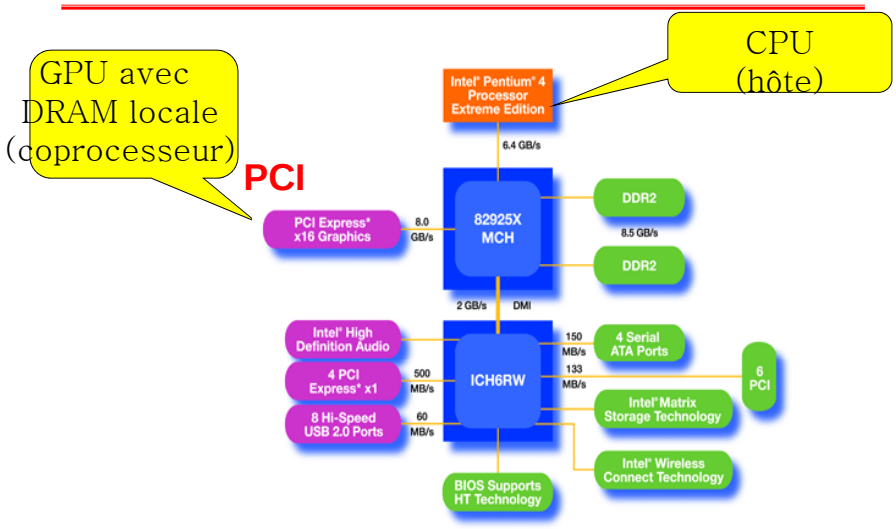
GPUs et CUDA

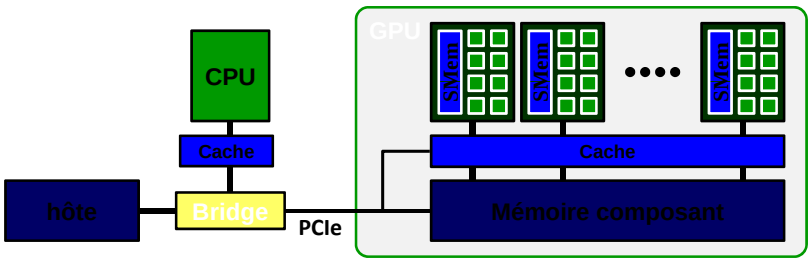
Daniel Etiemble
de@lri.fr

GPU = coprocesseur « graphique »

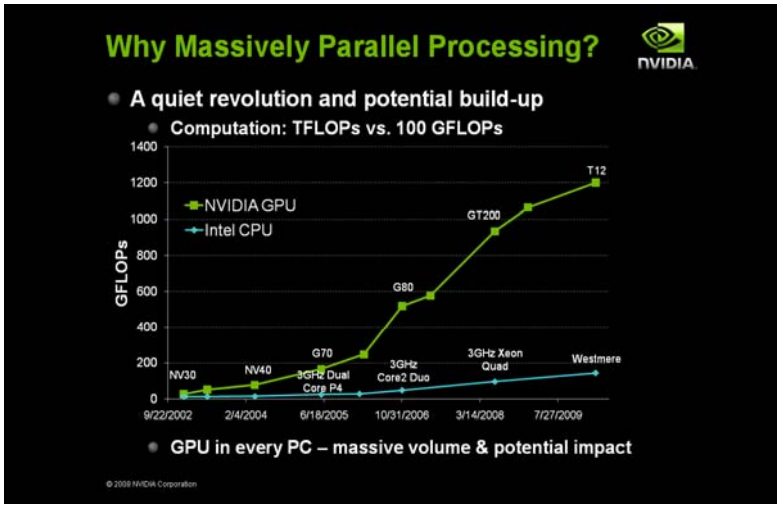


Architecture CPU+GPU

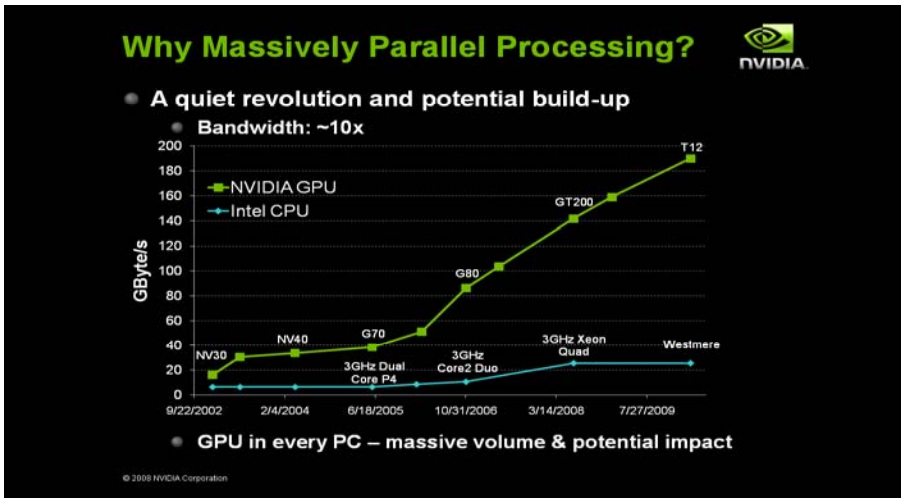
- Architecture hétérogène
- Le CPU exécute les threads séquentiels
  - Exécution séquentielle rapide
  - Accès mémoire à latence faible (hiérarchie mémoire)
- Le GPU exécute le grand nombre de threads parallèles
  - Exécution parallèle extensible
  - Accès mémoire parallèle à très haut débit



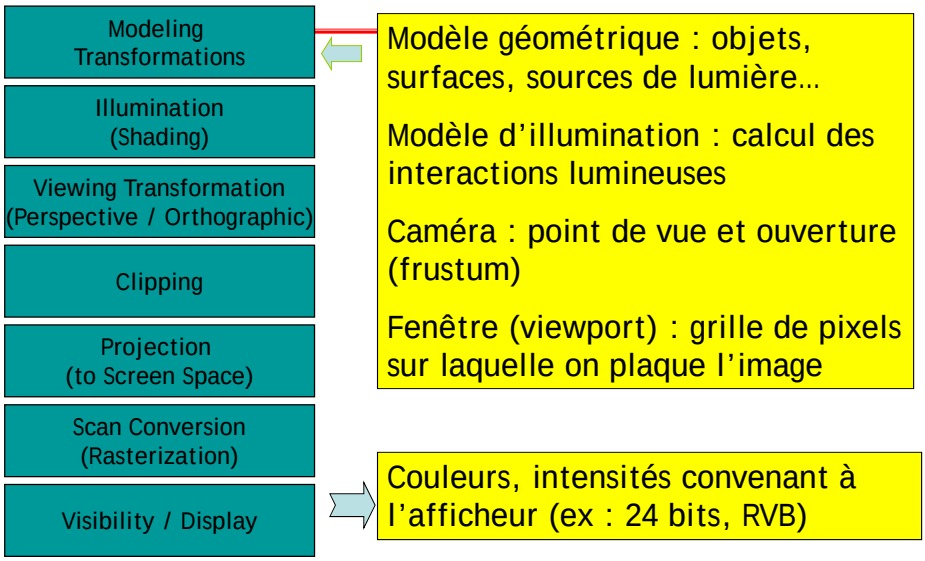
Performances CPU et GPU



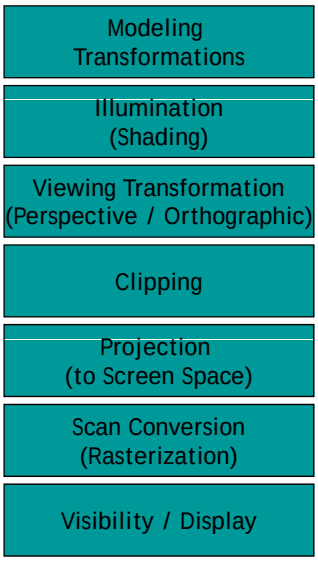
Débit mémoire CPU et GPU



Le pipeline graphique



Le pipeline graphique

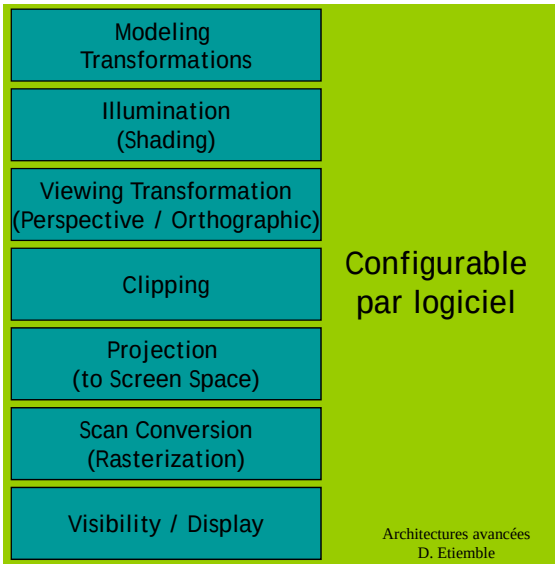


Chaque primitive passe successivement par toutes les étapes

Le pipeline peut être implémenté de diverses manières avec des étapes en matériel et d'autres en logiciel

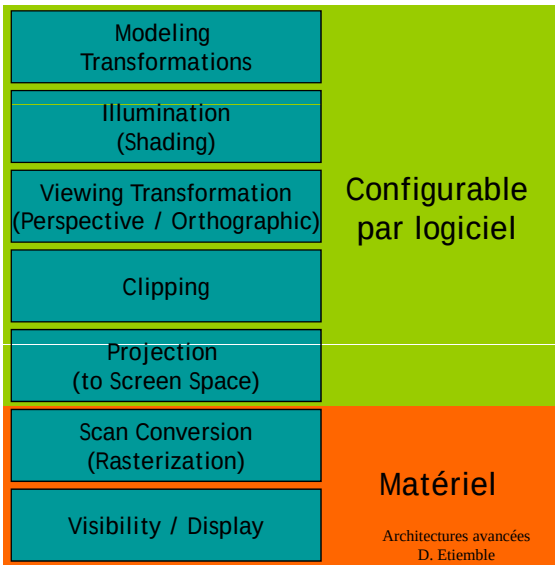
A certaines étapes, on peut disposer d'outils de programmation (ex : programme de sommets ou programme de pixels)

Le pipeline graphique



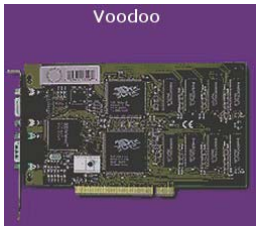
Sans carte graphique 3D

Le pipeline graphique



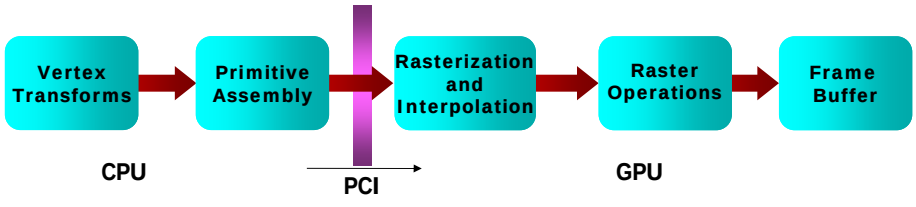
Cartes graphiques première génération

Generation I: 3dfx Voodoo (1996)

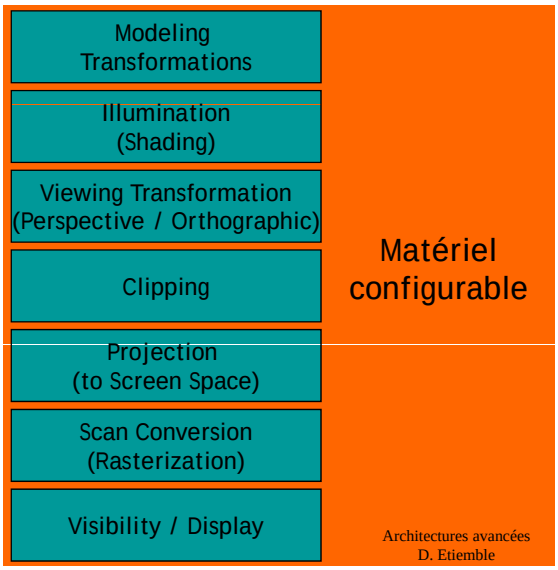


http://accelenation.com/?ac.id.123.2

- Une des premières vraies cartes de jeu 3D
- Ajoute à la carte vidéo standard 2D
- Ne faisait pas les transformations de sommets : faites par le CPU
- Faisait le mappage des textures, le z-buffering.



Le pipeline graphique



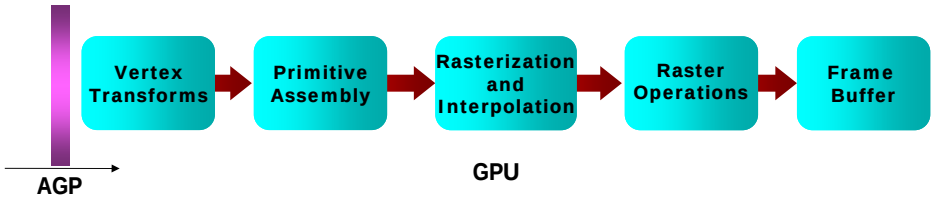
Cartes graphiques deuxième génération

Generation II: GeForce/Radeon 7500 (1998)

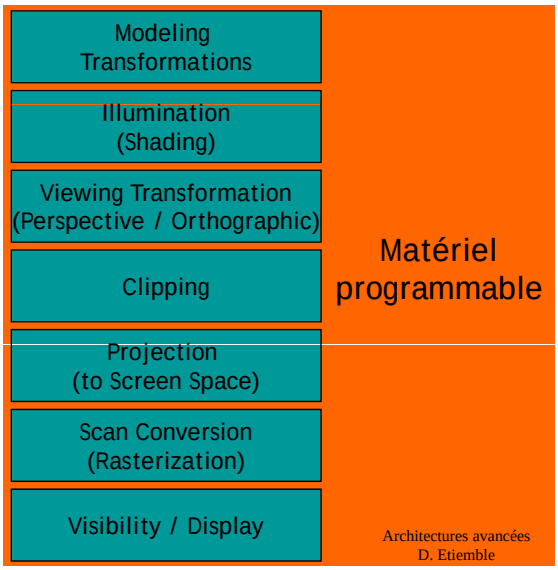
GeForce 256

http://accelenation.com/?ac.id.123.5

- Innovation principale : fait passer les calculs de transformation et d'éclairage au GPU
- Permettait plusieurs textures : bump maps, light maps, et autres.
- Bus AGP au lieu du bus PCI



Le pipeline graphique

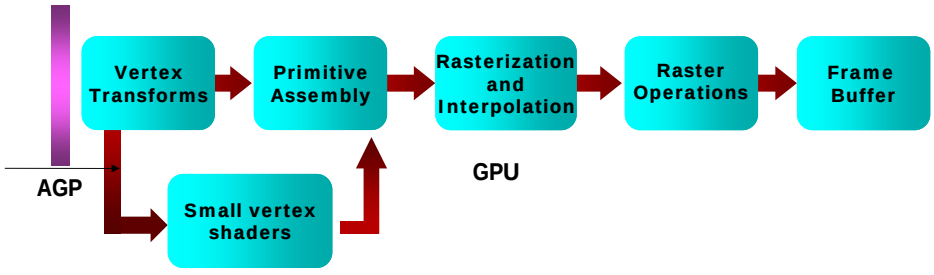


Cartes graphiques troisième génération

Generation III: GeForce3/Radeon 8500(2001)

http://accelenation.com/?ac.id.123.7

- Pour la première fois, permettait une programmation limitée au niveau du pipeline des sommets
- Permettait également les textures de volume et le "multi-sampling" pour l'antialiasing.

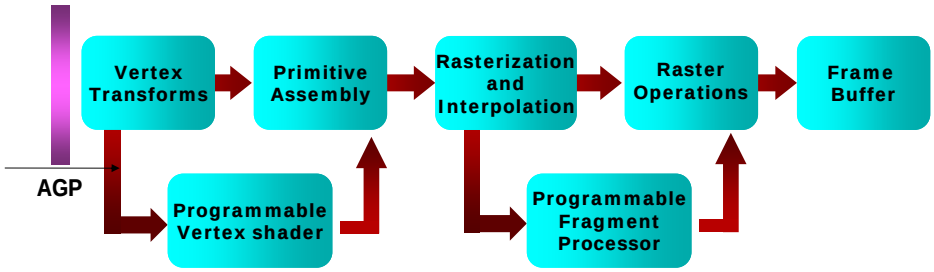


Generation IV: Radeon 9700/GeForce FX (2002)

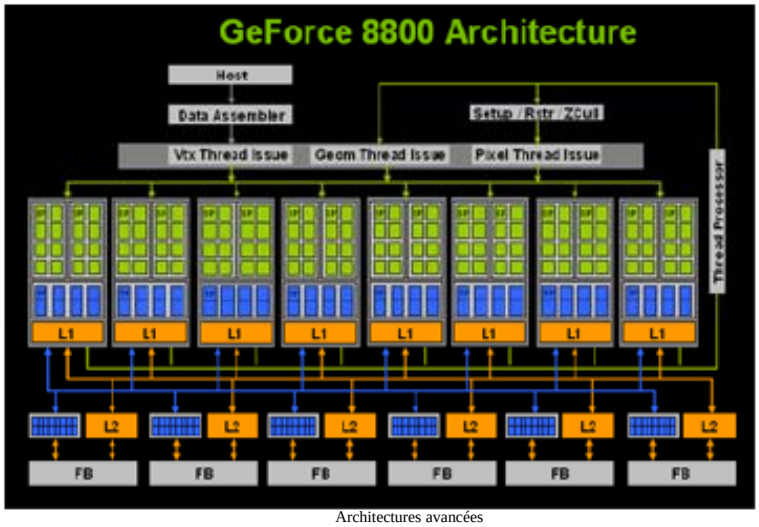
GeForce FX

http://accelenation.com/?ac.id.123.8

- C'est la première génération des cartes graphiques totalement programmables
- Les différentes versions ont des limites différentes sur les possibilités des programmes de fragment et de sommets.

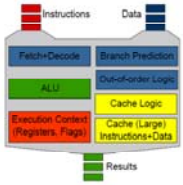


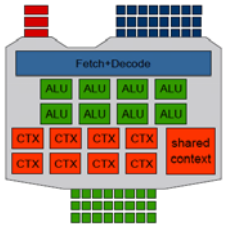
GeForce 8800



Les différents parallélismes

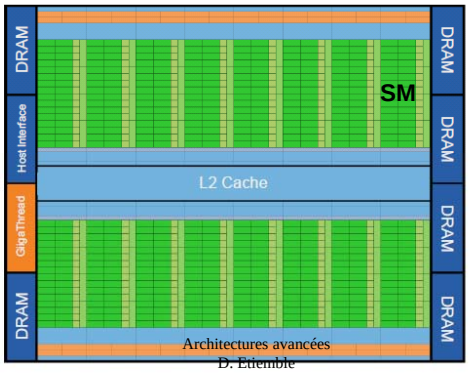
- Parallélisme d'instructions
  - Exécution non ordonnée, spéculation…
  - Moins d'intérêt avec les problèmes de puissance
- Parallélisme de données
  - Unités vectorielles
  - Importance croissante … SSE, AVX, Cell SPE, Clearspeed, GPU





Les différents parallélismes

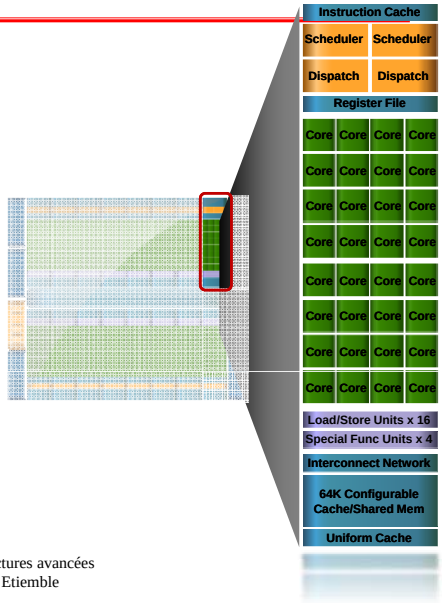
- Parallélisme de threads
  - croissant … multithreading, multicore, manycore
  - Intel Core2, AMD Phenom, Sun Niagara, STI Cell, NVIDIA Fermi, …



Nvidia Fermi

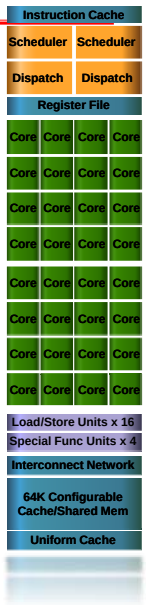
Fermi : Multiprocesseurs (SM)

- 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

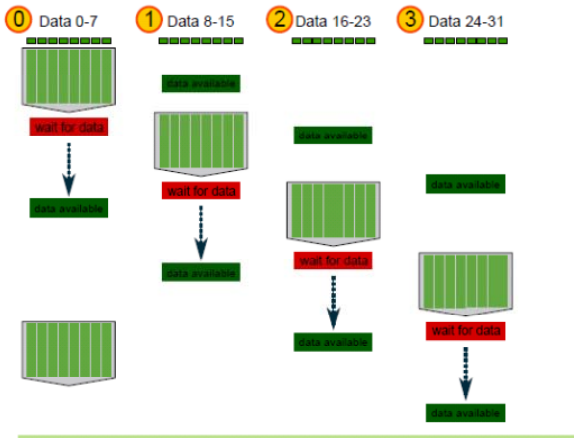


Les concepts clé

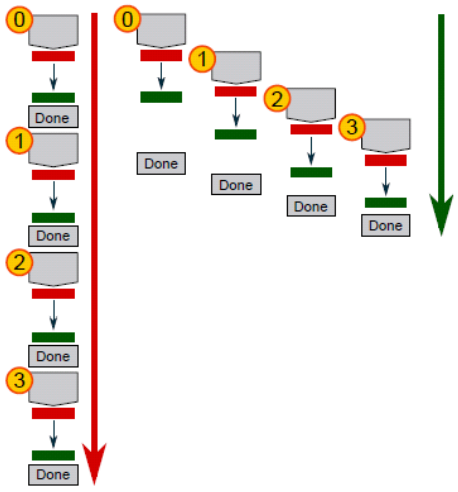
- Exécution SIMT (Single Instruction Multiple Thread)
  - Les threads s'exécutent par groupe de 32 (warps)
  - Une unité d'instructions (IU) par warp
  - Le matériel gère les divergences (if ….then….else)
- Multithreading par matériel
  - Allocation des ressources et ordonnancement des threads par matériel
  - 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



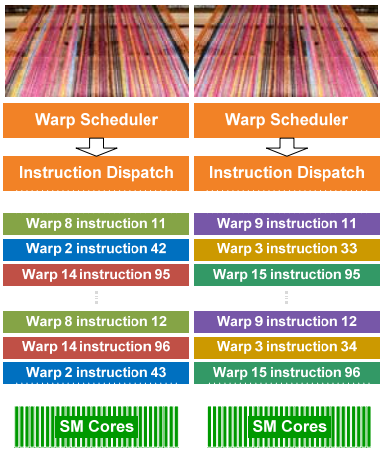
Cacher la latence



Cacher la latence



Exécution SIMT des warps dans les SM



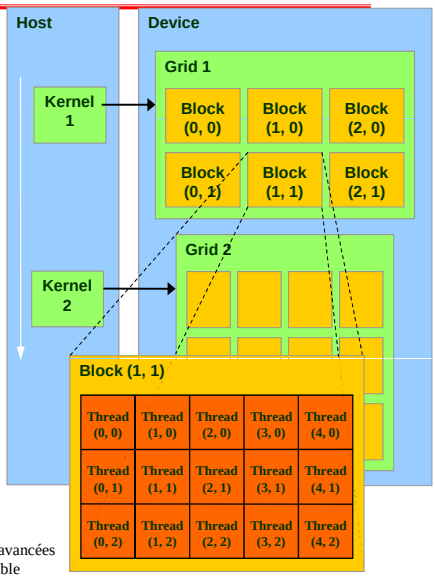
- Les pipelines deux accès choisissent deux warps à lancer aux cœurs parallèles
- Le warp SIMT exécute chaque instruction pour 32 threads
- Des prédicats autorisent ou non l'exécution individuelle des threads
- Une pile gère les branchements au niveau des threads
- Le calcul régulier redondant est plus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

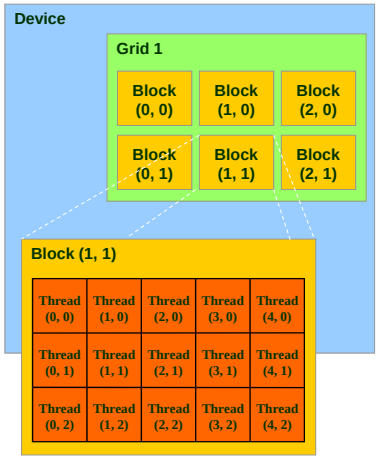
Répartition des threads : grilles et blocs

- 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.



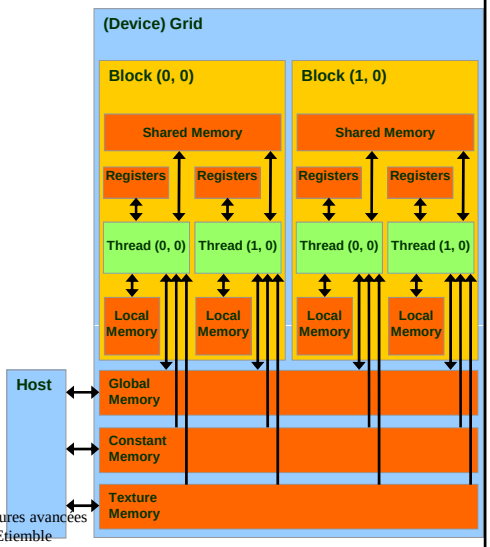
Identification des blocs et threads

- 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
  - …



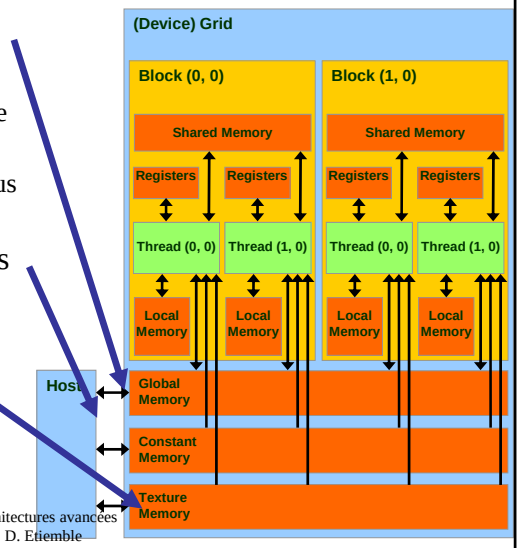
Espace mémoire CUDA

- 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.



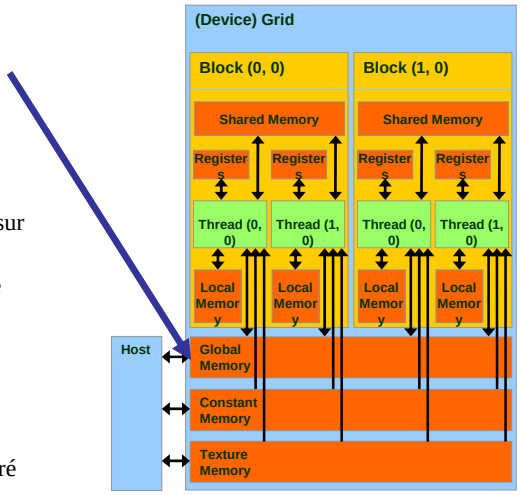
Les mémoires à latence longue

- 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



Allocation mémoire CUDA

- cudaMalloc()
  - Alloue des objets dans la mémoire globale du composant
  - Utilise deux paramètres
    - L'adresse d'un pointeur sur l'objet alloué
    - La taille de l'objet alloué
- cudaFree()
  - Libère des objets dans la mémoire globale du composant
    - Pointeur vers l'objet libéré



Allocation mémoire CUDA

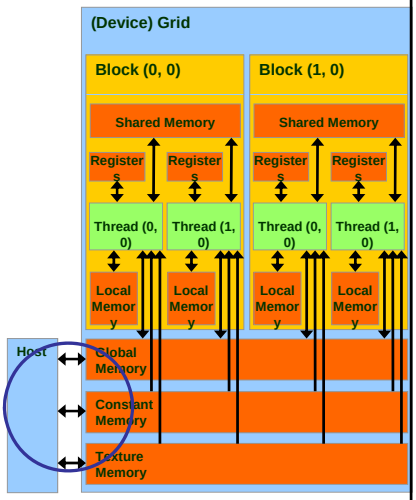
- Exemple de code
  - Alloue un tableau 64 x 64 de flottants simple précision
  - Attache la zone mémoire alloué aux éléments Md
  - "d" est souvent utilisé pour indiquer une structure de données du composant

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE *
sizeof(float);

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

CUDA : transfert hôte- composant

- cudaMemcpy()
  - Transfert de données mémoire
  - Utilise quatre paramètres
    - Pointeur vers la source
    - Pointeur vers la destination
    - Nombre d'octets à copier
    - Type du transferts
      - Hôte vers hôte
      - Hôte vers composant
      - Composant vers hôte
      - Composant vers composant
- Asynchrone dans CUDA 1.0



CUDA : transfert hôte- composant

- Exemple de code:
  - Transfère un tableau 64 *64 de flottants simple précision
  - M est dans la mémoire hôte et Md dans la mémoire du composant
  - cudaMemcpyHostToDevice et cudaMemcpyDeviceToHost sont des constantes symboliques

cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);

Déclarations de fonctions CUDA

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` définit une fonction noyau : doit retourner void
- `__device__` and `__host__` peuvent être utilisés ensemble
- `device__` functions cannot have their address taken
- Pour les fonctions exécutées sur le composant
  - Pas de récursion
  - Pas de déclaration de variables statiques à l'intérieur de la fonction
  - Pas de nombre variable d'arguement

Appel d'une fonction noyau – Création de threads

- Une fonction noyau doit être appelée avec un contexte d'exécution

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per
  block
size_t SharedMemBytes = 64; // 64 bytes of shared
  memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
  >>>(...);
```

- Tout appel à une fonction noyau est asynchrone depuis CUDA 1.0. Une synchronisation explicite n'est pas nécessaire pour bloquer.

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

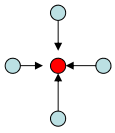
```
1.  #include<iostream>
2.
3.  void MatrixAdd(float *A, float *B, float *C, int N){
4.     int index;
5.     for(int i=0;i<N;i++) {
6.      for(int j=0;j<N;j++) {
7.       index = j*N + i;
8.       C[index] = A[index] + B[index];}}}
9.  int main(int argc,char **argv){
10.    int n=1001;
11.    float *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);}
```


Laplace



Y(i,j) = ¼ (X(i,j-1) + X(i,j+1) X(i-1,j) X(i+1,j))

Laplace (CPU -1)

```
1.  #include <iostream>
2.
3.  void Laplace_h(float *A, float *B, int N){
4.     int index,index1,index2,index3,index4;
5.     for(int i=1;i<N-1;i++) {
6.      for(int j=1;j<N-1;j++) {
7.       index = j*N + i; index1= j*N + i + 1; index2= j*N + i - 1;
8.       index3= (j+1)*N + i; index4= (j-1)*N + i;
9.  B[index] = 0.25*(  A[index1] + A[index2] + A[index3] + A[index4]);}}
10.
11. float Residual_h(float *A, float *B, int N){
12.    int index;
13.    float residual, max_res=0.0;
14.    for(int i=1;i<N-1;i++) {
15.     for(int j=1;j<N-1;j++) {
16.      index = j*N + i;
17.      residual = A[index] - B[index];
18.      if(residual>max_res) max_res  = residual;}}
19.    return max_res;}
20.
21. void Initialize(float *A, float *B, int N){
22. // initialisation }
```


Laplace (CPU – 2)

```
23. int main(int argc,char **argv){
24.    int k=0,n=10, blocksize=64;
25.    float max_residual, *phi1_h, *phi2_h;
26.    phi1_h = (float *)malloc(sizeof(float)*n*n);
27.    phi2_h = (float *)malloc(sizeof(float)*n*n);
28.    Initialize(phi1_h,phi2_h,n);
29.    while(k<100) {
30.      Laplace_h(phi1_h,phi2_h,n);
31.      Laplace_h(phi2_h,phi1_h,n);
32.      k+=2;}
33.    max_residual = Residual_h(phi1_h,phi2_h,n);
34.    printf("%d CPU residual=%f\n",k,max_residual);
35.    free(phi1_h); free(phi2_h);
36.    return 0;}
```

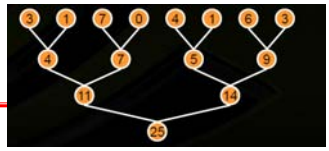
Laplace (GPU – 1)

```
1.  #include <iostream>
2.  #include <cuda.h>
3.  #include "sys/time.h"
4.  using namespace std;
5.
6.  __global__ void Laplace_d(float *A, float *B, int N){
7.     int i = blockIdx.x * blockDim.x + threadIdx.x ;
8.     int j = blockIdx.y * blockDim.y + threadIdx.y ;
9.     int index,index1,index2,index3,index4;
10.    index = i*N + j; index1= i*N + j + 1; index2= i*N + j - 1;
11.    index3= (i+1)*N + j; index4= (i-1)*N + j;
12.    if(i>0 && i<N-1 && j>0 && j<N-1) { B[index] = 0.25*(  A[index1] +
   A[index2] + A[index3] + A[index4] ); }
13. }
14. void Laplace_h(float *A, float *B, int N){
15.    int index,index1,index2,index3,index4;
16.    for(int i=0;i<N;i++) {
17.     for(int j=0;j<N;j++) {
18.      index = i*N + j; index1= i*N + j + 1; index2= i*N + j - 1;
19.      index3= (i+1)*N + j; index4= (i-1)*N + j;
20.      if(i>0 && j>0 && i<N-1 && j<N-1) { B[index] = 0.25*(  A[index1] +
   A[index2] + A[index3] + A[index4] ); }}}
21.
22. void Initialize(float *A, float *B, int N)
23. {/* initialisation*/}
```

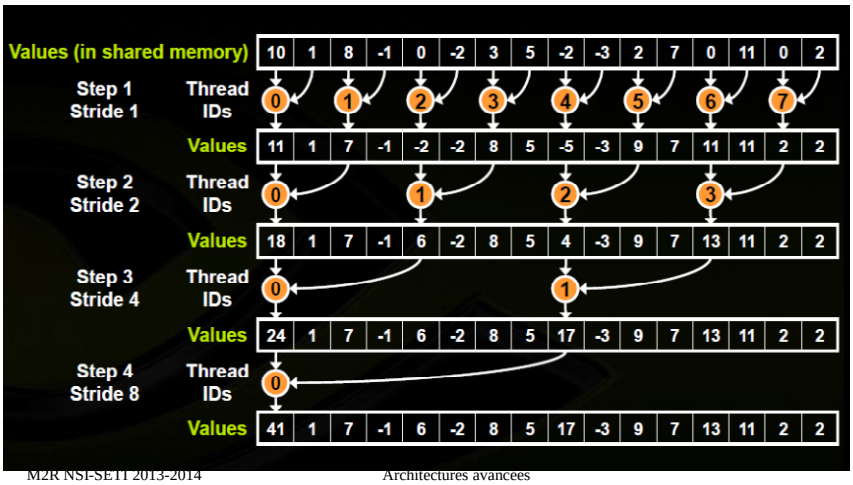

Laplace (GPU-2)

```
24. int main(int argc,char **argv){
25.    int k=0, iterations=100, n=64, ThreadsPerBlock=16;
26.    struct timeval t1_s,t1_e,t2_s,t2_e;
27.    float *phi1_h, *phi2_h; // pointers to host memory; a.k.a. CPU
28.    float *phi1_d, *phi2_d; // pointers to device memory; a.k.a. GPU
29.  // Allocate arrays on host and initialize
30.    phi1_h = (float *)malloc(sizeof(float)*n*n);
31.    phi2_h = (float *)malloc(sizeof(float)*n*n);
32.    Initialize(phi1_h,phi2_h,n);
33.    cudaMalloc((void **)&phi1_d,n*n*sizeof(float));
34.    cudaMalloc((void **)&phi2_d,n*n*sizeof(float));
35.    dim3 dimBlock( ThreadsPerBlock, ThreadsPerBlock );
36.    dim3 dimGrid( ceil(float(n)/float(dimBlock.x)),
   ceil(float(n)/float(dimBlock.y)) );
37.    cudaMemcpy(phi1_d,phi1_h,n*n*sizeof(float),cudaMemcpyHostToDevice);
38.    cudaMemcpy(phi2_d,phi2_h,n*n*sizeof(float),cudaMemcpyHostToDevice);
39.     k = 0;
40.    while(k<iterations){
41.      Laplace_d<<<dimGrid, dimBlock>>>(phi1_d,phi2_d,n);
42.      Laplace_d<<<dimGrid, dimBlock>>>(phi2_d,phi1_d,n);
43.      k+=2;}
44.    cudaMemcpy(phi2_h,phi2_d,n*n*sizeof(float),cudaMemcpyDeviceToHost);
45.    cudaThreadSynchronize();
46.    cudaFree(phi1_d);
47.    cudaFree(phi1_d);
48.    return 0;}
```

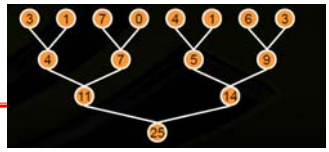
Réduction sur GPU



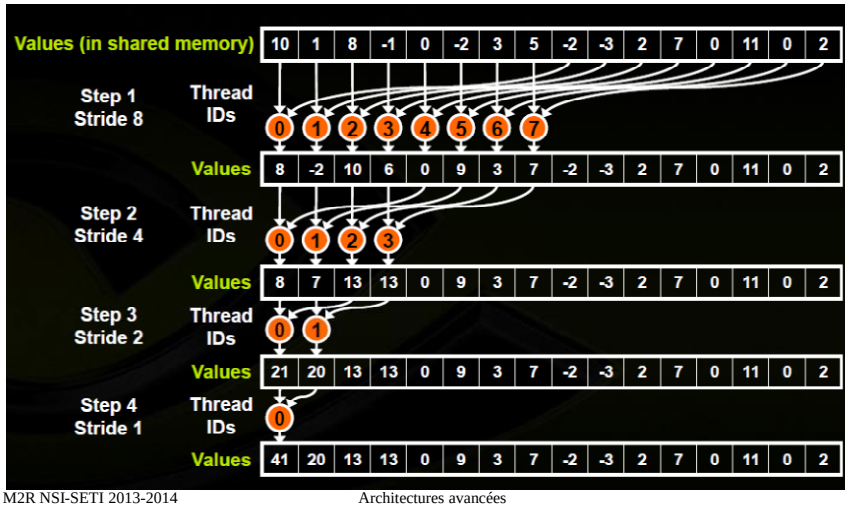
Entrelacement



Réduction sur GPU

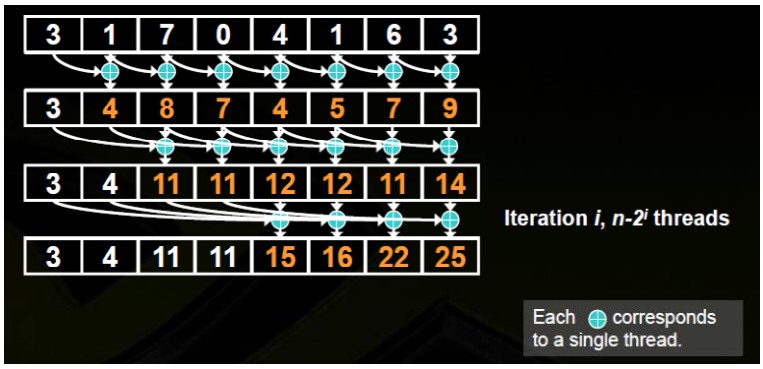


Continu



Algorithme Scan (GPU)

scan(A) = [I, a_0, ($a_0 \oplus a_1$), …, ($a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2}$)]



Références

- David Kirk/NVIDIA and Wen-mei W. Hwu, 2007, ECE 498AL, University of Illinois, Urbana-Champaign
- NVIDIA., CUDA Best Practices Guide, 3.0 edition, March 2010.
- http://www.starba.se/gpgpu/workshop.pdf
- http://www.nvidia.com/page/technologies.html