Problèmes Parallelisation

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

Histogramme

Histogramme

Contexte

Input : Un tableau de données *tabData* de taille *n*, contenant des nombres compris dans l'intervalle [0,255]

Output : Un tableau de fréquence *tabFrequence* de taille 256 indiquant le nombre d'apparition des données contenus dans *tabData* .

Code séquentiel

```
/**
 *  tabData in [0,255]
 */
void histo(unsigned char* tabData, long n, unsigned int* tabFrequence)
    {
    init(tabFrequence, 256, 0);

    for (long i = 0; i <n; i++)
       {
       tabFrequence[tabData[i]]++;
       }
    }
```

Validation

Méthode 1 *Empirique non Déterministe*

Tirer des nombres aléatoires uniformes entre 0 et 255.
Puis vérifier que le tableau des fréquences est presque une « droite », ie que les fréquences obtenues sont presque les même pour tout l'histogramme.

Cette validation n'est pas simple à automatiser, et pas déterministe.

Méthode 2 *Déterministe*

Peupler le tableau d'input avec les N éléments de la suite

$$u_i = i\%256 \in [0,255] \qquad i \in [1,N]$$

Puis vérifier que le tableau des fréquences est une « droite » exacte, ie que les fréquences obtenues sont exactement les même pour tout l'histogramme. Pour que cette propriété soit vrai, il faut évidemment que N soit un multiple de 256, ie que

$N\%256 = 0$

Cette validation est automatisable et déterministe. En effet, on doit avoir

tab[s]=N/256 $\forall s \in [0, 255]$

Mélanger ensuite votre tableau de données, pour « casser la systématique » de la construction. Par exemple, mélanger en permutant des éléments du tableaux, en tirant les indices de permutations de manières aléatoires. Effectuez par exemple *N* permutations, ça sera suffisant.

Implémentation OpenMP

Effectuez 3 implémentations différentes (cf cours) :

- *Entrelacer_PromotionTab*
- *For_Atomic*
- *For_PromotionTab*

et comparer les performances ! Pour l'étude des performances, varier aussi les compilateurs, et les OS ! N'oubliez pas l'implémentation séquentielle.

L'implémentation CPU est triviale, l'implémentation Cuda le sera moins.

Principe Cuda

Contexte

Supposons un tableau d'input `tabData` de taille quelquonque *n*. Prenons pas soucis de commodité pour le schéma :

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

Utilisons une organisation hiérarchique des threads monodimensionnel tant pour la *grille* que pour les *blocks*. On applique le pattern entrelacement standard afin de parcourir les *n* éléments de notre tableau d'input.

Pattern Entrelacement standard

```
const int tid=threadIdx.x+(blockIdx.x*blockDim.x) ;       //global à la grille
const int tidLocal=threadIdx.x;                           //local à un block
const int NB_THREAD= blockDim.x*gridDim.x;                //nbThreadTotal

int s=tid;
while(s<n)
      {
      // work with : s, tabData
```

```
s+=NB_THREAD;
}
```

Réduction IntraThread

Le thread jaune parcourt via le pattern d'entrelacement les cases jaunes de `tabData`.

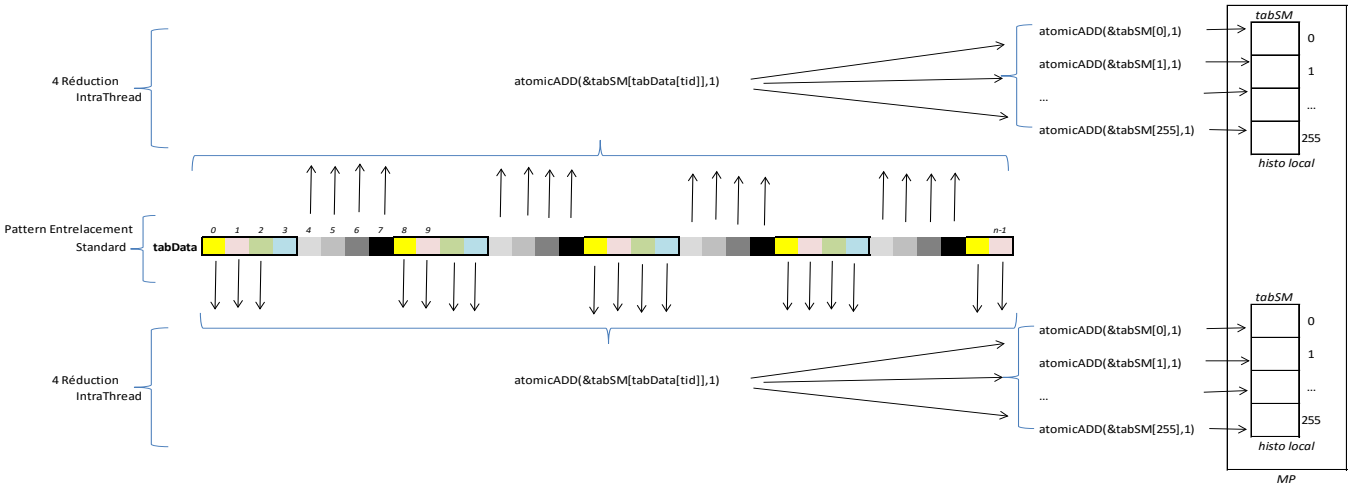
A chaque élément jaune parcouru, il update l'histogramme

en shared memory (SM)

qu'il partage avec les autres threads du même group. Plusieurs threads d'un même block peuvent donc en même temps incrémenter de 1 unité une des 256 cases de l'histogramme partagé en SM. Il y a donc problème de concurrence que l'on résout avec un

`atomicADD`

Toute synchronisation est couteuse. Elle est ici néanmoins moins couteuse en SM qu'elle ne l'aurait été en global memory (GM). En effet, en GM, tous les threads de la grille aurait du « séquentialiser » leur update de l'histogramme (au pire des cas), alors qu'en SM, seule les threads d'un même block peuvent être en conflit ! Comme il y a moins de thread dans un block que de thread totales, la version avec la SM est plus rapide, mais elle nécessite ensuite une réduction Interblock.



Réduction Intra block

Il n'y en as pas !

Réduction Inter block

L'algorithme expliqué au cours est générique et indépendant du TP ! Il s'agit juste de l'appliquer ! Les seuls paramètres d'entrée sont `tabSM` et `tabGM` et rien d'autres !

Implémentation Cuda

Implémenter les deux versions suivantes :

Version 1 : **Global Memory (GM)**

Version 2 : **MapMemory**

Notez que dans les deux versions vous devrez utiliser de la *shared memory* pour effectuer la réduction !

La version avec la *MapMemory* peut paraitre lente, car à priori plusieurs requêtes (une par case du tableau source) sont nécessaires (sur le pci-express) pour obtenir les données sources se trouvant sur le host, alors qu'avec la *global memory* une requête est suffisante pour obtenir tout le tableau en une fois. Cependant grâce au pattern d'entrelacement et au mode SIMD du GPU, les requêtes sur le *pci-express* peuvent être regroupés dans le cas de la *MapMemory*. Par ailleurs avec la technique de la GM, on lit les données sur le host, on les écrit sur la GM, puis on les relit une seconde fois pour calculer l'histogramme : il y a donc une opération de lecture de plus par case ! Au final la *MapMemory* ne serait-elle pas plus rapide que la GM. Qu'observez-vous ?

Notes :

(N1) La *MapMemory* est donc intéressante si les données ne doivent être lues qu'une fois de manière contiguë.

(N2) Cette technique est d'autant plus intéressante, s'il n'y a pas assez de place sur le GPU pour tout copier d'un cout !

Piège

N'oubliez pas l'initialisation de

- L'histogramme final en GM
- Des histogrammes locaux en SM

Dans le premier cas, un `cudaMemset` faira l'affaire.

```
size_t sizeOctet=sizeof(int)*256;      // poids histogramme en GM
HANDLE_ERROR(cudaMemset(ptrDevHistoGM,0,sizeOctet)) ;
```

Dans le second cas, initialiser à zéro les case de `tabSM`, en parallèles ! Chaque thread s'occupera d'une ou plusieurs cases de `tabSM`. Tout dépendera si vous avez plus de thread par blocks que les 256 cases de `tabSM` (représentant l'histogramme local au block).

Attention

N'oubliez pas les

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

Mais soyez minimaliste !

Variation

N'utilisez pas de SM, mais un unique histogramme en GM que tous les threads se partagent. Le problème de concurrence sera contourné avec un `atomicADD`. Comparer les performances avec la version en SM. Depuis cuda 5, `atomicADD` a été super optimisé en GM. Vous en pensez quoi ?

Généralisation

Généraliser votre code de telle sorte qu'il soit fonctionnel avec un tableau d'input `tabData` dont chaque élément est compris entre 0 et M-1 avec M une puissance de 2.

Speedup

Mesurer les coefficients de *speedup* des différentes implémentations.

Pour canevas, utiliser le document

speedup_simple.xls.

Au besoin adapter ce canevas.

End