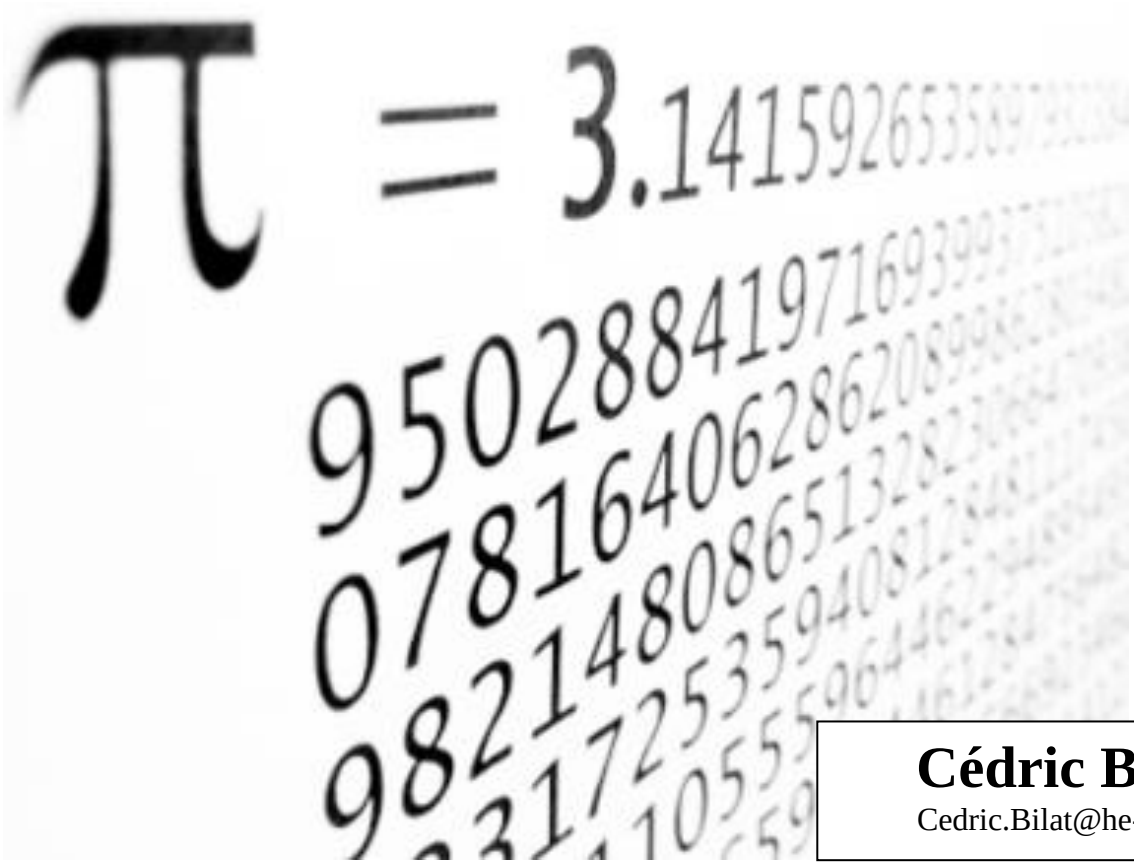
GPGPU



Cédric Bilat
Cedric.Bilat@he-arc.ch

Slice

SM

Prérequis

TP_Slice_Algo
TP_Slice_GM_Host
TP_Slice_GM

Cuda Pipeline Algo

But

Utilisation d'un algorithme de réduction parallèle, évitant le plus longtemps possible les problèmes de concurrences, donc évitant le plus longtemps possible les ralentissements dû à des barrières de synchronisation.

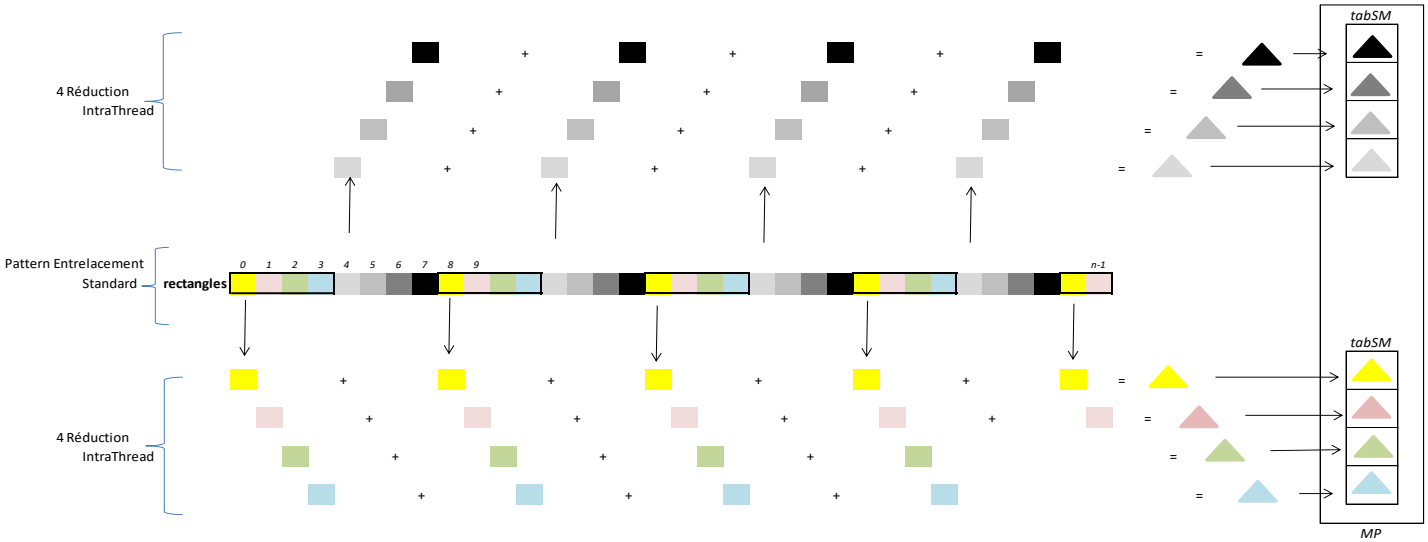
Le but est ici d'appliquer la réduction selon l'algorithme :

- Réduction **IntraThread**
- Réduction **IntraBlock**
- Réduction **InterBlock**

Utilisant de la SM pour maximiser les performances.

Réduction IntraThread

Le thread jaune s'occupe alors de calculer et d'additionner les rectangles jaunes de la surface d'intégration qu'il parcoure, et stocke par soucis de performance ces résultats intermédiaires dans une **variable locale** (dans les registres). Cette première réduction, dite réduction **intraThread** fournit le triangle jaune dans le schéma suivant :



Dans ce shéma on suppose par soucis de commodité :

- 4 threads par blocs
- 2 blocks par multi processeurs (MP)

Il suffit ensuite de copier ce triangle jaune en **shared memory** (SM)

Réduction Intra block

Au cours, on a vu comment effectuer une *réduction IntraBlock*. L'algorithme expliqué est générique et indépendant du TP ! Il s'agit juste de l'appliquer ! Le seul paramètre d'entrée est `tabSM` et rien d'autres !

```
/**
* Hyp : dimension(tabSM) est une puissance de 2, grid 1D
/*
__device__ void reductionIntraBlock(float* tabSM) ;
```

La dimension de celui-ci pourra se récupérer à l'intérieur avec

```
blockDim.x
```

Cet algorithme de *réduction intraBlock* est indépendant du nombre *n* de rectangles! Il s'occupe juste de réduire `tabSm`, quel que soit la manière dont `tabSM` a été peuplé. Cet algorithme de *réduction intraBlock* pourra être repris de TP en TP : il est générique !

L'algorithme de *réduction intraBlock* fait juste l'hypothèse que `tabSM` est

une puissance de 2.

A vous de choisir une dimension de block idéal satisfaisant cette contrainte. Cette hypothèse n'est pas réductrice. Il n'y a aucune relation entre cette taille *db.x* et le nombre *n* de rectangle.

Réduction Inter block

Cf cours. Cet algorithme est aussi générique !

Tip

Si vous avez déjà implémenté la classe *ReductionAdd*, utilisez-là !

Cuda ReductionAdd

Voir le TP séparé traitant de cette problématique !

Cuda Implémentation

Indication générales

(I1) Utiliser votre classe *ReductionAdd*

(I2) Structurer votre code comme suit

```
__global__ void XXX(…)
            {
            __shared__ extern float tabSM[] ;

            reductionIntraThread(tabSM, ….) ;

            ReductionAdd::reduce(tabSM, …) ;
            }
```

(I3) La fin du calcul peut se faire avantageusement cotée host de manière séquentielle

Pièges

(P1) Coté host, n'oubliez pas la création et surtout l'**initialisation à zéro** de la variable contenant le résultat final.

```
#include "GM.h"
...
float* ptrSumGM =NULL;
GM::mallocFloat0(&ptrSumGM);
```

(P2) Faut-il initialiser les *tabSM* à zéros lors d'une réduction additive ? Oui et non, tout dépend de votre technique de « peuplement »

Conseil : *Peuplement par écrasement*

Lors de la réduction *intraThread*, dont le but est le peuplement des *tabSM*, utiliser une variable locale dans les registres pour :

- Optimiser les performances de votre code
- Eviter l'initialisation de tabSM

Cette variable va accumuler la réduction du thread auquel elle est rattachée, et une fois sa valeur finale obtenue, il faut la déverser une seule fois, par écrasement dans la « bonne » case du tableau en SM. Chaque thread à une case en SM, « sa » case, elle est identifiée par son TID. Mais lequel : global ou local ? A vous de jouer ! La variable locale au thread doit, elle, être initialisé, puis peuplé, mais pas la case en SM :

```
__device__ void reductionIntraThread(…)
            {
            . . .
            int sumThread =0 ; // dans un register!

            while(s < …) // pattern entrelacement
                  {
                  . . .
                  sumThread +=   ;// incrémenter par le thread courant
                  . . .
                  }

            tabSM[ … ] = sumThread; // ecrasement, 1x !
            }
```

Parcimonie

N'oubliez pas les

```
__syncthreads() ;// barrière pour tous les threads d'un même block
```

Mais soyez minimaliste !

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)

Warning Nombre de Slice

Cf paragraphe du même nom dans le TP de base SliceGM, la problématique est la même !

Test unitaire Précision Problème

Cf paragraphe du même nom dans le TP de base SliceGM, la problématique est la même !

Cuda Variation

Utilisez en SM un tableau de 1 case. Tous les threads du même block écrivent leur résultat directement dans cette unique case. Le problème de concurrence se résout avec un

```
atomicAdd(&tabSM[0],aireSliceI) ;
```

ou de manière équivalente

```
atomicAdd(tabSM, aireSliceI) ;
```

Que peut-on dire des performances ?

End