GP sur GPU : notes d'implémentation

Denis Robilliard, Virginie Marion-Poty
(Laboratoire d'Informatique du Littoral,
Univ. Littoral-Côte d'Opale)

Ecole JET 2007, Yravals, France

Sommaire

- Portabilité matérielle
- Compatibilité ECJ
- Pipeline graphique OpenGL
- Langages de shader
- GPGPU en OpenGL
- Schéma d'intégration ECJ
- Codage GPGPU sous OpenGL
- Benchmark et résultats

Objectifs

- Tester l'accélération de la PG avec GPU
  - Papier de Harding et Banzhaf partiel : pas d'évolution, compilé ?
- Portabilité vs cartes graphiques
- Compatibilité avec ECJ (Sean Luke)
- Objectifs du tutoriel :
  - présenter travaux en cours
  - faciliter la prise en main (i.e. déminer le terrain!)

Portabilité matérielle

- Vaste gamme de GPU disponibles
- Tous ne sont pas programmables en flottants ! Certains ne supportent ni branchements ni boucles.
- => Support de OpenGL 2.0
  - Nvidia à partir de FX6xxx (FX5xxx limité)
  - ATI à partir de ATI 9500
- Plus de 100 modèles de GPUs... Pour s'y retrouver : GPGPU.org, FAQ, item 23.

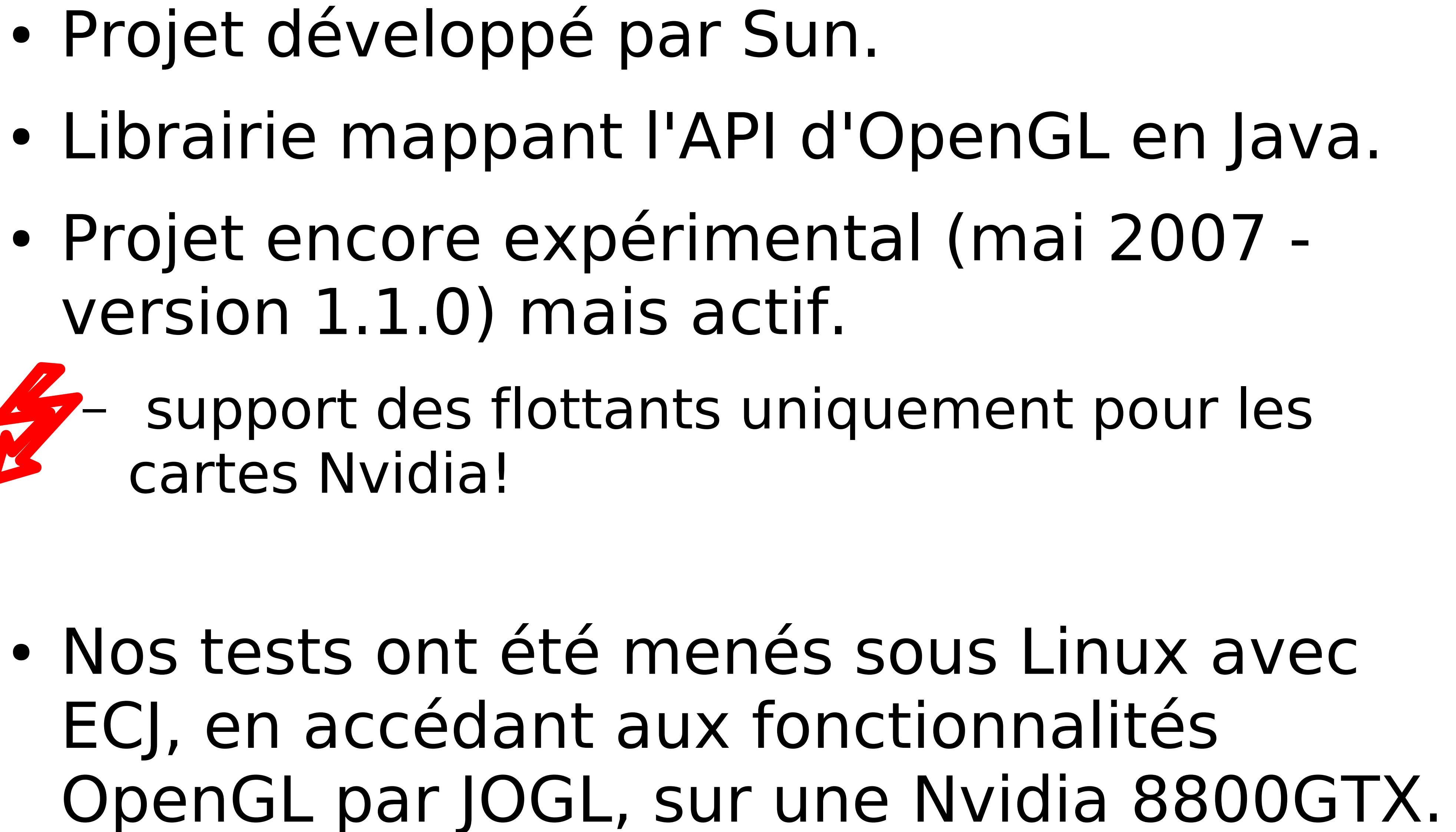
Portabilité matérielle (bis)

- Disponibilité des drivers :
  - DirectX/3D : Windows seulement.
  - CUDA : Nvidia 8xxx seulement.
- => OpenGL est l'API/driver le + portable
  - implanté sous Windows, Linux
  - versions propriétaires et libre (Mesa)
  - implanté dans les drivers propriétaires de ATI et Nvidia
- Sous Linux, Nvidia est fortement recommandé

Compatibilité ECJ

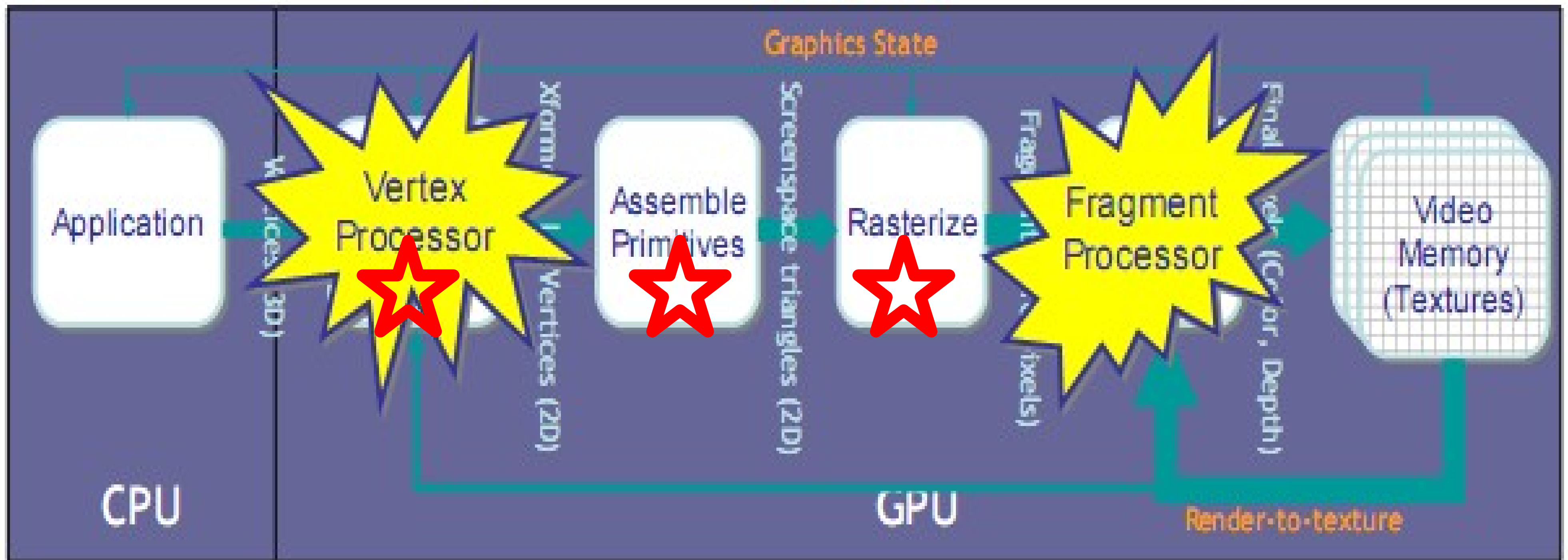
- ECJ : librairie de calcul évolutionnaire
  - développée par Sean Luke (Univ. Georges Mason, USA)
  - premièrement dédiée à la GP (mais nombreuses extensions GA, ES, multi-objectifs, co-évolution...)
  - écrite en Java (cf. Ecole d'été Yravals 2006)
- Les langages haut-niveau pour GPU (Brook, Sh, Accelerator,... cf. tutoriel C. Fonlupt) sont tous conçus comme des extensions de C/C++ et non pas Java.

Java OpenGL library - JOGL



Pipeline graphique OpenGL

- On ne programme que le fragment/pixel shader (processeur).



Ignoré (pipeline par défaut)

Langages de shader

- Sous OpenGL, on doit programmer les shaders directement (mais proche du C)
  - HLSL propriétaire Microsoft
  - Nvidia Cg ("C graphique") : largement disponible mais instable lors des compilations dynamiques (sous JOGL ?)
  - GLSL (OpenGL Shading Language) : standard OpenGL 2.0
- => programmation en GLSL
  - compilateur inclus dans les drivers sans installation de boîte à outils supplémentaire

Concepts GPGPU en OpenGL (1)

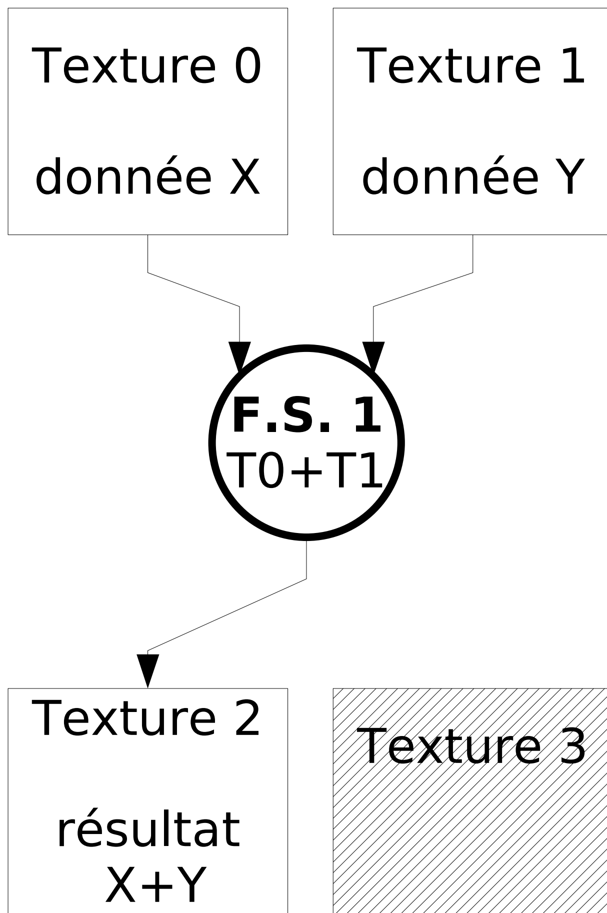
- Calcul "data stream" (data parallel, SIMD).
- Boucle itérant sur les données <=> flux parallèle dans les fragment shaders.
- Tableau de données = Texture
  - soit entières
  - soit RGBA = 4 flottants par texel
  - soit Luminance = 1 flottant par texel (mais en fait + 3 flottants ignorés) - pas très standard
- Couleur du pixel en sortie d'un shader en RGBA = 4 valeurs (normalement calculées en 4 itérations de boucle sur CPU).

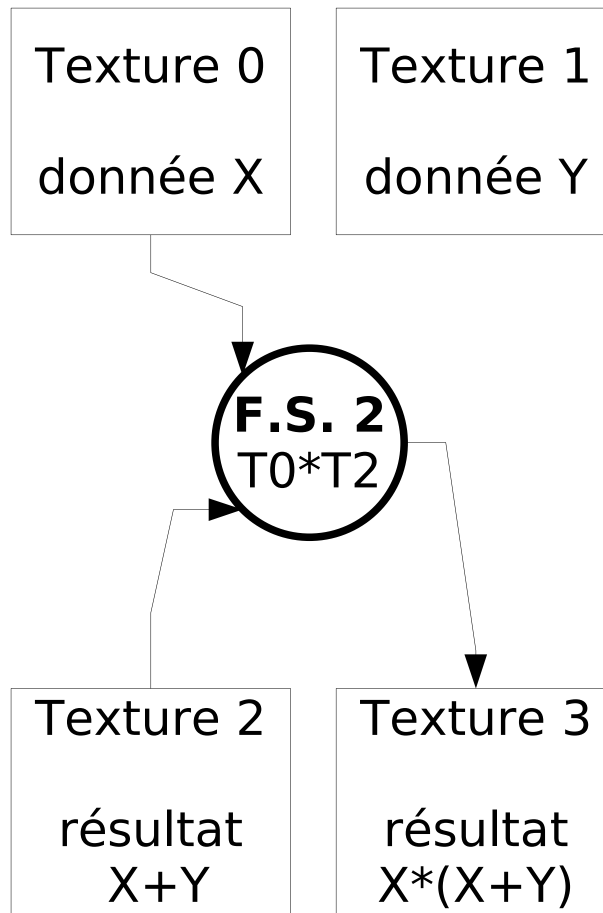
Concepts GPGPU en OpenGL (2)

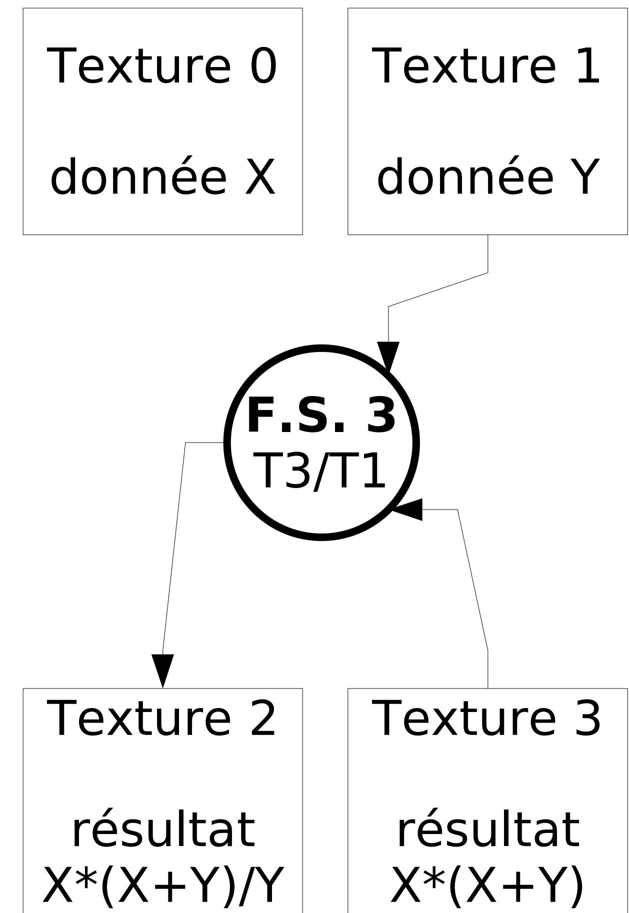
- Le rendu vers une texture (plutôt que vers l'écran), via un "framebuffer".
- Les textures doivent être chargées et déchargées du CPU vers le GPU, avant et après les calculs (coûteux).
- Le nombre de textures attachées à un framebuffer est limité par l'implémentation OpenGL (typiquement 4 ou 16).
- Les textures sont soit en lecture soit en écriture => ping-pong éventuel entre textures.

Ping-pong entre textures

Calcul de X(X+Y)/Y avec une opération par étape de rendu *(ce n'est pas la seule approche ! voir page suivante.)*

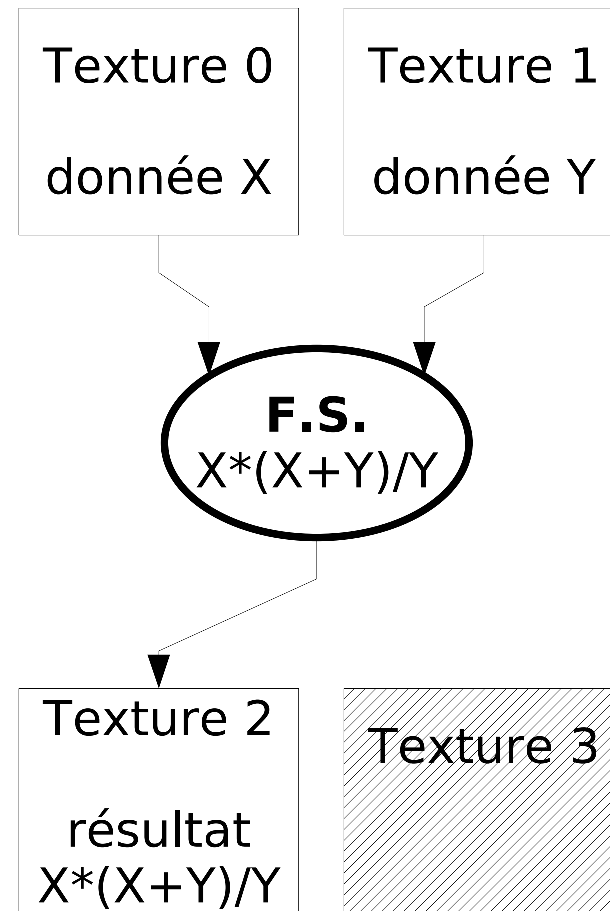






Solution sans ping-pong

- Coder tout le calcul dans le shader, avec utilisation de la mémoire locale (explicite ou implicite via compilation).



- En plus de l'accès aux textures, le pixel shader peut déclarer et utiliser des variables locales : int, float, arrays, vec2, vec3, vec4.

Concepts GPGPU en OpenGL (3)

- Le rendu est effectué selon une géométrie adaptée (ex : projection orthogonale d'un carré pour une matrice carrée).
- => pipeline par défaut associe une donnée à chaque fragment shader.

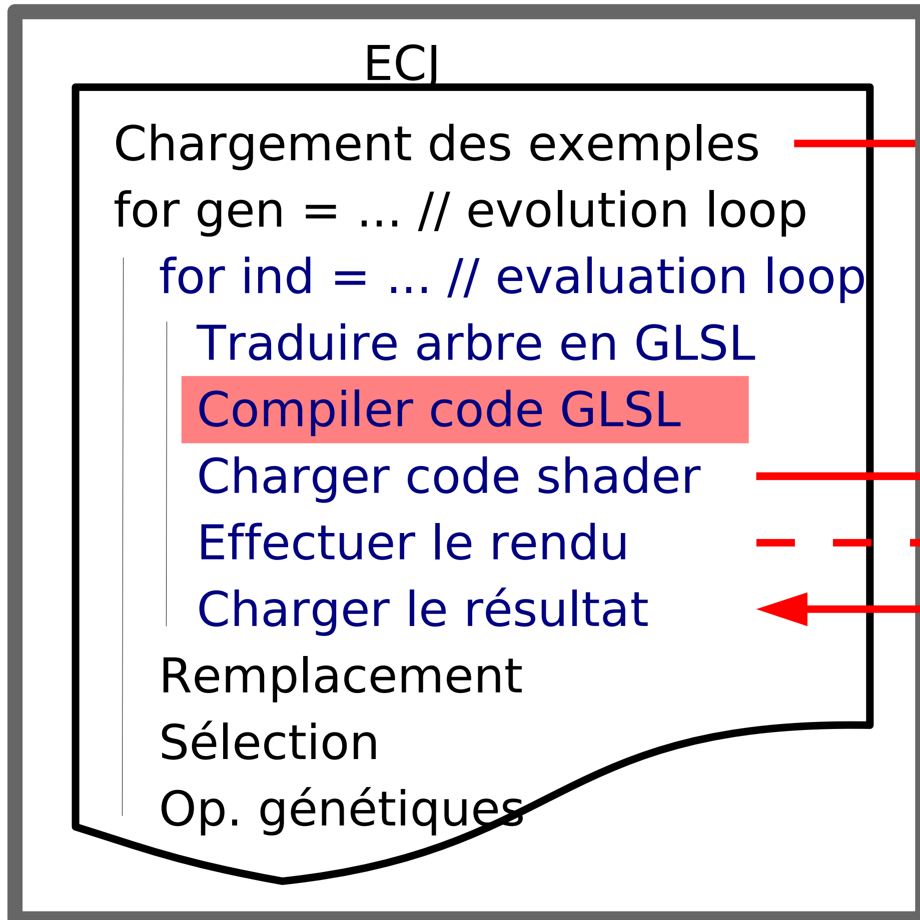
Intégration dans ECJ (1)

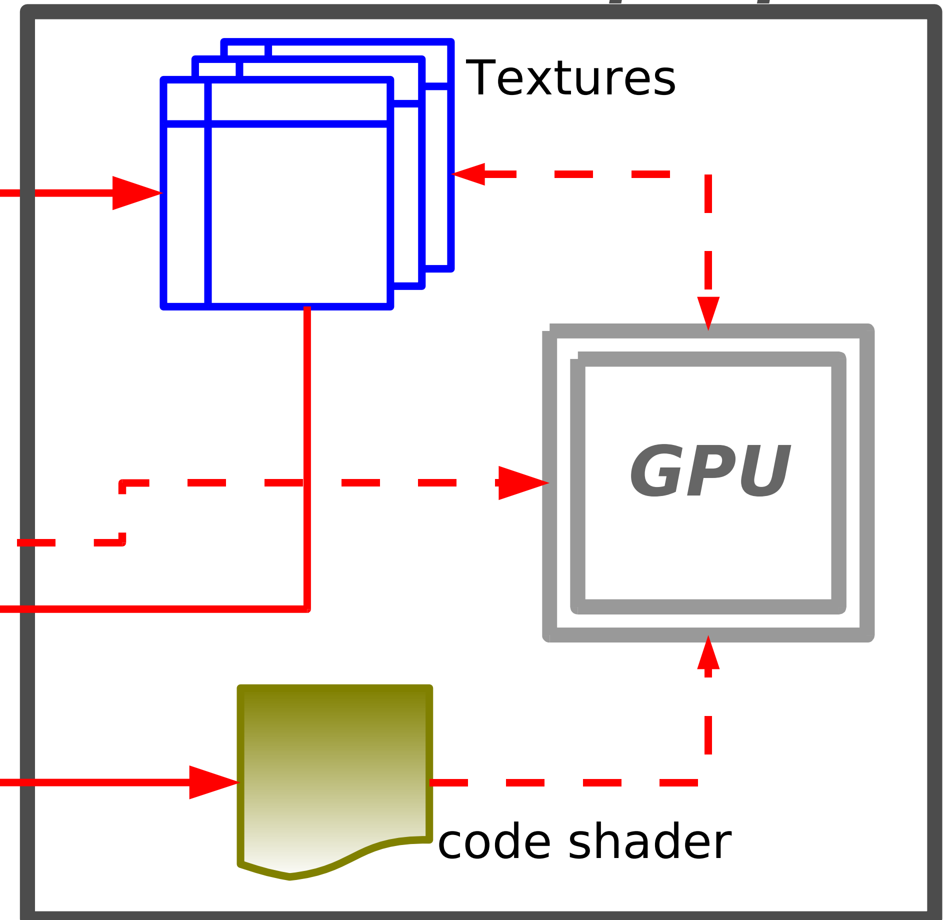
- Paralléliser le traitement de la population ?
  - Indirectement sous ECJ car arbres pointés => conversion nécessaire pour stocker en texture.
  - GP = population de programmes différents à exécuter => parallélisation MIMD !
- Paralléliser l'évaluation ?
  - Facile à réaliser de manière générique pour la régression/classification : un programme exécute les mêmes instructions sur tous les exemples.
  - Mal adapté à d'autre problèmes. Ex : simulation d'agent fourmi pour le "Santa Fe Trail".

Intégration dans ECJ (2)

- => le + simple : régression symbolique.
- Garder le moteur d'évolution d'ECJ sur CPU, déporter l'évaluation seule des individus sur le GPU :
  - traduire en GLSL, compiler et charger un à un chaque programme généré par ECJ.
  - une fois chargé dans le fragment shader, le programme est exécuté en parallèle sur tous les exemples d'apprentissage (fitness cases).

Schéma d'intégration ECJ-GPU





Nouvelle "boucle" d'évaluation

```
public void evaluate(final EvolutionState state,
  final Individual ind, final int threadnum)  {
    if (!ind.evaluated)  // don't bother reevaluating
    {
     listener.prog =
        makeGLSLprog(((GPIndividual)ind).trees[0].child);
     listener.progparamsrc = 0; // # texture entrée
     listener.progparamdest = 2; // # texture sortie
     pbuff.display();  // affiche

     ... // récupère l'erreur et assigne le fitness
    }
   }
```

Exemple de source GLSL

- La syntaxe est proche du C, la traduction arbre -> GLSL ne pose aucun problème.
- Ex: x*(0.9456502+x)

argument du shader, donne accès à la même texture pour tous les fragments.

variable réservée = couleur du pixel en sortie.

accès à un élément de texture aux coordonnées courantes du fragment.

```
uniform samplerRect input;

void main(void)
{
  gl_FragColor = (textureRect(input,
  gl_TexCoord[0].st) * (0.9456502 +
  textureRect(input, gl_TexCoord[0].st))));
}
```

Codage GPGPU sous OpenGL (1)

- Créer une "fenêtre" graphique (ex : canvas ou pbuffer) pour accéder à OpenGL.

```
GLCapabilities caps = new GLCapabilities();
caps.setDoubleBuffered(false);
pbuff =
  GLDrawableFactory.getFactory().createGLPbuffer(caps,
  null, width ,height, null);
```

- Créer une classe d'interface GLEventListener :

```
listener = new MyEventListener(myparams);
pbuff.addGLEventListener(listener);
....
class MyEventListener implements GLEventListener {
.... // code OpenGL  ..... }
```

Codage GPGPU sous OpenGL (2)

- Le "listener" implante 2 méthodes clefs.

public void init(GLAutoDrawable drawable) :

  - définit la géométrie des opérations de rendu (<=> paramétrage du vertex shader par défaut).
  - crée et initialise les textures (transfert des exemples CPU >> carte graphique), effectué une fois pour toutes si possible (! Turbo-cache).
  - crée un framebuffer pour effectuer le rendu hors-écran, auquel seront attachées les textures.
  - crée les structures pour recevoir un programme de fragment shader.

Codage GPGPU sous OpenGL (3)

public void display(GLAutoDrawable drawable) :

- charge et compile le code GLSL associé à chaque programme dont ECJ demande l'évaluation.
- associe aux arguments du programme les textures contenant les données nécessaires au calcul.
- définit la texture cible qui reçoit le résultat.
- effectue l'opération de rendu.
- transfère le résultat vers la mémoire CPU.

Codage GPGPU sous OpenGL (4)

- Convention de noms pour la suite :
  - width, height : dimension de la matrice/texture des données d'exemples.
  - data, result : pointeurs vers les données en entrées/en sortie sur le CPU.
  - texID : tableau de "handle" (int) sur les textures.
  - fb : framebuffer pour le rendu.
  - ProgramShaderID, FragmentShaderID : handles de shaders.
  - prog : chaîne contenant le source GLSL du programme GP

Codage GPGPU sous OpenGL (5)

```
class MyListener implements GLEventListener {
  // create suitable GL context
  private static GLU glu = new GLU();
  private static GL gl;
  public void init(GLAutoDrawable drawable) {
    gl = drawable.getGL();
    // manage geometry
    gl.glMatrixMode( GL.GL_PROJECTION);
    gl.glLoadIdentity();
    glu.gluOrtho2D( 0.0, width, 0.0, height);
    gl.glMatrixMode( GL.GL_MODELVIEW);
    gl.glLoadIdentity();
    gl.glViewport( 0, 0, width, height);
```

Codage GPGPU sous OpenGL (6)

```
public void init(GLAutoDrawable drawable) {
  ............
  // create, bind and parameterize one texture
  gl.glGenTextures(1, texID, 0);
  gl.glBindTexture(GL.GL_TEXTURE_RECTANGLE_ARB, texID[0]);
  gl.glTexParameteri(GL.GL_TEXTURE_RECTANGLE_ARB,
GL.GL_TEXTURE_MIN_FILTER, GL.GL_NEAREST);
  gl.glTexParameteri(GL.GL_TEXTURE_RECTANGLE_ARB,
GL.GL_TEXTURE_MAG_FILTER, GL.GL_NEAREST);
  gl.glTexParameteri(GL.GL_TEXTURE_RECTANGLE_ARB,
GL.GL_TEXTURE_WRAP_S, GL.GL_CLAMP);
  gl.glTexParameteri(GL.GL_TEXTURE_RECTANGLE_ARB,
GL.GL_TEXTURE_WRAP_T, GL.GL_CLAMP);
  // move data from CPU memory to GPU memory
  gl.glTexImage2D(GL.GL_TEXTURE_RECTANGLE_ARB, 0,
GL.GL_RGBA32F_ARB, width, height, 0, GL.GL_RGBA,
GL.GL_FLOAT, FloatBuffer.wrap(data));
```

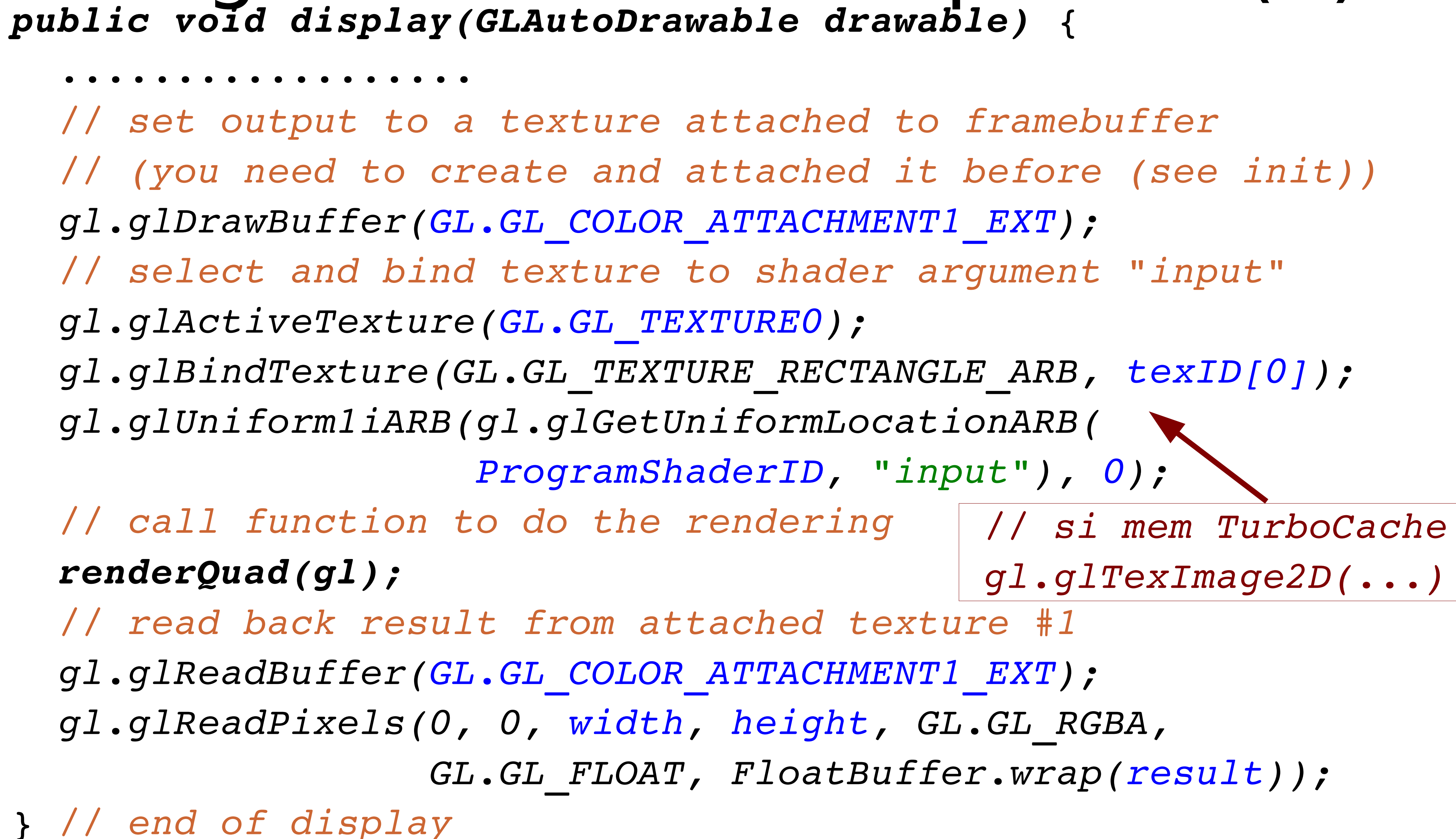
Codage GPGPU sous OpenGL (7)

```
public void init(GLAutoDrawable drawable) {
    .............
    // generate and bind one framebuffer
    fb = new int[1];
    gl.glGenFramebuffersEXT(1, fb, 0);
    gl.glBindFramebufferEXT(GL.GL_FRAMEBUFFER_EXT, fb[0]);
    // attach one given texture to the framebuffer
    gl.glFramebufferTexture2DEXT(GL.GL_FRAMEBUFFER_EXT,
  GL.GL_COLOR_ATTACHMENT0_EXT,GL.GL_TEXTURE_RECTANGLE_ARB,
  texID[0], 0);
    // prepare fragment shader and program shader
    FragmentShaderID =
  gl.glCreateShaderObjectARB(GL.GL_FRAGMENT_SHADER_ARB);
    ShaderProgramID = gl.glCreateProgramObjectARB();
} // end of init()
```

Codage GPGPU sous OpenGL (8)

```
public void display(GLAutoDrawable drawable) {

  // compile shader (stored in shared String "prog")
  gl.glShaderSourceARB(FragmentShaderID, 1, new
String[]{prog}, new int[]{prog.length()}, 0);
  gl.glCompileShaderARB(FragmentShaderID);

  // attach fragment to program (default vertex is used)
  gl.glAttachObjectARB(ProgramShaderID,FragmentShaderID);
  gl.glLinkProgramARB(ProgramShaderID);

  // verify and select program to use for shaders
  gl.glValidateProgramARB(ProgramShaderID);
  gl.glUseProgramObjectARB(ProgramShaderID);

  ............
```

Codage GPGPU sous OpenGL (9)



```
public void display(GLAutoDrawable drawable) {
   ....................
   // set output to a texture attached to framebuffer
   // (you need to create and attached it before (see init))
   gl.glDrawBuffer(GL.GL_COLOR_ATTACHMENT1_EXT);
   // select and bind texture to shader argument "input"
   gl.glActiveTexture(GL.GL_TEXTURE0);
   gl.glBindTexture(GL.GL_TEXTURE_RECTANGLE_ARB, texID[0]);
   gl.glUniform1iARB(gl.glGetUniformLocationARB(
                     ProgramShaderID, "input"), 0);
   // call function to do the rendering
   renderQuad(gl);
   // read back result from attached texture #1
   gl.glReadBuffer(GL.GL_COLOR_ATTACHMENT1_EXT);
   gl.glReadPixels(0, 0, width, height, GL.GL_RGBA,
                   GL.GL_FLOAT, FloatBuffer.wrap(result));
} // end of display
```

```
// si mem TurboCache
gl.glTexImage2D(...)
```

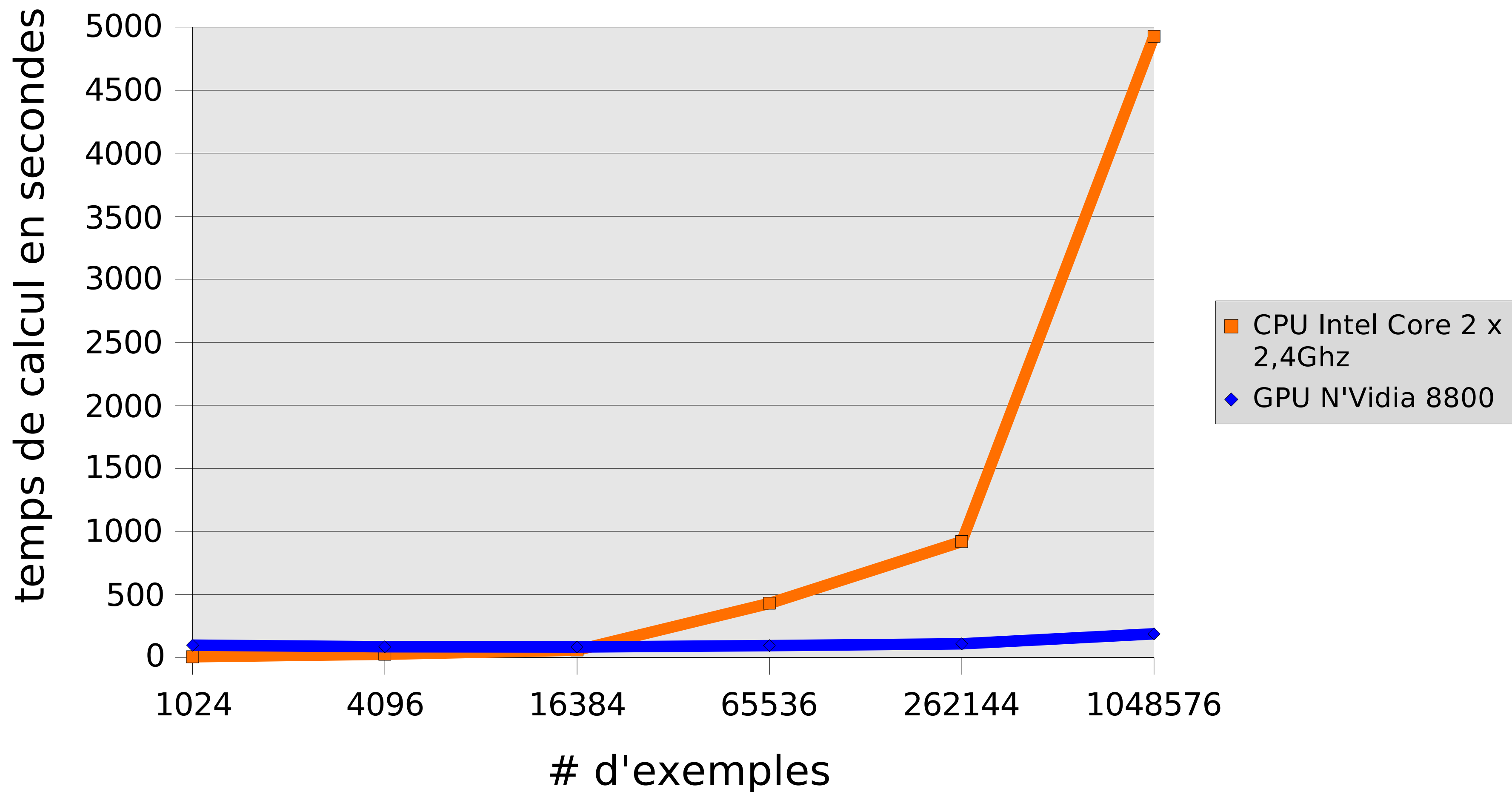
Codage GPGPU sous OpenGL (10)

```
public void renderQuad(GLAutoDrawable arg0) {
  // make quad filled to hit every pixel/texel
  gl.glPolygonMode(GL.GL_FRONT, GL.GL_FILL);

  // and render quad
  gl.glBegin(GL.GL_QUADS);
  gl.glTexCoord2f(0.0f, 0.0f);    gl.glVertex2f(0.0f, 0.0f);
  gl.glTexCoord2f(width, 0.0f);   gl.glVertex2f(width, 0.0f);
  gl.glTexCoord2f(width, height); gl.glVertex2f(width,
  height);
  gl.glTexCoord2f(0.0f, height);  gl.glVertex2f(0.0f, height);
  gl.glEnd();

}
```

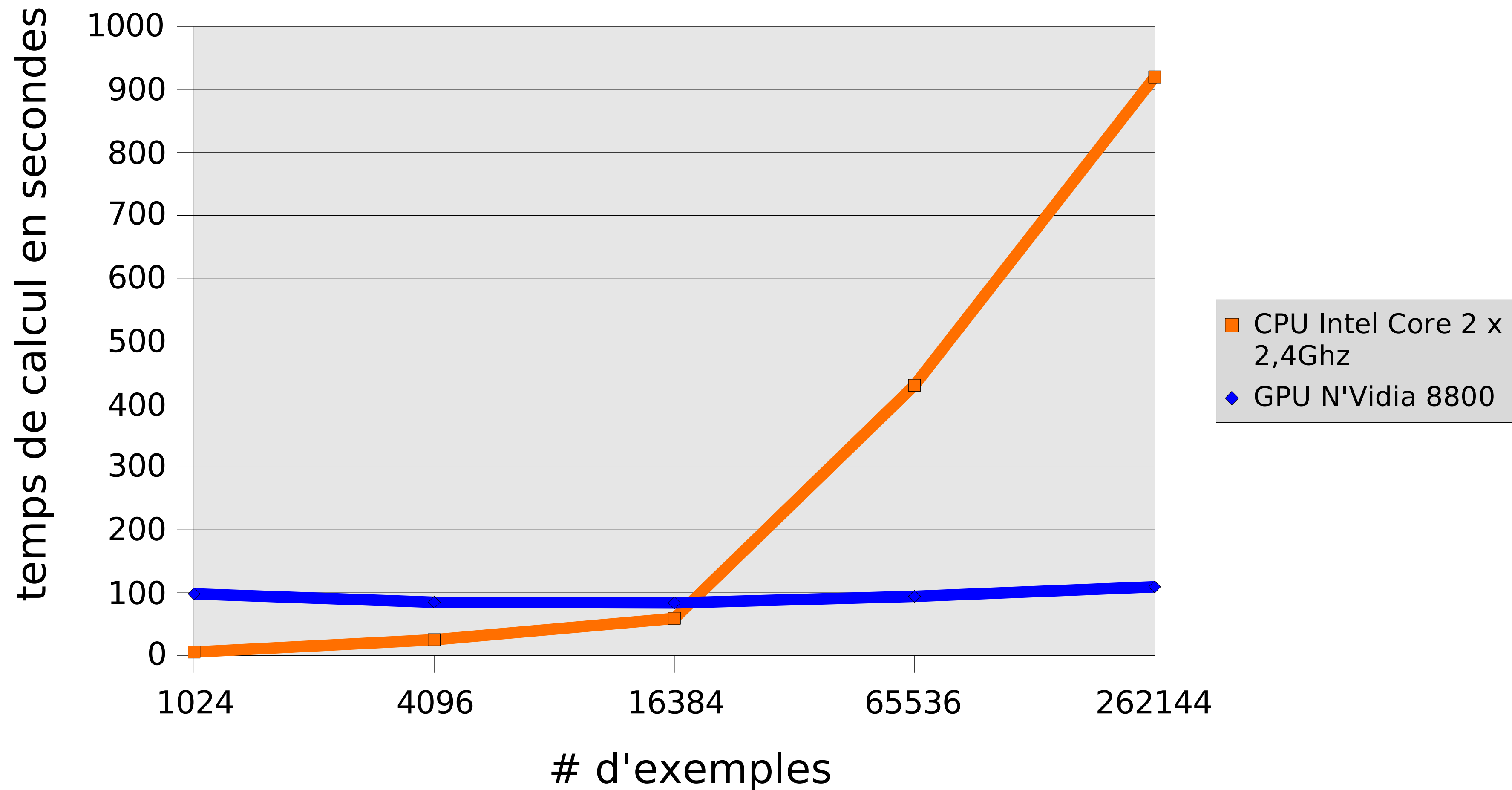
Benchmark

- Régression : $X^4 + X^3 + X^2 + X$
- Population : 100 individus
- Générations : 50
- Paramétrage évolutionnaire standard (démos ECJ)
- Nombre de points exemples : 1024, 4096, 16384, 65536, 262144, 1048576.
- Calcul du temps moyen sur 10 runs.

Résultats comparés CPU vs GPU



Zoom : petit nombre d'exemples



Conclusions

- La puissance de calcul est effectivement remarquable... à condition que le problème soit fortement parallélisé ;-)
- A-t-on souvent des jeux d'exemples de 1.000.000 de cas ?
- La compilation de chaque individu GP entraîne un surcoût important.
- La portabilité est problématique !!! (matériels, drivers (nv9751), librairies API)
  - Mémoire Turbo-cache => pas de mémoire GPU!

Conclusions (suite)

- L'utilisation de GLSL n'est pas difficile.
- On peut s'abstraire de certaines complications OpenGL via quelques routines d'encapsulation.
- Le nombre de textures et le ping-pong resteront à gérer.
- CUDA permet de gérer plus finement les types de mémoires (locale, partagée sur une grille, globale) et le découpage en blocs de threads.

Conclusions (fin)

- La parallélisation de l'évaluation se prête bien à la régression et à la classification GP.
- Ce n'est pas vrai en général : il faudrait paralléliser sur la population.
- GP = population de programmes <u>différents</u> à exécuter => parallélisation plutôt du type MIMD que SIMD!
- paradigme GA plus adapté que GP au type de parallélisation proposé par les GPU ?

-FIN -