Initiation à la GPGPU

Introduction

David Odin

Forma3Dev pour CPE–Lyon

2012

Qu'est-ce que la GPGPU ?

- **G**eneral **P**rograming on **G**raphic **P**rocessing **U**nits.
- Utilisation de la puissance des cartes graphiques pour autre choses qu'afficher des triangles.

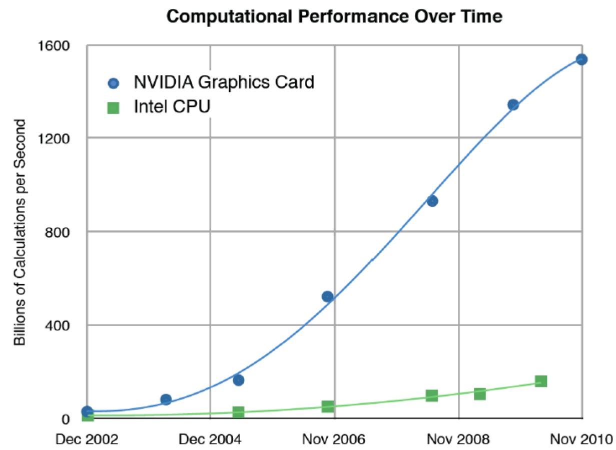
Programmation parallèle

- Idée assez vieille, au moins 30 ans (transputers)
- Implémenté via MMX, SSE sur les processeurs *x86* depuis des années.
- Activec sur PowerPC, Paired Single sur Wii, etc.
- **S**ingle **I**nstruction on **M**ultiple **D**ata.

Pourquoi sur GPU ?

- Les GPU sont SIMD par nature (*fragment shader*).
- Shaders unifiés.
- GPGPU commence à devenir mature (standardisation).
- CPU = 24 cœurs d'exécutions max.
- GPU = 3200 cœurs pour la AMD Radeon HD 5970 (comparable pour NVidia).

Comparaison CPU / GPU



Source: Cyprien Adnet.

Limitations

CPU $\neq$ GPU.

- Beaucoup de données en même temps, oui, mais un seul programme à la fois.
- Impossibilité d'écrire et de lire les mêmes données en même temps.
- Pas applicable à tous les algorithmes.

Traitements d'image simples

- Traitements les plus simple possible.
- Noir & blanc.
- Sépia.
- Ajustement de contraste.
- Ajustement de luminosité.
- Seuillage.

Convolutions

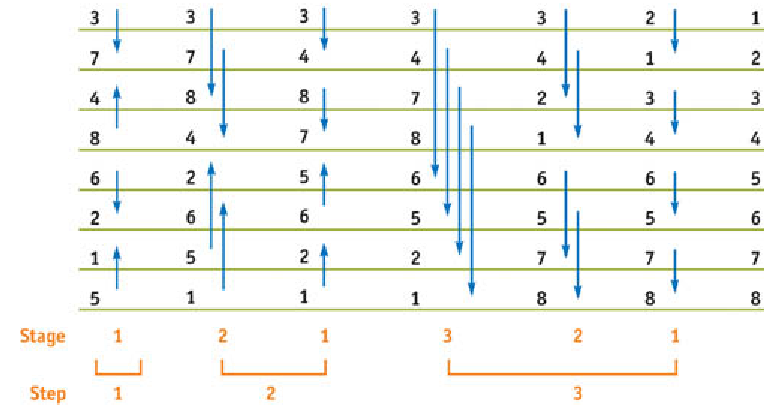
- Flou.
- Érosion / Dilatation.
- Post-effects :
  - DoF (profondeur de champ),
  - HDR (simulation de conditions de lumière extrèmes)

TRI ?

Algorithme de base important.

- Souvent peu parallélisable.
- Tri à bulle : pas terrible.
- Tri par insertion : pas mieux.
- Tri par tas : catastrophique.
- Tri ”rapide” : peu parallélisable.
- Tri fusion : vers la bonne voie.
- Tri bitonique : semble adapté !

TRI BITONIQUE



(image en provenance de NVidia)
Plus de détails : http://en.wikipedia.org/wiki/Bitonic_sorter

RENDU VOLUMIQUE

La GPGPU permet de faire du rendu volumique :

- Raycasting.
- Raytracing.
- Radiosité.
- Raymarching.
- Imagerie médicale.

AUTRES

Calculs dans un système de rendu.

- Création de terrains complexes.
- Système de particules.
- Physiques. (mass-spring, rigid body, etc.)
- Calculs de collisions.
- Skinning.

Plus ambitieux

- Calculs de pliage de protéines.
- Équation aux dérivées partielles.
- Calculs météo.
- Mécanique des fluides.
- Cryptage / décryptage.
- Détection de virus.

Mise en œuvre

- OpenGL 2.1 : fragment shader et FBO.
- OpenGL 3.0 : transform feedback.
- OpenCL : Computing library.
- OpenGL 4.3 : Compute shader.

Constat

- Donnée de base (sortie) : pixel/fragment.
- Programme identique pour beaucoup de données : *Fragment shader*.
- Données d'entrées :
  - Variables *uniform*,
  - Variables *varying*,
  - Textures 1D, 2D, 3D, etc.

Exemple : implémentation du flou

À partir d'une image stockée dans une texture, on affiche un *quad* de la même taille à l'écran avec le *fragment shader* suivant :

```
uniform sampler2D unite_texture;
uniform vec2      taille_texture;

void main(void)
{
  vec2 offset = vec2(1.0) / taille_texture;
  vec2 tex_coord = gl_TexCoord[0].xy;

  vec4 color  = texture2D(unite_texture, tex_coord + offset * vec2(-1.0, -1.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2(-1.0,  0.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2(-1.0,  1.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2( 0.0, -1.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2( 0.0,  0.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2( 0.0,  1.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2( 1.0, -1.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2( 1.0,  0.0));
       color += texture2D(unite_texture, tex_coord + offset * vec2( 1.0,  1.0));

  gl_FragColor = color / 9.0;
}
```

FBO

- Espace de rendu caché.
- Attaché à une ou plusieurs textures.
- Peut contenir 4 textures : ajoute des données de sorties !
- Peut contenir une ”texture de profondeur”.
- Peut contenir des textures de flottant.
- Se comporte comme le *framebuffer* ”normal”.
- Utilisation en ”Ping-Pong”.

Implémentation des FBO

```
GLuint fbo, texture;

glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo); // On dessine maintenant dans fbo.

glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_RECTANGLE, texture);
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, 800,600, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_RECTANGLE, texture, 0);
check_framebuffer_status();

glBindFramebuffer(GL_FRAMEBUFFER, 0); // On dessine de nouveau normalement.
```


Utilisation classique des FBOs

Rendu en deux passes :

- ```
  glBindFramebuffer(GL_FRAMEBUFFER, fbo);
  ...
  glDrawElements(GL_TRIANGLES, 24,
                 GL_UNSIGNED_SHORT, 0);
  ```
- ```
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  glBindTexture(GL_TEXTURE_2D, fbo_texture);
  ...
  // On dessin un quad qui recouvre tout.
  glDrawElements(GL_TRIANGLES, 6,
                 GL_UNSIGNED_SHORT, 0);
  ```


Utilisation ”GPGPU” en ping-pong

- Utilisation de 2 FBO.
- On utilise la texture du FBO 1 quand on écrit dans le FBO 2.
- Et réciproquement.
- Le résultat est utilisé pour le rendu.

Implémentation du ping-pong

- Création des deux FBO avec chacun leur texture.
- ```
  frame++;
  if (frame & 1)
  {
    i = 1; j = 0;
  } else {
    i = 0; j = 1;
  }

  glBindTexture(GL_TEXTURE_2D, fbo_texture[i]);
  glBindFramebuffer(GL_FRAMEBUFFER, fbo[j]);
  ...
  glDraw...
  ```
- ```
  glBindTexture(GL_TEXTURE_2D, fbo_texture[j]);
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  ...
  glDraw...
  ```


Transform Feedback

- Disponible depuis OpenGL 3
- Récupère la sortie du vertex shader.
- (Ou du geometry shader ou du tesselation shader)
- Sortie dans un VBO.
- Variables d'entrées : attribute, uniform, textures.
- Désactivation du dessin :

```
glEnable(GL_RASTERIZER_DISCARD);
```


Transform Feedback, implémentation

```
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, vbo);

glEnable(GL_RASTERIZER_DISCARD);

glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, query);
glBeginTransformFeedback(GL_POINTS);
...
glDraw...
glEndTransformFeedback();
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);

glDisable(GL_RASTERIZER_DISCARD);
glGetQueryObjectuiv(query, GL_QUERY_RESULT, &count);
```


Présentation

- Standard ouvert pour le calcul sur les GPU.
- Conçu pour s'exécuter sur GPU, CPU, SPU, DSP, etc.
- Utilisable sur GPU "compatible OpenGL 3.x+".
- Langage "parallèle" semblable à GLSL ou au C++.

ABSTRACTION

OpenCL prend en charge :

- La gestion de la mémoire.
- La gestion des textures.
- L'écriture et la lecture des données.
- La répartition de la charge entre les cœurs du GPU.
- L'utilisation de MMX, SSE, Altivec sur CPU.

UTILISATION D'OPENCL

Exemple d'addition de vecteur en OpenCL :

```
__kernel void square (__global float *input1,
                      __global float *input2,
                      __global float *output,
                      unsigned int     count)
{
  int i = get_global_id (0); // Le rang de notre processeur

  if (i < count)
    output[i] = input1[i] + input2[i];
}
```


QUELQUES LIENS

- http://gpgpu.org/
- http://www.khronos.org/opencl/