

Algorithmes évolutionnaires et GPU

- **Introduction : GPU et puissance calculatoire**
- **Principe du GPU : le pipeline graphique**
- **Introduction au parallélisme de données**
 - **Exemple de simulation**
 - **Les langages de haut-niveau pour le GPU**
 - **Accelerator, CUDA**
- **GPU et algorithmes évolutionnaires**
 - **Programmation génétique sur GPU**
 - **Algorithmes évolutionnaires sur GPU**

Introduction

- **Le GPU est devenu depuis années un processeur flexible, peu onéreux.**
 - Puissance
 - Programmable
 - Précision (float)
 - Calculs dédiés

Motivation : puissance calculatoire

➤ Les GPUs sont puissants

➤ Intel Core 2 Duo 3,5 Ghz (Woodcrest Xeon)

➤ Puissance 48 Gflops Peak

➤ Bande passante 21 Gbits/s Peak

➤ Prix 900 €

➤ NVIDIA GeForce 8800 GTX

➤ Puissance 330 Gflops Mesuré

➤ Bande passante 55,2 Gbits/s Mesuré

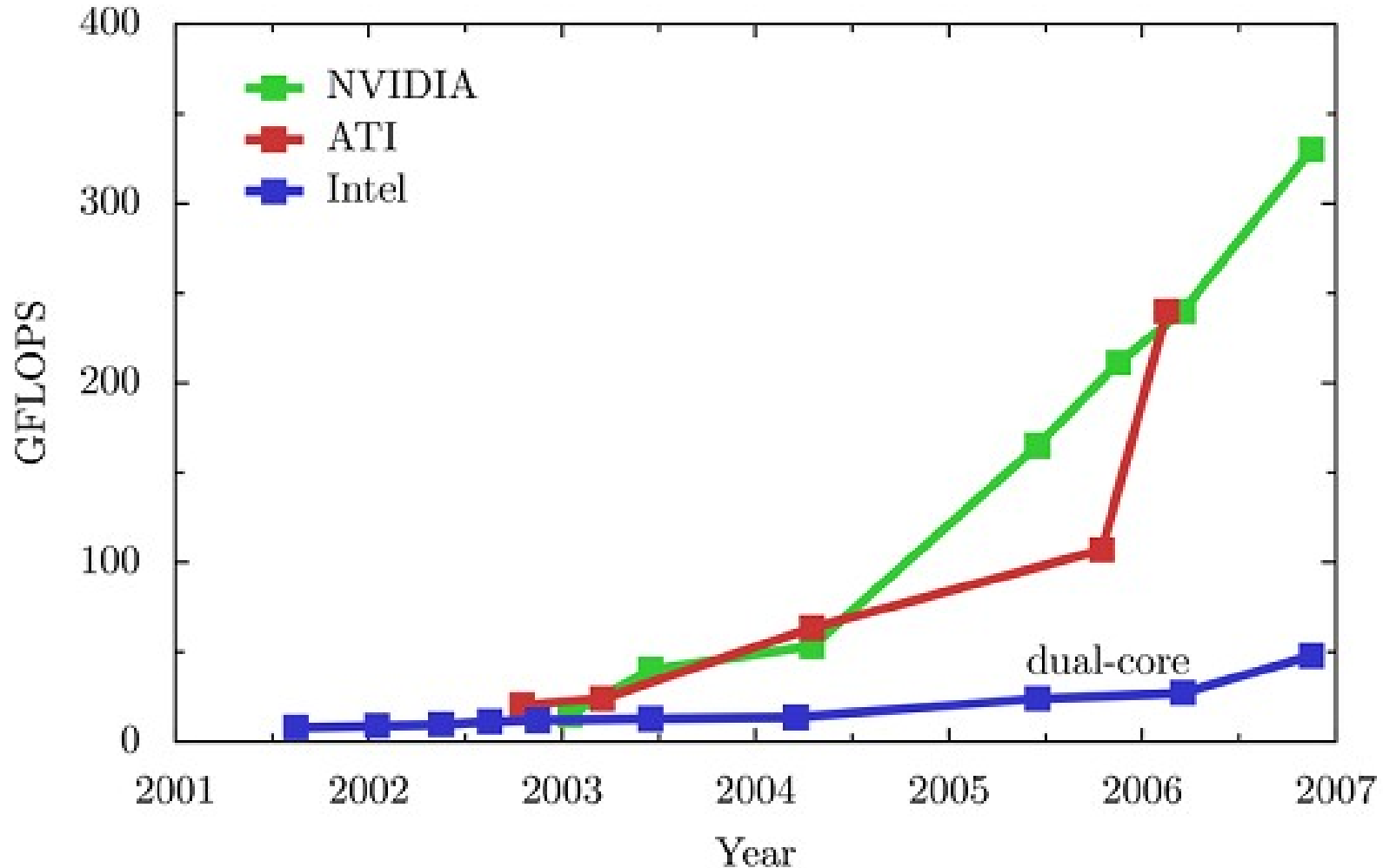
➤ Prix 600 €

➤ Les GPUs sont de plus en plus rapides

➤ CPU : $\sim x 1,4$ /an

➤ GPU : $\sim x 1,7$ /an (pixels) à $x 2,3$ (vertex)

Aperçu de la puissance calculatoire



Note

- ***Pourquoi les GPUs sont de plus en plus rapides ?***
 - Intensité arithmétique
 - Les calculs sont hyper-spécialisés et par conséquent les circuits sont simples (proches du RISC)
 - Économie
 - L'industrie du jeu vidéo pousse à l'innovation

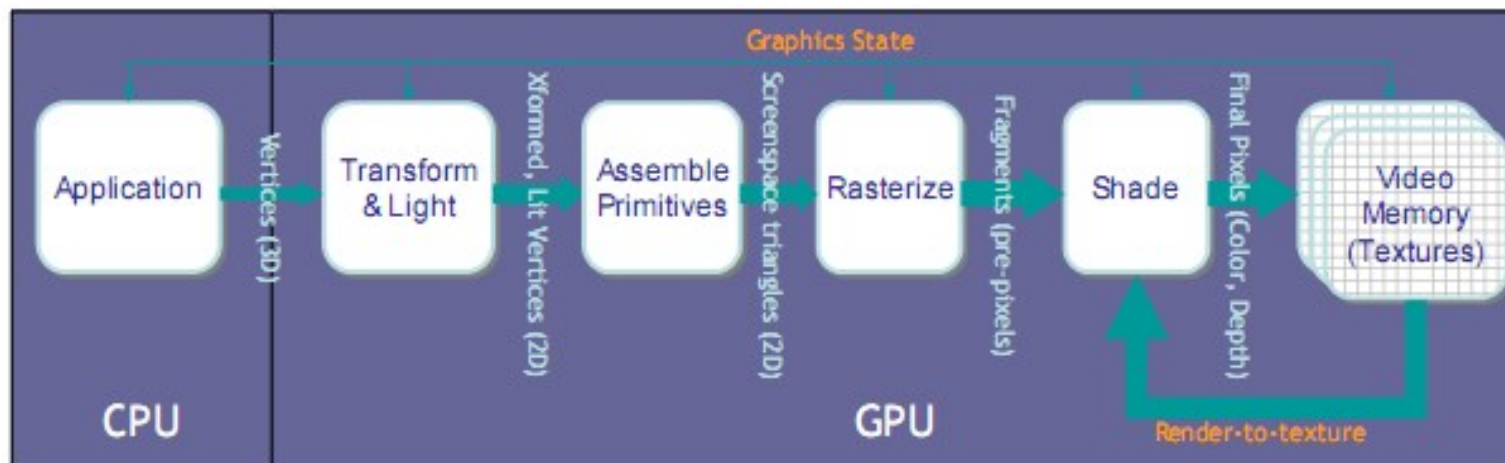
Motivations : flexibles et précis

- **Les nouveaux GPUs sont programmables**
 - Programmation au niveau du pixel, du vertex et (bientôt) shader de géométrie
 - Utilisation de langage de haut niveau
- **Précision importante**
 - Précision des réels sur 32 bits dans tout le pipeline
 - Suffisant pour la plupart des applications
 - Bientôt en double précision

Problème : difficile à utiliser

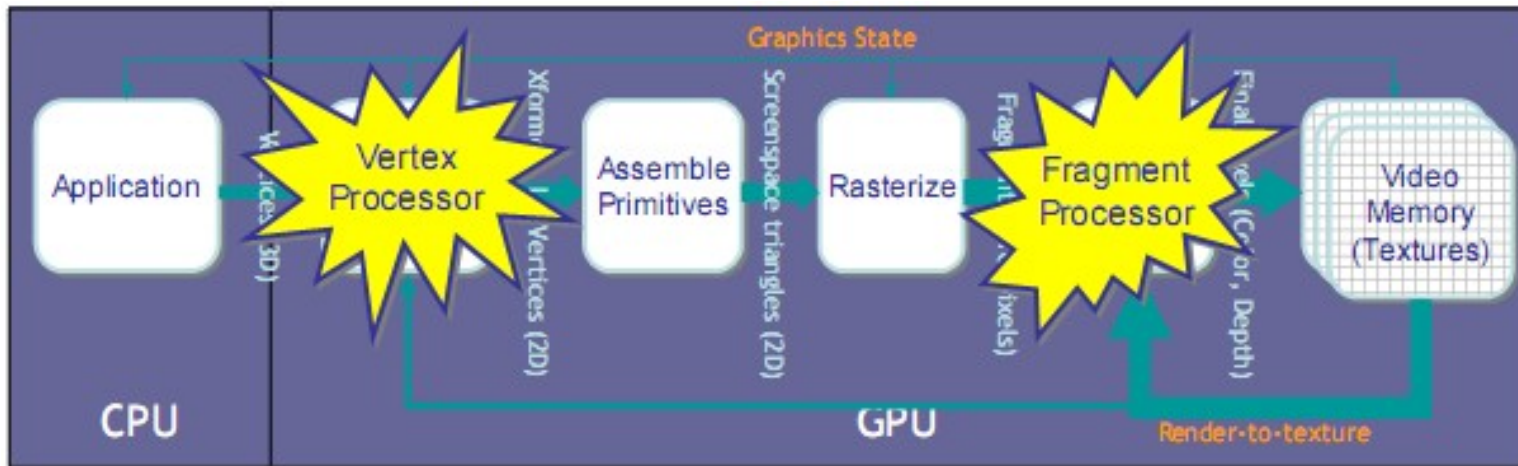
- **Les GPUs sont conçus pour les jeux vidéo**
 - Modèle de programmation atypique
 - Modèle de programmation très lié à la programmation graphique
- **L'architecture sous-jacente est :**
 - Basée sur le parallélisme de données
 - En évolution rapide
 - Relativement secrète !
- **On ne peut pas faire simplement du portage de code**

Le pipeline graphique



- **Le pipeline graphique simplifié**
 - Hautement parallèle
 - Beaucoup de caches, de FIFOs, ...

Le pipeline graphique récent

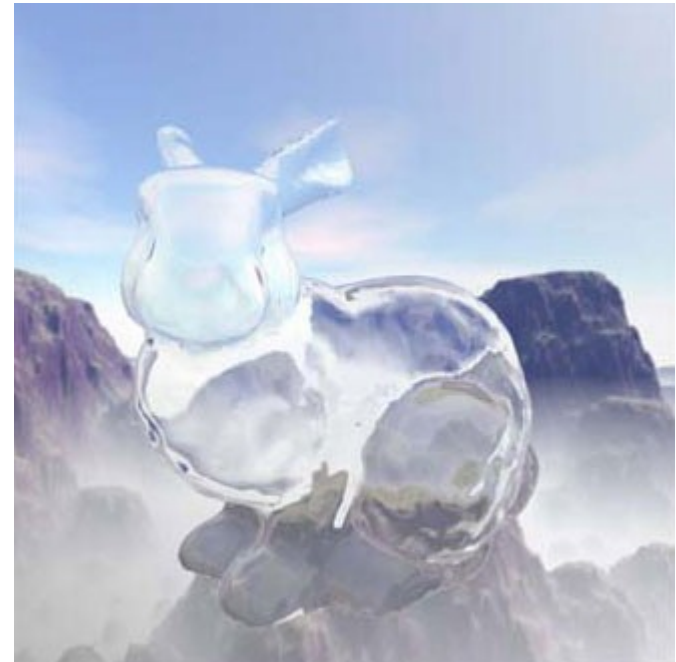


➤ **Processeur de vertex programmable (vertex shader)**

➤ **Pixel shader programmable**

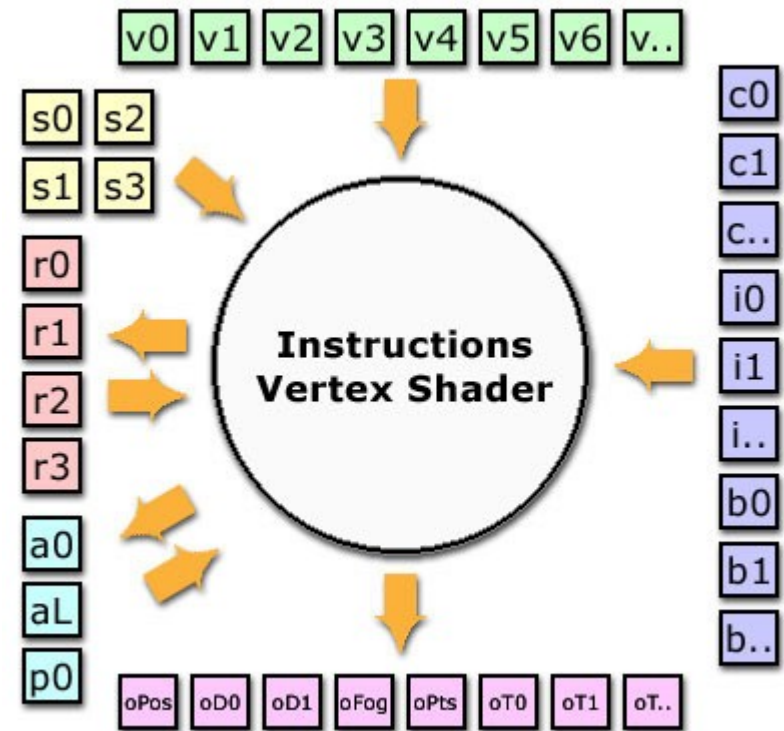
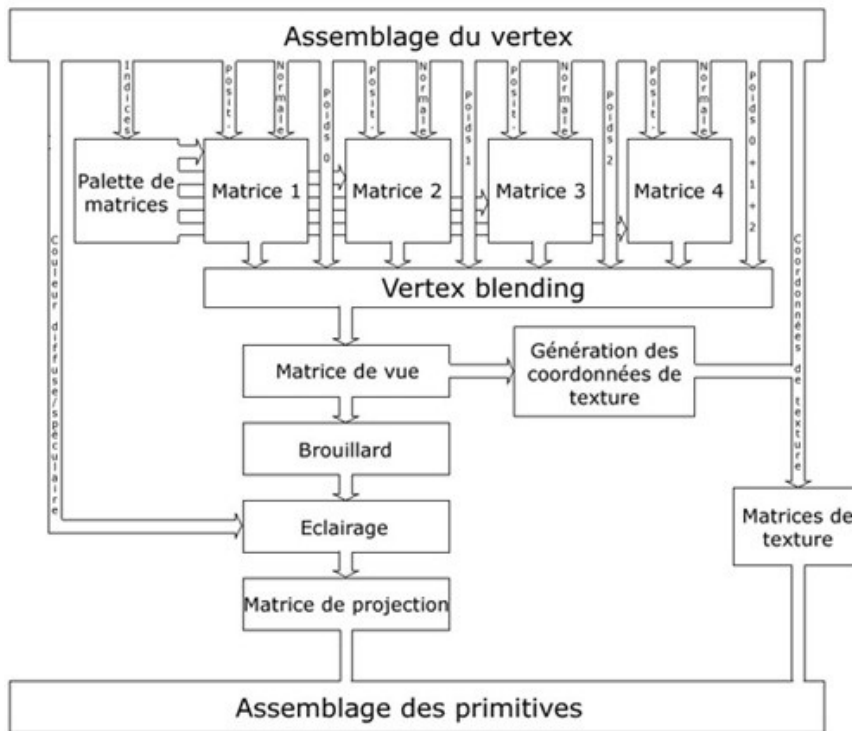
Processeur de vertex

- **But :**
 - Transformation sur les vertex
 - Rotations
 - Déplacements
 - Changements d'échelles
 - ...
 - Shaders programmables
 - Éclairage
 - Flou
 - ...



Vertex shader de réflexion/réfraction

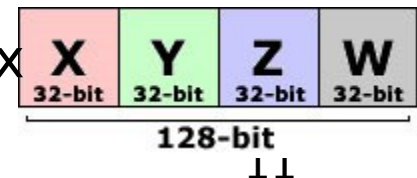
Le pipeline de vertex



Fixe (ensemble de matrices à fournir)

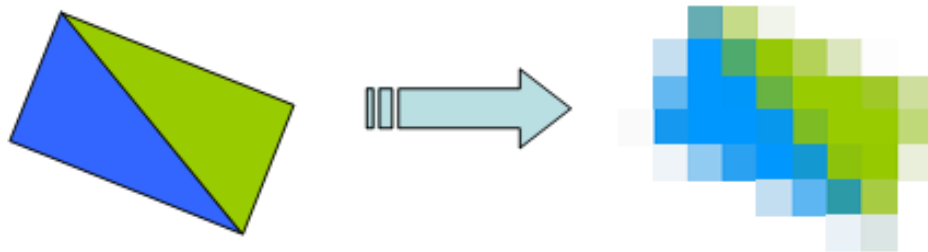
Programmable

v0 à v15 : vertex



GPU pipeline : rasterization

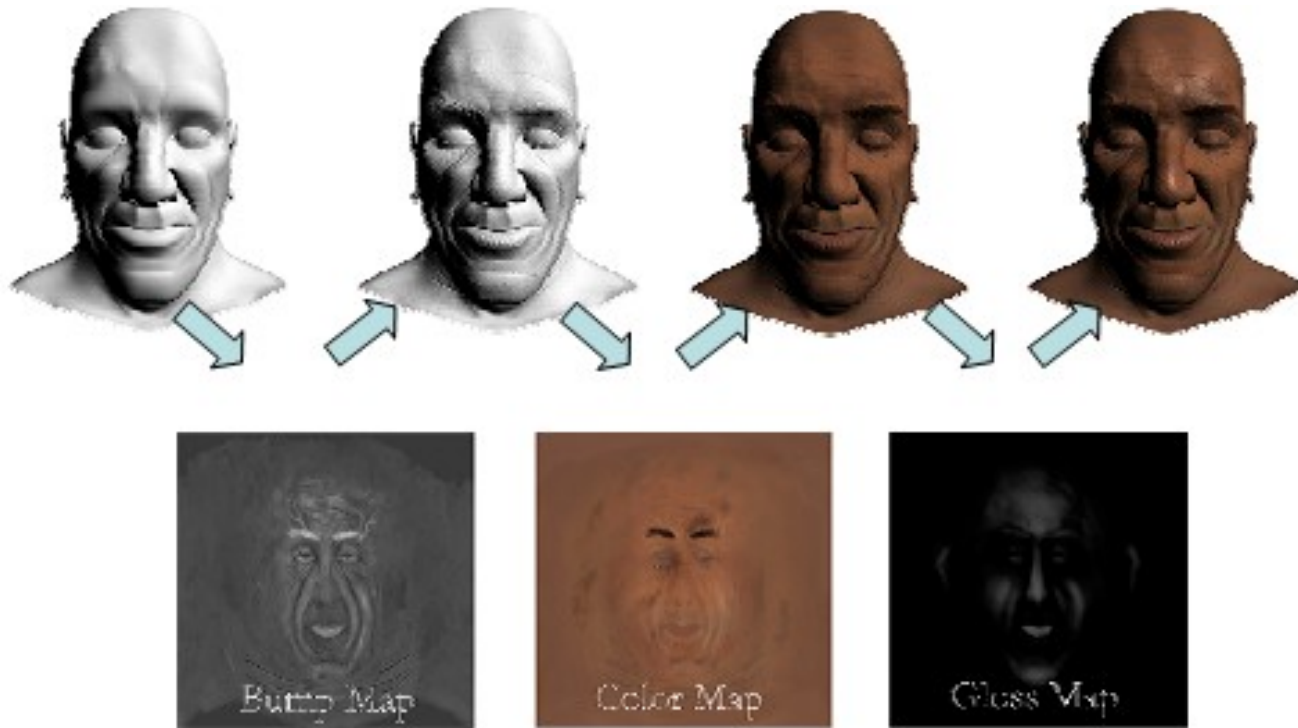
- **Génération de primitives**
 - Conversion des vertex en primitives géométriques
- **Rasterization (tramage)**
 - Génération de pixels à partir de la géométrie
 - Pixels appelés plus généralement **fragments** (pixel avec données associées, profondeur, couleur, ...)



Processeur de fragments

- **But très simple :**
 - Le pixel pipeline reçoit un pixel, ainsi que quelques informations l'accompagnant (les coordonnées des textures, par exemple)
 - Construire un pixel complet à partir de ces informations
 - Le pipeline de pixels peut être fixe ou programmable (appelé pixel shader)

Exemple



Importance du data-parallélisme

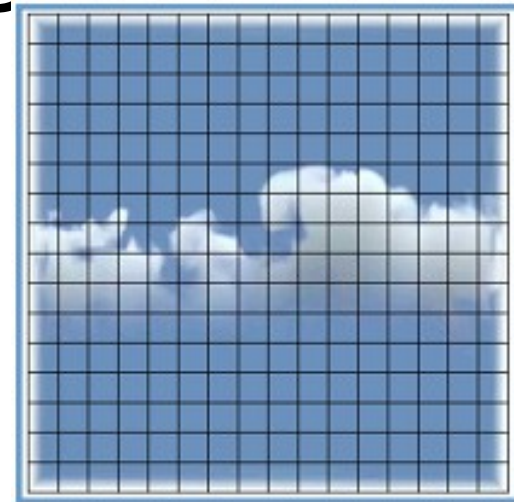
- **Les GPUs sont conçus pour le graphisme**
- **Traitement de vertex et de pixels *indépendants***
 - Pas de données statiques ou partagées
- **Traitement parallèle des données**
 - L'architecture des GPUs est basée sur une multiplications des ALUs
 - Plusieurs processeurs de vertex et pixels
 - La GeForce 8800 GTX a 128 ALUs

Applications idéales

- **Intensité calculatoire importante**
- **Les applications doivent avoir:**
 - Ensemble important de données
 - Beaucoup de parallélisme
 - Très grande indépendance entre les données à traiter

Exemple : calcul sur une grille

- **Programmation « classique » sur un GPU**
 - Les textures représentent la grille de calculs
- **Beaucoup de calculs peuvent se projeter sur une grille**
 - Calcul matriciel
 - Traitement d'images
 - Simulation physique
 - ...



Vocabulaire

- **Streams (ou grille de données)**
 - Ensemble de données nécessitant l'application du même traitement
 - A la base du parallélisme de données

- **Kernels**
 - Fonctions appliquées à chaque stream (transformations, ...)
 - Peu de dépendances entre les streams

Calcul sur GPU

➤ Calcul sur grille

- Réalisé en étapes
- Chaque étape met à jour la grille entière
- Chaque étape doit être terminée avant de passer à l'étape suivante

➤ La grille représente les stream les étapes sont les kernels



Algorithme de simulation de nuages

Ressources disponibles

- **Processeurs programmables parallèles**
 - Processeurs de vertex et de fragments
 - Ou conception unifiée (GeForce 8800 GTX, ATI Xenos)
- **Unité de texture**
 - Accès en lecture seule / écriture seule
 - Optimisée pour l'accès en 2D

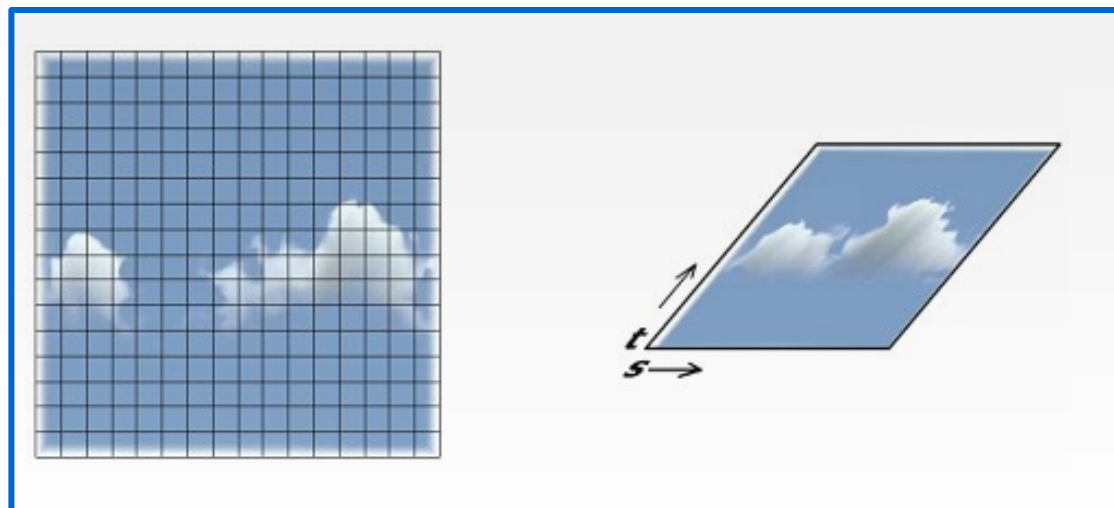
Processeur de vertex

- **Programmable (SIMD) voire MIMD**
- **Traitement de vecteurs à 4 coordonnées (RGBA / XYZW)**
- **Peut réaliser un scatter mais pas un gather**
 - Peut changer la position d'un vertex
 - Ne peut obtenir d'informations d'autres vertex
 - Peut récupérer des informations des textures

Processeur de fragments

- **Programmable en mode SIMD**
- **Traitement de vecteurs à 4 coordonnées (RGBA / XYZW)**
 - Note : la 8800 GTX est un processeur scalaire
- **Accès en lecture aux textures**
- **Capable de gather - scatter limités ?**
(mémoire partagée sur 8800 GTX)
 - Adresse de sortie fixée à une adresse
- **A priori plus utile que le processeur de vertex**
 - Plus de processeurs de fragments que de processeurs de vertex
 - Sortie directe (sortie du pipeline)

CPU – GPU analogies



CPU

GPU

- **Stream / Tableau de données = Texture**
- **Lecture en mémoire = Lecture de texture**

Kernels

advect

```
for (int j = 1; j < height - 1; ++j)
{
    for (int i = 1; i < width - 1; ++i)
    {
        // get velocity at this cell
        Vec2f v = grid(x, y);

        // trace backwards along velocity field
        float x = (i - (v.x * timestep / dx));
        float y = (j - (v.y * timestep / dy));

        grid(x, y) = grid.bilerp(x, y);
    }
}
```

C++

```
void advect(float2 uv : WPOS,
           out float4 xNew : COLOR,

           uniform float dt, // timestep
           uniform float dx, // grid scale
           uniform samplerRECT u, // velocity
           uniform samplerRECT x) // state
{
    // trace backwards along velocity field
    float2 pos = uv - dt * f2texRECT(u, uv) / dx;

    xNew = f4texRECT.bilerp(x, pos);
}
```

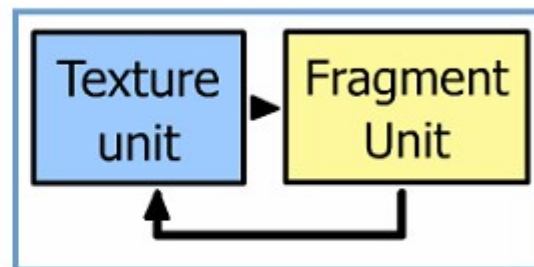
Cg

Kernel / Étape algorithmique = programmation du pixel shader

Retour de résultat

```

    .
    .
    Grid[i][j] = x;
    .
    .
    .
  
```

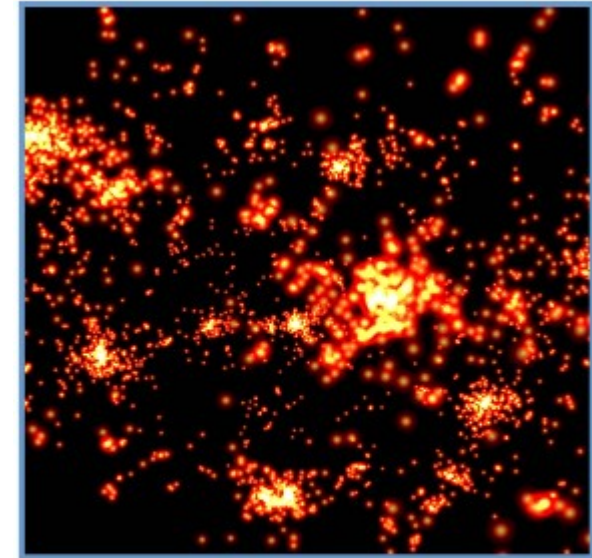


Écriture du résultat = mise à jour de textures

Exemple N-body simulation

- $N = 8192$ corps
- N^2 forces de gravitation
- 64 M forces/frame
- ~ 25 flops par force

- GeForce 8800 GTX
 - Précision réel 16 bits :
73 fps, **122,5 Gflops**
 - Précision réel 32 bits :
39 fps, **65,4 Gflops**



*Nyland, Harris, Prins,
 GP² 2004 poster*

Calcul des forces gravitationnelles

- **Chaque corps attire tous les autres corps**
 - N corps donc N^2 forces
- **Tampon de taille NxN**
 - Pixel(i,j) calcule la force entre le corps i et j
 - Programme de fragment (pixel shader) très simple
 - Limité par la taille maximale des textures

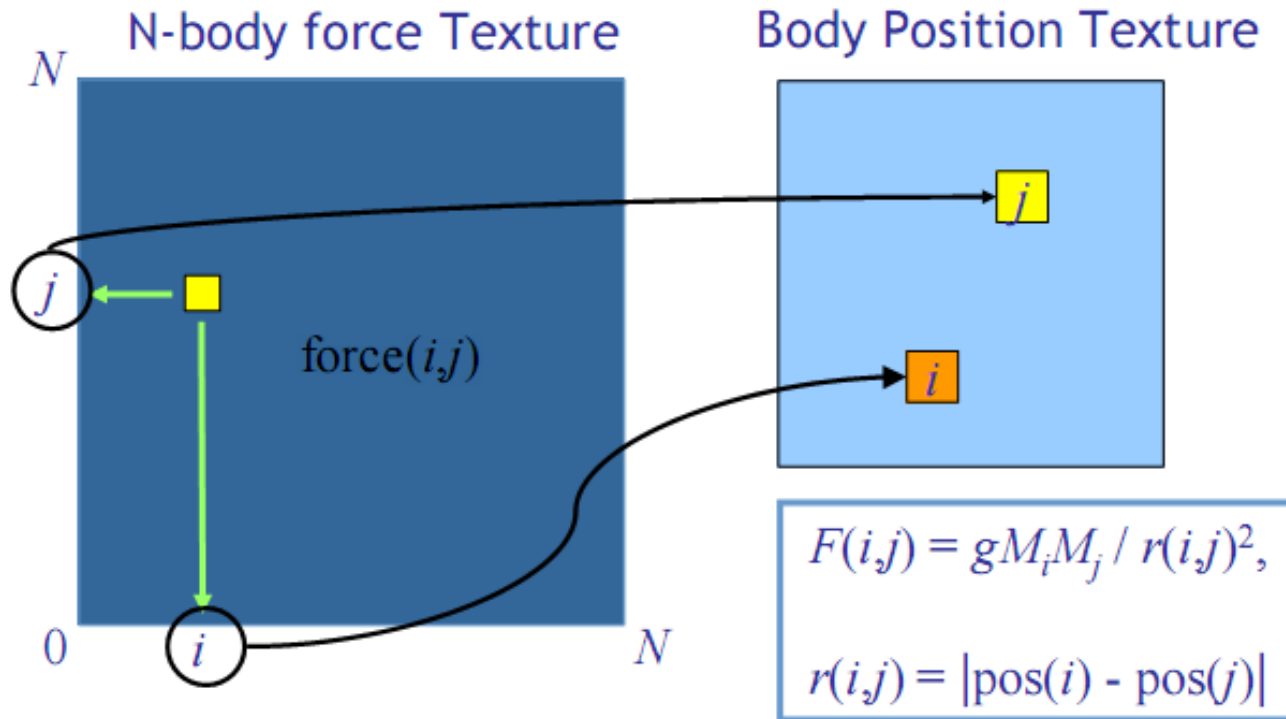
Calcul des forces gravitationnelles

$$F(i,j) = gM_iM_j / r(i,j)^2,$$

$$r(i,j) = |\text{pos}(i) - \text{pos}(j)|$$

La force est proportionnelle à l'inverse du carré de la distance entre les corps

Calcul des forces gravitationnelles



Les coordonnées (i,j) dans la texture gérant les forces sont utilisées pour récupérer la position des corps i et j

Calcul des forces gravitationnelles

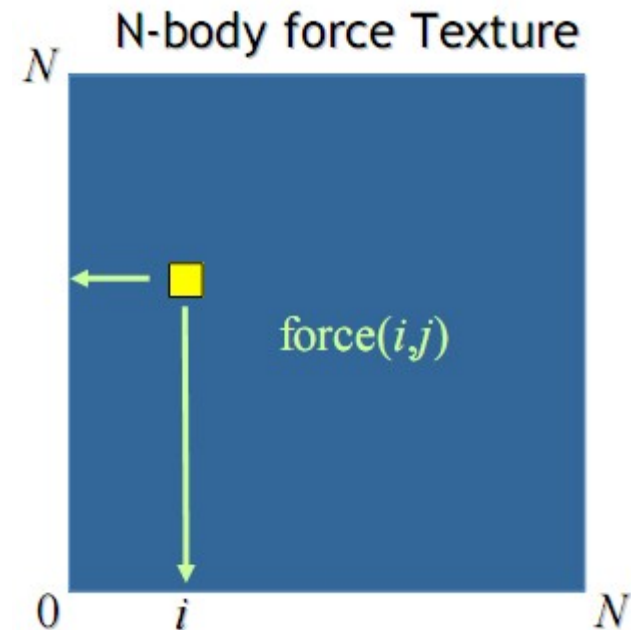
```

float4 force(float2 ij      : WPOS,
             uniform sampler2D pos) : COLOR0
{
    // Pos texture is 2D, not 1D, so we need to
    // convert body index into 2D coords for pos tex
    float4 iCoords = getBodyCoords(ij);
    float4 iPosMass = texture2D(pos, iCoords.xy);
    float4 jPosMass = texture2D(pos, iCoords.zw);
    float3 dir = iPos.xyz - jPos.xyz;
    float r2 = dot(dir, dir);
    dir = normalize(dir);
    return dir * g * iPosMass.w * jPosMass.w / r2;
}

```

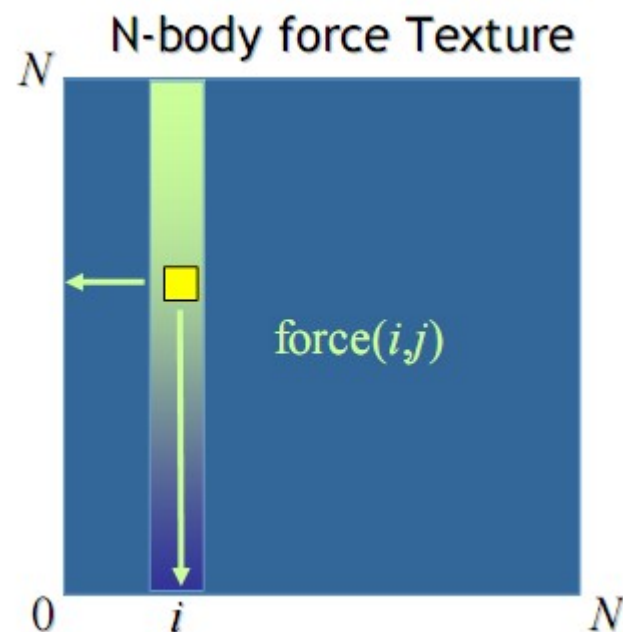
Calcul de la force

- On dispose du tableau de force (i,j)
- On souhaite calculer la force totale soumise sur la particule i
-



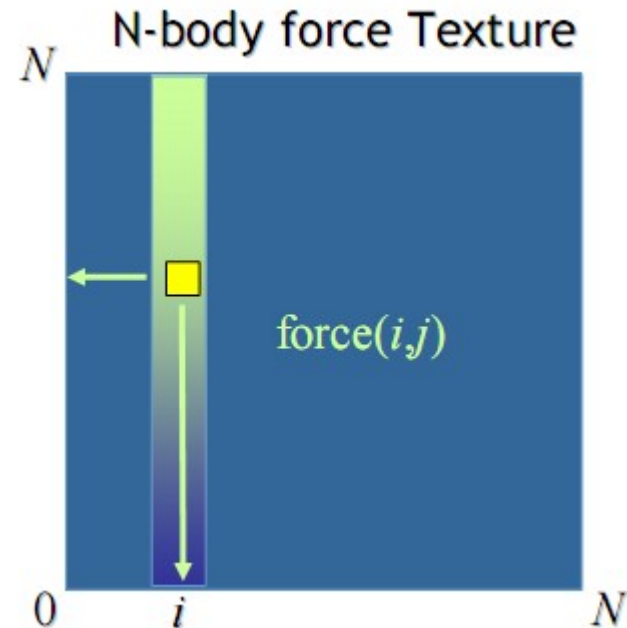
Calcul de la force

- **On dispose du tableau de force (i,j)**
- **On souhaite calculer la force totale soumise sur la particule i**
 - Sommation sur chaque colonne
 -



Calcul de la force

- **On dispose du tableau de force (i,j)**
- **On souhaite calculer la force totale soumise sur la particule i**
 - Sommation sur chaque colonne
 - Réduction parallèle
 -



Mise à jour des vitesses et positions

- **Nous avons un tableau 1D de forces**
 - Une par corps
- **Mise à jour de la vitesse**
 - $u(i,t+dt) = u(i,t) + F_{\text{total}}(i)*dt$
 - Simple programme du pixel shader
- **Mise à jour de la position**
 - $x(i,t+dt) = x(i,t) + u(i,t)*dt$
 - Simple programme du pixel shader qui lit la position précédente et la vitesse et met à jour la nouvelle position

Langages de GPU de haut niveau

Langages de shader

- **Cg, HLSL, OpenGL Shading Language**
 - Cg :
 - www.NVIDIA.com/cg
 - HLSL :
 - msdn.microsoft.com/library/default.asp?url=/library/enus/directx9_c/directx/graphics/reference/highlevellanguageshaders.asp
 - OpenGL Shading Language :
 - www.3dlabs.com/support/developer/ogl2/whitepapers/index.html

Langages de GPGPU

- **Pourquoi ?**
 - Rendre la programmation des GPU plus simples
 - Connaissance d'OpenGL, DirectX,... inutiles
 - Simplification des opérations communes
 - Travail sur l'algorithme et non sur l'implémentation

Quelques exemples

- Accelerator
 - Microsoft Research
- Brook
 - Stanford University
- CTM
 - ATI/AMD
- CUDA
 - NVIDIA
- Peakstream
- RapidMind
 - Version commerciale améliorée de sh

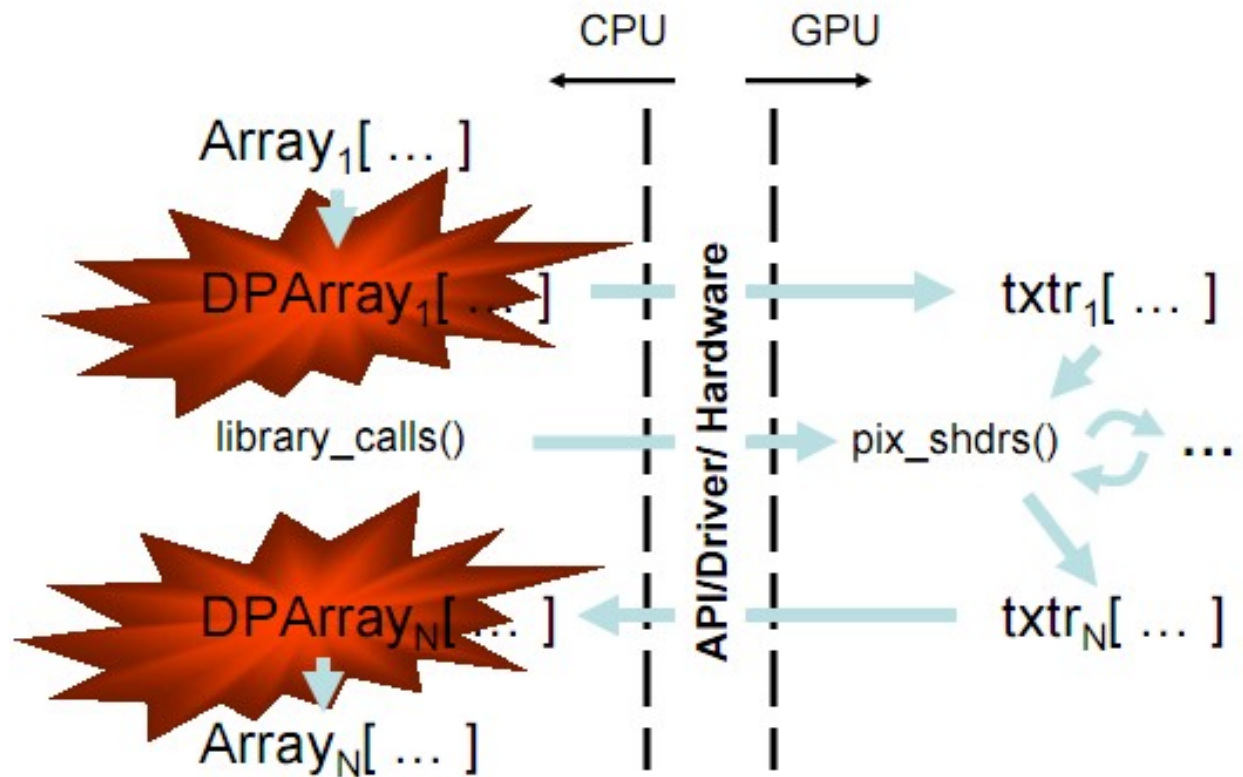
Accelerator

- **Programmation data-parallèle du GPU**
- **Librairie data-parallèle à la disposition du programmeur**
 - Simple, haut niveau de programmation
- **Compilateur JIT pour le CPU ou le GPU**
 - Fonctionne avec .NET

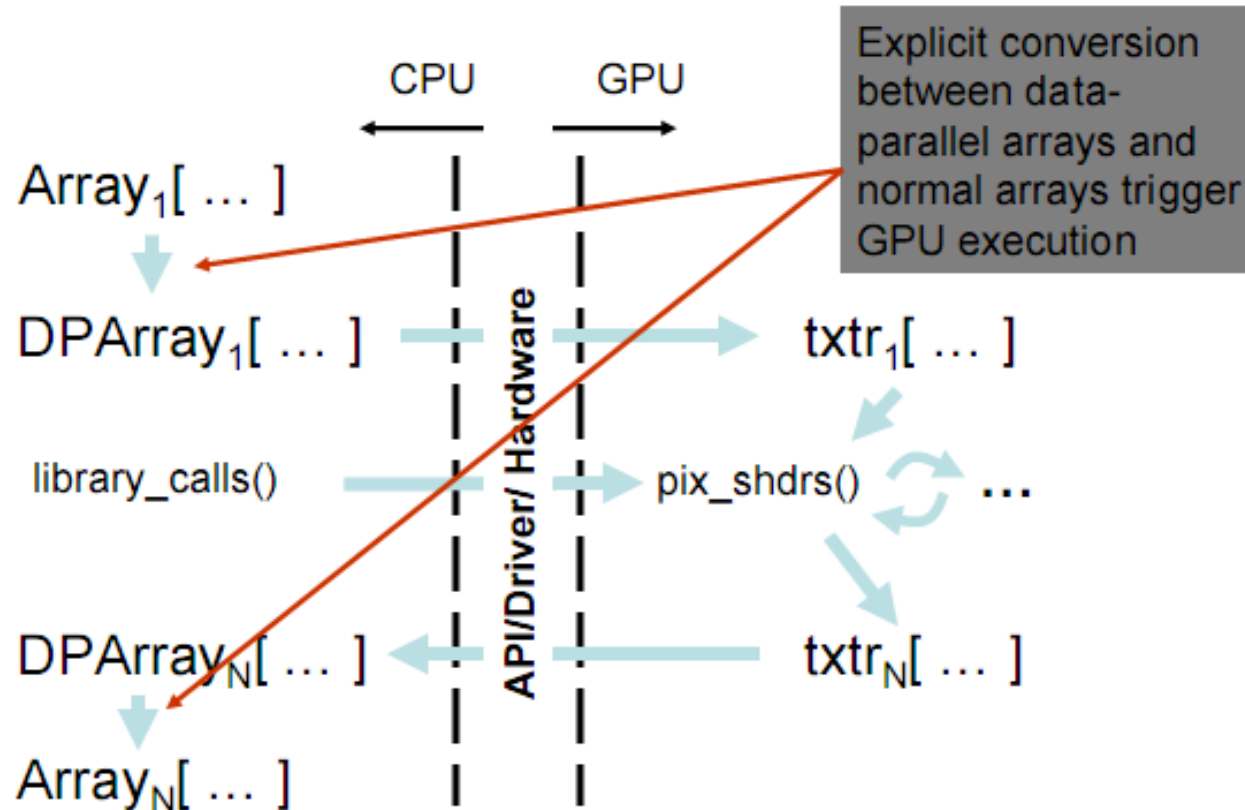
Bibliothèque data-parallèle

- **Conversion explicite entre tableaux DP et tableaux « classiques »**
- **Chaque opération produit un nouveau tableau DP**
- **Pas de pointeurs et d'accès aux éléments individuels**

Tableaux data-parallèles



Conversion explicite



CUDA et calcul sur GPU

Présentation

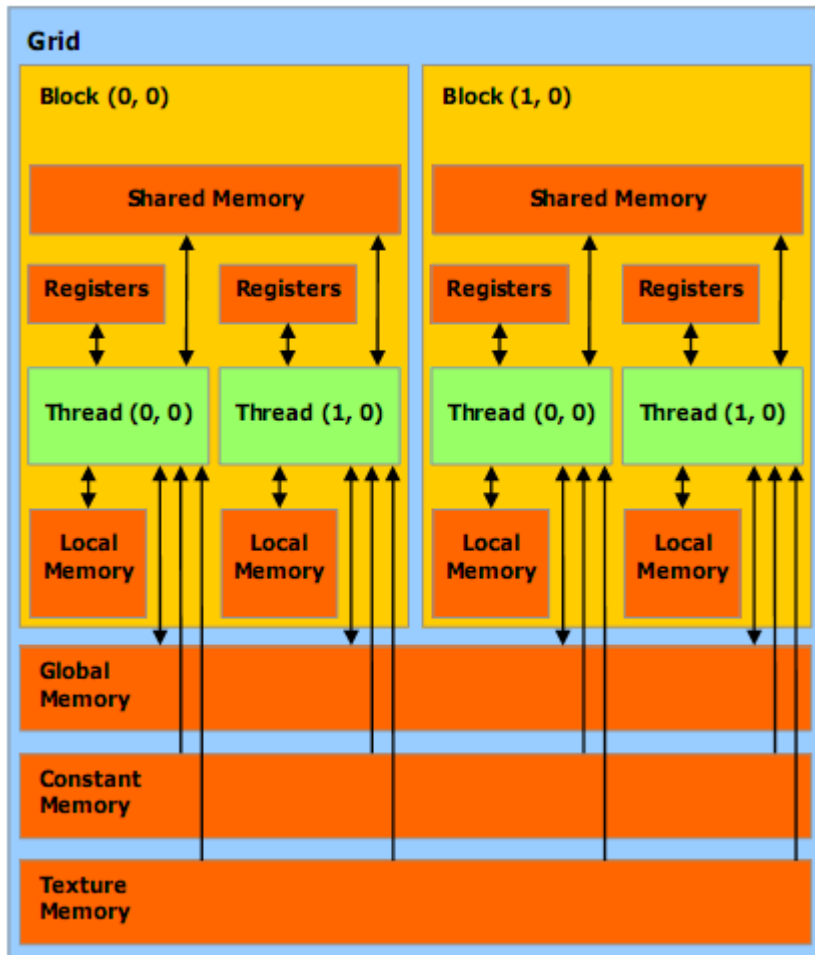
- **Programmation data-parallèle basée sur un système de threads**
 - Plusieurs milliers de threads possibles
- **Cache de données data-parallèle**
- **Programmation en C**
- **Bibliothèques d'algèbre linéaire et de FFT disponibles**
- **Utilisable uniquement sur GPU de type nvidia**

Exécution basée sur les threads

Programme de
threads

- Très grand nombre d'instructions
- Pas de limitation sur les alternatives et les boucles
- Allocation des threads suivant une logique 1D, 2D ou 3D
- Regroupement des threads en groupes logiques appelés blocs
 - Mémoire partagée
 - Les blocs sont exécutés en parallèle si des grilles de processeurs sont disponibles

Les blocs



- **Les blocs sont exécutés en parallèles si des grilles de processeurs sont disponibles**
- **Aucun ordre sur les blocs n'est garanti**

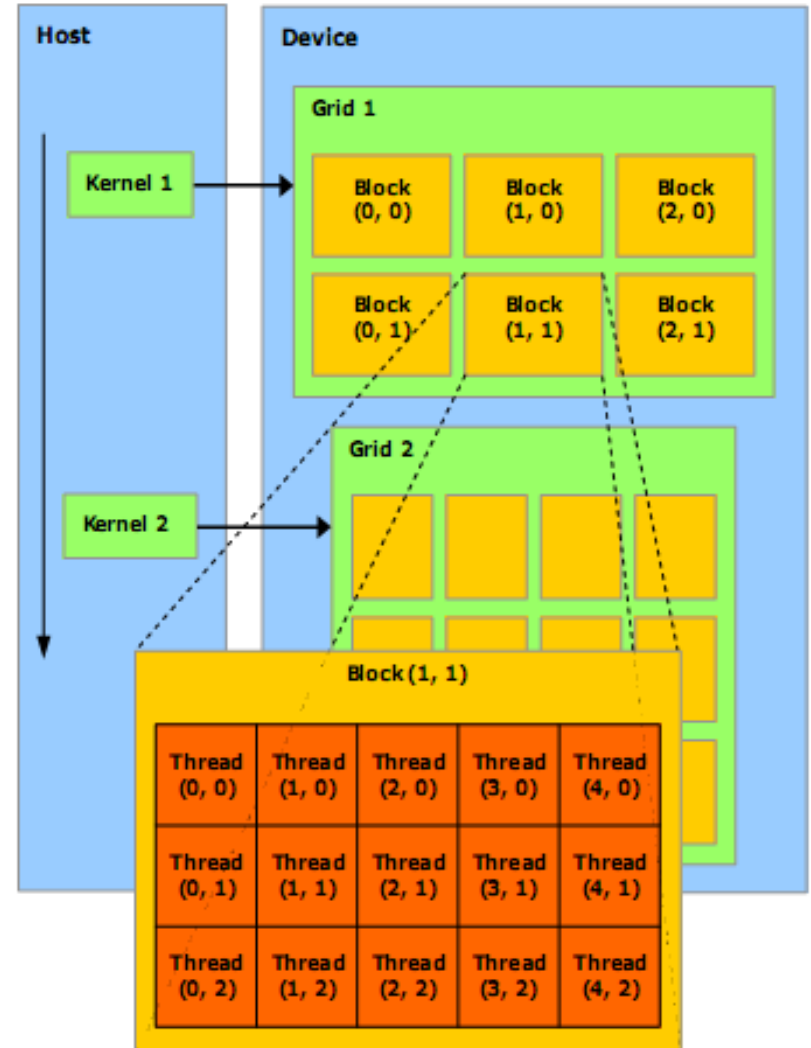
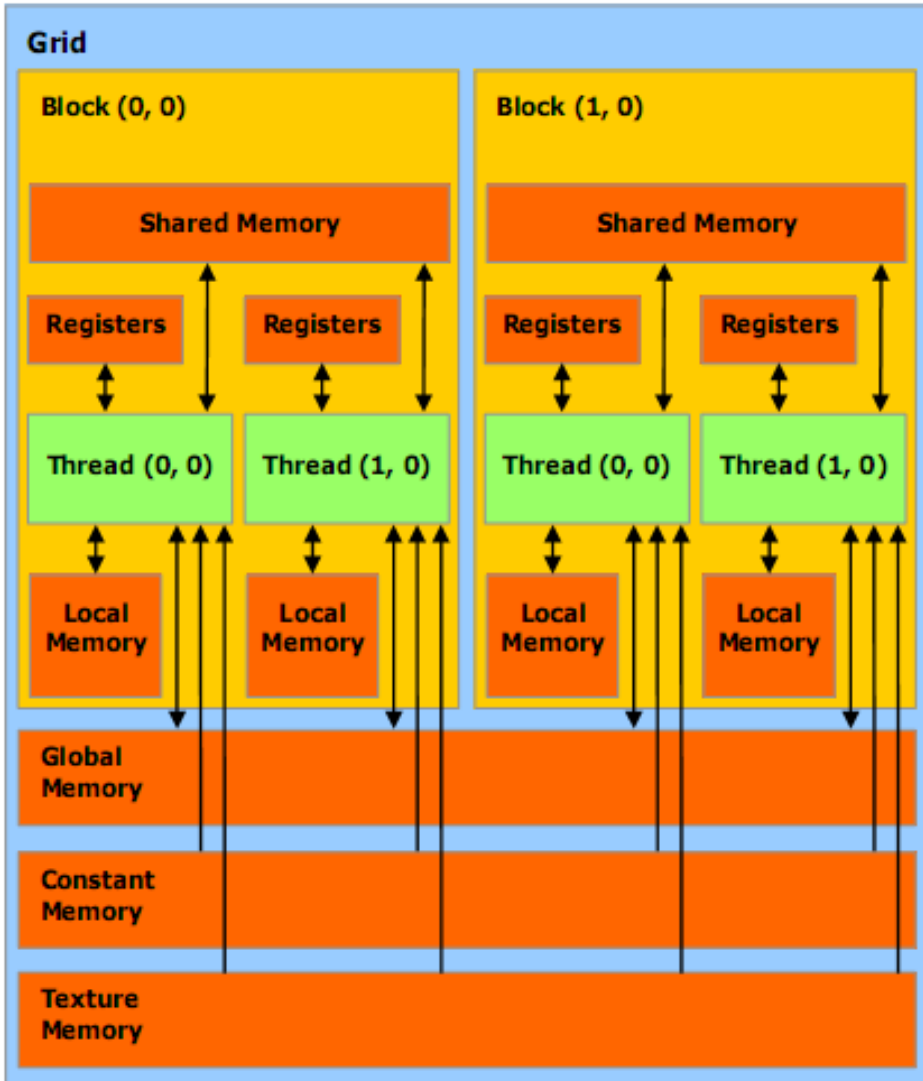
Mémoire partagée

- **Mémoire dédiée**
- **Partagée entre les threads et utilisée pour la communication inter-threads**
- **Manipulée de manière explicite**
- **Aussi rapide que les registres**

Mémoire globale

- **Mémoire générale du GPU : allocation – libération de mémoire**
- **Non typée (non limitée à la manipulation de texture)**
- **Support des pointeurs**

Modèle d'exécution



Exécution sur un GeForce 8800 GTX

- **768 Mo disponible**
- **16 grilles de 8 processeurs**
 - 128 processeurs au total à 1,35 Ghz (675 Mhz)
 - 16 ko par grille de mémoire partagée
 - 32 threads minimum par bloc
 - Cache de texture de 8 ko par grille
 - À partir de 16 blocs, le parallélisme maximum est atteint (pour moins de 16 blocs, certaines grilles seront inoccupées)

Programmation C du GPU

- **Assez simple à programmer, mais nécessite une connaissance approfondie pour une accélération optimale**
- **Extension de certains mots-clefs**
 - `__global__` void Kernelfunc(...);
 - `__device__` int Globalvar;
 - `__shared__` int Sharedvar;
 - KernelFunc <<<500,128 >>>(...);

Mesure de temps sur GPU

➤ **Exemple simple**

- Utilisation d'une ou plusieurs grilles de multi-processeurs ;
- Un fichier d'appel `clock.cu` exécuté sur le CPU ;
- Un fichier du kernel `clock_kernel.cu` exécuté sur le GPU

GPU et algorithmes évolutionnaires

- **Wong ML, Wong TT, Fok KL:** *Parallel evolutionary algorithms on GPU*, **Proceedings of IEEE CEC 2005, vol 3, pp 2286-2293**
- **Wong ML, Wong TT, Fok KL:** *Evolutionary computing on consumer-level graphics hardware*, **IEEE Intelligent Systems (to appear)**
- **Yu Q, Chen C., Pan Z. :** *Parallel Genetic Algorithms on Programmable Graphics Hardware*, **LNCS 3612, 2005**
- **Ebner M., Reinhardt M., Albert J.:** *Evolution of vertex and pixel shaders*, **EuroGP 2005, pp 261-270**
- **Harding S., Banzhaf W.:** *Fast Genetic Programming on GPUs*, **EuroGP 2007, LNCS 4445, pp 90-101**

GP sur GPU

- **Une unique (à ma connaissance) contribution partielle. Harding et Banzhaf 2007**
- **Idée relativement simple**
- **Les arbres GP sont évalués pour chaque élément du jeu d'entraînement**
 - Application parallèle du jeu d'entraînement sur le GPU
 - Pas de moteur d'évolution, l'évaluation de programmes aléatoires est réalisée sur le GPU
 - Le tableau de jeu d'entraînement est transformé en texture et chargé sur le GPU
 - Après l'évaluation, le résultat est transféré du GPU vers le CPU

Expériences

- **Expériences réalisées uniquement sur la phase d'évaluation**
- **Langage Accelerator™, parser GP écrit en C# et compilé avec Visual Studio 2005**
- **GPU Nvidia GeForce 7300 GO (512 Mo)**
- **CPU Intel Centrino T2400 @ 1,83 Ghz**
- **100 expressions générées aléatoirement pour calculer le facteur d'accélération (arbres a priori)**
- **Le programme du pixel shader correspond à un arbre**

Fonction $x^6 - 2x^4 + x^3$

	Nombre de cas de tests			
Longueur Max	10	100	1000	2000
10	0,02	0,08	0,7	1,22
100	0,07	0,33	2,79	5,16
1000	0,42	1,71	15,29	87,02
10000	0,4	1,79	16,25	95,37

Facteur d'accélération



Classification (2 spirales)

Jeu d'entraînement				
Taille de l'expression	194	388	970	1940
10	0,15	0,23	0,51	1,01
100	0,38	0,67	1,63	3,01
1000	1,77	3,19	9,21	22,7
10000	1,69	3,21	8,94	22,38

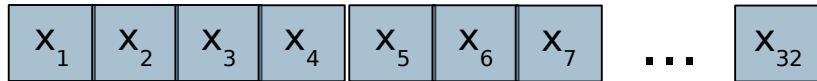
Conclusion

- **Peu de gains pour des jeux d'entraînement de petite taille (surcoût de transfert non négligeable)**
- **Pour des jeux de taille importante, le gain est conséquent**
- **Question ouverte : A-t-on souvent en GP l'occasion de tester des jeux d'entraînement de 10000 cas ou plus?**
- **Coût de la compilation ?**

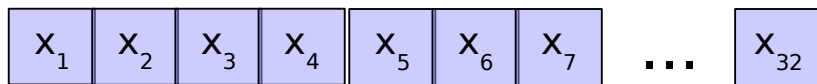
Travaux de Fok, Wong et Wong

- **Il s'agit de programmation évolutionnaire sur GPU**
- **Les individus sont chargés dans les textures**
- **Pas de recombinaison, mutation indépendante sur chaque individu**

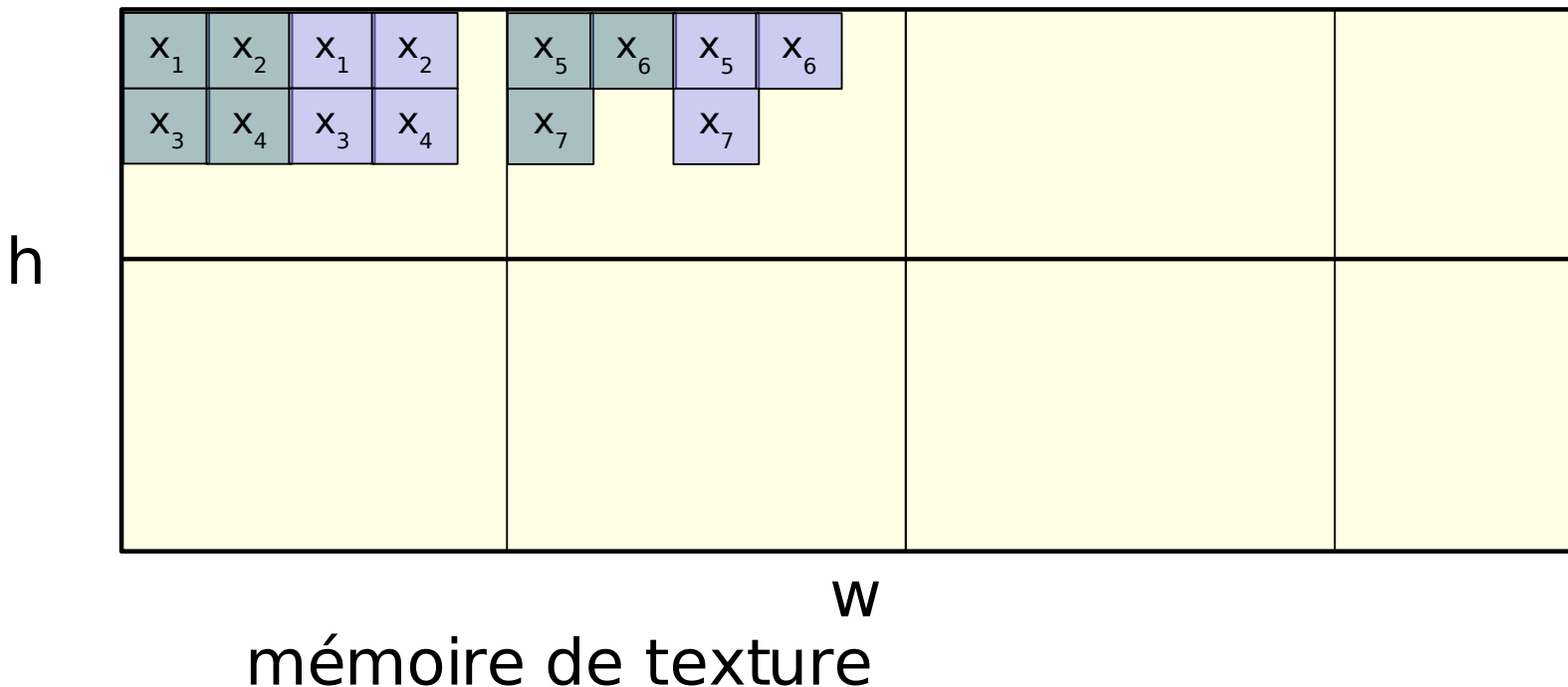
Représentation des individus



un individu de 32 gènes correspond à 8 pixels (r,g,b, α)



La mémoire est mieux utilisée avec des multiples de 4 (stockage r,g,b, α)



Réalisation de la mutation

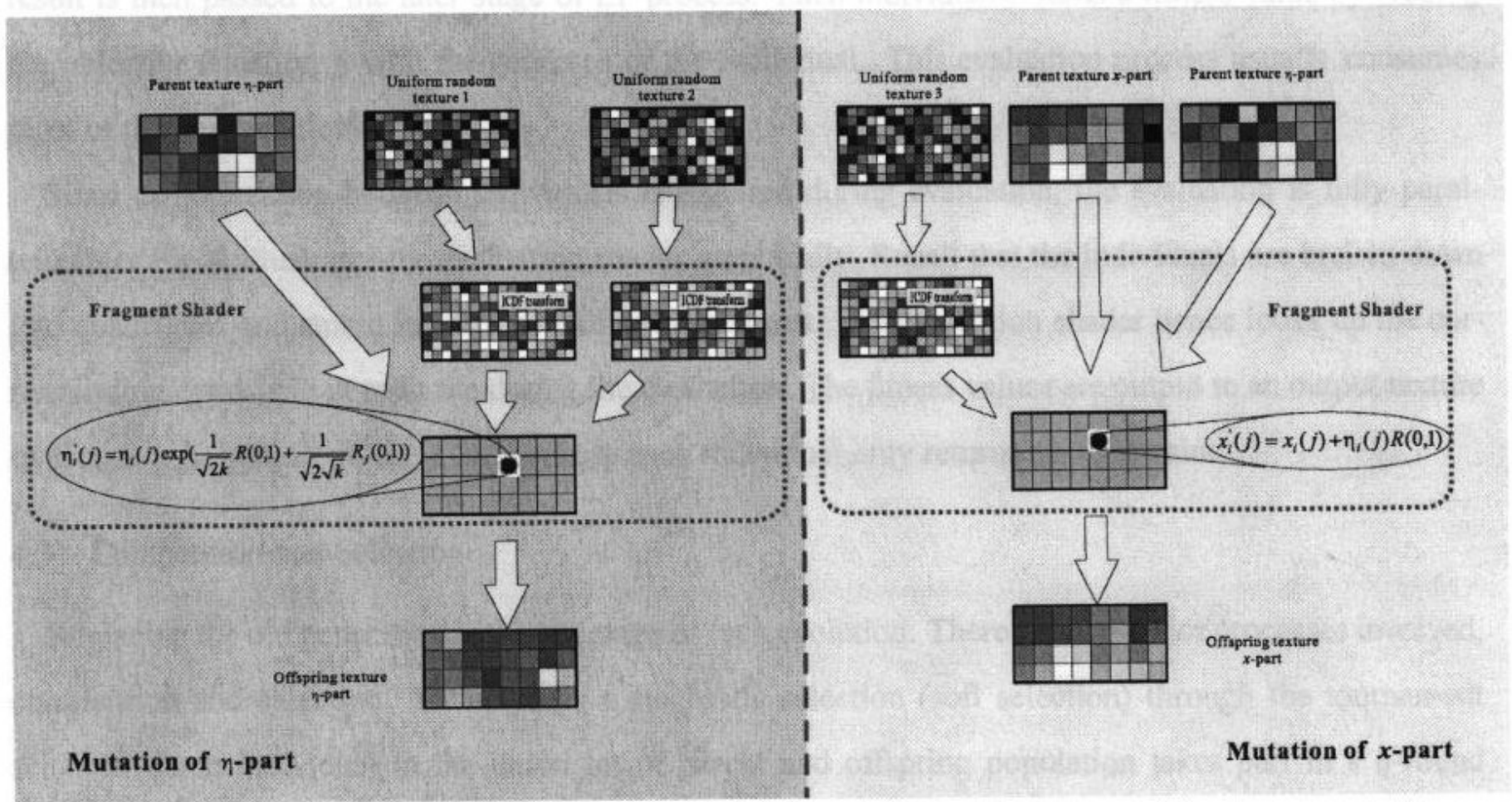
- **Mutation totalement parallélisable**
- **Pour chaque individu x**

$$x'(j) = x(j) + \eta(j)R(0,1)$$

$$\eta'(j) = \eta(j) \exp\left(\frac{1}{\sqrt{(2k)}} R_j(0,1)\right)$$

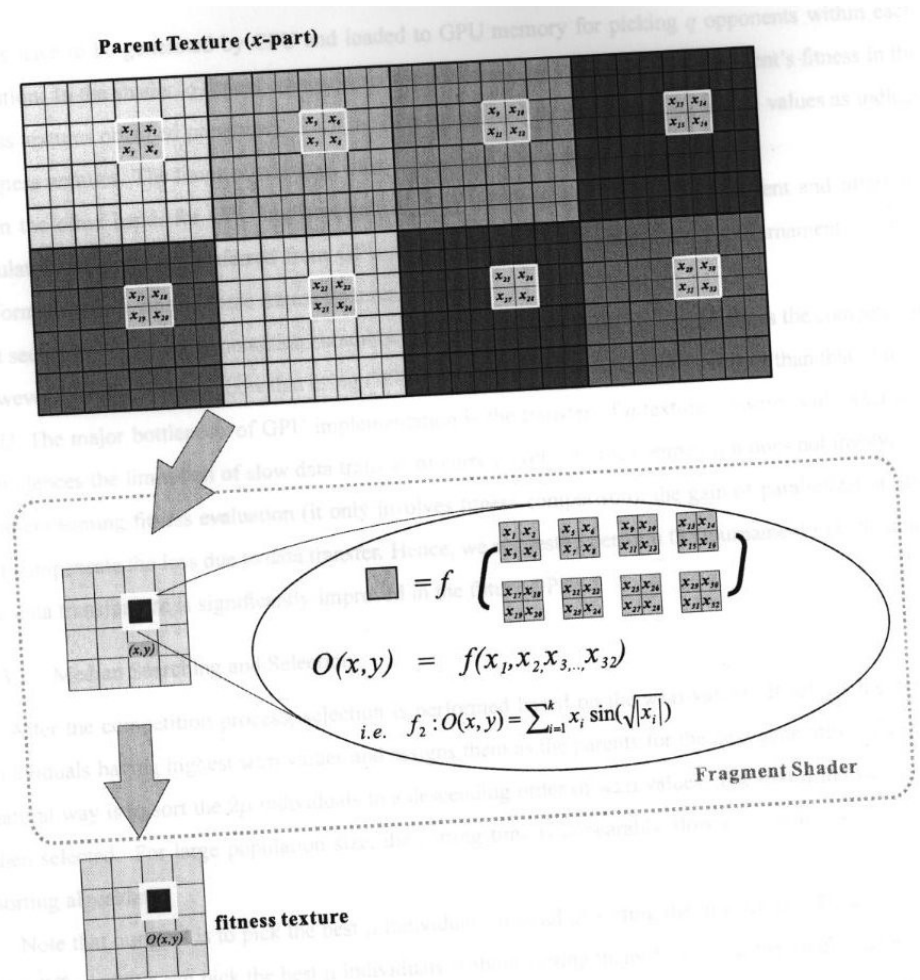
- **Deux programmes de fragment**
 - un pour générer η'
 - un pour générer x'

Visualisation



Évaluation du fitness

- Réalisation totalement parallèle de l'évaluation
- Le programme de shader calcule le déplacement correct pour le calcul du fitness



Compétition et sélection

- **Sélection stochastique à travers un tournoi entre les individus de la nouvelle génération et les parents**
- **Compétition avec q individus générés aléatoirement**
- **Ces deux opérations sont réalisées sur le CPU**
 - pas de fonction aléatoire sur le GPU
 - transfert coûteux entre le CPU et le GPU

Résultats

μ	GPU	CPU
800	1	2,02
3200	2,02	8,37
6400	3,04	16,75

Expériences sur Pentium IV 2,4 Ghz
GeForce 6800 Ultra avec 256 Mo

Temps d'exécution par rapport à une
exécution avec 400 individus

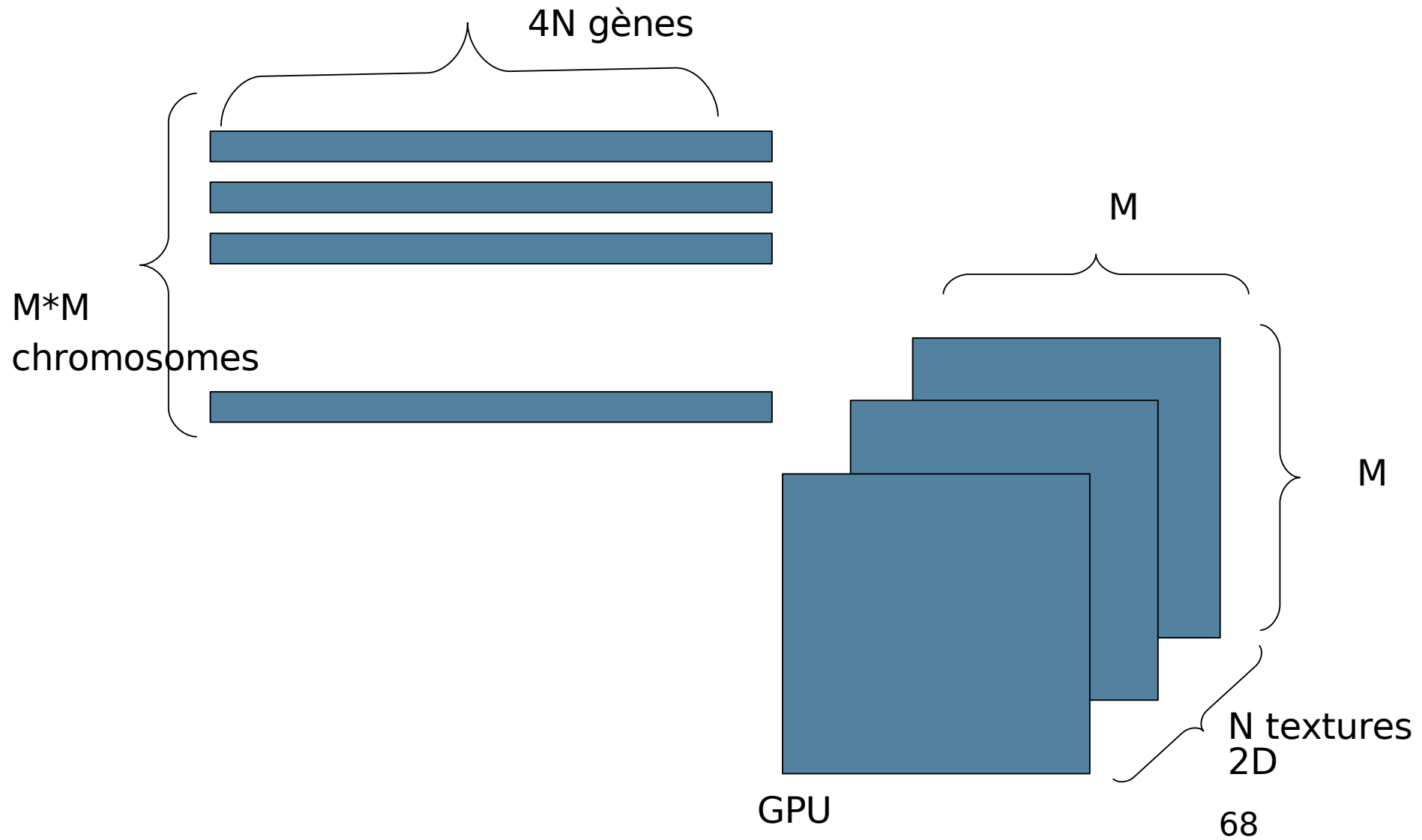
Travaux de Yu, Chen, Pan

- **Travaux moins avancés que le précédent**
- **Évaluation parallèle du fitness**
- **Génération des nombres aléatoires à partir d'une matrice de texture aléatoire**
- **Réalisation de la recombinaison et de la mutation sur le GPU**

Représentation de la population

- **4 gènes = 1 pixel au format (r,g,b,α)**
- **1 grille 2D de texture correspond aux quatre premiers gènes de chaque individu**
- **Il y a donc $n/4$ grilles de texture 2D (pour une population de n individus)**
- **Une grille supplémentaire est utilisée pour le fitness**

Visualisation



Quelques résultats

$$(1-x_2)^2 + (1-x_2^2) + 19.8(x_2-1)(x_4-1)$$

$$f(\bar{x}) = 100(x_1^2 - x_2)^2 + (1-x_1)^2 + 90(x_3^2 - x_4)^2 + 10.1i$$

Taille pop	Opérations		GAIN
	génétiques	Fitness	
32^2	1,4	0,3	
64^2	5,8	1,4	
128^2	11,8	7,9	
256^2	17,9	15,4	
512^2	20,1	17,1	