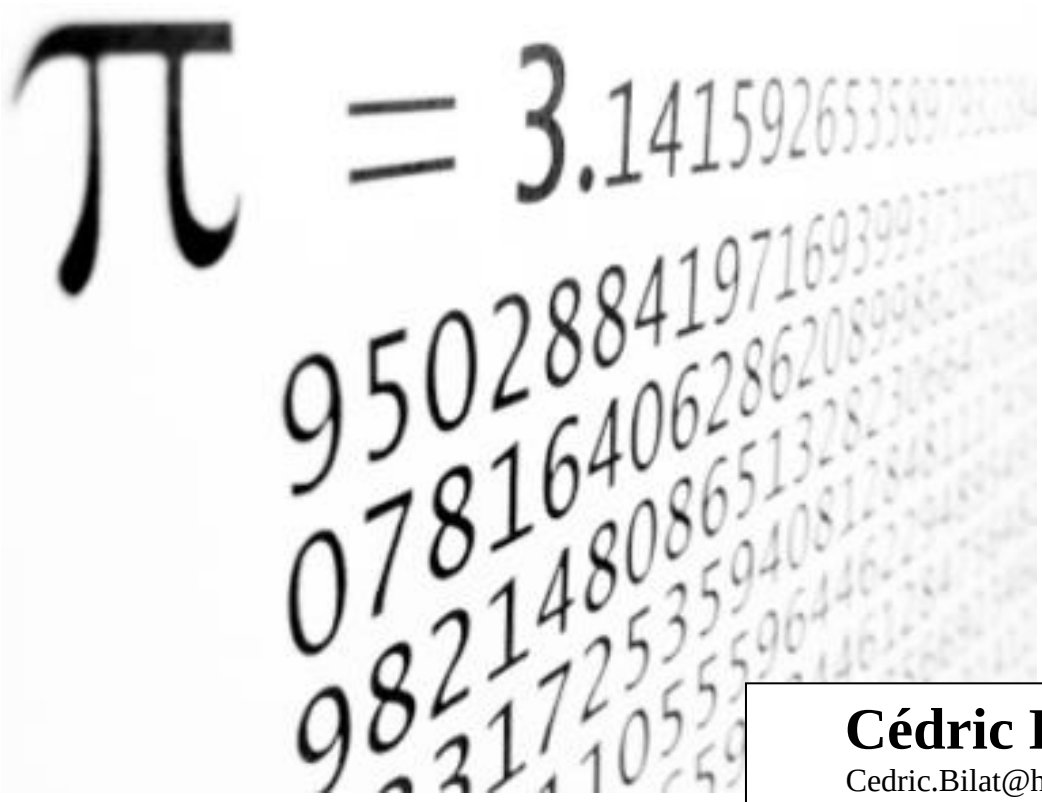
GPGPU



Cédric Bilat
Cedric.Bilat@he-arc.ch

Slice

GM_Host

Prérequis

TP_Slice_Algo

Cuda

Trois grandes étapes :

Etape 1 : ***Réduction IntraThread***

L'intervalle d'intégration est découpé en *n* slices. On applique le pattern d'entrelacement sur ces *n* slices. Ainsi chaque thread s'occupe de calculer et sommer l'aire de slices dont il a la charge. Ces slices sont non contiguës et séparées de NB_THREADS cases. Pour effectuer cette

Réduction IntraThread

on applique un pattern d'entrelacement standard :

```
const int TID= Thread2D::tid()                    //global à la grille
const int NB_THREAD= Thread2D::nbThread();        //nbThreadTotal

int s=TID;
while(s<nbSlice)
    {
    // work with : s
    . . .
    s+=NB_THREAD;
    }
```

Pour éviter les problèmes de concurrences, on utilise la même technique de

Promotion de tableau

qu'en OpenMP.

Etape 2 : ***Promotion de tableau et MM***

Il s'agit ici d'adapter la technique de promotion de tableau d'OMP, notamment en mettant en place du memory management (MM) pour :

- allouer sur le device le tableau promu
- copier coté host le tableau peuplé par la *réduction IntraThread*
- desalouer sur le device le tableau promu

Le chef d'orchestre est toujours le host. C'est lui qui mène la danse en effectuant le MM qui impacte le Device.

Etape 3 : ***Réduction finale coté host***

La réduction finale se fera cotée host. Il s'agit ici de réduire de manière parallèle le tableau promu issu de la réduction *IntraThread* effectuée sur le device. Chaque thread possédait alors une case « privé » dans le tableau promu, dans laquelle il est allé déposer son résultat.

Le tableau promu a autant de cases que de threads

En Cuda, il y a donc potentiellement beaucoup de thread, donc beaucoup de cases, selon la taille de la grille que vous allez choisir. La réduction finale a donc un certains prix ici en GPGPU, contrairement à la version OMP ou le nombre de thread était très petit.

Note :

Pour corriger ce défaut, un prochain TP vous sera proposé pour grandement optimiser la réduction sur GPU.

Optimisation

(O1) Réduction parallèle en OMP sur le host

(O2) Sur le device, lors de la réduction *IntraThread*, chaque thread utilisera

un **registre** (ie une variable locale)

pour cumuler la somme de ses slices. Chaque thread déposera une unique fois en GM dans le tableau promu son cumul. Un registre est évidemment bien plus rapide que la GM.

Squelettes de codes

Des squelettes de codes vous sont fournis dans le workspace.

1) Complétez les ***TODO*** (requiert de comprendre le code fournit et son organisation)
2) Faites passer les *uses*
3) Faites passer les *tests unitaires*

Concept

Les squelettes de codes vous sont fournis pour gagner du temps de saisie de code c++ standard. L'objectif dans ce cours est de mettre l'accent sur Cuda,

Il vous incombe néanmoins de comprendre les codes fournis, leur interaction et les structures sous-jacente.

Tests Unitaires

Les ***tests unitaires*** sont déjà codés.

Procédure

Dans le workspace, on peut spécifier deux types d'exécution :

- ***USE***
- ***TEST***

dans la méthode ***main*** à la racine du projet.

Etape 1

Passer le workspace en mode *test* dans ***main.cpp***, à la racine du projet.

Etape 2

Mettez en commentaire les tests qui ne vous intéressent pas dans ***mainTest.cpp***
Inutile de tous les lancer à chaque fois !
Idem pour ***mainUse.cpp***

Etape 3

Choisissez une rendu *console* ou *html* dans ***mainTest.cpp***
(html conseillé)

Test Performance

Warning

Parmi les tests unitaires, il y a un test performance. Celui-ci emploie

votre grille

ie celle que vous avez optimisée dans

```
Grid SliceGmHostUse::createGrid()
	{
	// TODO
	}
```

Ce warning est à prendre en compte pour tous les autres tests performances des futurs TP !

Debug/Validation

Validation 1 **1**

Dans la reduction ***intraThread***, remplissez *tabGM* avec des 1 partout !

La réduction de *tabGM* devra valoir NB_THREAD, car il y a autant de cases dans *tabGM* que de threads.

Note : Coté host on peut calculer NB_THREAD soit avec *dg*,*dg* soit en utilisant l'attribut nTabGM.

Warning : Dans le cas ou

NB_THREAD>nbSlice
nTabGM=min(nbSlice, nbThread)

il faut adapter le raisonnement ci-dessus !
Il faut remplacer NB_THREAD, par nbSlice

Validation 2 **TID**

Idem ci-dessus, mais on met TID dans tabGM. Cette fois-ci la réduction de tabGM devra valoir

$$\sum_{0}^{N-1} i = \frac{N*(N-1)}{2} \quad \text{avec} \quad \boxed{N = NB\_THREAD}$$

Warning : si N est trop grand, la formule ci-dessus risque de déborder du type int !

Warning : Dans le cas ou

NB_THREAD>nbSlice
nTabGM=min(nbSlice, nbThread)

il faut adapter le raisonnement ci-dessus !
Il faut remplacer N, par nbSlice.

Validation 3

Si vous n'avez toujours pas PI alors que vos validations ***intraThread*** ci-dessus fonctionnent, vérifier :

- votre finalisation coté host

Cuda Performance Register

Comme on le verra plus tard dans le cours, il est essentiel en Cuda d'utiliser à chaque instant la mémoire la plus appropriée possible. Dans le cadre de ce cours, on va notamment utiliser les mémoires suivantes :

- Global Memory (GM)
- Shared Memory (SM)
- Constant Memory (CM)

La GM est volumineuse et peut contenir plusieurs GO (jusqu'à 48GO en 2019), mais elle est très lente (500 GO/s en 2019) : lente ? La ram de votre portable doit plafonner en pratique à 5GO/s, donc 500GO/s ca paraissent énorme, mais en pratique c'est le goulet d'étranglement « intra-device ». On étudiera tout ca plus en détails dans la suite du cours. Ici on aimerait juste faire une petite optimisation qui va privilégier l'utilisation des registres, la mémoire la plus rapide !

Réduction Intra-Thread

Code naif

```
while(s<n)
    {
    tabGM[TID]+=aireSlice(s) ;
    }
```

Code Optimiser

```
float sumThread=0 ; // dans les register
while(s<n)
    {
    sumThread+=aireSlice(s) ;
    }
tabGM[TID]=sumThread ;// on accéde q'une seule fois en GM
```

Précision Nombre de Slice

Observation

Plus on prend de slices plus l'estimation du nombre PI est bon. Néanmoins, au-delà d'un certains seuils, ce n'est plus vrai !

En effet, avec un nombre de slices trop grands, on atteint les limites des nombres à virgules sur un ordinateur. La largeur des slices devient trop petite. Lorsque l'on flirte avec la limite du type ***float***, l'estimation du nombre PI devient moins bonne.

Il faut appliquer le principe de parcimonie !

Précision/Test unitaire Problème

Contexte

Dans les tests unitaires, on fait varier la grille. La taille de la grille influence le nombre de threads. Il se pourrait que le résultat soit correct avec certaines grilles, et pas avec d'autres. Deux cas se présentent alors :

- soit il y a juste une erreur de précision
- soit le résultat est totalement faux.

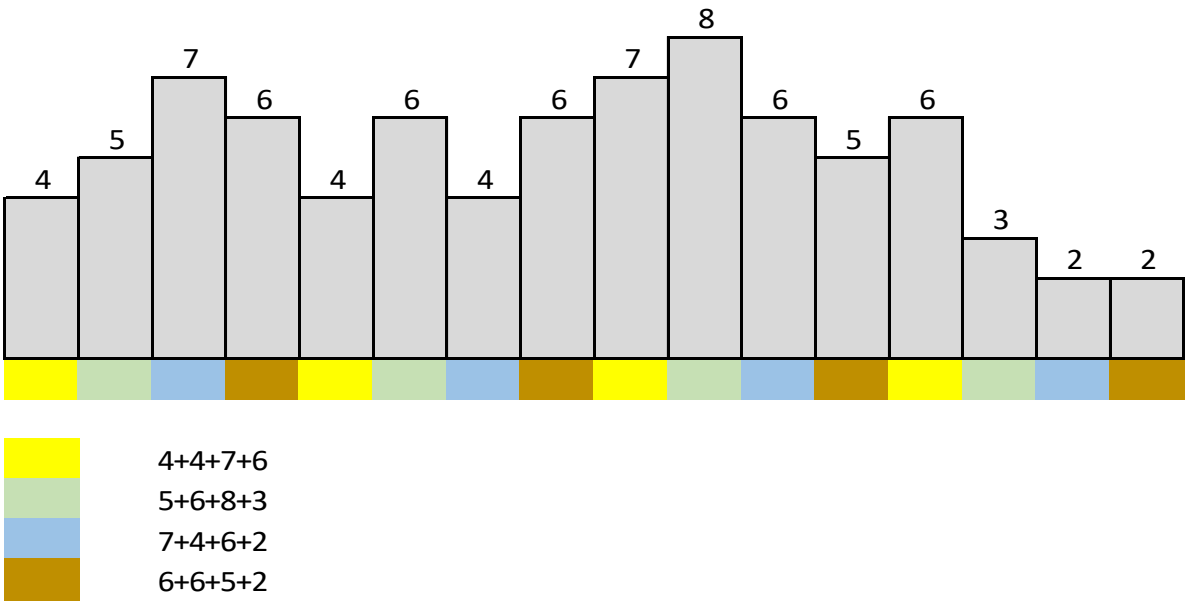
On s'intéresse ici juste au premier cas où le résultat est presque juste

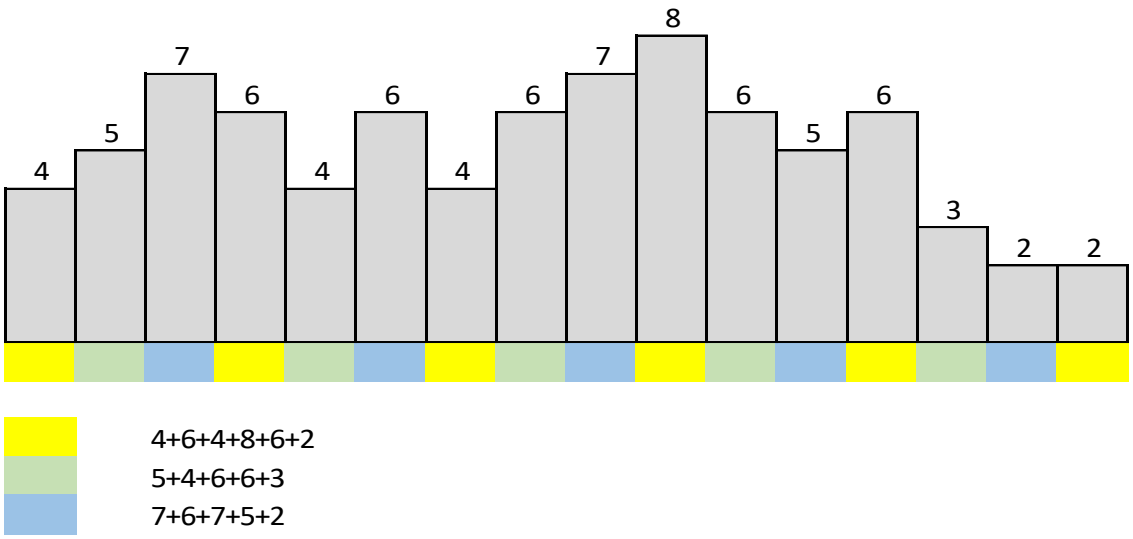
Différence de précision

Comment peut-on avoir des différences de précisions d'une grille à une autre, sachant que quelques soit la grille, on additionne toujours exactement les mêmes slices ? Si on additionne exactement les mêmes slices, ne devrait-on pas avoir toujours le même résultat ?

Explications

Comme le montre les deux schémas ci-dessous, selon la taille de la grille, on additionne toujours les mêmes nombres, mais dans le même ordre :





Or, sur une machine l'addition n'est pas associative :

$$(a+b)+c \neq a+(b+c)$$

surtout quand on se trouve proche des limites du type que l'on utilise. Dans ce TP, on prend beaucoup de slices, ils sont donc très étroits, et leur aire très petit, proche des limites du type *float* utilisée. L'addition n'est donc plus associative, et selon l'ordre d'addition des slices, on obtient un

résultat sensiblement différent.

Pour cette raison, on observe que le même critère de précision choisit pour le test unitaire peut faire échouer un test alors qu'il fonctionnait avec un autre ! Il faut donc relâcher le critère de précision en utilisant

epsilon = 1^{e}-3 au lieu de 1^{e}-4

Ce TP permet donc d'illustrer la non-associativité de l'addition sur une machine. Ce n'est pas tous les jours que l'on peut observer ca !

Workspace

Effectuez le changement dans

SliceGmUse::isOk(…)
SliceGmUse::isOk(…)

Note

Ce problème pourrait aussi arriver dans les futurs versions de slice, souvenez-vous en !

Problème possible

Globalement il faut se méfier d'un cas rare, mais qui pourrait se produire :

NB_SLICE<NB_THREAD

Coter host

On peut sécuriser le memory managment (MM)
S'il y a plus de thread que de slice, il est inutile de faire un tabGM trop grand

```
nTabGM=min(nbSlice, nbThread)
```

Note

Il n'est pas demandé de tenir compte de ce cas de figure. Si vos tests unitaires passent c'est ok !

Amélioration

Observation

(O1) Copier le tableau promu côté host, pas top !
(O2) La réduction finale se fait coté host, tous les calculs ne se font pas côté GPU, bof !

Amélioration

Attendez le prochain TP !

End