Architectures parallèles

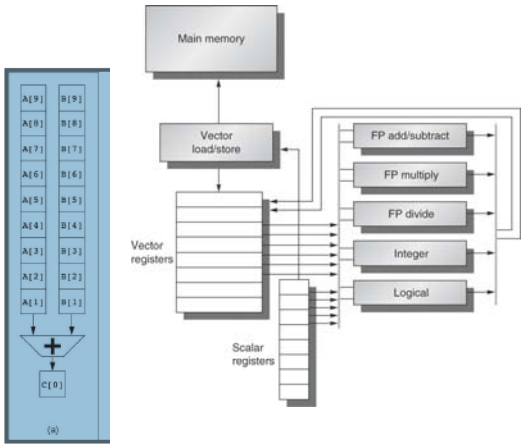
Daniel Etiemble
de@lri.fr

Classification de Flynn

- Flot d'instructions x Flot de données
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (SIMD)
  - Multiple Instruction Single Data
  - Multiple Instruction Multiple Data (MIMD)
- "Tout est MIMD" !

Architectures vectorielles

- Registres vectoriels
- Registres scalaires
- Chargement/rangement vectoriel sans caches. Permet les opérations de scatter-gather
- Les instructions vectorielles s'exécutent élément par élément dans les unités de calcul



Architectures vectorielles (2)

*Y = a * X + Y*

```
        LD      F0, a
        ADDI    R4, Rx, #512
Loop:   LD      F2, 0(Rx)
        MULTD   F2, F0, F2
        LD      F4, 0 (Ry)
        ADDD    F4, F2, F4
        SD      F4, 0 (Ry)
        ADDI    Rx, Rx, #8
        ADDI    Ry, Ry, #8
        SUB     R20, R4, Rx
        BNZ     R20, loop
```

Code scalaire (64 itérations)

*Y = a * X + Y*

```
LD      F0, a
LV      V1, Rx
MULTSV  V2, F0, V1
LV      V3, Ry
ADDV    V4, V2, V3
SV      Ry, V4
```

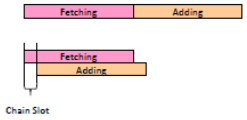
Code vectoriel (registres vectoriels de 64 éléments

Architectures vectorielles (3)

- Découpage des vecteurs

  Si N > taille des registres vectoriels, on itère
- Instructions conditionnelles
  - Instructions vectorielles de comparaison
    - Positionnent un vecteur de booléens
  - Les instructions vectorielles n'exécutent que les éléments pour lesquels le booléen est vrai
  - Les conditionnelles SI condition ALORS …. SINON exécutent la suite des instructions vectorielles avec le booléen Vrai pour ALORS puis la suite des instructions vectorielles avec le booléen Faux pour SINON
- Chaînage des opérateurs.

Mem → V0 (M-fetch)
V0 + V1 → V2 (V-add)
V2 < A3 → V3 (left shift)
V3 ^ V4 → V5 (logical product)

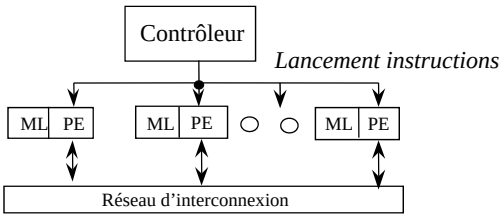


Architectures vectorielles (4)

- Heures de gloires
  - Machines vectorielles Cray…. (1975 : Cray-1)
- Disparition
  - Machines parallèles SIMD et MIMD
- Renaissance ?
  - L'extension MIC du jeu d'instructions Intel
  - 32 registres 512 bits (Zmm0 à Zmm31)
    - 16 int32, 8 int64, 16 float, 8 double
  - 8 registres de masque vectoriel k0 à k15 (16 bits)
    - 1 bit par élément du registre Zmm_i (16 ou 8 bits)
      - Contrôle des opérations « élément par élément »
      - Contrôle des écritures registre « élément par élément »
    - Exception
      - k0 correspond au « non masquage »
  - **DU SIMD au VECTORIEL**

Architectures SIMD

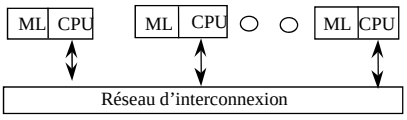
- Modèle d'exécution
  - CM2
  - Maspar



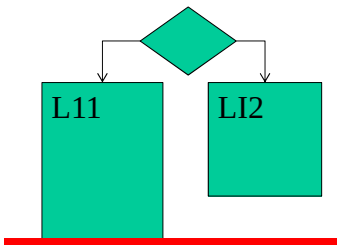
- Modèle obsolète pour machines parallèles
  - Processeurs spécifiques différents des CPU standards
  - Lancement des instructions incompatible avec les fréquences d'horloge des microprocesseurs modernes
  - Synchronisation après chaque instruction : les instructions conditionnelles sont synchronisées
- SPMD
- **Extensions SIMD dans les microprocesseurs**

Version SPMD du SIMD

- Modèle d'exécution

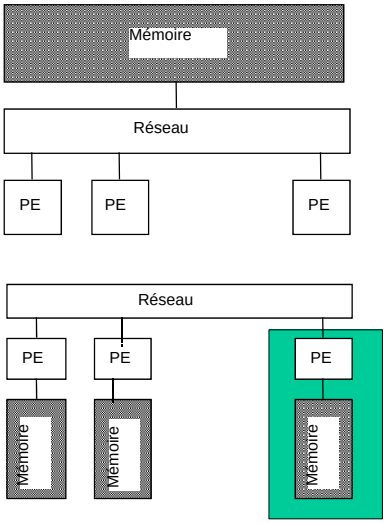


  - Chaque CPU exécute le même code.
  - Les CPU sont synchronisés par des barrières avant les transferts de données.




```
If condition
    Liste instructions 1
Else
    Liste instructions 2
Suite
```

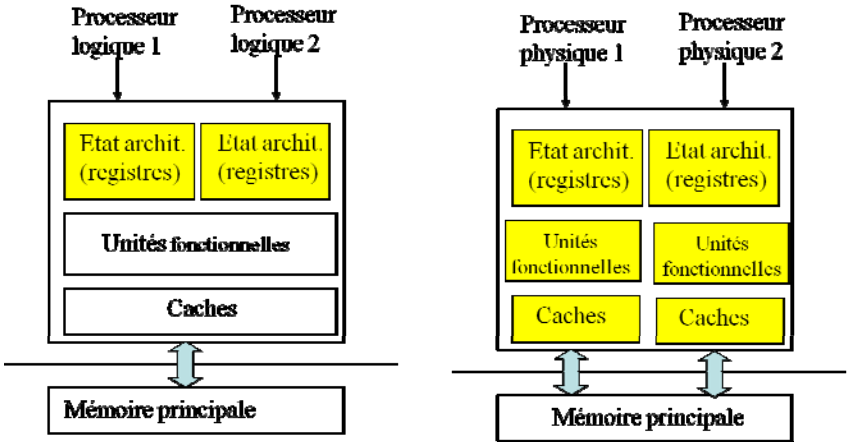
Les architectures MIMD



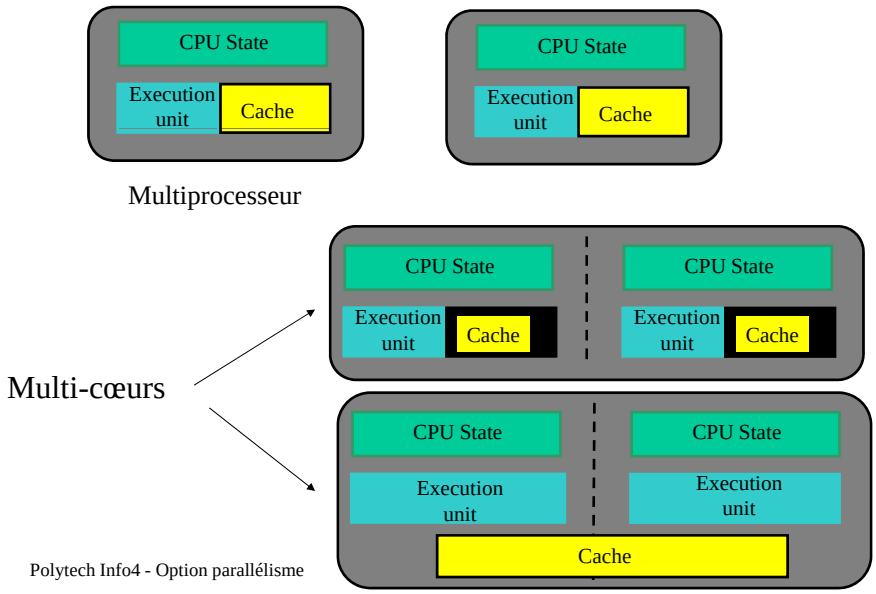
Multiprocesseurs
- OpenMP…
- Pthreads…

Multi-ordinateurs
- MPI (passage de messages)

Multi-thread et Multiprocesseur

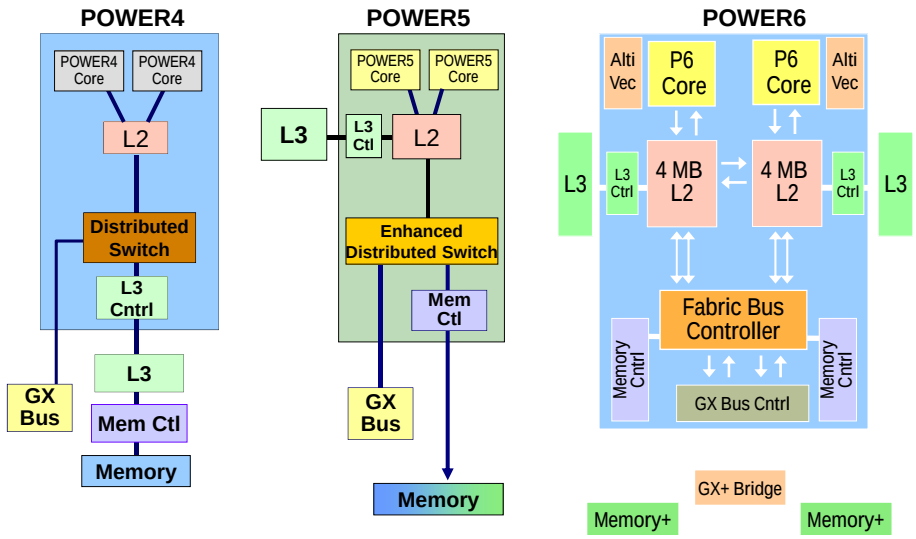


Multiprocesseur et Multi-cœur



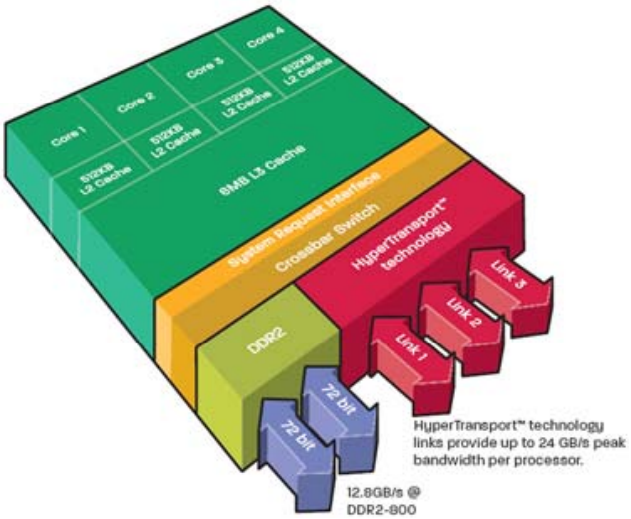
Polytech Info4 - Option parallélisme

Multi-coeurs : POWER4 / POWER5 / POWER6

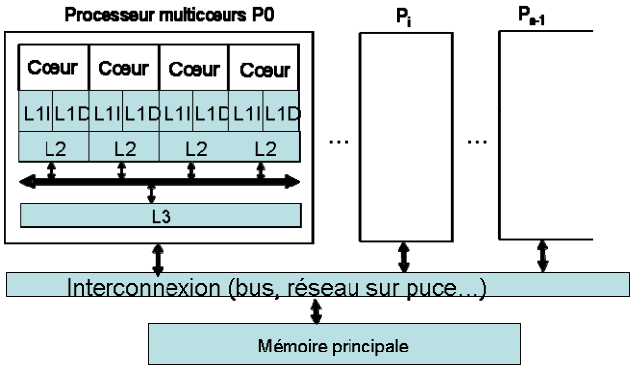


Polytech Info4 - Option parallélisme D. Etiemble 12

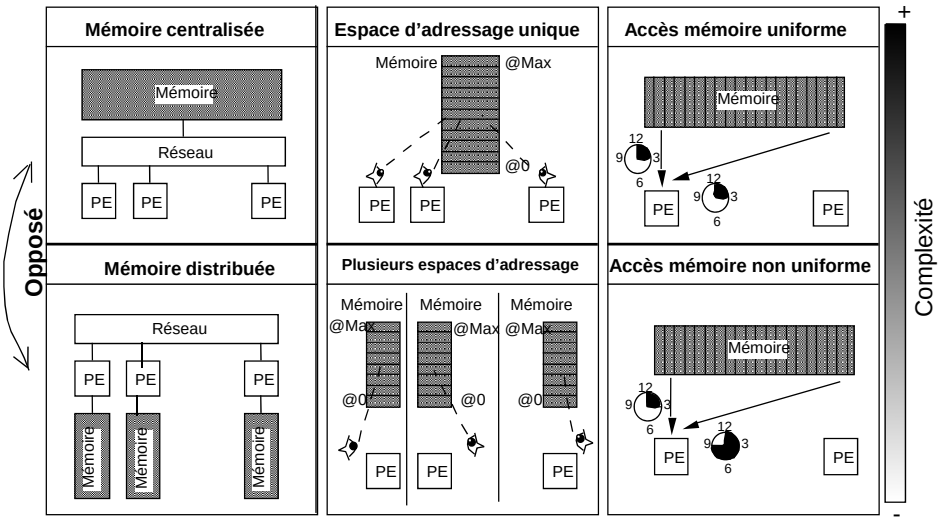
AMD Opteron



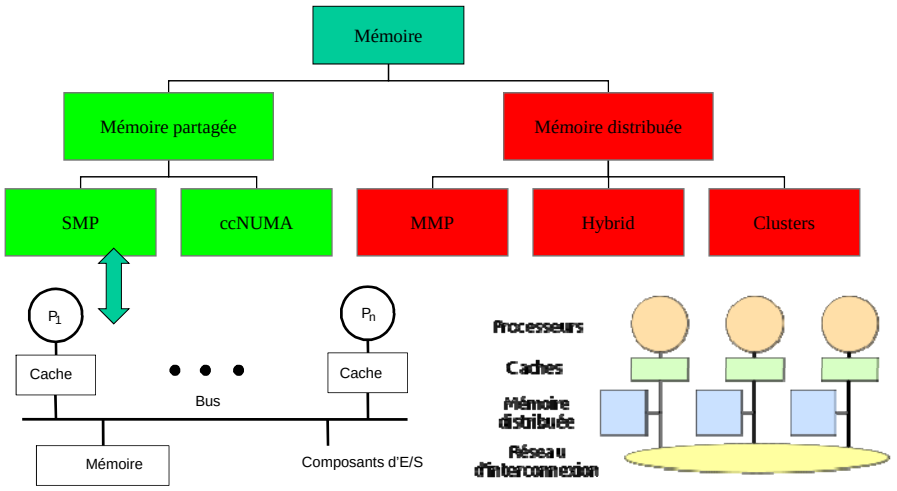
Clusters de multi-cœurs



Organisation mémoire

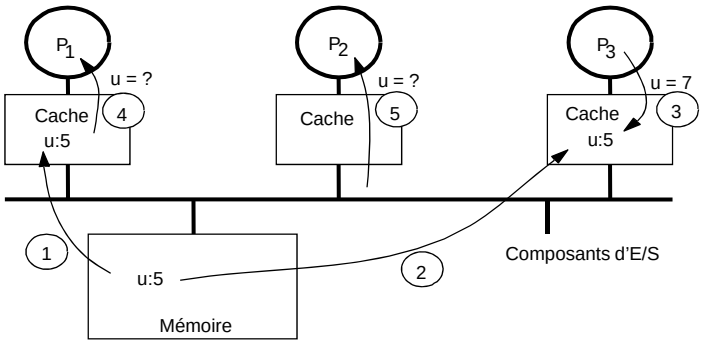


Classification mémoire



Modèle mémoire SMP **NUMA**

Le problème de cohérence des caches



1 P1 lit u
2 P3 lit u
3 P3 écrit dans u
4 P1 lit u
5 P2 lit u

Résultat des lectures 4 et 5
avec écriture simultanée ?
avec la réécriture ?

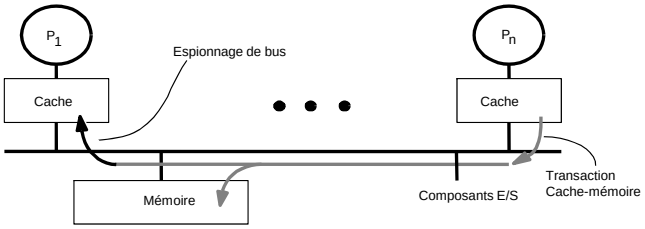
Cohérence des caches avec un bus

- Construite au dessus de deux fondements des systèmes monoprocesseurs
  - Les transactions de bus
  - Le diagramme de transitions d'états des caches
- La transaction de bus monoprocesseur
  - Trois phases : arbitrage, commande/adresse, transfert de données
  - Tous les composants observent les adresses ; un seul maître du bus
- Les états du cache monoprocesseur
  - Ecriture simultanée sans allocation d'écriture
    - Deux états : valide et invalide
  - Réécriture
    - Trois états : valide, invalide, modifié
- Extension aux multiprocesseurs pour implémenter la cohérence

La cohérence par espionnage de bus

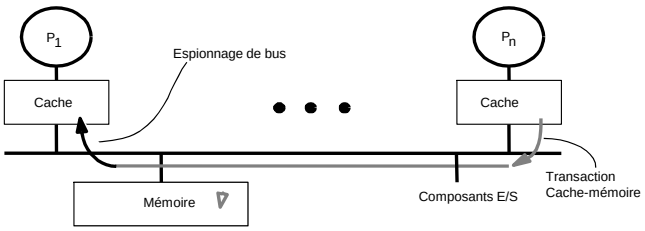
- *Idée de base*
  - Les transactions bus sont visibles par tous les processeurs.
  - Les processeurs peuvent observer le bus et effectuer les actions nécessaires sur les évènements importants (changer l'état)
- *Implémenter un protocole*
  - Le contrôleur de cache reçoit des entrées de deux côtés :
    - Requêtes venant du processeur, requêtes/réponses bus depuis l'espion
  - Dans chaque cas, effectue les actions nécessaires
    - Mise à jour des états, fourniture des données, génération de nouvelles transactions bus
  - Le protocole est un algorithme distribué : machines d'états coopérantes.
    - Ensemble d'états, diagramme de transitions, actions
  - La granularité de la cohérence est typiquement le bloc de cache

Cohérence avec cache à écriture simultanée



- Extension simple du monoprocesseur : les caches à espionnage avec invalidation ou propagation d'écriture
  - Pas de nouveaux états ou de transactions bus
  - Protocoles à diffusion d'écriture
- Propagation des écritures
  - la mémoire est à jour (écriture simultanée)
- Bande passante élevée nécessaire

Cohérence avec cache à invalidation



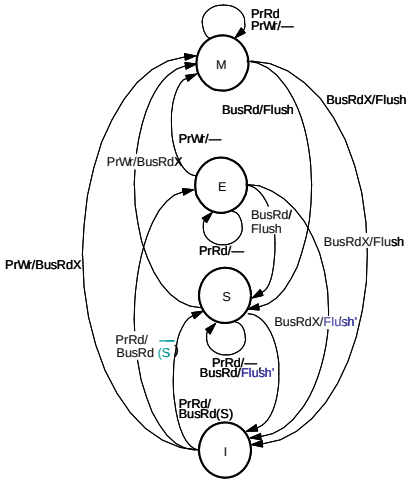
– Une écriture dans un cache invalide les copies de la ligne de cache présente dans les autres caches

Le protocole MESI

- Utilisé pour les monoprocesseurs et les multiprocesseurs
- Quatre états, à invalidation d'écriture
  - Invalide
  - Exclusif
  - Modifié
  - Partagé
- Différentes versions légèrement différentes du protocole MESI
  - Le protocole MESI du PowerPC 601 ne supporte pas les transferts de bloc de cache à cache.

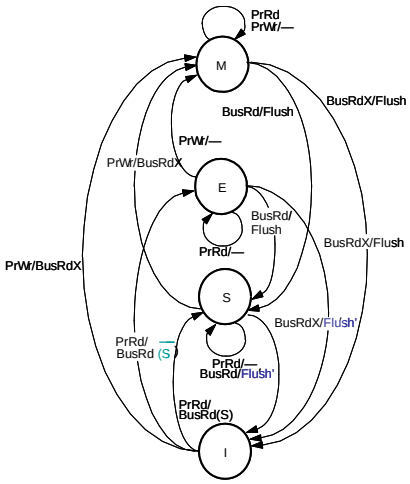
Le protocole MESI

- 4 états
  - Exclusif
    - Seule copie du bloc
    - Non modifié (= Mémoire)
  - Modifié
    - Modifié (!= Mémoire)
  - Partagé
    - Copie dans plusieurs blocs
    - Identique en mémoire
  - Invalide
- Actions
  - Processeurs
  - Bus



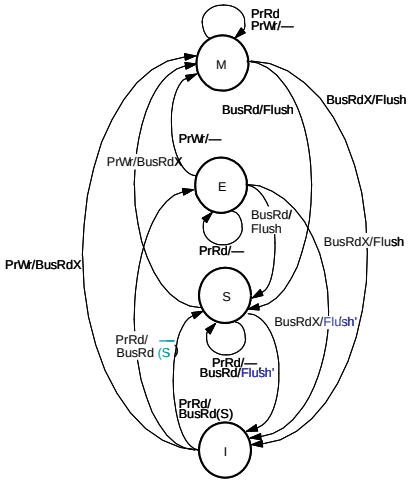
MESI – Succès local lecture

- Le bloc doit être dans l'un des états M, E ou S
- C'est la valeur correcte (si M, elle a été modifiée par une écriture locale)
- Renvoie la valeur
- Pas de changement d'état



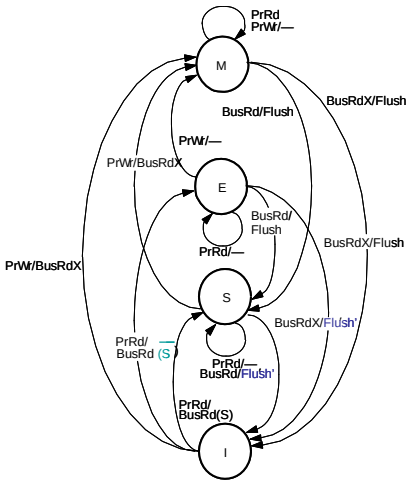
MESI – Défaut local en lecture (1)

- Pas d'autres copies dans un cache
  - Le processeur fait une requête bus vers la mémoire
  - Valeur lue dans le cache local (E)
- Un cache a une copie E
  - Le processeur fait une requête bus vers la mémoire
  - La copie E est mise sur le bus.
  - La requête mémoire est stoppée
  - Le cache local récupère la valeur
  - Les deux blocs passent à S



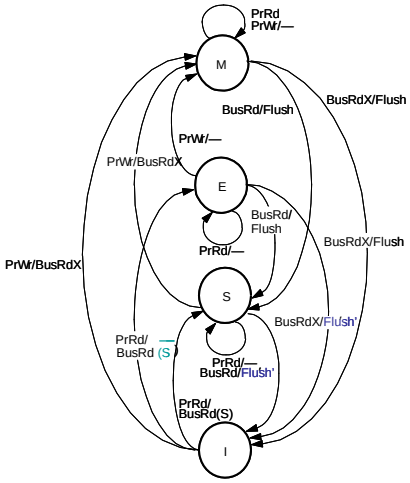
MESI – Défaut local en lecture (2)

- Plusieurs caches ont une copie S
  - Le processeur fait une requête bus vers la mémoire
  - Un cache met une copie sur le bus
  - L'accès mémoire est abandonné
  - Le cache local obtient la copie (S). Les autres copies restent (S)
- Un cache a une copie M
  - Le processeur fait une requête bus vers la mémoire
  - Un cache met la copie M sur le bus
  - L'accès mémoire est abandonné
  - Le cache local obtient la copie (S).
  - Le bloc M est écrit en mémoire.
  - Le bloc M passe à S.



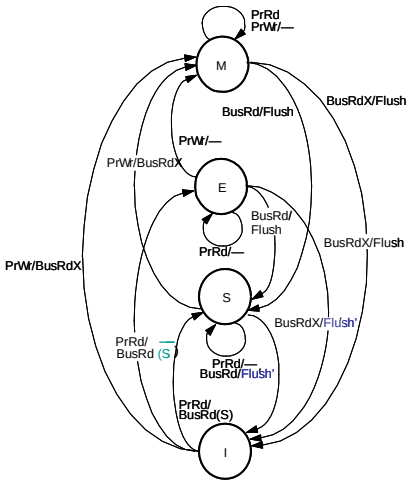
MESI – Succès local en écriture

- Les blocs doivent être M, E ou S
- M
  - le bloc est exclusive et déjà modifiée
  - mettre à jour la valeur locale
  - Pas de changement d'état
- E
  - Mettre à jour la valeur locale
  - L'état passe de E à M
- S
  - Le processeur diffuse une invalidation sur le bus
  - Les caches avec une copie S passe à I
  - Le bloc local est mis à jour
  - L'état du bloc local passe de S à M.



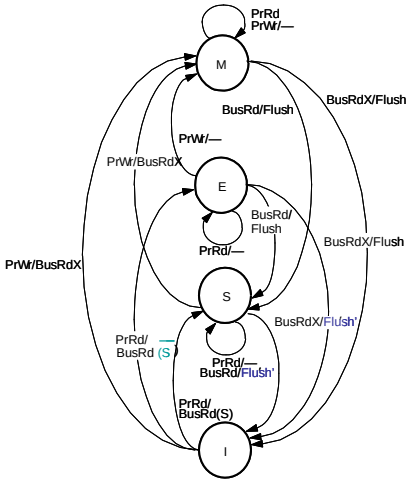
Echec local en écriture

- Pas d'autres copies
  - Valeur lue depuis la mémoire vers cache local
  - Etat bloc local à M
- D'autres copies, soit E (1) soit S (n)
  - Valeur lue depuis la mémoire vers cache local
  - Transaction bus RWITM (lecture avec intention de modifier)
  - Les autres caches mettent leur copie à I
  - La copie locale est mise à jour et état M

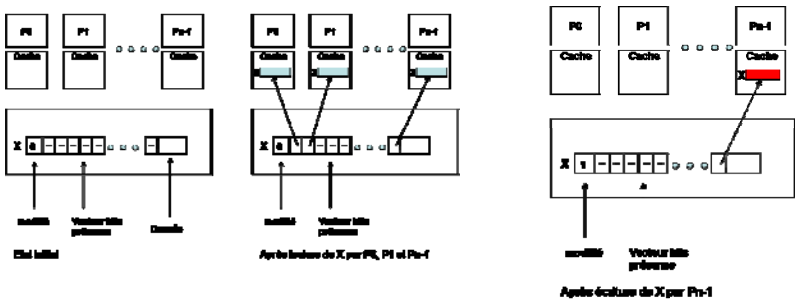


Echec local en écriture (2)

- Une autre copie M
  - Le processeur local envoie une Transaction bus RWITM (lecture avec intention de modifier)
  - Le cache avec copie M la voit, bloque RWTIM, prend contrôle du bus et écrit sa copie en mémoire. Le bloc passe en I.
  - Le processeur local réémet RWITM : c'est maintenant le cas sans copie
  - Valeur lue depuis la mémoire vers cache local
  - Etat bloc local à M



Cohérence par répertoire



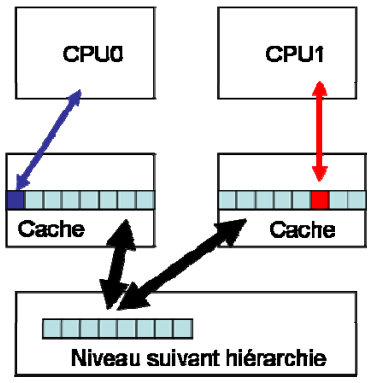
- Répertoire dans la mémoire principale
  - Chaque ligne de la MP a un vecteur de K bits (présence ou non de la ligne dans les K caches) et un bit modifié
  - Chaque ligne des processeurs a un bit valide et un bit privé (seul possesseur)

Extensions pour les clusters

- Extension du protocole MESI
  - MOESI
  - MES

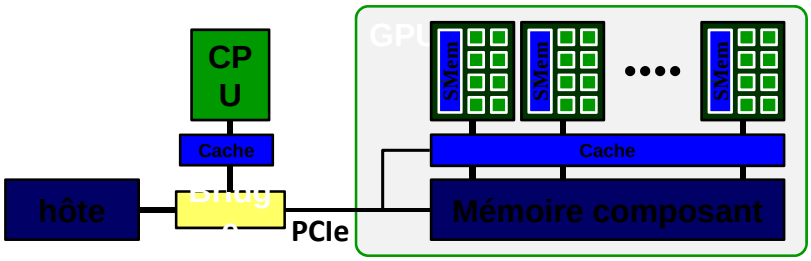
Le problème du faux partage

- Un processeur écrit dans une partie d'une ligne de cache.
- Un autre processeur écrit dans une autre partie d'une ligne de cache
- Même si chaque processeur ne modifie que sa « partie » de la ligne de cache, toute écriture dans un cache provoque une invalidation de « toute » la ligne de cache dans les autres caches.

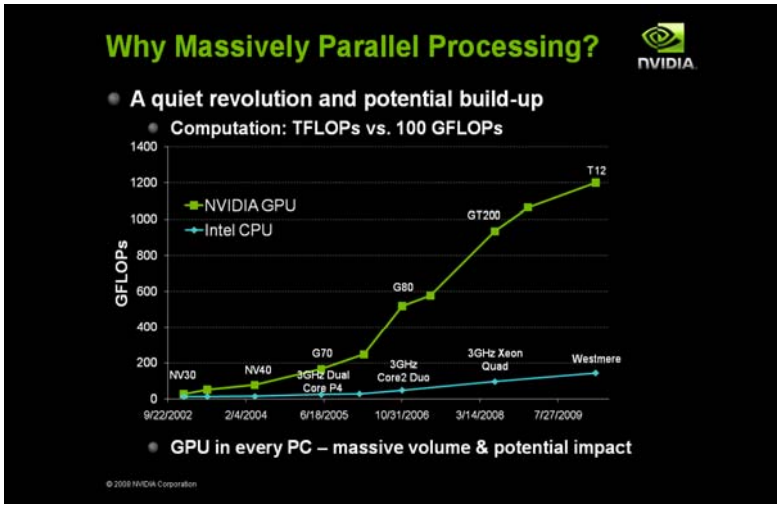


Architecture CPU+GPU

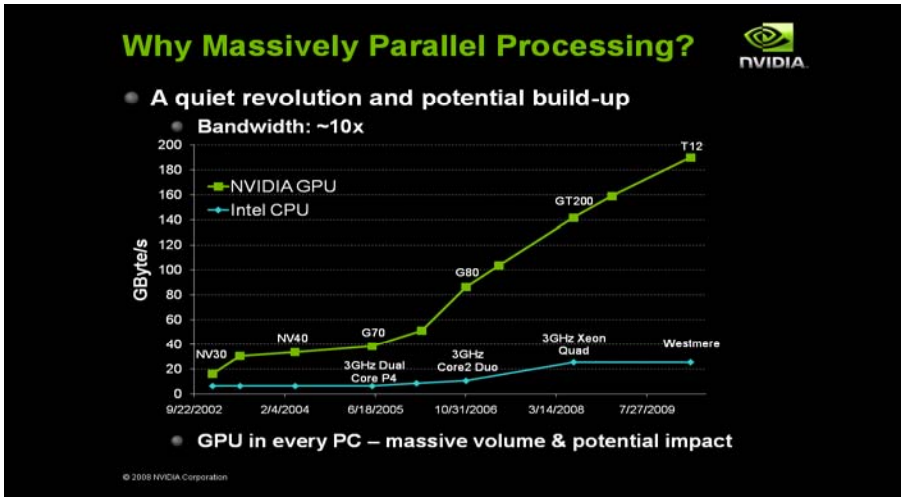
- Architecture hétérogène
- Le CPU exécute les threads séquentiels
  - Exécution séquentielle rapide
  - Accès mémoire à latence faible (hiérarchie mémoire)
- Le GPU exécute le grand nombre de threads parallèles
  - Exécution parallèle extensible
  - Accès mémoire parallèle à très haut débit



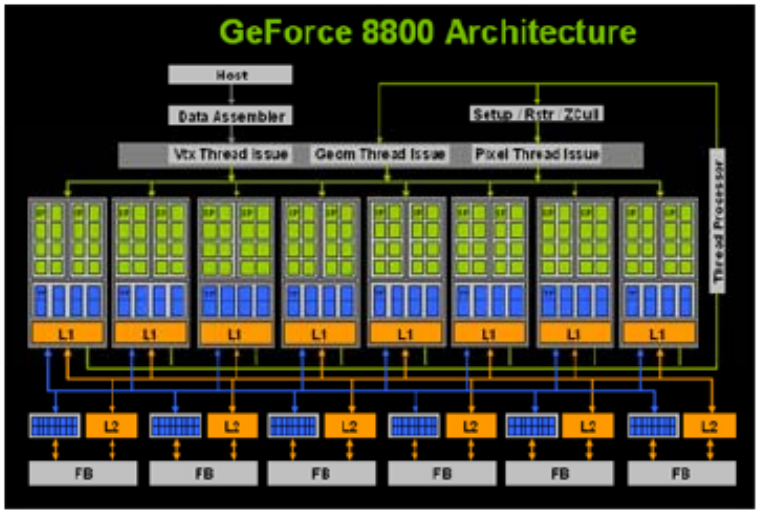
Performances CPU et GPU



Débit mémoire CPU et GPU

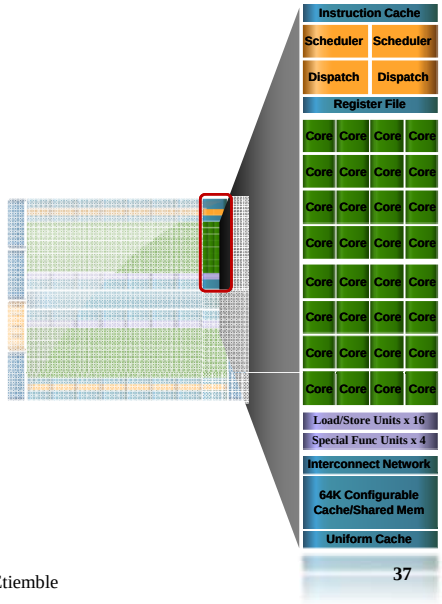


GeForce 8800



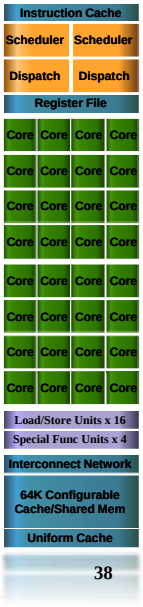
Fermi : Multiprocesseurs (SM)

- 32 cœurs CUDA par SM (512 total)
- Load/store Direct vers mémoire
  - Séquence linéaire classique d'octets
  - Débit élevé (Centaines GB/sec)
- 64 Ko de RAM rapide sur puce
  - Gérée par matériel ou logiciel
  - Partagée entre les coeurs
  - Permet la communication des threads

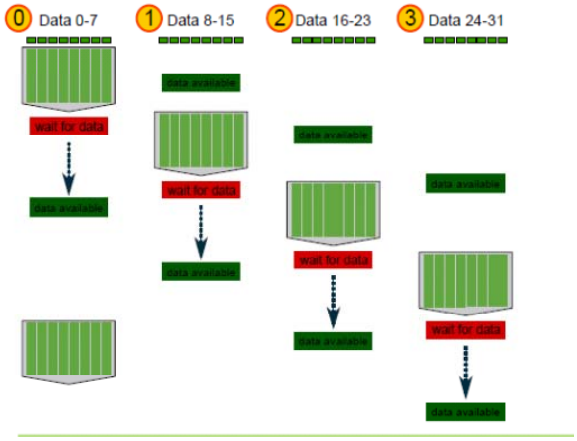


Les concepts clé

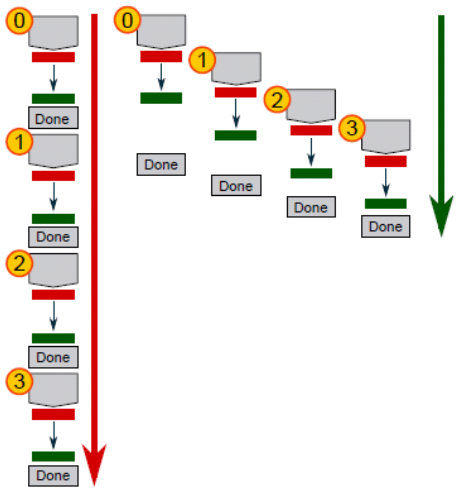
- Exécution SIMT (Single Instruction Multiple Thread)
  - Les threads s'exécutent par groupe de 32 (warps)
  - Une unité d'instructions (IU) par warp
  - Le matériel gère les divergences (if ….then….else)
- Multithreading par matériel
  - Allocation des ressources et ordonnancement des threads par matériel
  - Le matériel utilise des commutations de threads pour cacher les latences
- Les threads ont toutes les ressources nécessaires pour s'exécuter.
  - Tout thread qui n'attend pas peut s'exécuter
  - La commutation de contextes est quasi gratuite



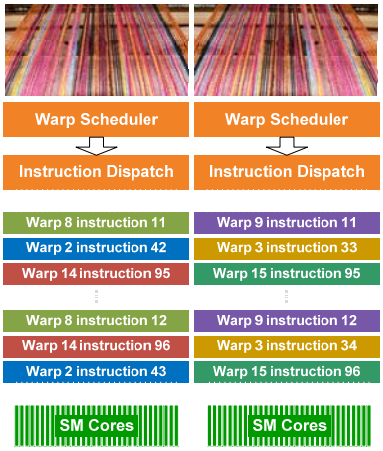
Cacher la latence



Cacher la latence



Exécution SIMT des warps dans les SM



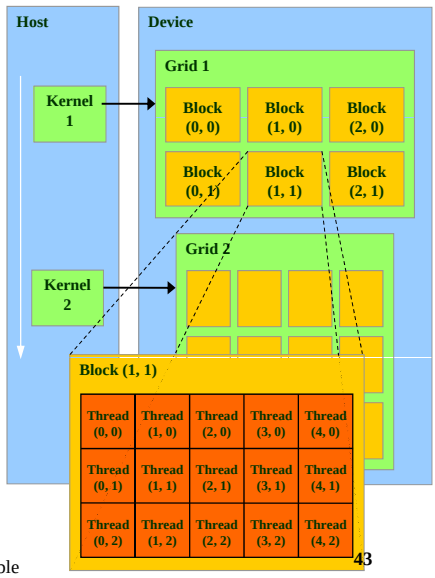
- Les pipelines deux accès choisissent deux warps à lancer aux cœurs parallèles
- Le warp SIMT exécute chaque instruction pour 32 threads
- Des prédicats autorisent ou non l'exécution individuelle des threads
- Une pile gère les branchements au niveau des threads
- Le calcul régulier redondant est plus rapide qu'une exécution irrégulière avec branchements

Modèle de programmation CUDA

- Le GPU est vu comme un composant de calcul qui
  - Est un coprocesseur du CPU ou hôte
  - A sa propre DRAM (composant mémoire)
  - Exécute beaucoup de threads en parallèle
- Les portions "data parallèles" d'une application sont exécutées par le composant comme des noyaux (kernels) qui s'exécutent en parallèle sur beaucoup de threads
- Les différences entre les threads GPU et CPU
  - Les threads GPU sont très légers
    - Très peu de surcoût de création
  - Les GPU ont besoin de milliers de threads pour être efficaces.
    - Les threads CPU en ont besoin de quelques uns.

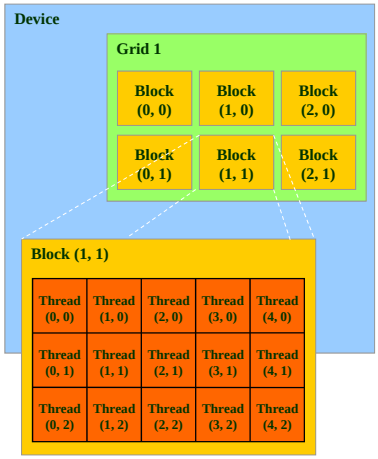
Répartition des threads : grilles et blocs

- Un noyau est exécuté comme une grille de blocs de threads
  - Tous les threads partagent l'espace données mémoire
- Un bloc de threads est un ensemble de threads qui peuvent coopérer ensemble en
  - Synchronisant leur exécution
    - Pour des accès sans aléas à la mémoire partagée
  - Partager efficacement les données via une mémoire partagée à faible latence
- Deux threads de deux blocs différents ne peuvent coopérer.



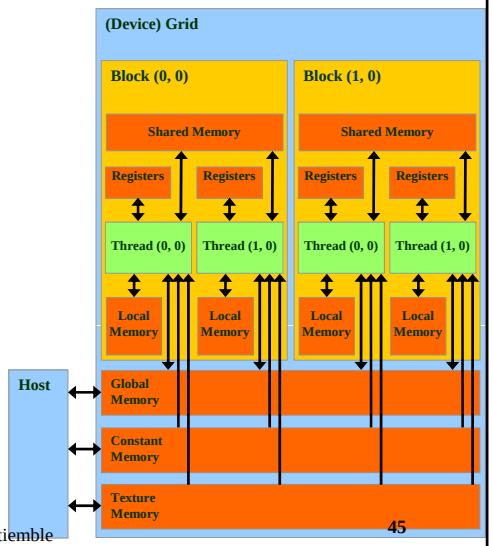
Identification des blocs et threads

- Les threads et les blocs ont un identifiant (ID)
  - Chaque thread peut décider sur quelles données il travaille
  - Blocs ID : 1D ou 2D
  - Thread ID : 1D, 2D ou 3D
- Simplifie l'adressage mémoire lorsqu'on travaille sur des données multidimensionnelles
  - Traitement d'images
  - Résolution d'équations aux dérivées partielles sur des volumes
  - …



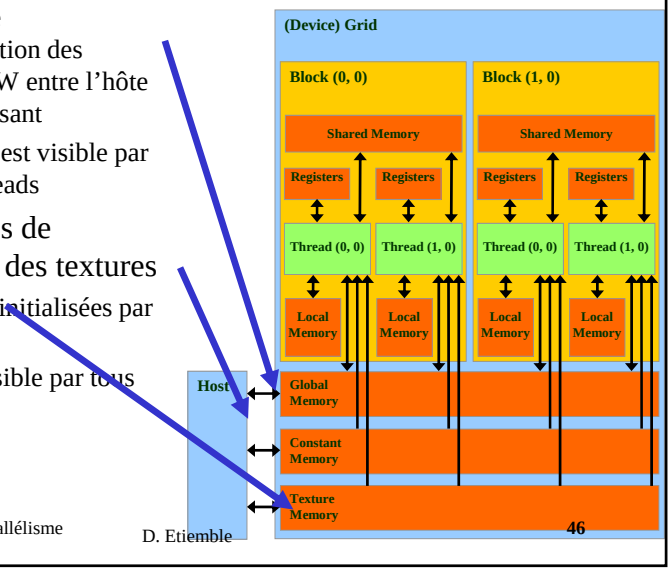
Espace mémoire CUDA

- Chaque thread peut
  - R/W des registres/thread
  - R/W la mémoire locale/thread
  - R/W la mémoire partagée/bloc
  - R/W la mémoire globale/grille
  - Lire la mémoire des constantes/grille
  - Lire la mémoire des textures/grille
- L'hôte peut lire et écrire dans les mémoires globales, des constantes et de textures.



Les mémoires à latence longue

- La mémoire globale
  - Méthode de communication des données R/W entre l'hôte et le composant
  - Le contenu est visible par tous les threads
- Les mémoires de constantes et des textures
  - Constantes initialisées par l'hôte
  - Contenu visible par tous les threads



Addition de vecteurs

- Programme C

```
void addVector (fload *a, float *b,
                fload *c, int N)
{
    int i, index,
    for (i = 0; i<N, i++) {
          c[index] = a[index] + b[index];
    }
}

void main()
{
  .....
  addVector(a, b, c, N);
  .....
}
```

- Programme CUDA

```
__global__ void addVector (fload *a, float *b,
                           float *c)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    c[i] = a[i] + b[i];
}
```

Device code

```
Void main()
{
  ...
  // allocation & transfer data to GPU
  //Excuate on N/256 blocks of 256 threads each
  addVector << N/256, 256 >> ( d_A, d_B, d_C);
  .....
}
```

Host code

Addition matrices (CPU)

Version CPU

```
1.   #include<iostream>
2.
3.   void MatrixAdd(float *A, float *B, float *C, int N){
4.      int index;
5.      for(int i=0;i<N;i++) {
6.       for(int j=0;j<N;j++) {
7.         index = j*N + i;
8.         C[index] = A[index] + B[index];}}}
9.   int main(int argc,char **argv){
10.     int n=1001;
11.     float *a, *b, *c;
12.     a = (float *)malloc(sizeof(float)*n*n);
13.     b = (float *)malloc(sizeof(float)*n*n);
14.     c = (float *)malloc(sizeof(float)*n*n);
15.     for(int j=0;j<n*n;j++) {
16.      a[j]=rand()%35;
17.      b[j]=rand()%35;}
18.     MatrixAdd(a,b,c,n);
19.     free(a); free(b); free(c);
20.     return 0;}
```

Addition de matrices (GPU)

Version GPU

```
1.  #include <iostream>
2.  #include <cuda.h>
3.  __global__ void MatrixAdd_d(float *A, float *B, float *C, int N){
4.     int i = blockIdx.x*blockDim.x + threadIdx.x;
5.     int j = blockIdx.y*blockDim.y + threadIdx.y;
6.     int index = i*N + j;
7.     if(i<N && j<N) { C[index] = A[index] + B[index]; }}
8.   int main(){
9.     float *a_h, *b_h, *c_h; // pointers to host memory; a.k.a. CPU
10.    float *a_d, *b_d, *c_d; // pointers to device memory; a.k.a. GPU
11.    int blocksize=16, n=1001, i, j, index;
12. // allocate arrays on host
13.    a_h = (float *)malloc(sizeof(float)*n*n);
14.    b_h = (float *)malloc(sizeof(float)*n*n);
15.    c_h = (float *)malloc(sizeof(float)*n*n);
16. // allocate arrays on device
17.    cudaMalloc((void **)&a_d,n*n*sizeof(float));
18.    cudaMalloc((void **)&b_d,n*n*sizeof(float));
19.    cudaMalloc((void **)&c_d,n*n*sizeof(float));
20.    dim3 dimBlock( blocksize, blocksize );
21.    dim3 dimGrid( n/dimBlock.x, n/dimBlock.y );
22. //  dim3 dimGrid( ceil(float(n)/float(dimBlock.x)),
   ceil(float(n)/float(dimBlock.y)) );
23. // initialize the arrays
24.    for(j=0;j<n;j++) {
25.     for(i=0;i<n;i++) {
26.      index = i*n+j;
27.      a_h[index]=rand()%35; b_h[index]=rand()%35;}}
28. // copy and run the code on the device
29.    cudaMemcpy(a_d,a_h,n*n*sizeof(float),cudaMemcpyHostToDevice);
30.    cudaMemcpy(b_d,b_h,n*n*sizeof(float),cudaMemcpyHostToDevice);
31.    MatrixAdd_d<<<dimGrid, dimBlock>>>(a_d,b_d,c_d,n);
32.    cudaThreadSynchronize();
33.    cudaMemcpy(c_h,c_d,n*n*sizeof(float),cudaMemcpyDeviceToHost);
34. // cleanup...
35.    free(a_h); free(b_h); free(c_h);
36.    cudaFree(a_d); cudaFree(b_d); cudaFree(c_d);
37.    return(0);}
```