel

Une fonction noyau est définie en utilisant la déclaration __global__ et en indiquant le nombre de threads CUDA pour chaque appel. Pour cela on utilise une nouvelle syntaxe <<<...>>>.

Principes CUDA

- Un seul programme C contenant le code CPU et le code GPU
- Les parties séquentielles exécutées sur le CPU (nommé HOST)
- Les parties parallèles exécutées sur le GPU (nommé DEVICE) suivant un mode SPMD
- Code Série (host)
- Code Parallèle (device)
- KernelA<<< nBlk, nTid >>>(args)
- Code Série (host)
- Code Parallèle (device)
- KernelB<<< nBlk, nTid >>>(args)

Device

- C’est un coprocesseur du CPU ou hôte (host)
- Il a sa propre mémoire
- Exécuté plusieurs threads en parallèle
- C’est typiquement un GPU mais peut aussi bien être un autre type de ressources de calcul parallèle.

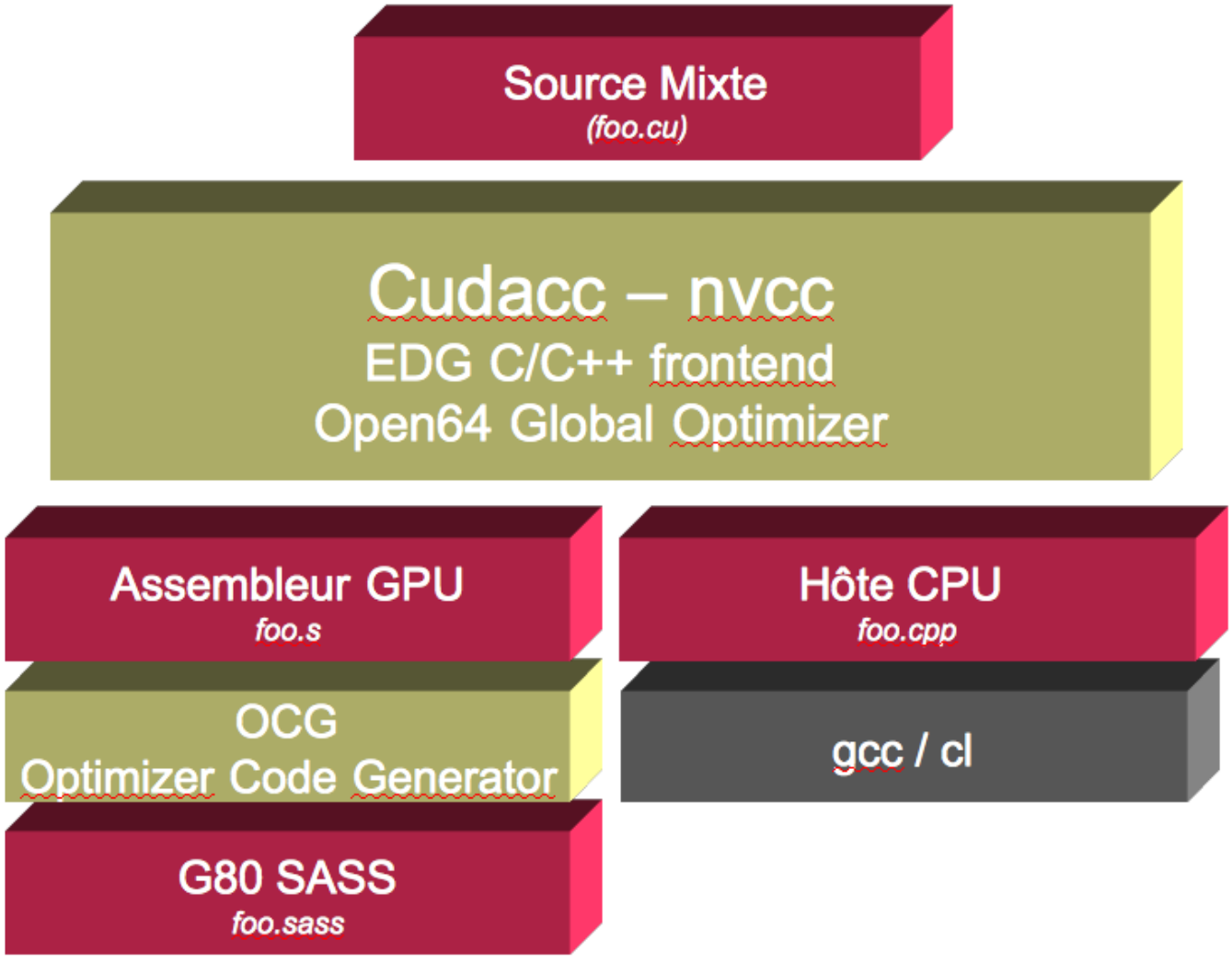
Device

- Les portions parallèle (parallélisme de données) d'une application sont exprimées sous forme de noyau device qui s'exécute sur plusieurs threads.
- Différences entre les threads GPU et CPU
  - les threads GPU sont extrémement léger
  - très peu d'overhead à leur création
  - un GPU a besoin de 1000 thread pour une efficacité maximale
  - le CPU Multi-coeur en on besoin seulement de quelque uns

CUDA - Extension du C

- Déclarations Spécifiques global, device, shared, local, constant
- Mots Clés threadIdx, blockIdx
- Intrinsics __syncthreads
- API Memory, symbol, execution management
- Démarrage fonction

Chaine de Compilation



Exécution d'un Noyau

- Un noyau CUDA est exécuté par un tableau de threads
- Tous les thread exécutent le même code (SPMD)
- Chaque thread a son propre identifiant qu'il utilise pour calculer les adresses mémoire et prendre des décisions de contrôle.

Exécution d'un Noyau

- Le tableau monolitique de Thread est divisé en plusieurs bloc (block)
- Les threads à l'intérieur d'un même bloc peuvent travailler ensemble via la mémoire partagée, des opérations atomique et la barrière de synchronisation.
- Les threads dans des blocs différents ne peuvent pas coopérer.

Exécution d'un Noyau

- Chaque thread utilise son identifiant pour décider des données surlesquelles travailler
  - il possède un identifiant de bloc BlockId : 1D ou 2D
  - il possède un identifiant de thread ThreadId : 1D, 2D ou 3D.
- Cela permet d'avoir un adressage aux données multidimensionnelles simplifié
  - En traitement d'image
  - en résolution d'équations volumiques
  - ...

Exécution d'un Noyau

- Le programeur déclare
  - La taille des blocs de 1 à 512 threads concurrents
  - La forme des blocs : 1D, 2D ou 3D
  - La taille des grids en nombre de blocs
  - La forme des grids : 1D ou 2D
- Chaque bloc peut s'exécuté dans n'importe quel ordre par rapport aux autres blocs.

Définition d'un Noyau

```
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i]=A[i]+B[i];
}
```

threadIdx

Chaque thread exécutant une fonction noyau possède un unique identifiant accessible via la variable threadIdx

Appel d'un Noyau

```
int main()
{
VecAdd<<<1, N>>>(device_A,device_B,device_C);
}
```

CUDA et Mémoire

Mémoire globale

- Mémoire principale pour communiquer des données entre l'hôte (CPU) et la device (GPU)
- Son contenu est visible par tous les threads
- Elle a une longue latence d'accès

CUDA et Mémoire

Gestion mémoire

- fonction d'allocation : `cudaMalloc()`
- Allocation d'objets dans la mémoire globale de la device
- Requiert deux paramètres
  - l'adresse d'un pointeur sur l'objet alloué
  - la taille de l'objet alloué
- fonction de désallocation : `cudaFree()`
  - Requiert l'objet en paramètre

Appel d'un Noyau

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

CUDA Transferts Mémoire

Transfert CPU<->GPU

- une fonction de copie : `cudaMemcpy()`
- Nécessaire pour les transfers mémoire entre hôte et device
- Requiert 4 paramètres
  - un pointeur sur la destination
  - un pointeur sur la source
  - Nombre d'octets copiés
  - le type de transfert
    - "Host to Host"
    - "Host to Device"
    - "Device to Host"
    - "Device to Device"
  - Transferts asynchrones

Transferts mémoire

```
TILE_WIDTH = 64;
Float* Md,M;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);
Malloc(M,size);
cudaMalloc((void**)&Md, size);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
cudaFree(Md);
Free(M);
```

CUDA declaration

	Exécuté sur	Appelable uniquement de
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__ défini une fonction noyau **doit retourner** `void`
- __device__ and __host__ peuvent être utilisé ensemble

CUDA declaration

- les fonctions __device__ ne peuvent avoir leur adresse utilisées
- Pour les fonctions exécutées sur le device
  - Pas de récursion
  - Pas de déclaration de variables statique dans la fonction
  - Pas de nombre variable d'arguments

CUDA declaration

Une fonction noyau doit être appelée avec une configuration d'exécution

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 blocs
dim3   DimBlock(4, 8, 8);   // 256 threads par bloc
size_t SharedMemBytes = 64; // 64 octets de mémoire partagée
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- N'importe quel appel à une fonction noyau est asynchrone.
- Il est nécessaire d'avoir une synchronisation explicite

Compilation

- N'importe quelle source contenant des contructions CUDA doit être compilée avec NVCC
- NVCC est un compilateur de pilote
- Il travaille en invoquant tous les outils et compilateurs nécessaires, comme cudacc, g++, cl
- NVCC produit
  - du code C (code CPU)
  - Il doit être compilé avec le reste de l'application en utilisant un autre outil : PTX
    - Code objet directement
    - PTX source, interpreté à l'exécution

Compilation

- N'importe quel exécutable CUDA nécessite deux bibliothèques
  - CUDA runtime library (cudart)
  - CUDA core library (cuda)

Debug

- En utilisant l'option -deviceemu du compilateur nvcc (nvcc -deviceemu), le mode émulation est choisi
- Dans ce mode l'application s'exécute entièrement sur l'hôte
- Pas de besoin de carte de du CUDA driveer
- Chaque thread GPU est émulé par un thread CPU
- Exécuter un code en mode émulation, peut
  - permettre l'utilisation d'un support de debug natif
  - permettre l'accès aux données spécifique device à partir du code hôte et vice-versa
  - Appeler n'importe quelle fonction hôte du device, par exemple `printf`
  - Détecter des situation d'interblocage du à une mauvaise utilisation de __syncthreads

API

- L'API est une extension du langage C, elle consiste en
  - des extensions pour détecter les portions de code exécutées sur le device
  - Une librairie dynamique scindée en
    - un composant commun fournissant les types vectoriels et des fonctions dynamiques pour l'hôte et a device
    - un composant hôte pour contrôler et accèder à une ou plusieurs devices
    - un composant device fournissant les fonctions spécifiques device

type dimension

- dim3 gridDim;
  - Dimensions du grid en nombre de blocs (gridDim.z non utilisé)
- dim3 blockDim;
  - Dimensions du bloc en nombre de threads
- dim3 blockIdx;
  - Index du Bloc à l'intérieur du Grid
- dim3 threadIdx;
  - Index du thread à l'intérieur du bloc

API mathématique

- les fonctions mathématiques standards sont accessibles
  - pow, sqrt, cbrt, hypot
  - exp, exp2, expm1
  - log, log2, log10, log1p
  - sin, cos, tan, asin, acos, atan, atan2
  - sinh, cosh, tanh, asinh, acosh, atanh
  - ceil, floor, trunc, round
  - Etc.
- quand ces fonctions sont exécutées sur l'hôte elles utilisent la mise en oeuvre standart si elle est disponible
- ces fonction sont définies uniquement pour des scalaires pas des vecteurs

API

- Quelques fonctions mathématiques ont une version plus rapide bien que moins précise
  - __pow
  - __log, __log2, __log10
  - __exp
  - __sin, __cos, __tan

Barrière de Synchronisation

Synchronisation Exécution

- `void __syncthreads();`
- synchronise tous les threads d'un bloc
- Une fois que tous les threads arrivent à ce point l'exécution continue sinon elle attend
- Utile pour éviter les aléas d'accès aux données en mémoire partagée et globale
- Autorisé dans des parties conditionnelle, uniquement si cette partie conditionnel est uniforme dans le bloc

Instructions mémoires

- Débit mémoire : 8 opérations par cycle
- Accès à la mémoire locale ou globale : de 400 à 600 cycles de latence
- Accès masquables par l'ordonnanceur de Threads CUDA si il y a suffisamment d'instruction arithmétiques indépendantes pour cela

Mémoire globale

Taille accès

- Les accès ne sont pas cacheables
- Lecture par mots de 32, 64 ou 128 bits
- Accès alignés sur la taille du transfert : 4, 8 ou 16 octets
- Type prédéfinis pour alignement `float2, float4`
- Possibilité d'aligner des stuctures `__align__(8), __align__(16)`

Alignement Accès Mémoire Structure

- il existe un mot clé pour garantir un alignement de données dans les stuctures, c'est le mot `__align__`.

1 accès mémoire

l'accès aux structures suivantes ne génère qu'un accès mémoire au lieu de respectivement 2 et 3.

```
struct __align__(8) {
        float a;
        float b;
};
```

ou

```
struct __align__(16) {
        float a;
        float b;
        float c;
};
```

2 accès mémoire

l'accès à la structure suivante ne génère que deux accès mémoire au lieu de 5.

```
struct __align__(16) {
        float a;
        float b;
        float c;
        float d;
        float e;
};
```

Mémoire globale

Accès simultanés

- `Warp` ensemble de 32 Threads CUDA
- Accès mémoire par `demi-Warp`
- Accès mémoire possiblement coalescés en une simple transaction 32, 64 ou 128 octets

Mémoire globale

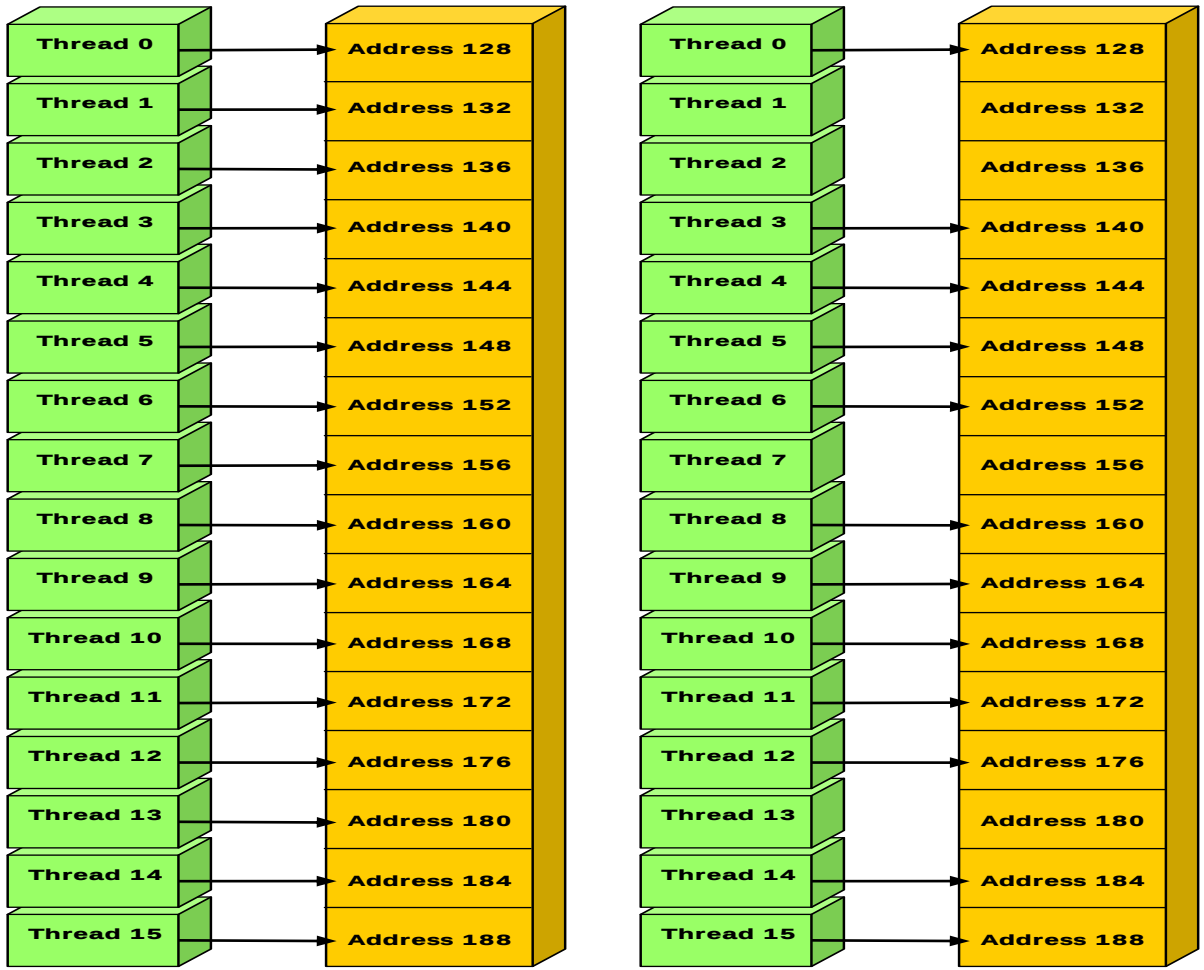
Coalescence sur une Device 1.0 ou 1.1

- Les Threads CUDA doivent accéder
  - à des mots 32 bits pour une transaction 64 octets
  - à des mots 64 bits pour une transaction 128 octets
  - à des mots 128 bits pour deux transactions 128 octets
- Les données doivent être dans le même segment mémoire de 16 mots
- Les Threads CUDA doivent accèder aux données en séquence.
- Si le `demi-Warp` ne respecte pas toutes ces restrictions des transactions séparées sont générées.

Coalescence sur une Device 1.2

- Plus souple basée sur l'algo suivant
- Trouver le segment mémoire qui contient les adresses requises par le Thread CUDA d'identifiant le plus petit. Le segment est de taille 32 octets pour les données 8bits, 64 octets pour les données 16 bits, 128 octets pour les données 32, 64 et 128 bits
- Trouver tous les autres Threads CUDA actifs qui accèdent au même segment
- Réduire la transaction si possible
  - Si la transaction est de 128 octets et que seulement la partie haute ou basse du segment est utilisée alors réduire à 64 octets la transaction.
  - Si la transaction est de 64 octets et que seulement la partie haute ou basse du segment est utilisée alors réduire à 32 octets la transaction.
- Traiter la requête et marquer comme inactif les Threads CUDA servis
- répéter tant qu'il y a des Threads CUDA non servis

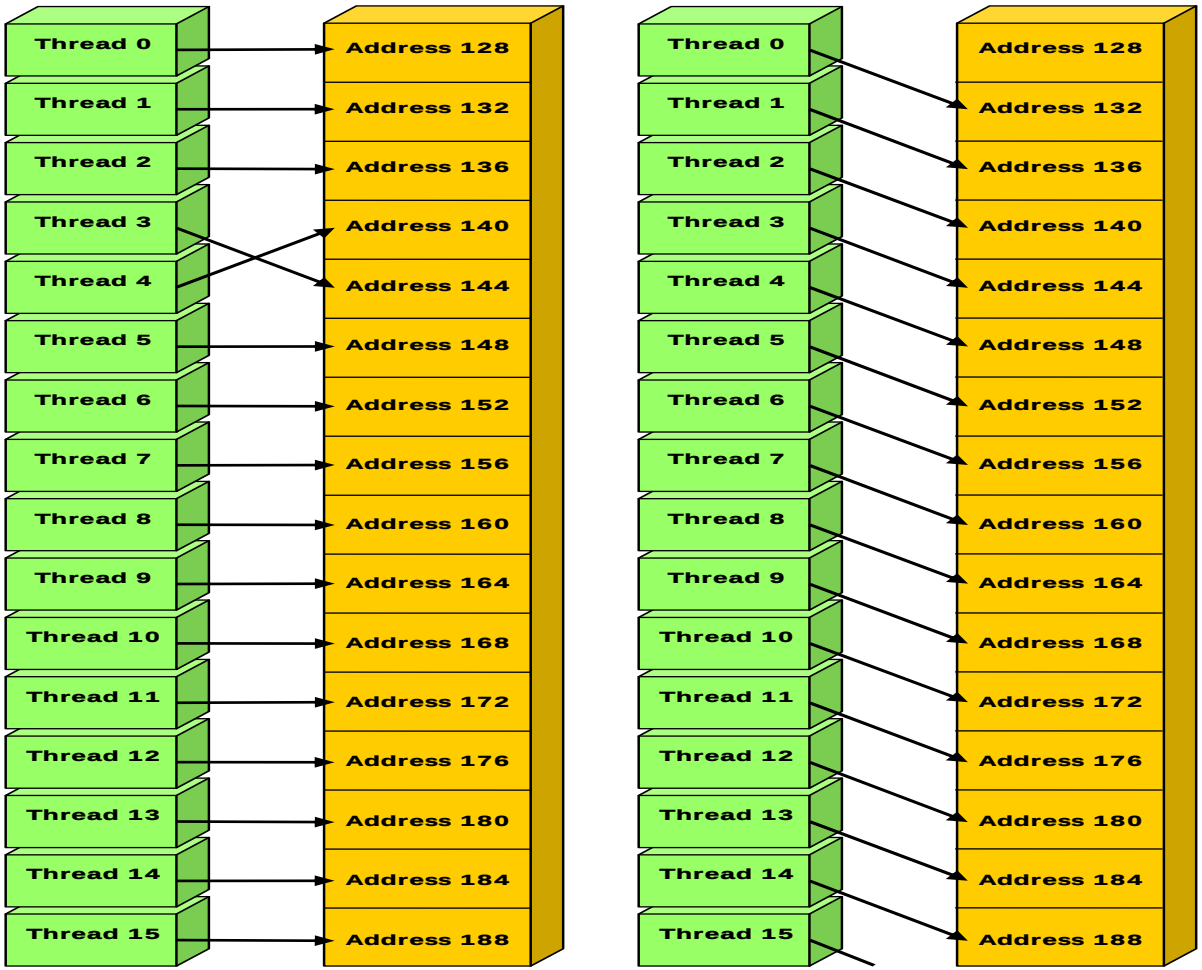
Accès Mémoire Globale



Left: coalesced `float` memory access, resulting in a single memory transaction.
Right: coalesced `float` memory access (divergent warp), resulting in a single memory transaction.

Figure 5-1. Examples of Coalesced Global Memory Access Patterns

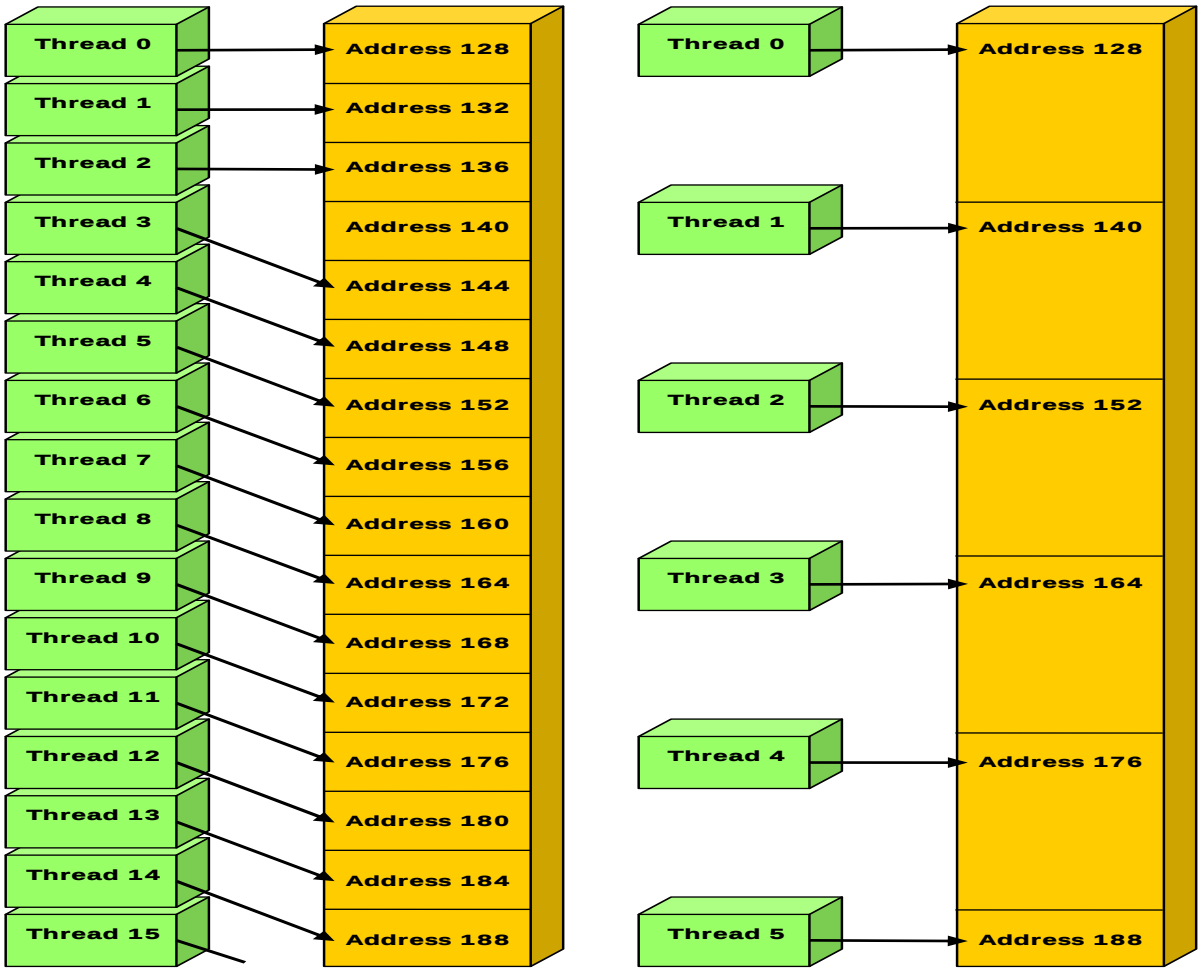
Accès Mémoire Globale



Left: non-sequential `float` memory access, resulting in 16 memory transactions.
Right: access with a misaligned starting address, resulting in 16 memory transactions.

Figure 5-2. Examples of Global Memory Access Patterns That Are Non-Coalesced for Devices of Compute Capability 1.0 or 1.1

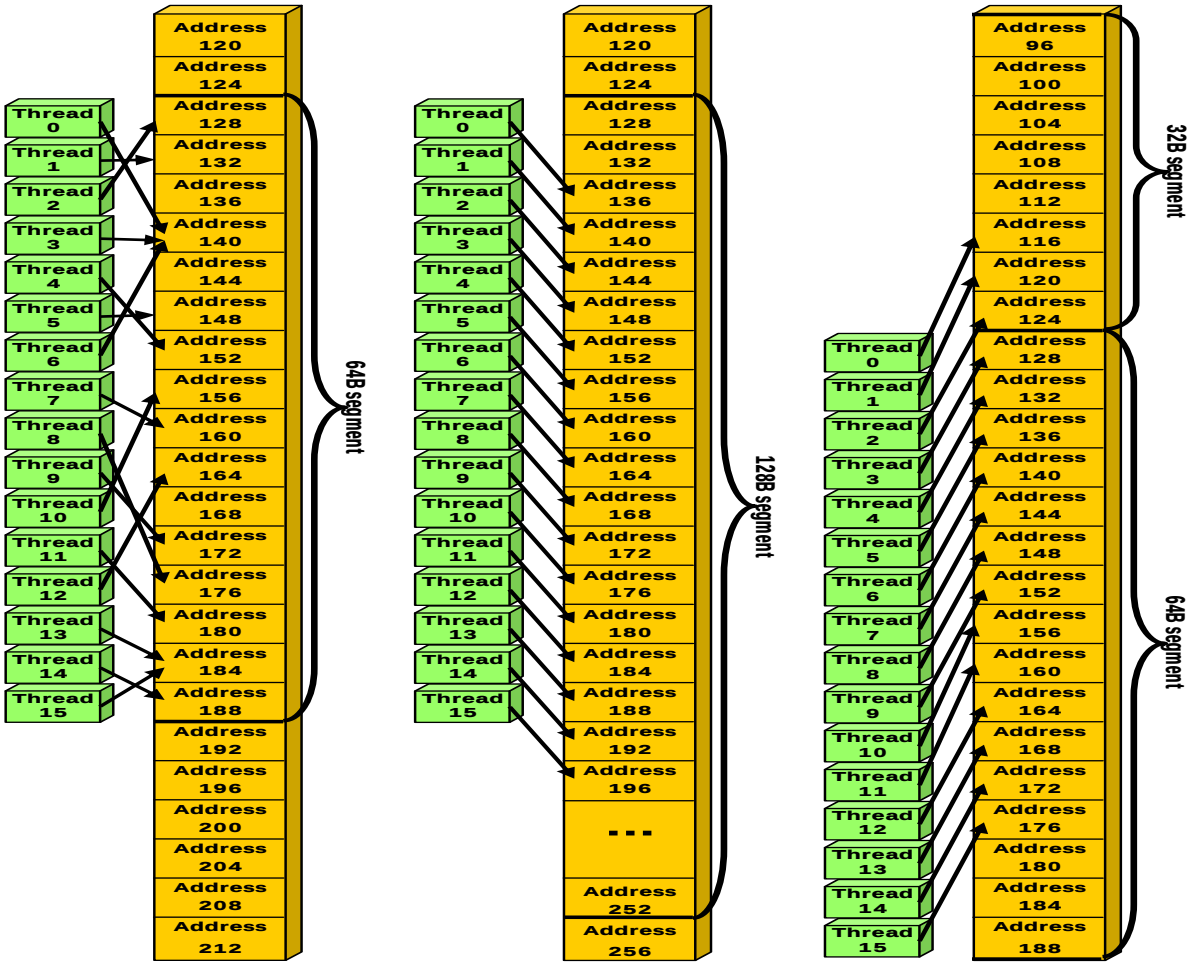
Accès Mémoire Globale



Left: non-contiguous `float` memory access, resulting in 16 memory transactions.
Right: non-coalesced `float3` memory access, resulting in 16 memory transactions.

Figure 5-3. Examples of Global Memory Access Patterns That Are Non-Coalesced for Devices of Compute Capability 1.0 or 1.1

Accès Mémoire Globale



Left: random **float** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **float** memory access, resulting in one transaction.
Right: misaligned **float** memory access, resulting in two transactions.

Figure 5-4. Examples of Global Memory Access by Devices with Compute Capability 1.2 and Higher

Mémoire de constantes

- Mémoire cacheable
- Pour le `demi-Warp` accès aussi rapide que l'accès aux registres

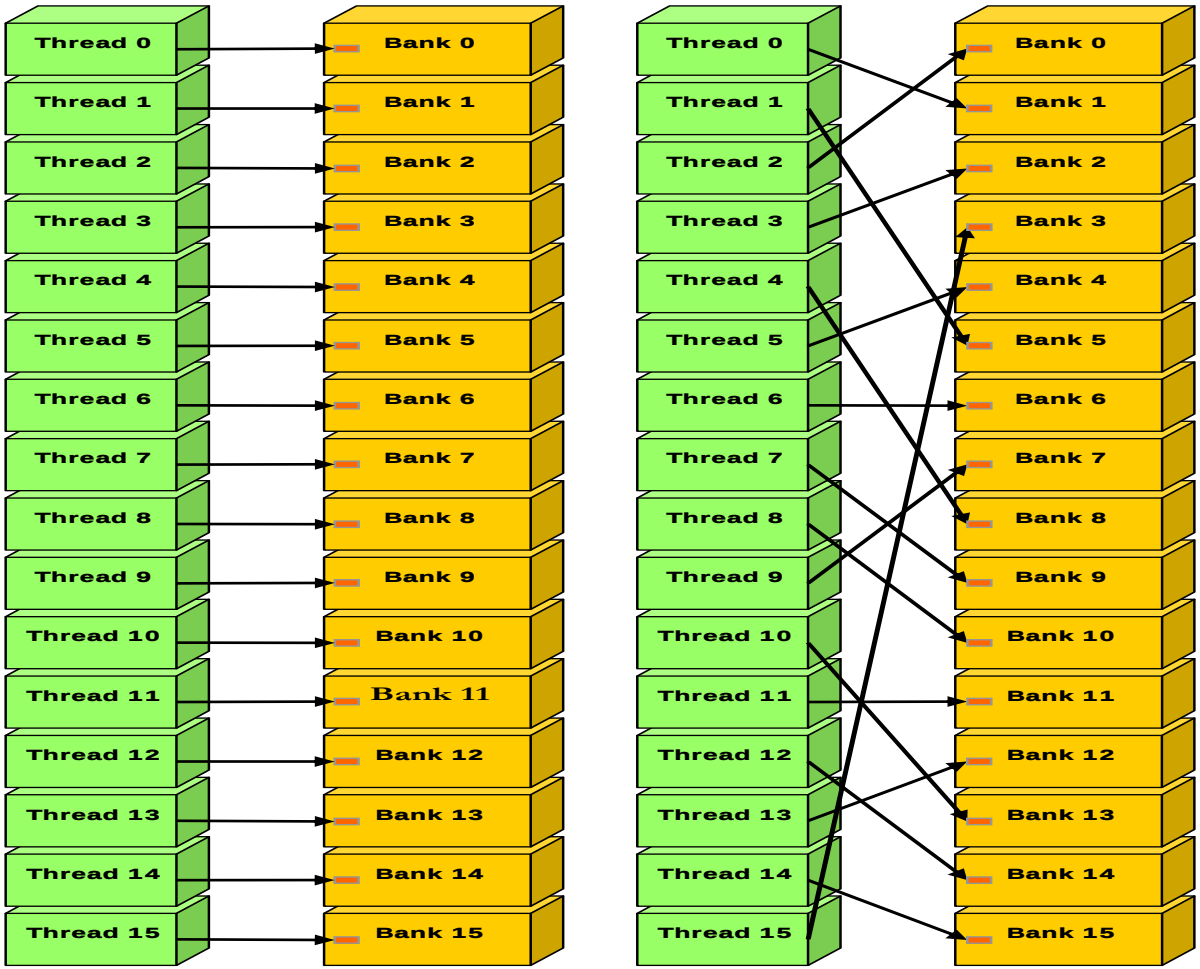
Mémoire de textures

- Mémoire à accès en lecture seule

Mémoire partagée

- Plus rapide que la mémoire locale ou globale
- Divisée en bancs de taille identique
- Possibilité de conflit si accès au même banc
- Accès simultané si accès à des bancs différents
- Organisé de telle façon que l'accès à des mots successifs de 32 bits sont effectués dans différents bancs
- Accès 32 bits en 2 cycles d'horloge
- Device 1.x , taille des `Warp` égale 32 et nombre de bancs égal 16
- Pas de conflit si les deux Threads CUDA sont dans deux différentes moitiés du `Warp`

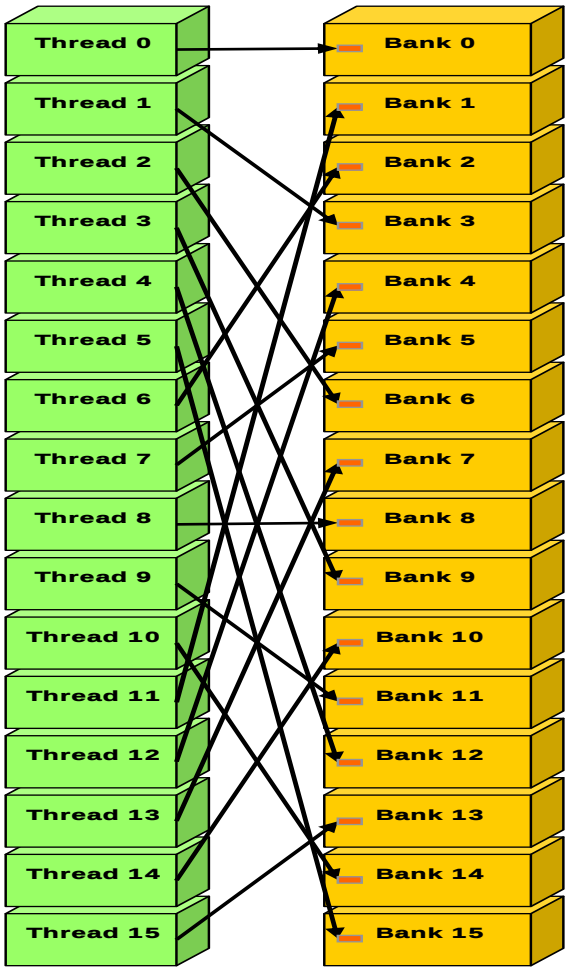
Accès Mémoire Partagée



Left: linear addressing with a stride of one 32-bit word.
Right: random permutation.

Figure 5-5. Examples of Shared Memory Access Patterns without Bank Conflicts

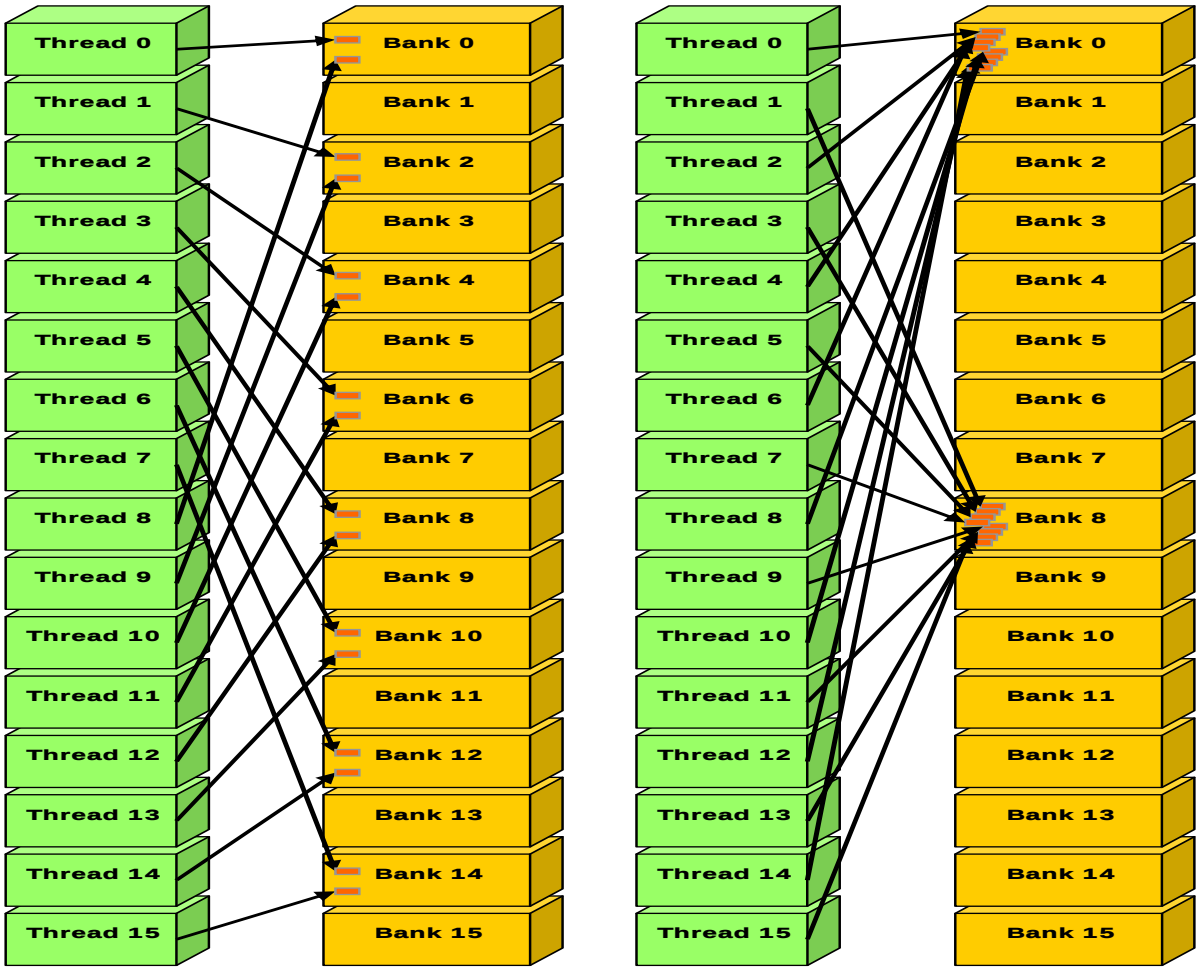
Accès Mémoire Partagée



Linear addressing with a stride of three 32-bit words.

Figure 5-6. Example of a Shared Memory Access Pattern without Bank Conflicts

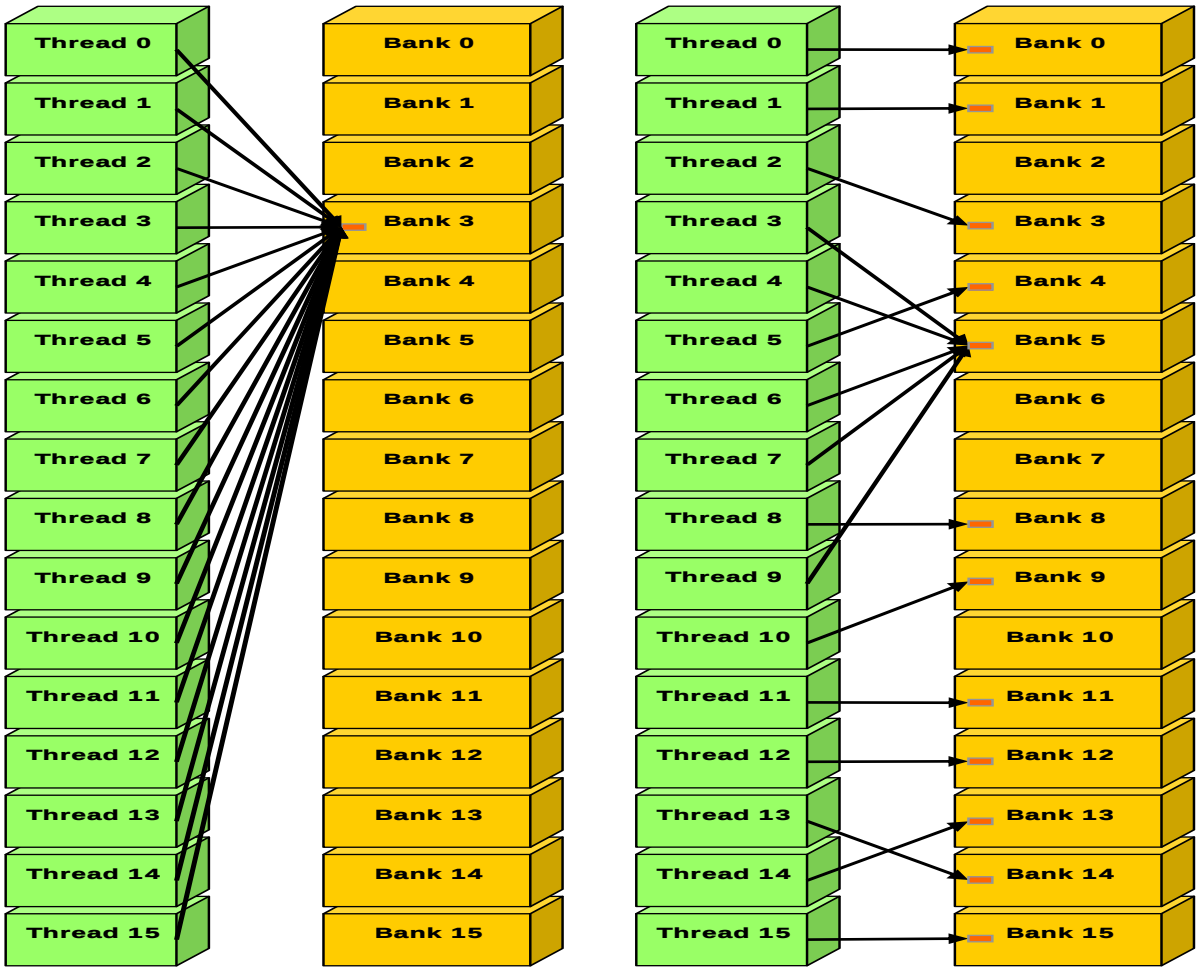
Accès Mémoire Partagée



Left: Linear addressing with a stride of two 32-bit words causes 2-way bank conflicts.
Right: Linear addressing with a stride of eight 32-bit words causes 8-way bank conflicts.

Figure 5-7. Examples of Shared Memory Access Patterns with Bank Conflicts

Accès Mémoire Partagée



Left: This access pattern is conflict-free since all threads read from an address within the same 32-bit word.
Right: This access pattern causes either no bank conflicts if the word from bank 5 is chosen as the broadcast word during the first step or 2-way bank conflicts, otherwise.

Figure 5-8. Example of Shared Memory Read Access Patterns with Broadcast

Barrière de Synchronisation

Synchronisation Accès Mémoire Globale

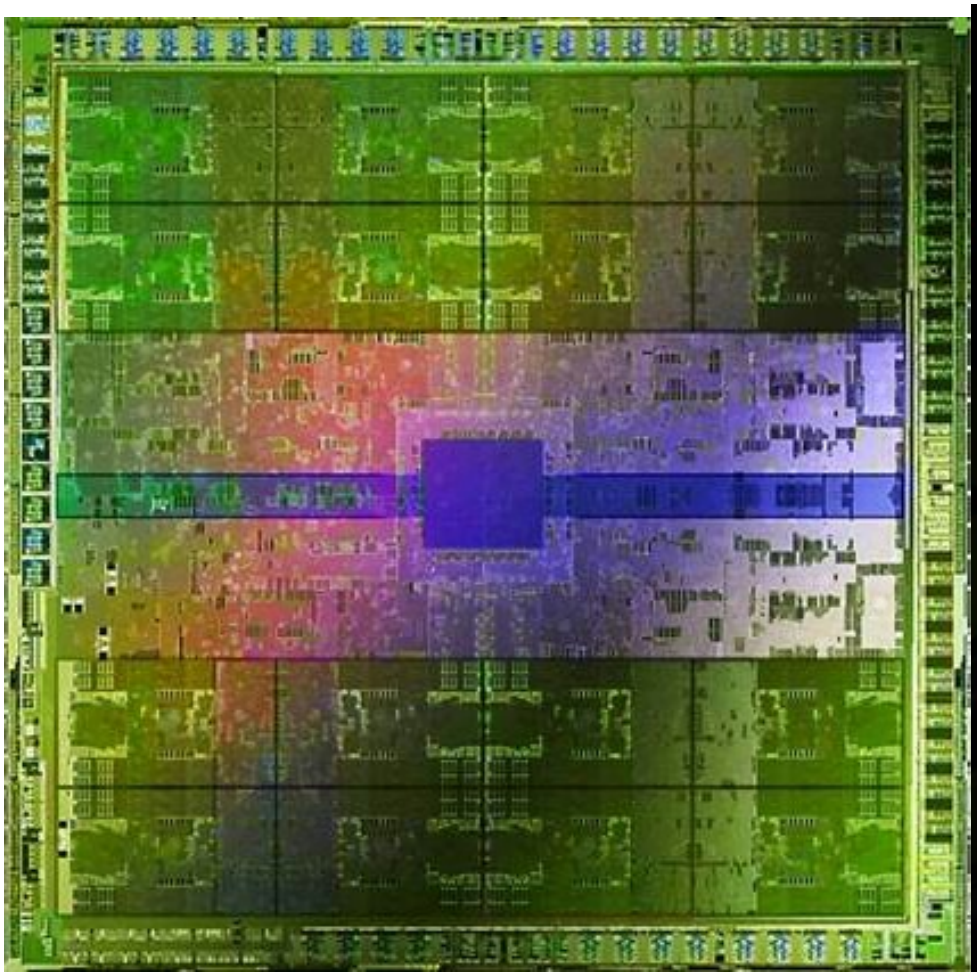
- `void __threadfence();`
- synchronise tous les accès mémoires des threads à la mémoire globale
- Une fois que tous les accès mémoire à la mémoire globale en amont de cette instruction sont terminés l'exécution continue sinon elle attend

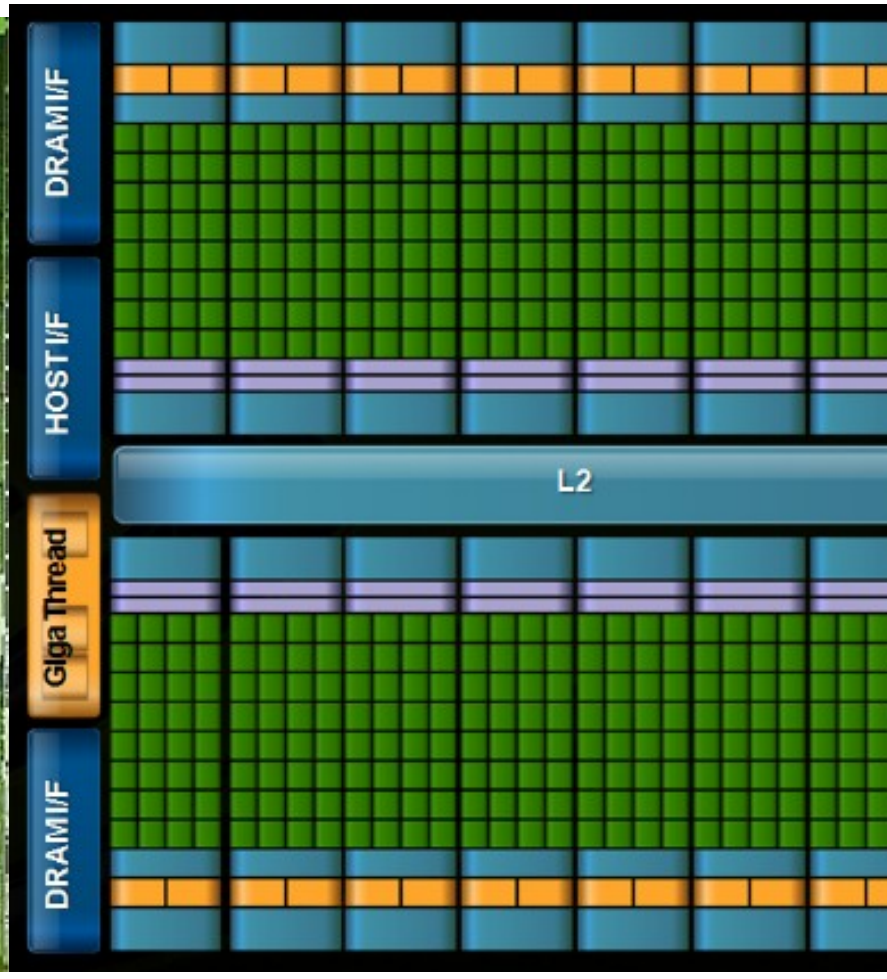
Barrière de Synchronisation

Synchronisation Accès Mémoire Partagée

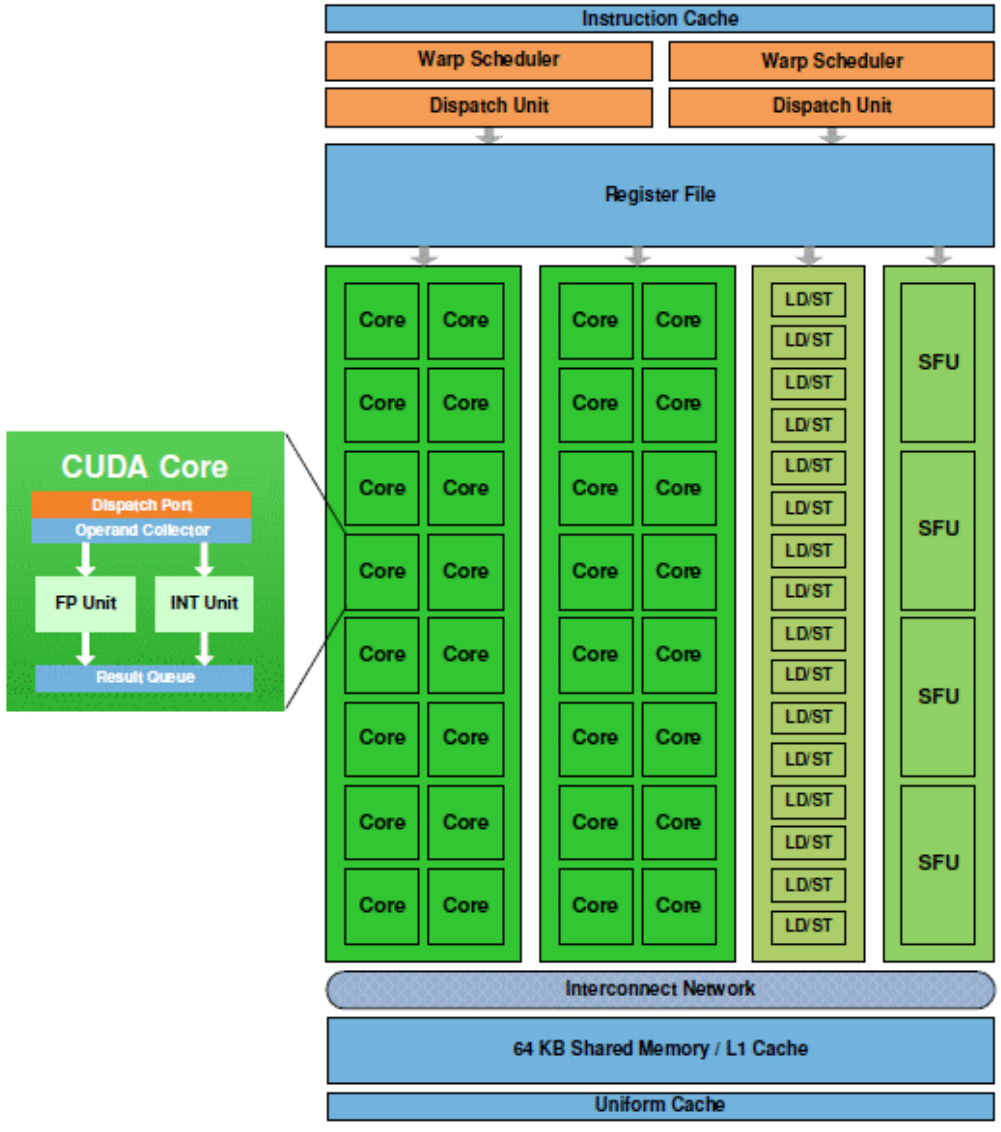
- `void __threadfence_block();`
- synchronise tous les accès mémoires des threads à la mémoire partagée
- Une fois que tous les accès mémoire à la mémoire partagée en amont de cette instruction sont terminés l'exécution continue sinon elle attend

Die Chip



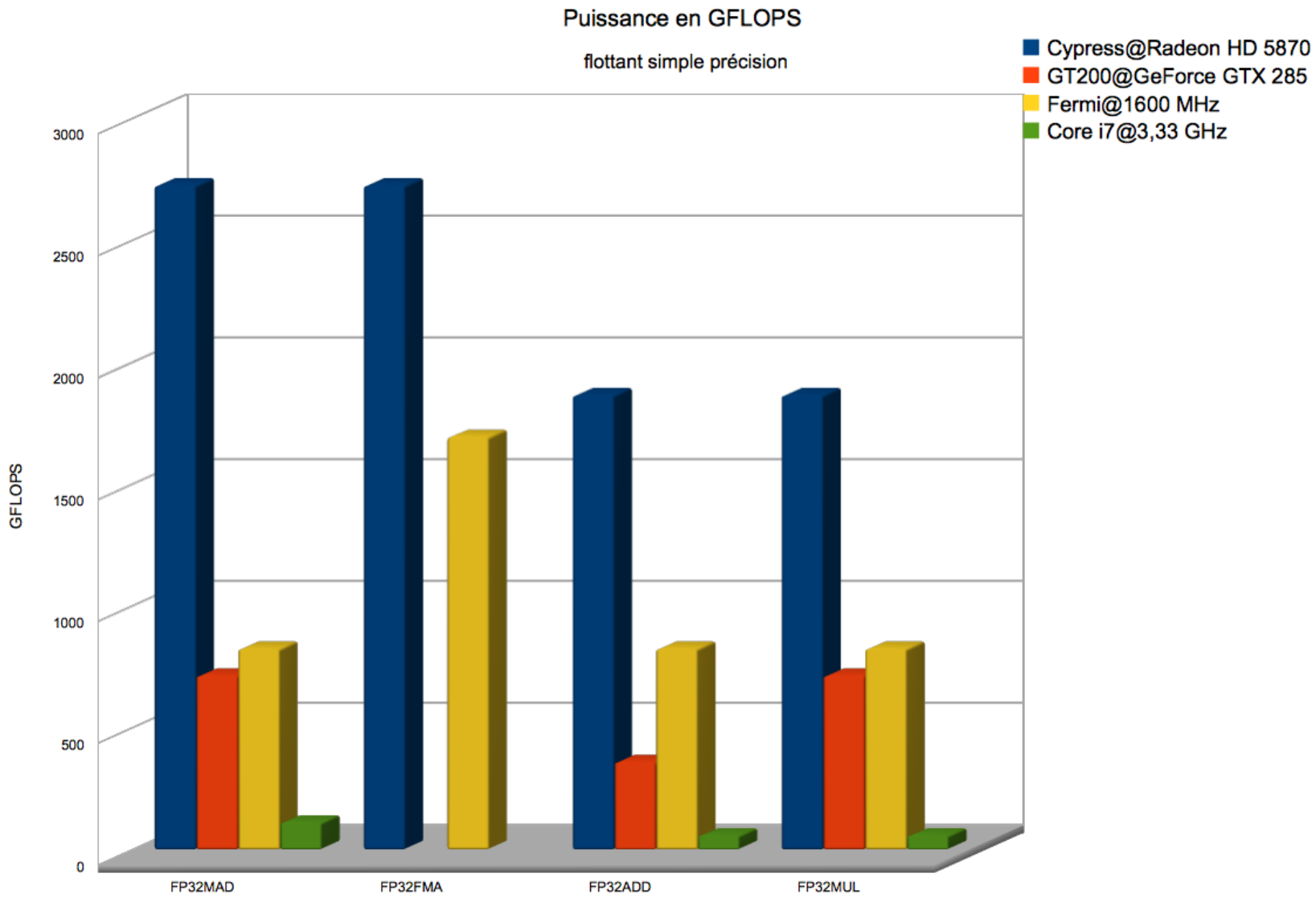


Coeur Fermi

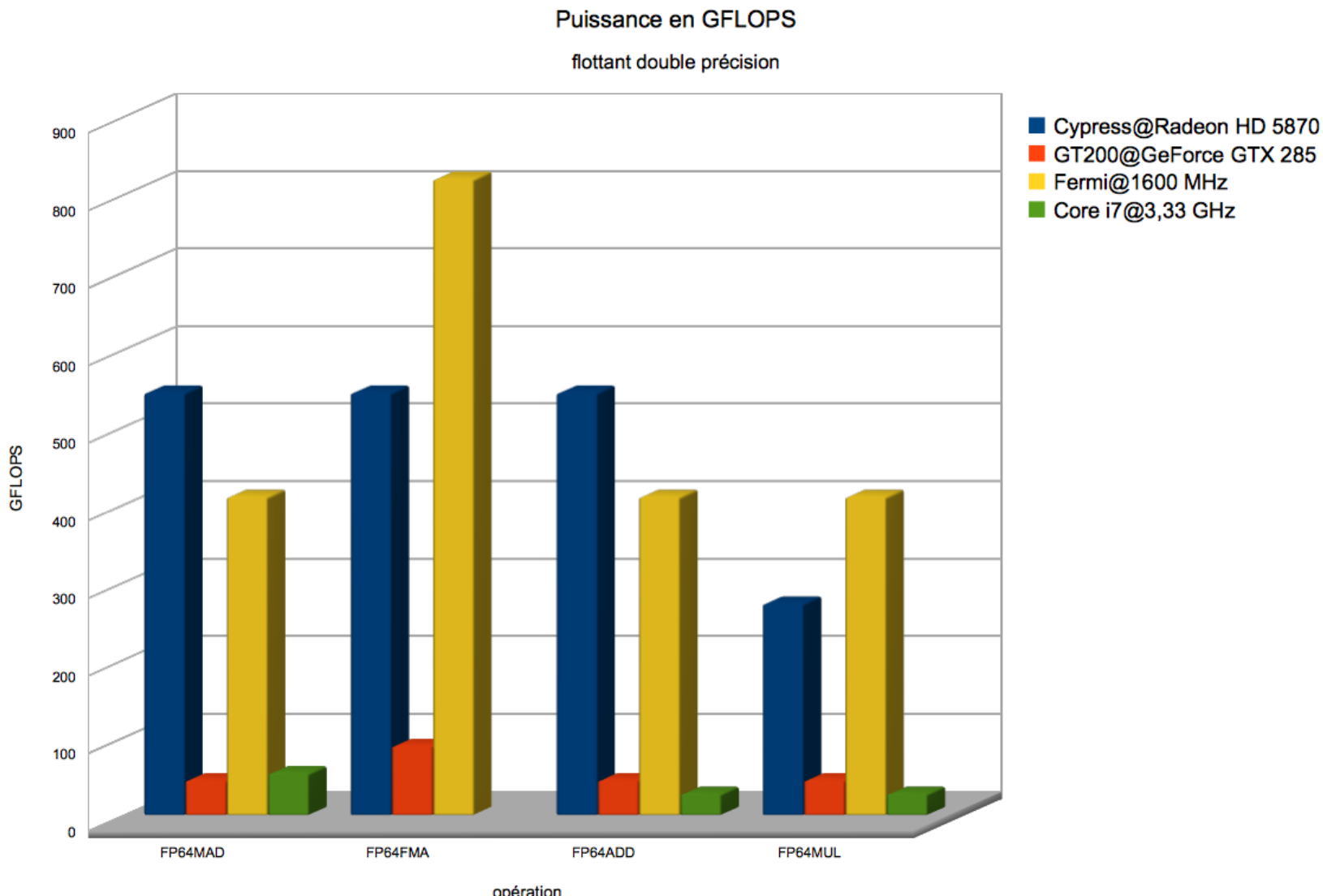


Fermi Streaming Multiprocessor (SM)

Performances Flottantes Fermi



Performances Flottantes Fermi



Performances Entières Fermi