

M2-Images

OpenGL 4.3 : Compute Shaders et mémoire partagée

J.C. lehl

November 30, 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,
- ▶ accès mémoire...

mémoire partagée

les groupes de W shaders s'exécutent sur un processeur :

- ▶ ils partagent des ressources...
- ▶ mémoire partagée !
- ▶ tous les threads du groupe peuvent lire et modifier les variables partagées,
- ▶ mais attention à la synchronisation !

déclaration avec le mot-clé `shared`, variable globale d'un groupe de shaders.

32K de mémoire partagée par processeur pour OpenGL.

exemple

```
#version 430

shared int group_data[256];

layout(local_size_x= 256) in;
void main( )
{
    ...
}
```

accès parallèle à la mémoire partagée

mais :

- ▶ l'accès *parallèle* à la mémoire partagée est soumis à conditions,
- ▶ les threads doivent accéder à des adresses consécutives,
- ▶ sinon les accès mémoire sont sérialisés...
- ▶ selon les architectures, d'autres types d'accès sont parallèles :
- ▶ par exemple, tous les threads lisent la même variable,
- ▶ ...

exemple

```
#version 430

shared int group_data[256];

layout(local_size_x= 256) in;
void main( )
{
    int ID= int(gl_LocalInvocationID.x);
    // acces parallele, les threads ecrivent une sequence d'adresses
    //      memoires
    group_data[ID]= { ... };
}
```

et alors ?

à quoi ça sert ?

- ▶ à limiter les accès à la mémoire globale,
- ▶ qui sont très lents...
- ▶ à échanger des résultats intermédiaires !

synchronisation ?

pourquoi ?

- ▶ les valeurs écrites par un sous-groupe ne sont pas directement visibles par un autre sous-groupe,
- ▶ si un sous-groupe doit relire des valeurs partagées, il faut attendre que le sous-groupe qui écrit / modifie les valeurs,
- ▶ soit exécuté !
- ▶ donc *barrière* d'exécution,
- ▶ et que les données soient visibles !
- ▶ donc *barrière* mémoire.

synchronisation ?

rappel :

- ▶ les groupes de threads sont exécutés par sous-groupes indépendants,
- ▶ chaque sous groupe est ordonnancé indépendamment des autres,
- ▶ si un sous groupe écrit une valeur dans une variable partagée,
- ▶ et qu'un autre sous groupe veut relire sa valeur,
- ▶ il faut s'assurer que le sous groupe qui écrit soit exécuté,
- ▶ *avant* le sous groupe qui relit la variable.

et oui, si le groupe n'est composé que d'un seul sous-groupe d'exécution, pas de problème. mais comme on doit écrire le shader sans connaître cette taille...

exemple

```
#version 430

shared int group_data[16];

layout(local_size_x= 256) in;
void main( )
{
    // initialisation des donnees partagees
    int ID= int(gl_LocalInvocationID.x);
    if(ID < 16)
        group_data[ID]= 0;

    barrier();
    // attendre que le sous groupe thread ID < 16 s'execute

    // tous les threads peuvent maintenant utiliser group_data[]
    ... ;
}
```

exemple

calculer l'histogramme d'une image

- ▶ en c++, facile !
- ▶ avec un shader... facile !!

exemple c++

```
// version c++
Image image= read_image( "..." );

int histogram[16]= { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
for(int y= 0; y < image.height(); y++)
for(int x= 0; x < image.width(); x++)
{
    Color pixel= image(x, y);
    float grey= (pixel.r + pixel.g + pixel.b) / 3; // entre 0 et 1
    int bin= grey * 15; // entre 0 et 15
    histogram[bin]++;
}
```

exemple compute shader

```
// compute shader
#version 430

layout(std430, binding= 0) coherent buffer HistogramData
{
    int histogram[16];
};

layout(binding= 0, rgba8)  readonly uniform image2D image;

layout(local_size_x= 8, local_size_y= 8) in;
void main( )
{
    vec3 pixel= imageLoad(image, ivec2(gl_GlobalInvocationID.xy)).rgb;

    // calculer la cellule de l'histogramme pour le pixel
    float grey= (pixel.r + pixel.b + pixel.b) / 3;      // entre 0 et 1
    int bin= int(grey * 15);

    atomicAdd(histogram[bin], 1);
}
```

quoi, ça marche ?!

ben oui :

- ▶ image 3200×2400 :
- ▶ cpu 1 thread : $\approx 30\text{ms}$
- ▶ gpu : 5ms

pas mal... mais un code multi-threadé sur cpu à de bonnes chances d'être au moins aussi rapide...

et ça marche vraiment bien sur gpu ?

quoi, ça marche ?!

ben non :

- ▶ le gpu est quasi bloqué par les additions atomiques,
- ▶ et les processeurs attendent...

GPUbusyCycles		8 011 577
VALUCycles		26 666
MemBusyCycles		7 286 963
MemStalledCycles		6 802 301

bon qu'est ce qu'on fait ?!

trop d'écritures en mémoire video :

- ▶ comment limiter ?
- ▶ ??
- ▶ décomposer :
- ▶ chaque groupe calcule un histogramme 'local',
- ▶ et ajoute ses résultats à l'histogramme 'global' une seule fois (au lieu de 1 fois par pixel...)

comment ?

bon qu'est ce qu'on fait ?!

stocker un histogramme par groupe ?

- ▶ facile, mémoire partagée !!
- ▶ attention à la synchro !
(pour initialiser l'histogramme du groupe)

exemple histogramme partagé

```
// compute shader
#version 430
...

shared int group_histogram[16]; // shared !!

layout(local_size_x= 8, local_size_y= 8) in;
void main( )
{
    int bin= ... ;

    uint ID= gl_LocalInvocationIndex;
    if(ID < 16)
        group_histogram[ID]= 0;
    // attendre tous les sous groupes
    barrier();

    // construire l'histogramme du groupe
    atomicAdd(group_histogram[bin], 1);

    // attendre
    barrier();
    // accumuler l'histogramme du groupe
    if(ID < 16)
        atomicAdd(histogram[ID], group_histogram[ID]);
}
```

et alors ?

ben oui :

- ▶ image 3200×2400 :
- ▶ cpu 1 thread : $\approx 30\text{ms}$
- ▶ gpu : 5ms
- ▶ gpu + histogramme partagé : 0.450ms

c'est mieux non ??

et alors ?

c'est mieux non ??

GPUbusyCycles	8 011 577	412 249
VALUCycles	26 666	40 833
MemBusyCycles	7 286 963	100 657
MemStalledCycles	6 802 301	1 343

mais :

l'accès à la mémoire partagée est soumise à condition...

GPUbusyCycles	8 011 577	412 249
VALUCycles	26 666	40 833
MemBusyCycles	7 286 963	100 657
MemStalledCycles	6 802 301	1 343
LDSConflictCycles	0	44 810
VALUStalledbyLDS	0	122 165

et alors ?

limiter les écritures en mémoire :

- ▶ utiliser la mémoire partagée,
- ▶ 1 écriture par groupe,
- ▶ au lieu de 1 écriture par thread,
- ▶ on pourrait pas limiter les écritures dans la mémoire partagée ?

et oui ! dans ce cas, le shader est très simple, il utilise < 10 registres, on peut stocker plus de variables sans perdre en parallélisme...

1 histogramme par thread...

pour réduire les écritures en mémoire partagée :

- ▶ on peut aussi utiliser les variables locales du thread,
- ▶ mais :
- ▶ on a découpé avec 1 thread par pixel ?
- ▶ ben, on va découper différemment :
- ▶ par exemple 4×4 pixels par thread...

1 histogramme par thread...

```
// compute shader
#version 430
...

layout(local_size_x= 8, local_size_y= 8) in;
void main( )
{
    ...
    int thread_histogram[16]= int[16](0, 0, 0, 0, ... );
    for(int i= 0; i < 16; i++)
    {
        ivec2 offset= ivec2((i % 4), (i / 4));
        vec4 pixel= imageLoad(image, ivec2(gl_WorkGroupID.xy)*32 +
            ivec2(gl_LocalInvocationID.xy) + offset*8);
        int bin= ...;

        thread_histogram[bin]++;
    }

    for(int i= 0; i < 16; i++)
        if(thread_histogram[i] > 0)
            atomicAdd(group_histogram[i], thread_histogram[i]);

    // attendre
    barrier();
    // accumuler l'histogramme du groupe
    if(ID < 16)
        atomicAdd(histogram[ID], group_histogram[ID]);
}
```


et alors ?

ben oui :

- ▶ image 3200×2400 :
- ▶ cpu 1 thread : $\approx 30\text{ms}$
- ▶ gpu : 5ms
- ▶ gpu + histogramme partagé : 0.450ms
- ▶ gpu + histogramme partagé + thread : 0.200ms

GPUbusyCycles	8 011 577	412 249	256 575
VALUCycles	26 666	40 833	85 274
MemBusyCycles	7 286 963	100 657	126 579
MemStalledCycles	6 802 301	1 343	887
LDSConflictCycles	0	44 810	0
VALUStalledbyLDS	0	122 165	401