UF I5AISE51
Dimensionnement et évaluation des architectures
Introduction à la programmation massivement parallèle

—

Partie 7
Instructions atomiques
Exemple sur le calcul d'histogramme

—

P.-E. Hladik

INSA Toulouse

11 janvier 2021

Plan

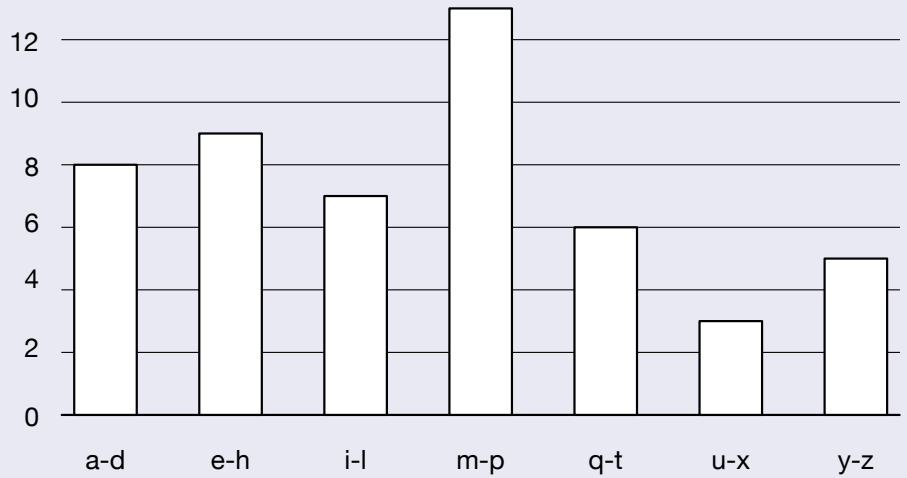
Calcul d'un histogramme

Principe

Chaque élément dans un ensemble de données est associé à un "conteneur".

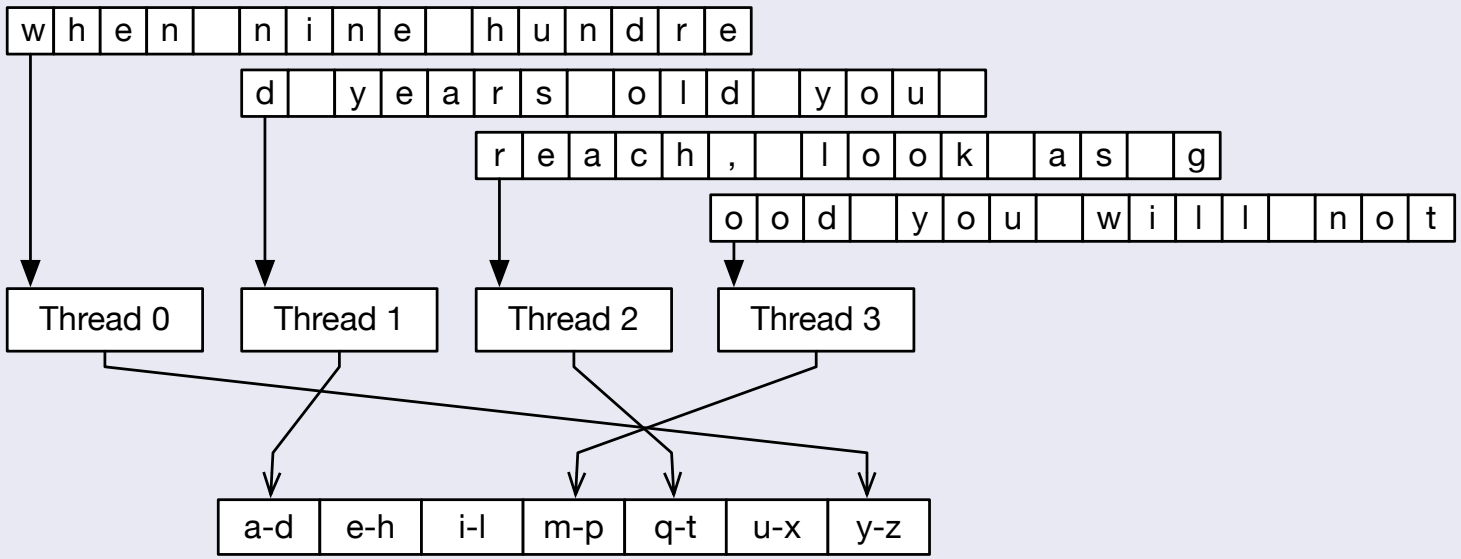
Exemple :

- Conteneurs : sections de 4 lettres de l'alphabet : a-d, e-h, i-l,...
- Pour chaque caractère d'une chaîne, on incrémente le compteur du conteneur correspondant
- Dans la phrase "When nine hundred years old you reach, look as good you will not" on obtient :



Algorithme parallèle simple : histogramme

- Les entrées sont divisées en sections
- Chaque thread prend une section comme entrée
- Chaque thread itère dans sa section
- Pour chaque lettre le compteur de conteneur est incrémenté



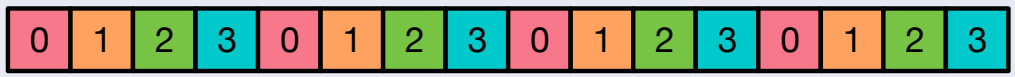
Accès mémoire sous-optimale

Le découpage par sections entraîne une faible efficacité d'accès à la mémoire

- Les threads n'accèdent pas à des données adjacentes
- les accès ne sont pas regroupés (coalesced)
- La bande passante des DRAM est sous-utilisée

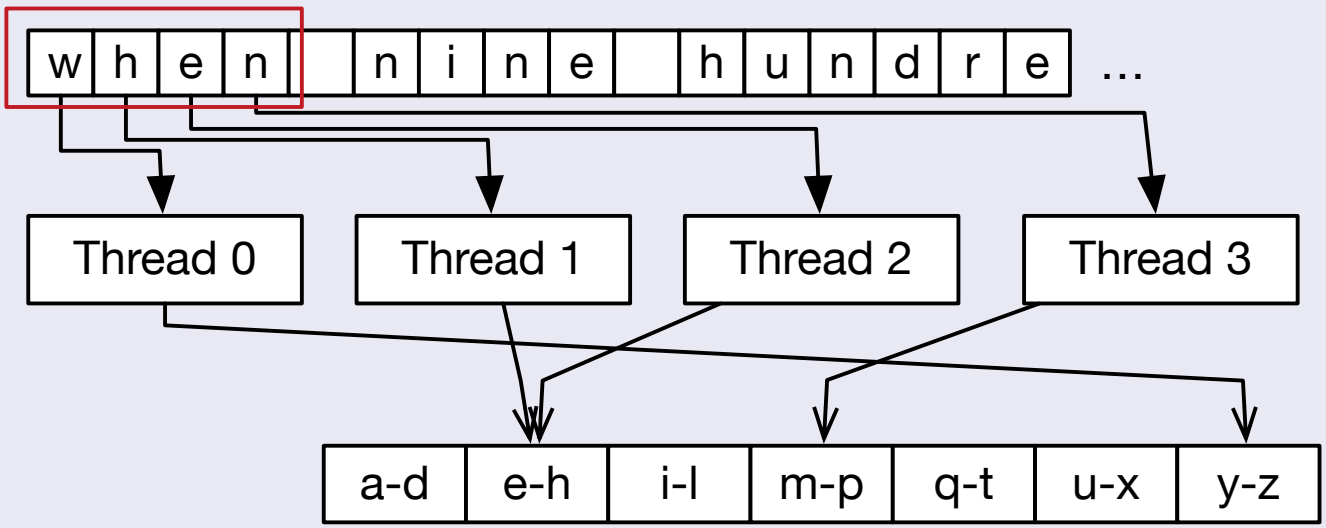
Optimisation : entrelacer les sections

- Les threads procèdent par sections contiguës
- les accès sont ainsi regroupés (coalesced)



Entrelacement des sections

- Les entrées sont divisées en sections
- Chaque thread prend une section comme entrée
- Chaque thread itère dans sa section
- Pour chaque lettre le compteur de conteneur est incrémenté



Accès concurrent

La mise à jour des compteurs se fait en trois étapes :

1. old <- Mem[x]
2. new <- old + 1
3. Mem[x] <- new

Il peut y donc y avoir une incohérence dû à l'accès concurrent des threads.

Accès concurrent

La mise à jour des compteurs se fait en trois étapes :

1. old <- Mem[x]
2. new <- old + 1
3. Mem[x] <- new

Il peut y donc y avoir une incohérence dû à l'accès concurrent des threads.

Instant	Thread 0	Thread 1
1	(0) old <- Mem[x]	
2	(1) new <- old + 1	
3	(1) Mem[x] <- new	
4		(1) old <- Mem[x]
5		(2) new <- old + 1
6		(2) old <- Mem[x]

Accès concurrent

La mise à jour des compteurs se fait en trois étapes :

1. old <- Mem[x]
2. new <- old + 1
3. Mem[x] <- new

Il peut y donc y avoir une incohérence dû à l'accès concurrent des threads.

Instant	Thread 0	Thread 1
1	(0) old <- Mem[x]	
2	(1) new <- old + 1	
4		(0) old <- Mem[x]
3	(1) Mem[x] <- new	
5		(1) new <- old + 1
6		(1) old <- Mem[x]

Principe des opérations atomiques

- Une opération de lecture-modification-écriture est réalisée en une seule instruction matérielle
- Le matériel garantit qu'aucun autre thread ne peut effectuer une autre opération de lecture-modification-écriture sur le même emplacement jusqu'à ce que l'opération atomique en cours soit terminée.
  - Tout autre thread qui tente d'effectuer une opération atomique au même endroit sera placé dans une file d'attente
  - Tous les threads effectuent leurs opérations atomiques en **série** pour une même adresse mémoire

Opérations arithmétiques atomiques en CUDA

- Effectuées par l'appel de fonctions qui sont traduites en une unique instruction :
  - addition , soustraction, incrémentation, décrémentation , min, max, échange, CAS (comparaison et échange)

Addition atomique

```
int atomicAdd(int* adresse, int val);
```

- lit le mot de 32 bits ancien à partir de l'emplacement pointé par l'adresse dans la mémoire globale ou partagée,
- calcule (ancien + val),
- stocke le résultat en mémoire à la même adresse
- retourne l'ancienne valeur.

Ces trois opérations sont effectuées en une seule instruction.

Kernel simple pour le calcul d'un histogramme

```
__global__
void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  // stride is total number of threads
  int stride = blockDim.x * gridDim.x;

 // All threads handle blockDim.x * gridDim.x
 // consecutive elements
 while (i < size) {
    atomicAdd( &(histo[buffer[i]]), 1);
    i += stride;
  }
}
```

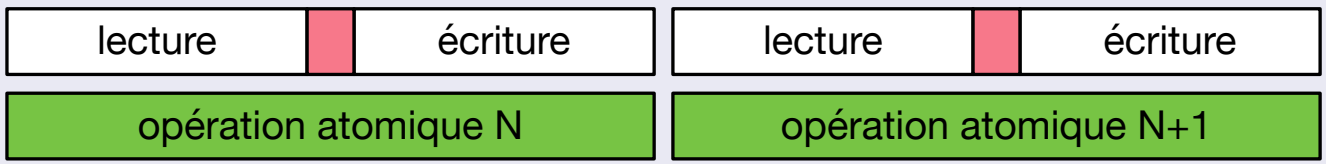
Exemple sur une chaîne de caractères

```
__global__
void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  // stride is total number of threads
  int stride = blockDim.x * gridDim.x;

 // All threads handle blockDim.x * gridDim.x
 // consecutive elements
   while (i < size) {
      int alphabet_position = buffer[i] - 'a';
      if (alphabet_position >= 0 && alpha_position < 26)
                atomicAdd(&(histo[alphabet_position/4]), 1);
       i += stride;
   }
}
```

Quelques éléments de performance

- Une opération atomique commence par une lecture en DRAM, qui a une latence de quelques centaines de cycles
- L'opération atomique se termine par une écriture au même endroit, avec une latence de quelques centaines de cycles
- Pendant tout ce temps, personne d'autre ne peut accéder à l'adresse

- Chaque lecture-modification-écriture induit deux délais d'accès à la mémoire
- Toutes les opérations atomiques sur une même variable (emplacement de la DRAM) sont sérialisées



Exemple de privatisation

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *
     histo)
{
  __shared__ unsigned int histo_private[7];
  if (threadIdx.x < 7) histo_private[threadidx.x] = 0;
  __syncthreads();

  int i = threadIdx.x + blockIdx.x * blockDim.x;
  // stride is total number of threads
  int stride = blockDim.x * gridDim.x;

  while (i < size) {
    atomicAdd( &(private_histo[buffer[i]/4), 1);
    i += stride;
  }
     // wait for all other threads in the block to finish
  __syncthreads();

  if (threadIdx.x < 7) {
          atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x] );
  }
}
```