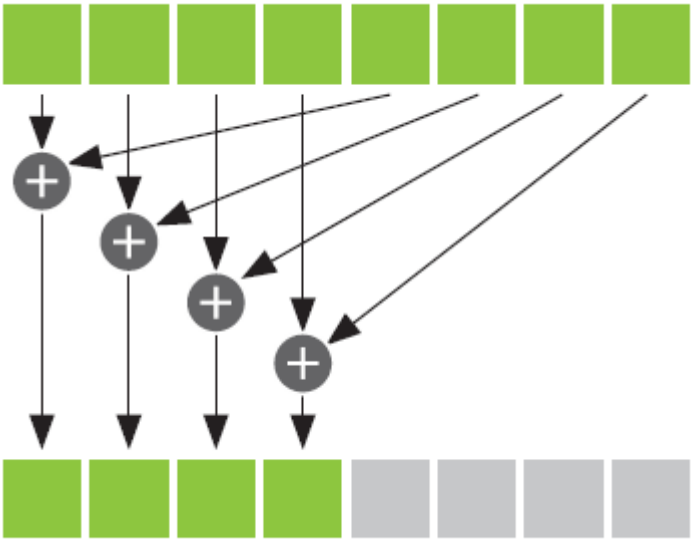
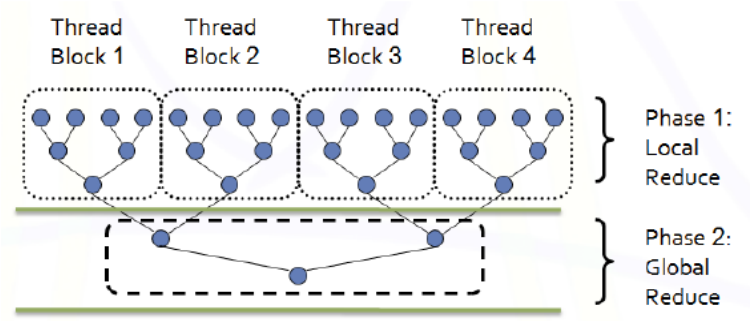
GPGPU



By Professeur
Cédric Bilat
Cedric.Bilat@he-arc.ch



Réduction

Hiérarchique par écrasement

Contexte

La réduction est un algorithme incontournable dont on a toujours besoin, un peu partout.

Par exemple, en traitement d'image on aimerait « contraster » une image « fade ». On cherche alors la valeur min et max de l'intensité entre tous les pixels, puis on amplifie par une transformation « affine » chaque pixel de telle sorte que le min devienne 0 et le max devienne 255 (image en uchar). De cette façon tout le spectre [0,255] de couleur est utilisé, et l'image sera plus « dense ». Ici il s'agit d'un exemple de réduction de type MinMax, le plus souvent la réduction est additive, mais pas toujours, il est utile d'en être conscient.

Rappel

La réduction se divise en trois phases

Réduction Intra Thread

Réduction Intra Block

Réduction Inter Block

La réduction Intra Thread dépend du contexte et eu problème à résoudre. Elle ne peut être implémentée sous forme « générique ». En opposition,

Réduction Intra Block

Réduction Inter Block

sont des algorithmes génériques qui peuvent être implémentés une fois pour toute !

Objectif

L'objectif de ce TP est l'implémentation des deux algorithmes de réduction générique vu au cours

Reduction Intra Block

Reduction Inter Block

On se placera dans un premier temps dans un scénario de réduction additive. Dans les futurs TP ou la réduction sera nécessaire, il suffira d'employer l'implémentation générique réalisée ici. Il ne restera alors plus qu'à travailler sur la

Réduction Intra Thread

Cuda ReductionAdd

Implémenter les deux algorithmes génériques

- *reductionIntraBlock*
- *reductionInterBlock*

sous forme *static* dans une classe *ReductionAdd*. Attention, coder cette classe dans un fichier .h, et utiliser les **templates** du C++ pour pouvoir réduire tant des *int* que des *float* par exemple. Finalement, implémenter

reduce

qui regroupe l'appel et le code nécessaire à

reductionIntraBlock
reductionInterBlock

Soyez le plus parcimonieux possible en terme d'input dans la signatures de méthodes, pour faciliter leur utilisation et minimiser les erreurs d'utilisation ou la confusion « de variables »

```
class ReductionAdd
      {
      public:

      template <typename T>
      __device__ static void reduce(T* tabSM, T* ptrResultGM)
            {
            // TODO
            }

      private :

      template <typename T>
      __device__ static void reductionIntraBlock(T* tabSM)
            {
            // TODO
            }
```

```
        template <typename T>
        __device__ static void reductionInterblock(T* tabSM,
                                                   T* ptrResultGM)
        {
        // TODO
        }

};
```

Rappel

- Un kernel cuda est mono fichier.
- Pour malgré tout faire du multifichier, il faut utiliser des .h pour les fichiers secondaires. *ReductionAdd* sera donc codé dans un .h et utiliser dans un .cu

Validation Protocoles

Il serait bien de trouver un protocole de test pour valider hors TP votre classe *ReductionAdd*.

Protocole I

Principe :

Pour la réduction *intraThread*, chaque thread dépose 1 dans « sa case » en SM. Le résultat de la réduction totale additive doit valoir dans ce cas

$$\sum_{tid=0}^{NB\_THREAD\_TOTAL} 1 = NB\_THREAD\_TOTAL$$

ie le nombre de thread total !

Debug

Pour debuguer, commencer avec 1 seul block, pour minimiser les risques au niveau de la *réductionInterblock* !

Protocole II

Principe

Chaque thread peuple « sa case » en SM par son *tidGlobal*. Le résultat de la réduction totale additive doit valoir dans ce cas

$$\sum_{tid=0}^{NB\_THREAD\_TOTAL-1} tid = \sum_{i=0}^{N-1} i = \frac{(N-1)N}{2} \qquad où\ N = NB\_THREAD\_TOTAL$$

Variation de la grille :

Il se peut que votre code fonctionne avec de petites grilles, mais plus avec de grande.
Il est donc important de valider votre code avec des grilles de tailles différentes et grandes.

Les tests unitaires fournit s'en chargeront, mais il est conseillé que vous expérimenter vous même quelques dg,db.

Double Piège des protocoles

Peut-être que vos implémentations sont justes, mais que le résultat des tests indique *loupé*. Ce cas pourrait se produire pour le *protocole II*, alors que pour le *protocole I* le test réussissait. Ce problème pourrait apparaitre notamment quand vous utilisez beaucoup de threads !

Vérifier que vous ne déborder pas la ***limite du type*** avec lequel vous travaillez !

Exemple 1 : *int* *protocole II*

Posons n est le nombre de thread total. Le résultat du *protocole II* est

$$reultatTheorique = \frac{n(n-1)}{2}$$

Comme pour la réduction, le nombre de thread par block doit être une puissance de 2, et comme $db_{\max} = 1024$, dans vos tests, le pire des cas est 1024 pour *db*.

Quel *dg* max peut-on prendre pour ne pas que le résultat déborde le type *int* dans le *protocole II*? Posons M la plus grande valeur possible du type avec lequel vous travailler (ici int). Il suffit de résoudre l'inéquation

$$\frac{n(n-1)}{2} \leq M$$

Résolvons à cet effet l'équation quadratique

$$x^2 - x - 2M = 0$$

En int

M=2147483647

Ici seule la solution positive nous intéresse

$$x = \frac{-1+\sqrt{1+8M}}{2} = 65535.5$$

On peut donc avoir au maximum 65535 thread. Comme $db_{\max} = 1024$, on a finalement

$$\boxed{dg_{\max} = \lfloor 65535/1024 \rfloor = 63}$$

Piège 2

Attention, un deuxième piège se situe au niveau du calcul de la valeur théorique elle-même :

$$resultatTheorique = \frac{n(n-1)}{2}$$

Avec $db_{max} = 1024$ et $dg_{max} = 63$, le produit du numérateur déborde le type *int*, alors que le diviser par 2 le fait revenir dans le type *int* !

Indication :

(I1) Faites les opérations dans le bon ordre !

(I2) Faites le calcul en long

(I3) Un nombre pair est divisible par 2

Exemple 2 : *long protocole II*

$$x = \frac{-1+\sqrt{1+8M}}{2} = ?$$

En long,

M=9223372036854775807

En employant par exemple la classe *BigInteger*, ou mieux *BigDecimal* en *Java*, et en calculant la racine avec l'algorithme de *Newton*, on obtient :

$$x = \frac{-1+\sqrt{1+8M}}{2} = 4194303.99951...$$

$$\boxed{dg_{max} = \lfloor 4194303.99951.../1024 \rfloor = 4194303}$$

La limite, si elle existe belle et bien aussi en long, est très lointaine !

Rappel

Contraintes

(R1) Grille et block de dim1.

(R2) Taille de block en puissance de 2 (2, 4, 8 ,16 ,32 ,64 ,128 ,256 ,1024).

(R3) Autant de cases en SM que de thread par block.

Warning *Côté Host*

(W1) N'oubliez pas l'allocation de la SM

```
size_t sizeTabSM= … // en octets
myKernel<<<dg,db, sizeTabSM >>>(…) ;
```

(W2) N'oubliez pas l'initialisation de la SM

```
#include « GM.h »
...
Int* ptrGM=null ;
GM ::mallocInt0(&ptrGM) ; // ou mallocFloat0
…
GM ::memcpy_int(ptrHost, ptrGM) ;
```

Reduction Generic/Mutex

Observations

(O1) Une réduction n'est pas toujours ***additive***

(O2) Une réduction n'est pas toujours sur des ***type simples***

(O3) Selon la version de cuda, tous les types simples ne possèdent pas forcément de méthode ***atomicAdd***

Solutions

(S1) Sous la forme de pointeurs de fonction. Templatées, utiliser comme inputs :
- un ***opérateur binaire***
- un ***opérateur binaire atomic***

(S2) Utiliser un ***mutex GPU*** pour combler le manque d'une méthode *atomic*
(Laisser à l'utilisateur qui doit fournir une méthode atomic add)

Pointeurs Fonctions *Opérateur Binaire & Opérateur Binaire atomic*

```
#define BinaryOperator(name) T (*name)(T, T)
#define AtomicOp(name) void (*name)(T*, T)

class Reduction // Generic
    {
    public:

      template <typename T>
      static __device__
      void reduce(BinaryOperator(OP) , AtomicOp(ATOMIC_OP),
                                          T* tabSM, T* ptrResultGM)
          {
            . . .
            // examples
            T result = OP(tabSM[0],tabSM[1]) ;
            . . .
            ATOMIC_OP(ptrResultGM,tabSM[0]) ;
            . . .
          }
. . .
}
```

Lock/Mutex

```
#include "Lock.h"

// variable global .cu
__device__ int volatile mutex=0;    // Attention a l'initialisation

__device__ static
void kernel( ... )
      {
      Lock locker(&mutex);
      ...
      locker.lock();
      // code a protéger
            {
            ...
            }
      locker.unlock();
      ...
      }
```

Exemple1

```
#include "Lock.h"

// variable global .cu
__device__ int volatile  mutex=0;   // Attention a l'initialisation

__device__
void atomicAddLong(long* ptrResult, long value)
      {
      Lock locker(&mutex);
      locker.lock();
      *ptrResult += value;
      locker.unlock();
      }
```

Exemple2

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)

Cuda Défi I

On pourrait avoir besoin de plusieurs réductions en même temps, par exemple:

- Min
- Max

C'est par exemple assez souvent le cas en traitement d'image. Il n'y a donc plus ici, un output, mais deux et deux réductions à faire !

Ce défi devra être relevé dans le *TP_Video_Convolution*

Cuda Défi II

Dans le cas du max (ou du min), trouver non pas seulement le *max*, mais aussi le lieu où se passe le max, ie *l'argmax*. Il n'y a donc plus ici, un output, mais deux !

Il faut faire suivre aux indices le même chemin que la réduction

End