M2-Images

OpenGL 4.3 : Compute Shaders

J.C. Iehl

November 4, 2020

résumé des épisodes précédents...

- vertex shader : transforme les sommets,
- fragment shader : calcule une couleur,
- autres calculs ? animation ? visibilité ?
- sur cpu ?
- en détournant un fragment shader ?
- opengl 4.3 : compute shader !

compute shader

compute shader :

- nouveau type de shader,
- ne fonctionne pas dans le pipeline graphique,
- donc pas d'entrées / sorties "classiques", cf vao et fbo,
- mais des (storage) buffers et des (storage) images...
- accessibles en lecture / écriture,
- et des uniforms, des textures, des uniform buffers (en lecture) comme d'habitude.

mais calculs parallèles sur les processeurs de la carte graphique !!

(en général plus rapides que sur cpu classique, mais pas toujours...)

compute shader

calculs parallèles ?

- implicite, comme pour les autes shaders,
- cf transformer tous les sommets, "colorier" tous les fragments,
- comment ca marche ?

calculs parallèles ?

idée principale :

remplacer chaque iteration par un thread parallèle...

```
// executer n iterations en sequence
for(i= 0; i < n; i++)
    r[i]= f(i);

// executer n threads en parallele
i= thread ID;
r[i]= f(i);
```

calculs parallèles ?

exécuter n threads ?

- pas de `glDraw()`,
- mais : `glDispatchCompute()`.

mais :

- les (compute) shaders s'exécutent par groupes de W threads,
- l'application exécute N groupes de W threads, N*W $>=$ n,
- `glDispatchCompute(N)`,
- et le shader déclare W, le nombre de threads par groupe : `layout(local_size= W) in;`

exemple : transformer les sommets d'un mesh

```
#version 430                                    // tutos/vertex_compute.glsl

#ifdef COMPUTE_SHADER
layout(std430, binding= 1) readonly buffer vertexData
{
    vec3 data[];
};

layout(std430, binding= 0) writeonly buffer transformedData
{
    vec4 transformed[];
};

uniform mat4 mvpMatrix;

layout(local_size_x= 256) in;
void main( )
{
    // recupere l'indice du thread
    const uint ID= gl_GlobalInvocationID.x;

    // chaque thread transforme le sommet d'indice ID...
    if(ID < data.length())
        transformed[ID]= mvpMatrix * vec4(data[ID], 1);
}
#endif
```

exemple : transformer les sommets d'un mesh

```
                                        // tutos/tuto_vertex_compute.cpp

// recupere le nombre de threads de chaque groupe du compute shader
GLint threads[3]= { };
glGetProgramiv(m_program, GL_COMPUTE_WORK_GROUP_SIZE, threads);
printf("threads / group x %d, y %d, z %d\n",
    threads[0], threads[1], threads[2]);

// nombre de groupes de threads a executer pour transformer les sommets
m_compute_threads= threads[0];
m_compute_groups= m_mesh.vertex_count() / m_compute_threads;
if(m_mesh.vertex_count() % m_compute_threads)
    m_compute_groups= m_compute_groups +1;

printf("groups %d= %d threads\n", m_compute_groups,
    m_compute_groups*m_compute_threads);

// go !!
glDispatchCompute(m_compute_groups, 1, 1);

// ou plus direct :
// glDispatchCompute(m_mesh.vertex_count() / 256 +1, 1, 1);
```

calculs parallèles ?

pourquoi ?

- `if(ID < data.length())` ?
- des groupes de W threads ?

calculs parallèles ?

pourquoi ?

- `if(thread ID < data.length())` ?
- ou thread ID < n ?

si N*W > n, on exécute plus de threads que nécessaire pour transformer les n sommets...

calculs parallèles ?

pourquoi ?

- des groupes de W threads ?

les threads d'un groupe s'exécutent sur le même processeur !
ils partagent des ressources, peuvent communiquer et se synchroniser ! (les autre shaders ne peuvent pas le faire...)

calculs parallèles ?

W ? combien de threads par groupe ?

- ça dépend de l'architecture :
- 32 pour Nvidia,
- 64 (GCN), ou 32 (RDNA) pour AMD,
- 8, 16, ou 32 pour intel (à voir pour X^e)

si W est plus petit que la taille de l'architecture :
perte de performance...
si W est trop grand, perte de performance aussi :
mais pas pour les memes raisons...

- en pratique : 256, ou 8×8 marche très bien sur toutes les architectures.

en pratique :

thread ID :

- peut être 1d, 2d, ou 3d,
- les threads sont numérotés globalement : `uvec3 gl_GlobalInvocationID.xyz`
- les threads sont numérotés localement (dans leur groupe) : `uvec3 gl_LocalInvocationID.xyz`

et aussi :

- `uint gl_SubGroupSize` : taille de l'archicture,
- `uvec3 gl_WorkGroupSize` : taille du groupe,
- `uvec3 gl_WorkGroupID` : indice du groupe,

thread ID : 1d, 2d, ou 3d ?

pratique :

- pour travailler sur un groupe de pixels dans une image,
- pour travailler sur un groupe de voxels dans une grille,

iterations en 1d, 2d, ou 3d...

compiler un compute program

créer et compiler un computer shader / program :

- 1 program avec un seul (compute) shader :
- `glCreateShader(GL_COMPUTE_SHADER)`,
- seul shader présent dans le program,
- `glCreateProgram()`, `glAttachShader()`, `glLinkProgram()`,
- ou `read_program()` avec `#ifdef COMPUTE_SHADER`

exécuter un compute program

exécuter un compute program :

- `glDispatchCompute(Nx, Ny, Nz)`
- avec Nx, Ny, Nz, nombre de groupes 1d, 2d, 3d, ou 1
- `glDispatchCompute(Nx, 1, 1)`, exécution 1d,
- `glDispatchCompute(Nx, Ny, 1)`, exécution 2d,
- `glDispatchCompute(Nx, Ny, Nz)`, exécution 3d,
- `layout(local_size_x= Wx, local_size_y= Wy, local_size_z= Wz) in;`
  dans le shader pour déclarer le nombre de threads d'un groupe.

synchronisation externe ?

synchro quoi ?

- les résultats calculés par le compute shader, sont en général utilisés par un autre shader (compute, ou fragment),
- et il faut attendre que les résultats sont disponibles avant de les utiliser...
- `glMemoryBarrier(` *flags* `)`,
- *flags* indique qui consomme les résultats,
- *flags* = `GL_ALL_BARRIER_BITS` pour tout synchroniser, utile pour debuger.

oui : ça veut dire que l'exécution des compute shaders peut etre asynchrone ou parallèle à l'exécution du pipeline graphique...

synchronisation interne ?

et revoila les mutex !!

- argh !!
- *pourquoi ?*
- exemple : écrire dans un buffer les triangles qui sont visibles par la camera ?
- on ne connait pas à l'avance la place du résultat dans le tableau / buffer,
- ni le nombre de résultats...
- il n'y a plus séparation des données / résultats lors de l'exécution parallèle, r[i]= f(i);
- donc synchronisation...

exemple :

```
// en sequentiel, pas de problemes
int output[n];

int k= 0;
for(int i= 0; i < n; i++)
    if(test(i))
        output[k++]= f(i);

// ou pour les faineants :
std::vector<int> output;
for(int i= 0; i < n; i++)
    if(test(i))
        output.push_back( f(i) );
```

operations atomiques ?

quel est le problème ?

- exécuter k++ en parallèle ne donne pas le bon résultat...
- c'est équivalent à 3 operations :
- int a= k; a= a+1; k= a;
- lorsque plusieurs threads exécutent ces instructions en même temps, k ne peut pas être correct...

opérations atomiques ?

et ?

- int index= atomicAdd(k, 1);
- GLSL fournit des opérations arithmétiques *atomiques*,
- résultat correct avec plusieurs threads,
- mais la variable k, doit etre visible pour tous les groupes,
- donc ce n'est pas une variable locale du shader...
- c'est une variable dans un buffer !!

cf #pragma omp atomic, avec openMP en c++

exemple :

```
int output[n];              // a declarer dans un buffer
int k;                      // a declarer dans un buffer

layout(local_size_x= 256) in;
void main()
{
    uint ID= gl_GlobalInvocationID.x;
    if(test(ID))
    {
        int index= atomicAdd(k, 1);
        output[index]= f(ID);
    }
}
```

exemple complet dans tuto multi draw indrect, cf tutos/M2/tuto_mdi_count.cpp + tutos/M2/indirect_cull.glsl

opérations atomiques ?

c'est magique ?!

- ben non, ça sérialise l'exécution des threads...
- et c'est lent,
- par exemple : sur une radeon, ≈2000 threads s'exécutent par processeur et il y a entre 32 (radeon 570) et 64 processeurs (vega 64)...
- si tous les threads réalisent une opération atomique en même temps, la machine complète est bloquée...
- mais bien sur, on peut faire mieux...

mais ce sera pour plus tard...

en résumé

- nouveau shader,
- nouveaux buffers et textures (écriture),
- parallélisme de données, pleins de threads numérotés,
- synchronisation.

+ détails techniques :
manipulation des storage buffers et des (storage) textures...

+ détails pas techniques :
parallélisme et synchronisation...

exemple

comment ecrire un shader :

- qui calcule l'intersection rayon / triangle,
- pour un ensemble de rayons et un ensemble de triangles ?

2 solutions : boucle sur les rayons, ou sur les triangles...

exemple : solution 1

```
// pour chaque rayon : tester tous les triangles

Ray rayons[R]= { ... };
Triangle triangles[T]= { ... };
Hit hits[R]= {};

for(int r= 0; r < R; r++)
{
    // distance max
    float tmax= rays[r].tmax;

    for(int t= 0; t < T; t++)
    {
        Hit h= intersect(triangles[t], rayons[r], tmax);
        if(h)
        {
            // intersect() ne renvoie vrai que si l'intersection
                  trouvee est plus petite que tmax
            assert(h.t < tmax);
            tmax= h.t;
            hits[r]= h;
        }
    }    // a la fin de la boucle sur les triangles, on connait l'
         intersection la plus petite.
}
```

exemple : solution 2

```
// pour chaque triangle : tester tous les rayons

Ray rayons[R]= { ... };
Triangle triangles[T]= { ... };
Hit hits[R]= {};

// on commence par initialiser la distance max
for(int r= 0; r < R; r++)
    hits[r].t= rays[r].tmax;

for(int t= 0; t < T; t++)
{
    for(int r= 0; r < R; r++)
    {
        Hit h= intersect(triangles[t], rayons[r], hits[r].t);
        if(h)
        {
            // intersect() ne renvoie vrai que si l'intersection
                  trouvee est plus petite que tmax
            assert(h.t < hits[r].t);
            hits[r]= h;
        }
    }
}
```

exemple : et alors ?

ben quoi, ça marche non ?

- oui, quand un seul thread fait tous les calculs...
- mais on va remplacer la premiere boucle par n threads parallèles,
- pour la solution 2 :
  que se passe-t-il lorsque plusieurs threads trouvent une intersection du meme rayon sur des triangles differents ?
- oui, ça va arriver à chaque fois...

exemple : et alors ?

quelle solution ?

- solution 1 : à priori sans surprise,
- solution 2 : synchronisation nécessaire lorsque plusieurs threads trouvent une intersection avec le meme rayon ?
- il faut toujours renvoyer la plus petite...

donc pour démarrer : solution 1 plus simple à écrire...
(et sans doute plus efficace aussi)

autre remarque : comment initialiser le tableau hits[] dans la solution 2 ?