M1 - Aquisition, Analyse et Traitement d'Images

OpenGL et compute shaders

J.C. Iehl

April 23, 2014

Résumé des épisodes précédents

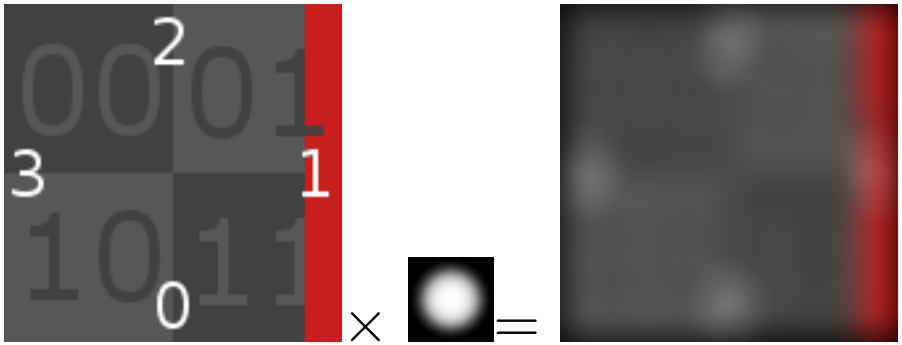
programmation GPU :

- GPU ?
- API 3D,
- pipeline graphique, pipeline calcul,
- shaders + GLSL,
- parallélisme de données,
- espace d'itération, threads et tache associée.

Exemple précédent : filtrage par convolution

rapide ?

- cpu 200ms, gpu 1ms
- peut mieux faire ?



Exemple précédent : filtrage par convolution

peut mieux faire ?

- complexité de l'algorithme ?
- combien de fois est lue l'image décrivant le filtre ?
- combien de fois est lue l'image ?
- combien d'opérations sont redondantes pour traiter des pixels voisins ? dans les threads d'un meme groupe ?

Exemple précédent : filtrage par convolution

filtre 32×32 :

- = 1024, l'image est lue 1024 fois...

test : image 4M pixels, filtrée en 150ms

- si on remplace la lecture d'une valeur du filtre par une constante : 100ms
- si on remplace la lecture d'un pixel de l'image par une constante : 60ms
- si on remplace les 2 lectures par une constante : 40ms

Exemple précédent : filtrage par convolution

quel est le facteur limitant ?

- les calculs,
- la lecture du filtre,
- la lecture de l'image ?

cf cours de complexite de licence...

Exercice : ecrire un test pour identifier le facteur limitant

par exemple :

- remplacer la lecture des coefficients du filtre par un calcul :
- le filtre est une gaussienne, il suffit de l'évaluer.

résultat attendu :

- soit le temps d'exécution augmente,
- soit ...
- il diminue,
- ou reste stable.

Filtre gaussien

$$= f(x,y) = g(x)g(y)$$

$$g(x,\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

Filtre gaussien

résulat :

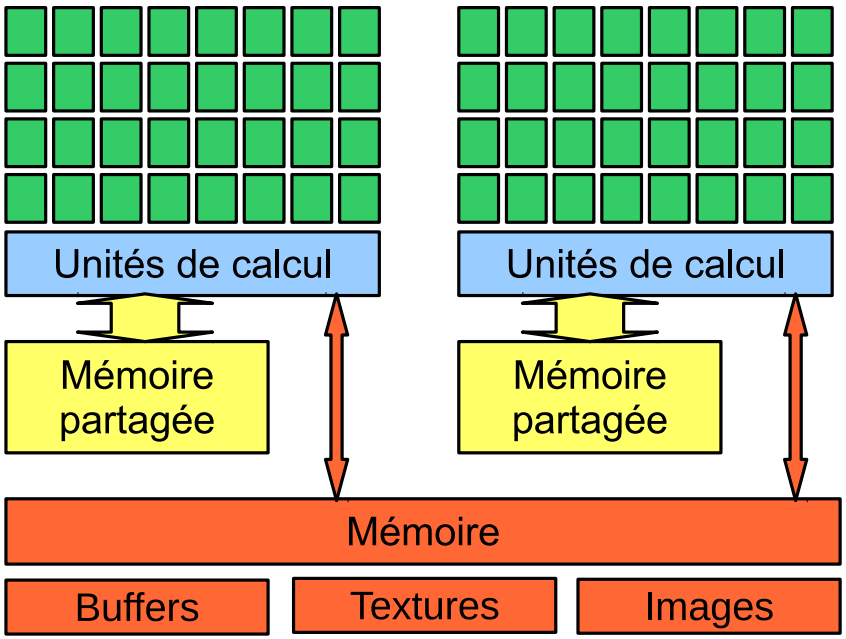
- 145ms,
- le temps d'execution ne change pas...
- donc ce ne sont pas les calculs qui ralentissent l'execution du programme...

mais : ce n'est pas gratuit non plus...
(cf 100ms si on remplace la lecture du filtre par une constante)

Architecture

les threads sont groupés :

- ils partagent des ressources...



Mémoire partagée ?

chaque processeur :

- est composé de plusieurs unités de calcul (1 thread),
- et d'une (petite $\approx$ 64Ko) zone de mémoire partagée...
- accessible rapidement par tous les threads d'un groupe,
- en lecture,
- et en écriture (synchronisation ?)...

Mémoire partagée

- plus rapide d'accès que la mémoire centrale,
- permet une "communication" entre threads.

communication ?

- les threads ne sont pas obligés de tous faire exactement la même chose !
- calculer une partie du resultat,
- échanger les valeurs intermédiaires,
- finir le calcul.

Utilisation en GLSL

GLSL :

- déclarer un tableau global avec le mot clé `shared`.

```
#version 430

shared float weights[32];

layout( local_size_x= 16, local_size_y= 16 ) in
void main( )
{

}
```

Exercice

la fonction $g(x, \mu, \sigma)$:

- est évaluée par tous les threads / taches,
- pour les mêmes valeurs de x et y,
- calculer une seule fois chaque valeur $g(x)$, $g(y)$...
- et partager les résultats.

quelle taille de mémoire partagée ? (pour un filtre 32x32, par exemple)

Exercice

comment répartir les calculs ?

- décomposer l'itération sur le filtre...
- et affecter une évaluation de $g(x)$, $g(y)$ à chaque thread,
- écrire le résultat dans un tableau partagé,
- attendre,
- finir le calcul / filtrage de l'image utilisant les valeurs $g(x)$, $g(y)$ stockées dans le tableau partagé.

comment répartir l'évaluation du filtre ?

qui fait quoi ?

- à vous de choisir :
- par exemple, le thread i calcule $g(i)$
- et écrit $g(i)$ dans le tableau partagé $weights[i]$,
- que font les autres threads (lorsque $i >$ support filtre) ?

attendre :

- barrier()

rappel: `gl_LocalInvocationID`

Exemple :

```
#version 430
#define PI 3.1415
float square( const float x ) { return x*x; }
float g( const float x, const float mean, const float var ) {
    return exp( - square(x - mean) / (2.0*var) ) / sqrt(2.0*PI*var); }

// tableau partage, 1 valeur par thread du groupe
shared float weights[32];

// nombre de threads par groupe
layout( local_size_x= 32, local_size_y= 16 ) in;
void main( ) {
    for(int i= 0 ; i < 32; i++) {
        vec3 gy= g(i);

        for(int j= 0 ; j < 32; j++) {
            vec3 gx= g(j);
            // filtre, a modifier pour utiliser la memoire partagee
            vec3 c= gx * gy;

            // lire un pixel dans l'image
            vec3 p= texelFetch(...).rgb
            {...}
        }
    }
    imageStore(...);     // ecrire le resultat
}
```

et alors ?

bilan :

- 100ms
- le plus interressant est bien sur de faire la même chose pour limiter le nombre de lectures de l'image en cours de traitement...

l'acces mémoire est bien le plus gros facteur limitant dans cet exemple. donc il faut éliminer les lectures "inutiles" et exploiter la mémoire partagée.

Exemple :

```
#version 430
#define PI 3.1415
float square( const float x ) { return x*x; }
float g( const float x, const float mean, const float var ) {
  return exp( - square(x - mean) / (2.0*var) ) / sqrt(2.0*PI*var); }

// tableau partage, 1 valeur par thread du groupe
shared float weights[32];

// nombre de threads par groupe
layout( local_size_x= 32, local_size_y= 16 ) in;
void main( ) {
  vec3 r;
  for(int i= 0; i < window; i++) {
    vec3 gy= g(i);

    if(gl_LocalInvocationID.y == 0) // calcul intermediaire
      weights[gl_LocalInvocationID.x]= g(gl_LocalInvocationID.x);
    barrier();          // tous les threads attendent

    for(int j= 0; j < window; j++) {
      vec3 f= texelFetch(....).rgb;
      float c= weights[j] * gy;
      r+= c * f;
    }
  }
  imageStore(..., r);    // ecrire le resultat
}
```