M2-Images

OpenGL 4.3 : Compute Shaders, exécution cohérente

J.C. Iehl

November 4, 2020

résumé des épisodes précédents...

- compute shaders,
- remplace une séquence de n iterations en séquence, par n threads parallèles,
- synchronisation interne et externe,
- cf opérations atomiques et barrières...

exécution cohérente

les threads sont ordonnancés :

- par le matériel, par groupes de W threads sur chaque processeur, (cf work group),
- qui exécute `gl_SubGroupSize` threads par cycle : un sous-groupe, (cf warp ou wavefront),
- (un thread est exécutable lorsque ses opérandes sources sont disponibles),
- chaque sous-groupe est indépendant des autres sous-groupes du groupe de W threads,
- chaque processeur est indépendant des autres processeurs de la carte graphique,

les W threads d'un groupe sont ordonnancés par blocs / sous-groupes cohérents...

exécution cohérente d'un sous-groupe

??

- les threads du sous-groupe exécutent la **même** instruction,
- au **même** cycle...
- cf SIMD, Single Instruction Multiple Data,
- et les unités de calcul sont vectorielles :
- r= a + b;
- exécute une addition sur W valeurs :
  r, a, et b sont des vecteurs de W composantes...

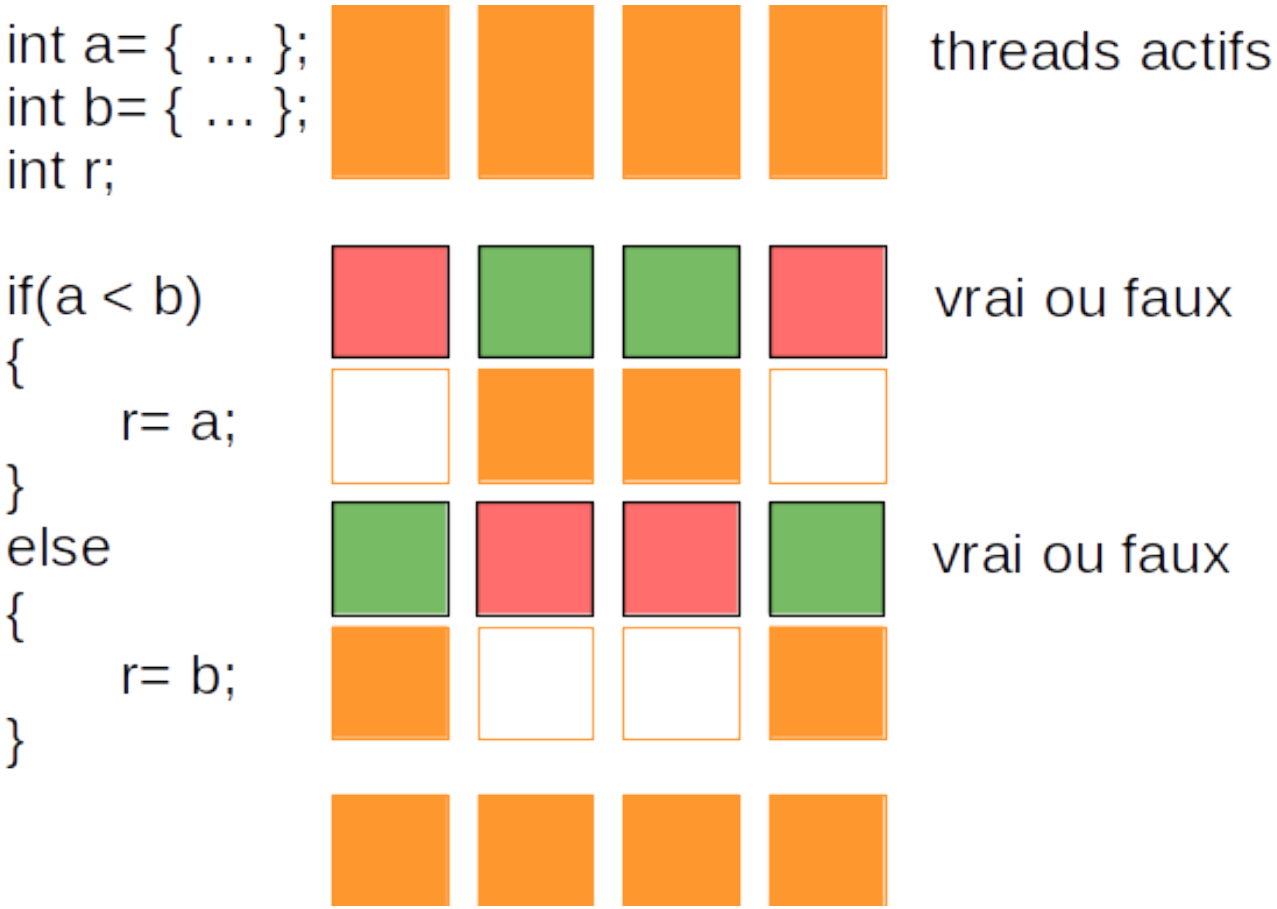
exécution cohérente d'un sous-groupe

ok, on peut croire que ça marche pour du code de calcul...

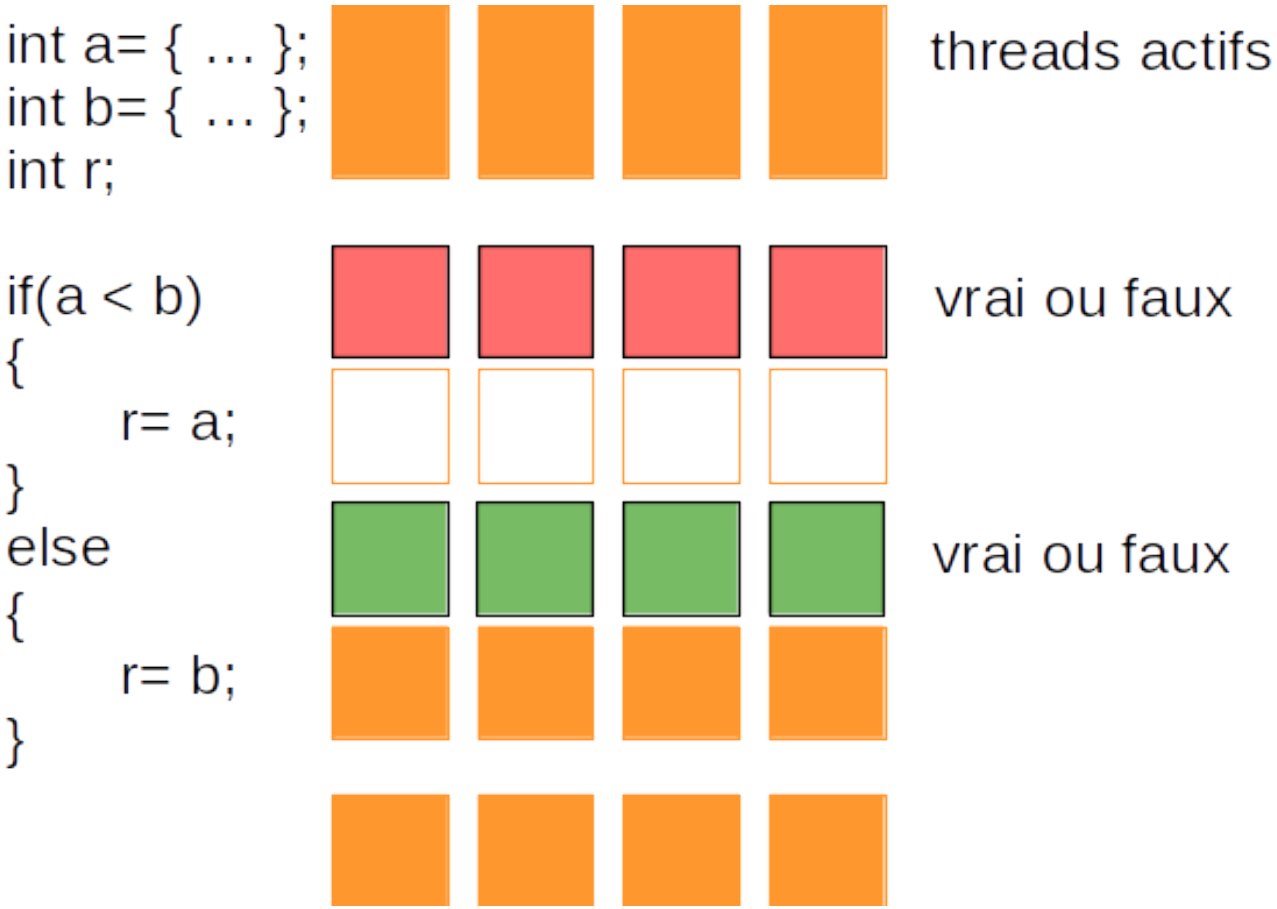
et les tests ??

- `if(a < b) { r= a; } else { r= b; }`
- tous les threads qui vérifient a $<$ b exécutent r= a;
- tous les threads qui ne vérifient pas a $<$ b exécutent r= b;
- en gros, le sous-groupe exécute les 2 parties du test...
  si un seul thread vérifie a $<$ b, il faut exécuter r= a;
  et si un thread ne vérifie pas a $<$ b, il faut aussi exécuter r= b;
- argh, mais c'est lent ?
- par contre, si tous les threads vérifient a $<$ b, il suffit d'exécuter r= a; (idem dans l'autre cas...)

et les tests ?


int a= { … };
int b= { … };
int r;
if(a < b)
{
r= a;
}
else
{
r= b;
}
threads actifs
vrai ou faux
vrai ou faux

et les tests ?


int a= { … };
int b= { … };
int r;
if(a < b)
{
r= a;
}
else
{
r= b;
}
threads actifs
vrai ou faux
vrai ou faux

exécution cohérente d'un sous-groupe

et pour les boucles ?

- c'est pareil, si les threads ne font pas le même nombre d'iterations,
- le sous-groupe exécute la boucle tant qu'un thread doit exécuter la boucle...
  (tous les threads du sous-groupe...)
- argh, mais c'est lent ?

si tous les threads du sous-groupe vérifient les mêmes conditions, l'exécution est *cohérente*, sinon elle est incohérente, et donc lente... (dans le pire cas, elle redevient séquentielle, 1 seul thread travaille par cycle)

et alors ?

en résumé :

- on fait quoi avec *ça* ?
- avoir une (petite) idée de l'architecture de la machine qui exécute le shader,
  permet de comprendre pourquoi certains programmes fonctionnent lentement...
- et d'autres, très vite !
- dans certains cas, le code n'est pas vraiment parallèle, son exécution est incohérente... (trop de conditions, trop de boucles, etc...)

limites materielles

limites ?

- un processeur ne peut *ordonnancer* qu'un certain nombre de sous-groupes (ou threads),
- si nombre de sous-groupes (ou W) est plus grand, le shader ne peut pas s'exécuter,
- un gpu ne peut ordonnancer qu'un certain nombre de groupes,
- si N est plus grand, le shader ne peut pas s'exécuter...
- il faudra re-découper le problème...

et le nombre de sous-groupes *exécutable* par processeur dépend du code du shader !
quoi ?

limites materielles

shader :

- $\equiv$ code compilé pour les instructions, l'architecture du gpu,
- une instruction charge des valeurs depuis la mémoire,
- réalise une opération,
- écrit le résultat en mémoire...
- mais pas tout à fait :
- les variables locales du shader ne sont pas allouées dans la mémoire principale du gpu, (uniquement buffers et textures),
- elles sont allouées sur le processeur qui exécute le shader,
- et le nombre de variables locales est limité...
  (les variables locales sont allouées sur des registres)

limites materielles

pourquoi ?

- les registres sont la mémoire la plus rapide,
- donc c'est bien ?
- mais il n'y a que 64K registres, 256KB par processeur,
- donc c'est pas bien ?
- réponse : ça dépend du code exécuté / du shader.

chaque *sous-groupe* de threads alloue une *copie* des variables locales du shader dans les 64K registres.
résultat : il y a une limite aux nombres de sous-groupes et de groupes *exécutables* par processeur.
(et il y a une limite au nombre de sous-groupes *ordonnancables* par processeur...)

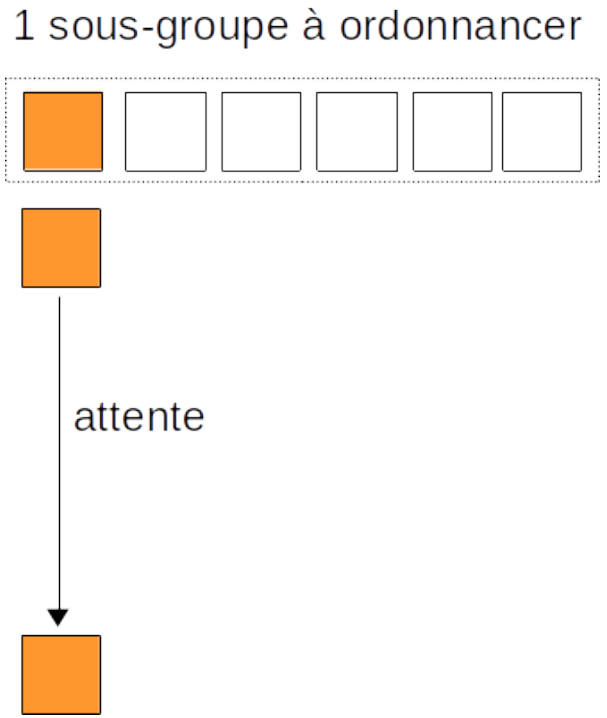
limites materielles

mais *pourquoi* ?

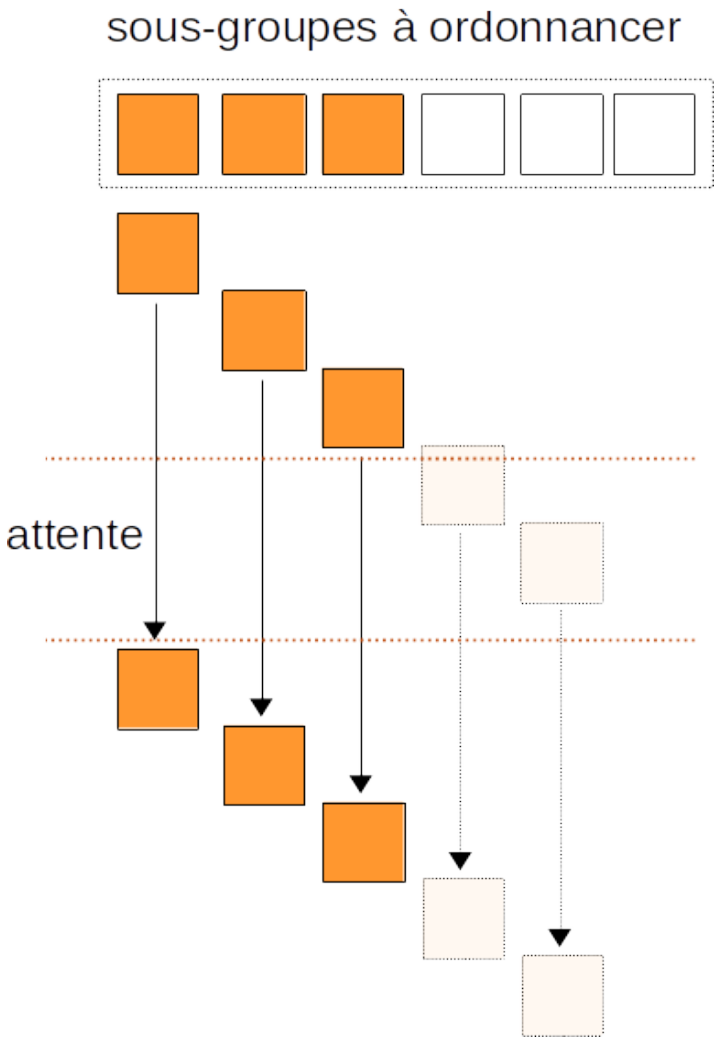
- chaque processeur tente d'exécuter du code à chaque cycle,
- un sous-groupe n'est exécutable que si ses données sont disponibles, soit variable locale dans un registre, soit une donnée chargée depuis la mémoire principale,
- les sous groupes / ou threads qui attendent des données sont bloqués,
- si un seul sous-groupe ordonnancé par processeur : pas d'instruction exécutable pendant les accès mémoire...
- donc plusieurs sous-groupes s'exécutent en cascade au fur et à mesure que les données arrivent de la mémoire principale...

plusieurs sous-groupes sont nécessaires pour trouver une instruction exécutable à chaque cycle...

ordonnanceur : quel sous-groupe executer ?



ordonnanceur : cascade de sous-groupes...



et alors ?

en résumé :

- accès mémoire lent,
- chaque processeur doit trouver un sous-groupe exécutable (par cycle),
- les variables locales / les registres sont *toujours* disponibles,
- lire des valeurs depuis des buffers ou des textures demande plusieurs *centaines* de cycles,
- si aucun sous-groupe n'est exécutable, le processeur attend...
- et *ça* c'est lent...

donc plusieurs sous-groupes par processeur sont nécessaires, mais le nombre de sous-groupe est limité par le nombre de variables locales du shader...

et alors ?

N et W :

- ont finalement une grosse influence sur l'exécution des shaders,
- W sur le nombre de sous-groupes (et de groupes) exécutés par processeur,
- N sur le nombre de groupes exécutés par les processeurs du gpu...

mais pour raison de portabilité, les conditions réelles ne sont pas exposées / connues...

cf RenderDoc ou outils / profilers des constructeurs : AMD Radeon Gpu Profiler, Intel Graphics Performance Analyzer, Nvidia Nsight

et GLSL ?

mais GLSL ?

- comme les autres langages gpu (openCL, CUDA, HLSL, etc.) propose une vue simplifiée du problème :
- un modèle de programmation simplifié,
- on écrit du code pour un thread,
- le compilateur génère du code SIMD,
- et l'ordonnanceur exécute les sous-groupes de threads.