M2-Images

OpenGL 4.3 : Shaders et partage de données

J.C. Iehl

December 1, 2020

résumé des épisodes précédents...

- compute shaders,
- remplace une séquence de n iterations en séquence, par n threads parallèles,
- synchronisation interne et externe,
- opérations atomiques et barrières,
- exécution cohérente SIMD des shaders,
- limites d'ordonnancement,
- mémoire partagée,
- accès mémoire...

exécution incohérente

exécution incohérente :

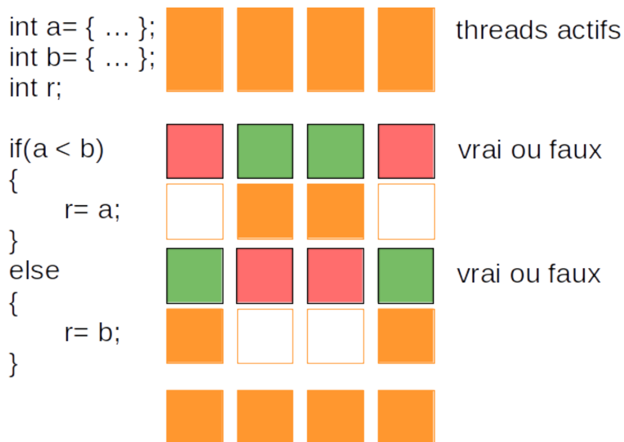
- trouver un autre algorithme avec une exécution différente ?
- qui sera peut etre cohérente ?
- ou :
- détecter les cas incohérents et modifier à la volée ?

tests incohérents

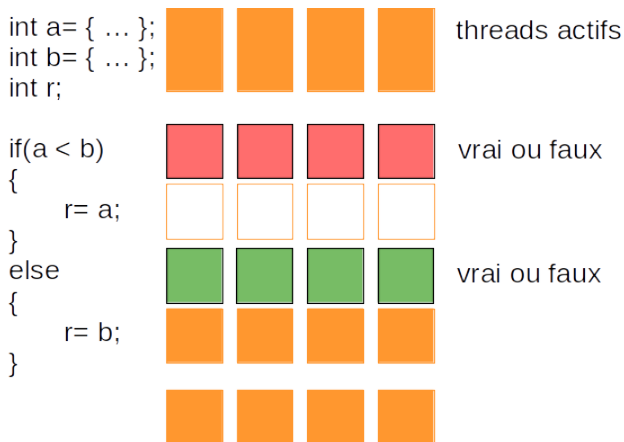
tests :

- on utilise souvent un test pour choisir entre :
- une version simplifiée et,
- une version complète (plus longue à calculer),
- mais l'exécution SIMD force à exécuter les 2 branches,
- et l'exécution est incohérente / lente.

rappels : exécution SIMD des tests



rappels : exécution SIMD des tests



tests incohérents

mais extension GLSL ARB_shader_group_vote :

- on peut détecter si tous les threads évaluent la condition du test de la même manière (ou pas) :
- `anyInvocationARB(bool)`,
- `allInvocationsARB(bool)`,
- `allInvocationsEqualARB(value)`

déclaration obligatoire dans le shader :

```
#extension GL_ARB_shader_group_vote : require
```

et alors ?

à quoi ça sert ?

- si un thread du sous-groupe doit exécuter la version complète,
- on peut forcer tous les threads à exécuter la version complète,
- et retrouver (un peu) de cohérence...

disponible pour tous les shaders
pas uniquement les compute shaders.

peut mieux faire...

on peut aussi :

- savoir quels threads du sous-groupe évaluent une condition de la même manière :
  `uint64_t mask= ballotARB(bool)`
- renvoie un *masque* de 64 bits,
- chaque bit correspond à un thread du sous-groupe,
- 1 si la condition est vraie,
- 0 si la condition est fausse,
- combien de threads veulent faire la même chose :
  `int n= bitCount(mask)`

déclaration obligatoire dans le shader:

```
#extension GL_ARB_shader_ballot : require
#extension GL_ARB_gpu_shader_int64 : require
```

manipulation des masques

opérations sur le masque :

- chaque thread 'id' est identifié par un masque :
- `thread_mask= 1 << id`
- `id= gl_SubGroupInvocationARB`
  ou `thread_mask= gl_SubGroupEqMaskARB`
- `(mask & thread_mask) != 0`
  le thread 'id' vérifie la condition,
- autres masques pour identifier les autres threads :
  `gl_SubGroupGeMaskARB`, gl_SubGroupLeMaskARB, etc.
- premier thread à vérifier la condition :
  `findLSB(mask)`

manipulation des variables

on peut aussi échanger des variables :

- entre threads du meme sous-groupe ??
- un peu comme la mémoire partagée,
- mais pour tous les shaders :
- extension GLSL : ARB_shader_ballot.

les fonctionnalités portables sont assez limitées en openGL, plus d'opérations disponibles en vulkan, + slides
ou direct3d, shader model 6.

extensions constructeurs : NV_shader_thread_group + tutorial

manipulation de variables

- `T v= readFirstInvocationARB(value)`
  valeur du premier thread actif,
- `T v= readInvocationARB(value, id)`
  valeur du ieme thread du sous-groupe.

permet d'énumérer les valeurs d'une variable pour les threads du sous-groupe.

et alors ?

on fait quoi avec ça ?

- itérer sur les valeurs des threads du sous-groupe,
- permet de forcer une exécution cohérente,
- mais : pas toujours interessant de sérialiser l'exécution...

exemple : parcours de BVH

```
#version 430

void main( )
{
    int index= root;
    while(index != -1)
    {
        Node node= nodes[index];
        if(is_leaf(node))
        {
            // == if(anyInvocationARB(leaf))
            // intersection avec le triangle de la feuille
            ...
            index= node.skip;
        }
        else
        {
            // == if(anyInvocationARB(!leaf))
            // visite des fils
            ...
            index= node.[next|skip];
        }
    }
}
```

exemple : parcours de BVH

```
#version 430

void main( )
{
    int index= root;
    while(index != -1)
    {
        while(true)
        {
            node= nodes[index];
            if(is_leaf(node))
                break;

            // visite des fils
            ...
            index= node.[next|skip];
        }
        if(is_leaf(node))
        {
            // intersection avec le triangle de la feuille
            ...
            index= node.skip;
        }
    }
}
```

exemple : parcours de BVH

```
#version 430
#extension ARB_shader_ballot : require
#extension ARB_gpu_shader_int64 : require

uint64_t exec= ballotARB(true);   // threads actifs ?
int threads= bitCount(exec);                // combien ?

bool leaf= is_leaf(node); // qui visite une feuille ?
uint64_t mask_leaf= ballotARB(leaf);     // lesquels ?
int leafs= bitCount(mask_leaf);            // combien ?


if(leafs >= threads / 2)
    // la majorite des threads teste une feuille
else
    // la majorite des threads visite un noeud
```

parcours de BVH sur gpu

à lire sur le sujet :

"Understanding the Efficiency of Ray Traversal on GPUs"
T. Aila, S. Laine, 2009

les architectures ont évolué (mémoire, cache, ordonnanceur), bitCount() / popcount() existe, et ballot() aussi...

remarque : packet traversal, une seule pile par sous-groupe de theads, au lieu de 1 pile par thread...

exemple : histogramme

```
#version 430
#extension ARB_shader_ballot : require
#extension ARB_gpu_shader_int64 : require

void main( )
{
    int bin= ... ;
    while(true)
    {
        // recupere la variable 'bin' du premier thread actif...
        int bin1= readFirstInvocationARB(bin);

        // threads qui ont la meme valeur...
        uint64_t mask1= ballotARB(bin == bin1);
        // nombre de threads qui ont la meme valeur
        int n1= bitCount(mask1);

        // selectionner un thread parmi ceux qui ont la meme valeur
        if(gl_SubGroupInvocationARB == findLSB(mask1))
            atomicAdd(histogram[bin1], n1);

        // terminer les threads bin == bin1
        if((mask1 & gl_SubGroupEqMaskARB) != 0)
        // ou if(bin == bin1)
            break;
    }

    ...
}
```