Calcul GPU – Cours 3: Patterns de calcul

Jonathan Rouzaud-Cornabas

LIRIS / Insa de Lyon – Inria Beagle

Cours inspiré de ceux de Prof Wen-mei Hwu
(University of Illinois at Urbana–Champaign)

Références

- D. Kirk and W. Hwu, "Programming Massively Parallel Processors – A Hands-on Approach," Morgan Kaufman Publisher, 3rd edition

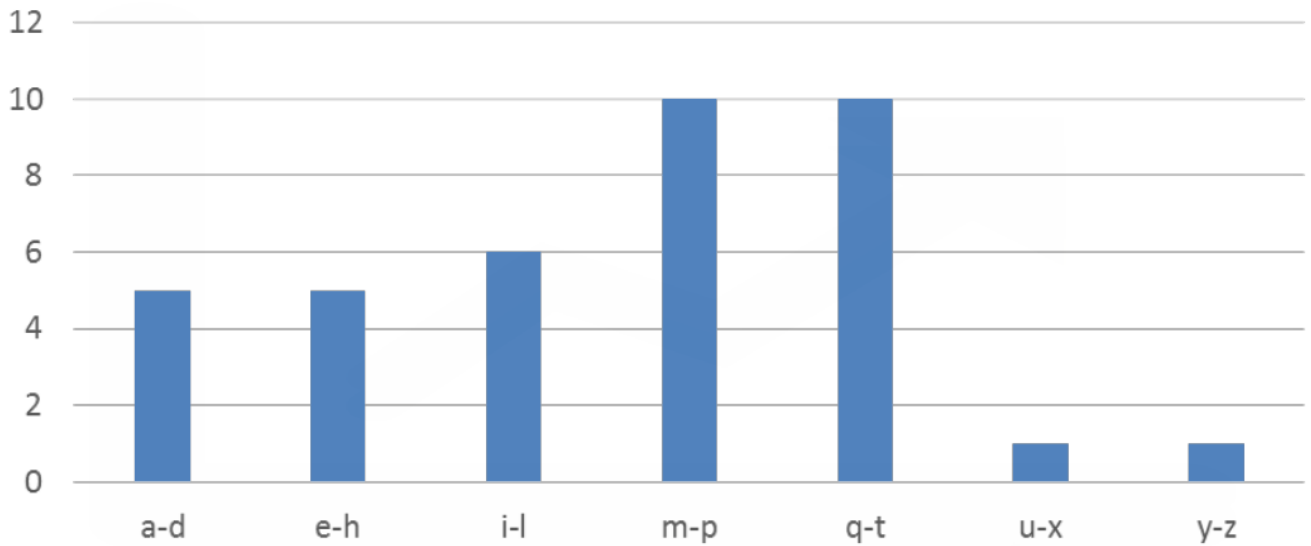
- NVIDIA, NVidia CUDA C Programming Guide, version 8.0, NVidia, 2016 (reference book)

Histogramme

- Une méthode d'extraction de caractéristique et de modèle pour les grands ensembles de données
  - Extraction de caractéristiques pour la reconnaissance d'objets dans des images
  - Détection de fraudes dans les transactions par cartes de crédit
  - Corrélation des mouvements d'objets célestes en astrophysique
  - ...
- Fonctionnement de base des histogrammes: Pour chaque élément dans l'ensemble de données, utiliser une valeur pour identifier un/une sac/caractéristique à incrémenter

Exemple: un Histogramme de texte

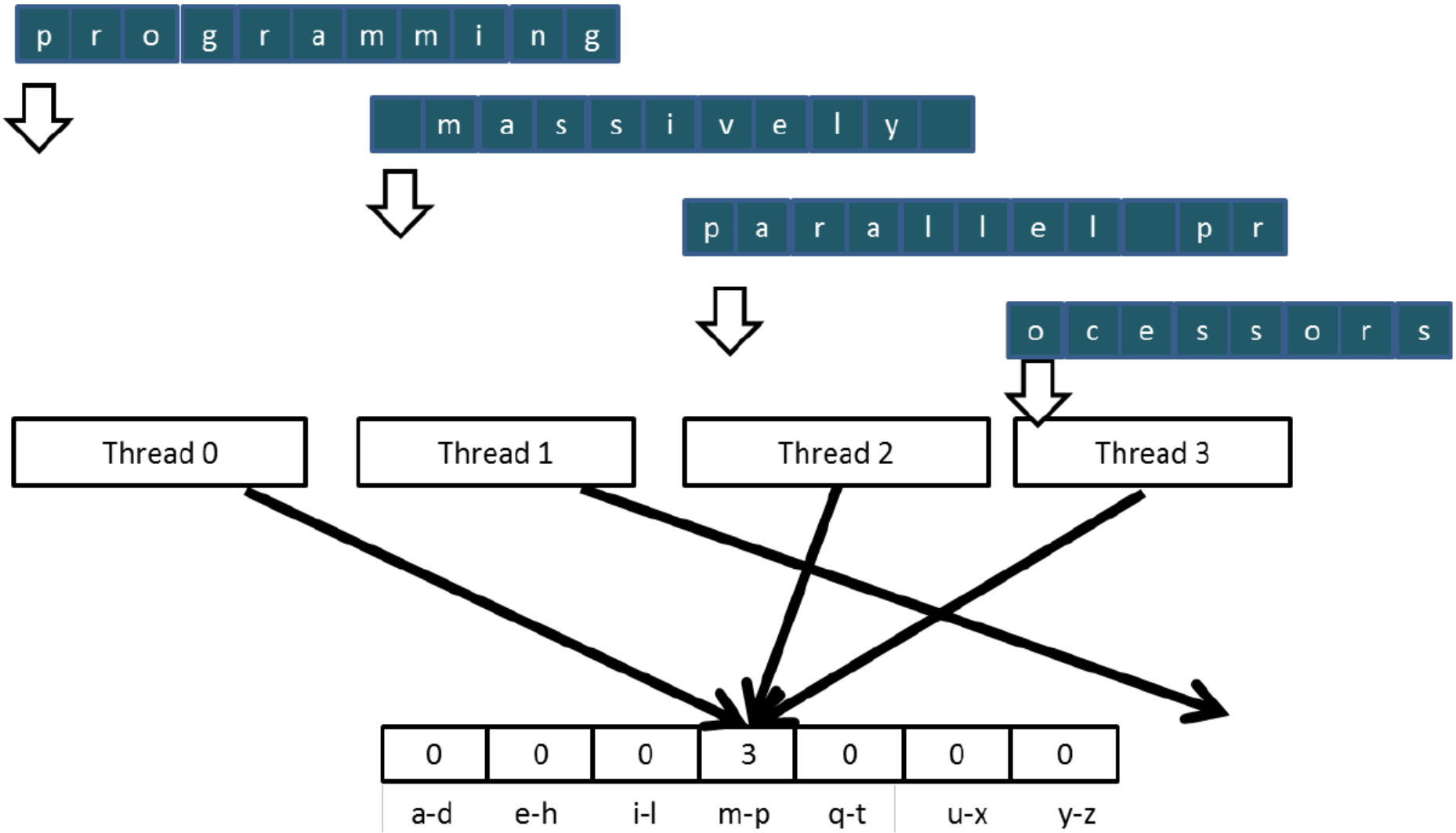
- Définir les sacs comme des ensembles de 3 lettres de l'alphabet: a-d, e-h, i-l, n-p, ...
- Pour chaque caractère dans la chaine en entrée, incrémenter le compteur du sac correspondant
- Exemple de résultats pour la phrase "Programming Massively Parallel Processors"



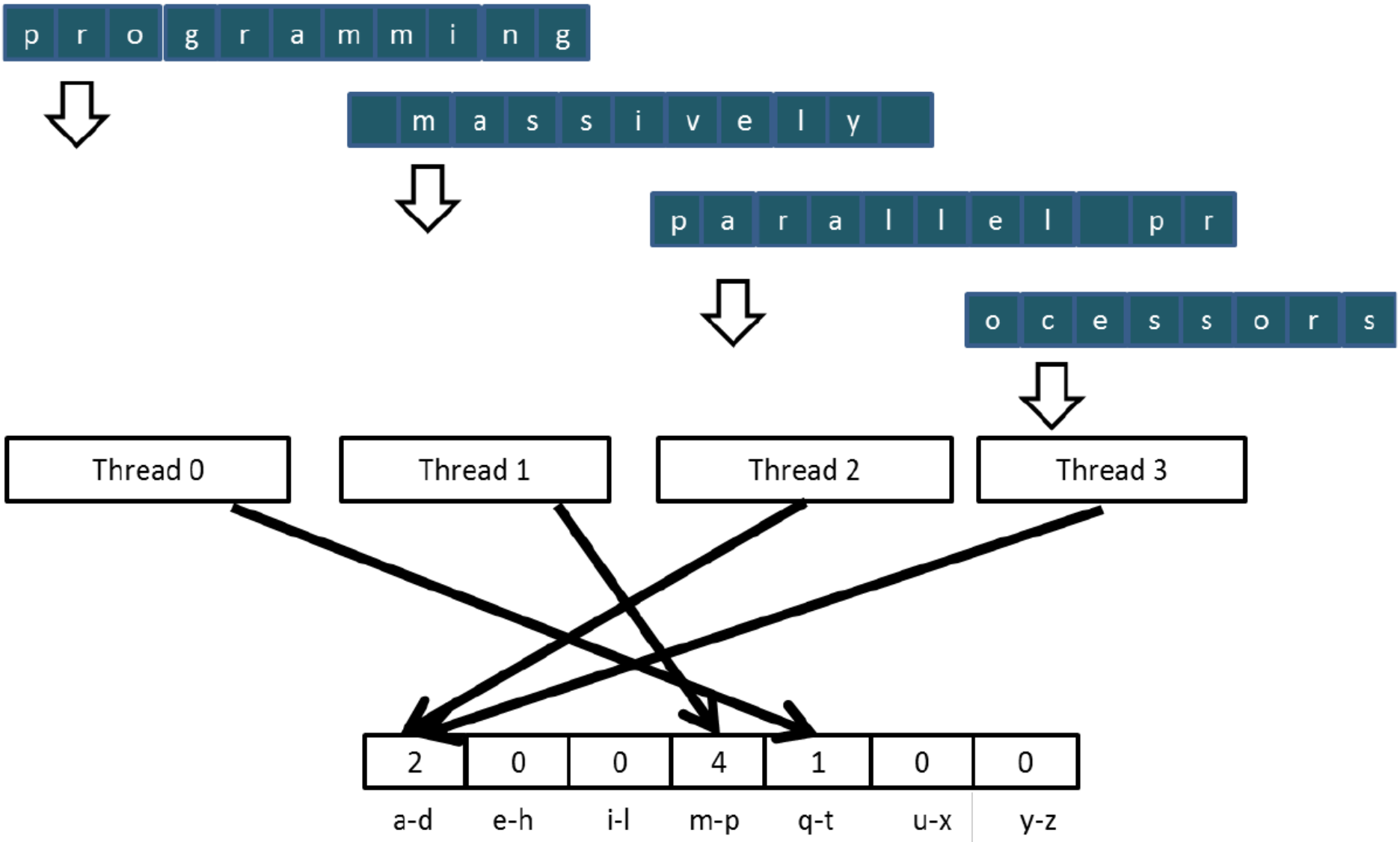
Algorithme simple de parallélisation d'un histogramme

- Partionnement de l'entrée en sections
- Chaque thread est en charge de traiter une section
- Chaque thread parcourt sa section
- Pour chaque lettre, le compteur du sac correspondant est incrémenté

Sections partionnées (Iteration #1)

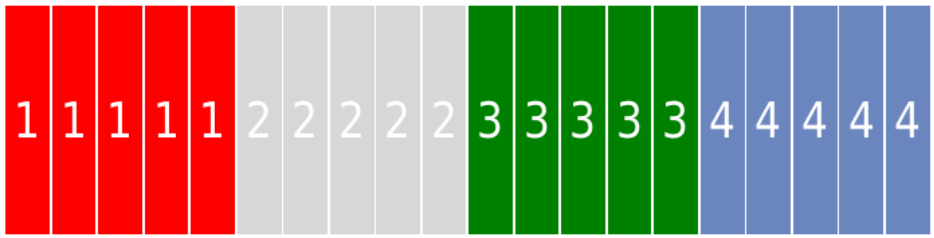


Sections partionnées (Iteration #2)



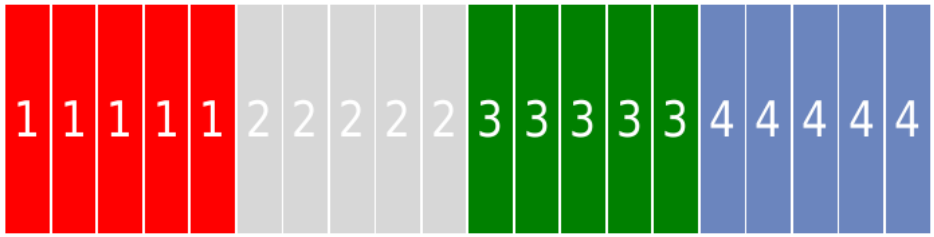
Effets du partionnement sur l'efficacité des accés mémoires

- Le partionnnement en sections peut avoir un impact négatif sur l'efficacité des accès mémoires
  - Les threads adjacents n'accédent pas à des adresses mémoires adjacentes
  - Les accés ne sont pas coalescent
  - La bande passante mémoire n'est pas utilisé efficacement

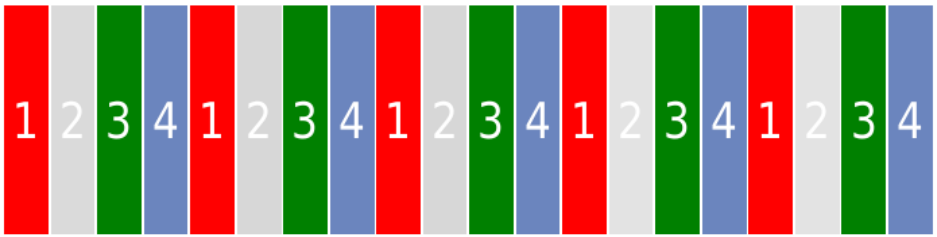


Effets du partionnement sur l'efficacité des accés mémoires

- Le partionnnement en sections peut avoir un impact négatif sur l'efficacité des accès mémoires
  - Les threads adjacents n'accédent pas à des adresses mémoires adjacentes
  - Les accés ne sont pas coalescent
  - La bande passante mémoire n'est pas utilisé efficacement

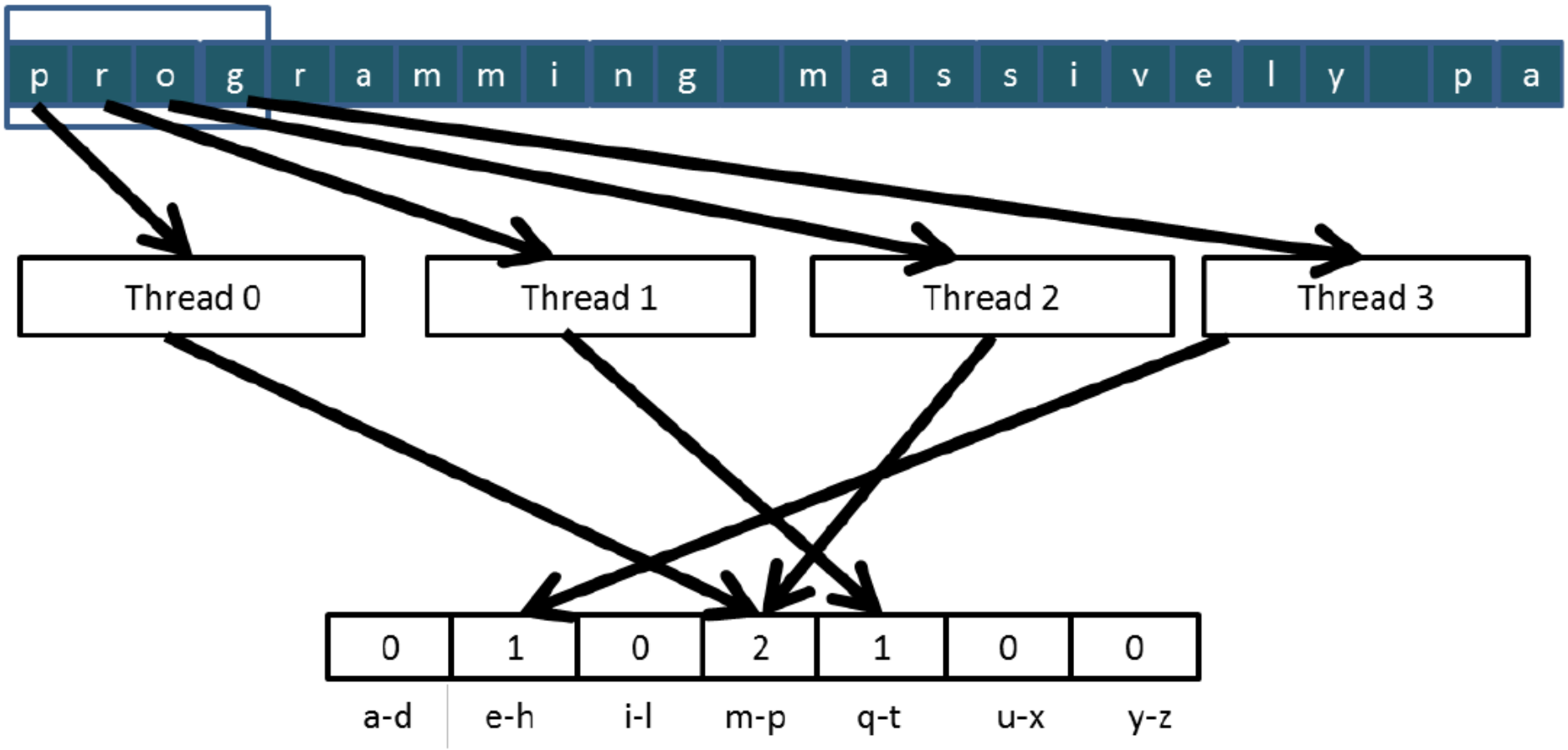


- Changement pour un algorithme de partionnement entrelacé
  - Tous les threads calculent une section contigue d'éléments
  - Ils se déplacent tous vers la prochaine section et recommencent
  - La mémoire est accédée de manière coalescente



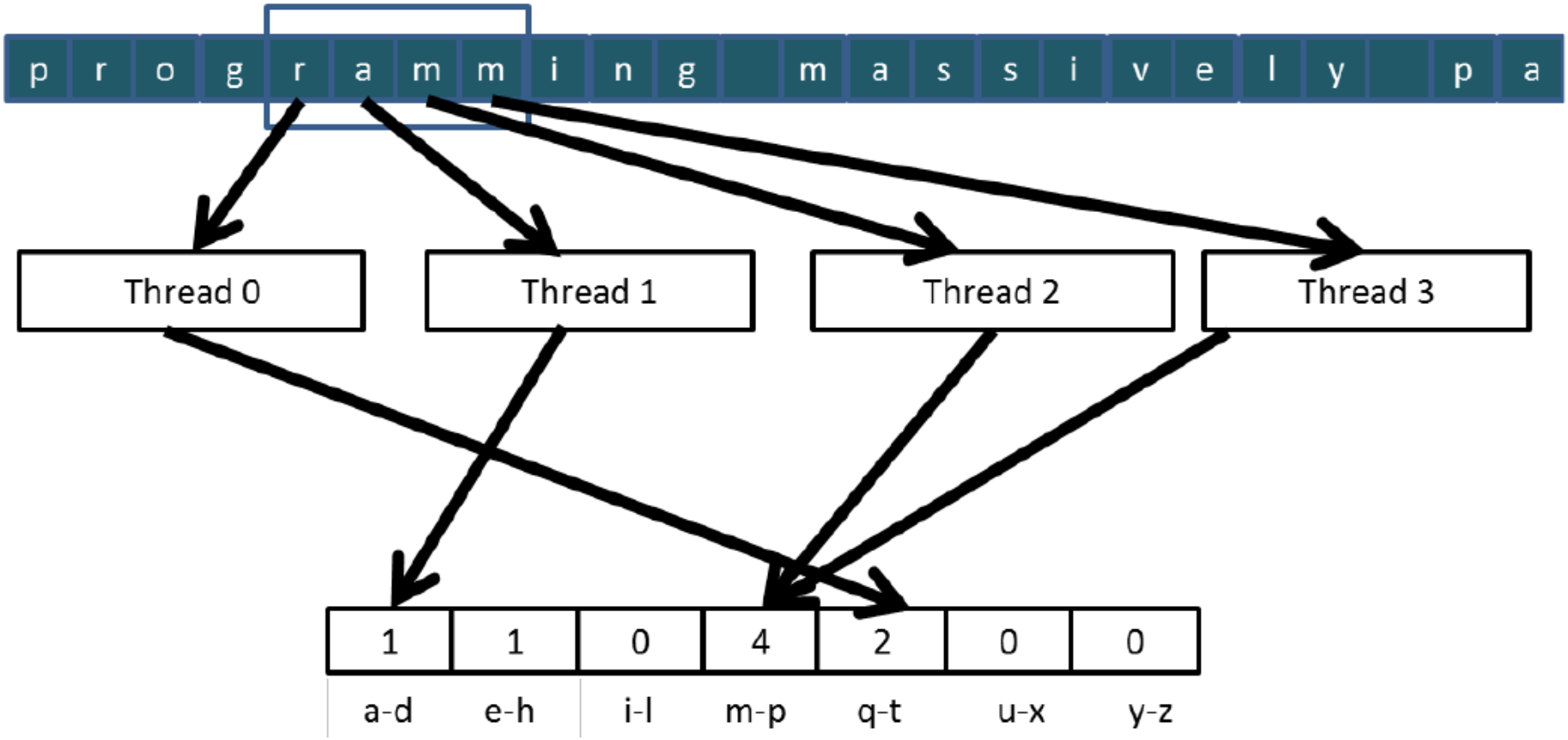
Partionnement entrelacé de l'entrée (Iteration #1)

Pour des accés mémoire coalescent et de meilleur performance



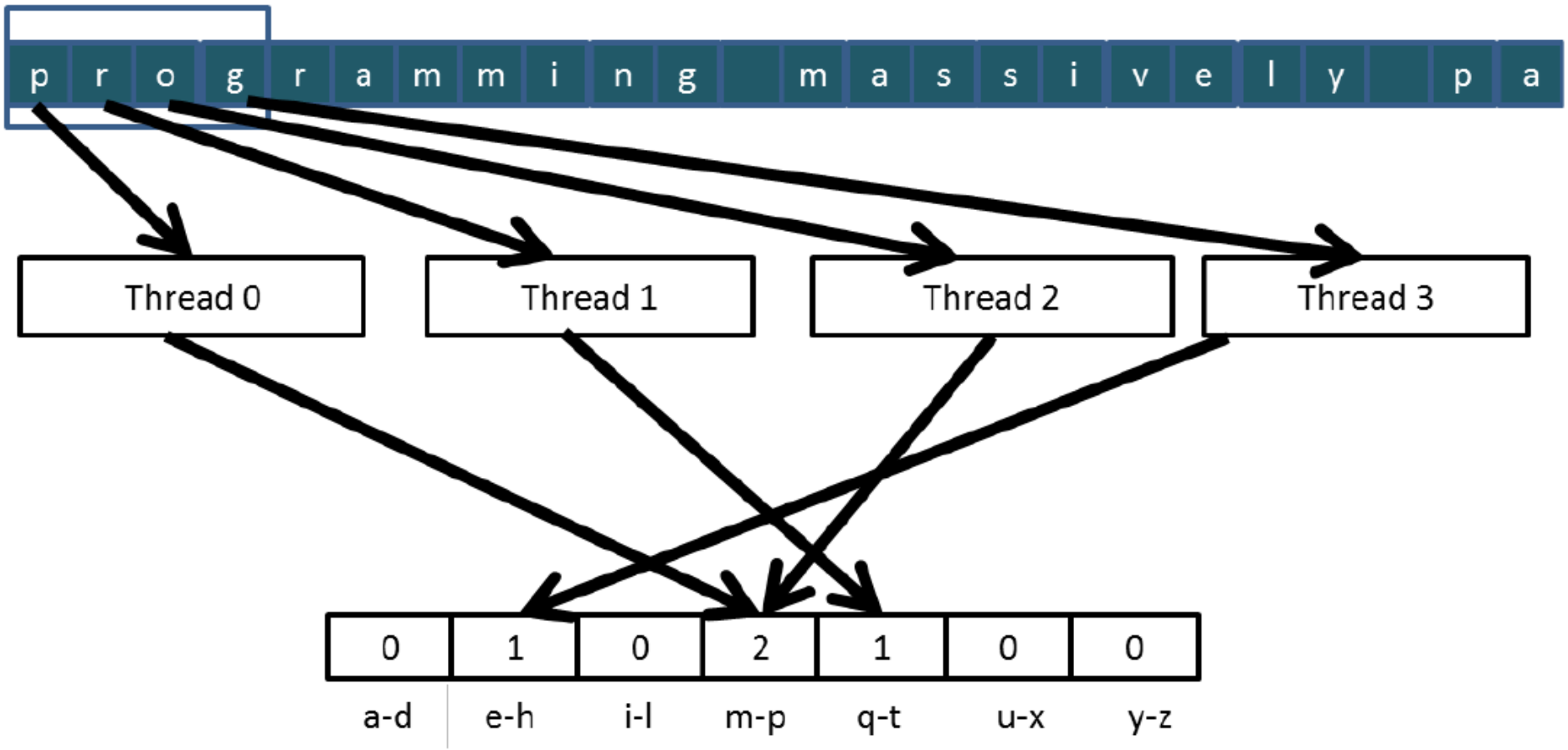
Partionnement entrelacé de l'entrée (Iteration #2)

Pour des accés mémoire coalescent et de meilleur performance



Example d'histogramme: opération Lecture-Modifier-Ecrire

Pour des accés mémoire coalescent et de meilleur performance



L'opération Lecture-Modifier-Ecrire utilisée dans les modèles collaboratifs

- Par exemple, plusieurs caissiers compte la quantité totale de liquide dans un coffre
- Chacun prend une pile et la compte
- Il y a un affichage centralisé de la somme déjà calculée
- Chaque fois que quelqu'un finit de compter une pile, il lit la somme totale courante (lecture), y ajoute le sous-total de sa pile (modifier-écrire)
- Un problème récurrent : Le sous-total de certaines piles n'est pas présent dans le somme final

Un service commun et parallèle de caractérisation

- Par exemple, plusieurs personnels de service client servent se partage une file d'attente
- Le système doit maintenir 2 nombres:
  - le nombre à donner à la prochaine personne entrant dans la file d'attente (I)
  - le nombre de la prochaine personne à être servie (S)
- Le système donne un numéro à chaque personne entrant dans la file (lecture I) et incrémente de 1 ce nombre pour le donner au prochain client entrant (modifier-écrire I)
- Un panneau d'affichage affiche le nombre du prochain client à servir
- Quand un personnel est disponible, il appelle le nombre (read S) et incrémente de 1 le nombre affiché (modifier-écrire S)
- Problèmes
  - Plusieurs clients recoivent le même nombre, seulement un est servi
  - Plusieurs personnels servent le même nombre

Une régle commune d'arbitrage

- Par exemple, plusieurs clients réservent des billets d'avion en parallèle
- Chacun
  - Regarde le plan des sièges dans l'avion (lecture)
  - Décide quelle siége ils souhaitent
  - Mets à jour le plan des sièges et marque celui sélectionné comme pris (modifier-écrire)
- Problèmes
  - Plusieurs passagers peuvent avoir réservés le même siége

Situation de concurrence lors de l'exécution de threads en parallèle

thread1:
Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2:
Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

- *Old* et *New* sont des variables allouées dans des registres et donc spécifique à un thread

- Question #1 : Si *Mem*[*x*] était initialisé à 0; qu'elle serait la valeur de *Mem*[*x*] après que les threads 1 et 2 aient fini leur exécution ?
- Question #2 : Qu'est-ce que les threads ont dans leur variable *Old* ?

Situation de concurrence lors de l'exécution de threads en parallèle

thread1:
Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2:
Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

- *Old* et *New* sont des variables allouées dans des registres et donc spécifique à un thread

- Question #1 : Si *Mem*[*x*] était initialisé à 0; qu'elle serait la valeur de *Mem*[*x*] après que les threads 1 et 2 aient fini leur exécution ?
- Question #2 : Qu'est-ce que les threads ont dans leur variable *Old* ?

- Malheuresement, la réponse peut varier suivant le minutage relatif entre l'exécution des 2 threads *i.e.*, nous somme dans une situation de concurrence sur les données

Scénario #1 du minutage des threads

Temps	Thread 1	Thread 2
1	(0) $Old \leftarrow Mem[x]$	
2	(1) $New \leftarrow Old + 1$	
3	(1) $Mem[x] \leftarrow New$	
4		(1) $Old \leftarrow Mem[x]$
5		(2) $New \leftarrow Old + 1$
6		(2) $Mem[x] \leftarrow New$

- Thread 1 $Old = 0$
- Thread 2 $Old = 1$
- $Mem[x] = 2$ après la séquence d'exécution

Scénario #2 du minutage des threads

Temps	Thread 1	Thread 2
1		(0) $Old \leftarrow Mem[x]$
2		(1) $New \leftarrow Old + 1$
3		(1) $Mem[x] \leftarrow New$
4	(1) $Old \leftarrow Mem[x]$	
5	(2) $New \leftarrow Old + 1$	
6	(2) $Mem[x] \leftarrow New$	

- Thread 1 $Old = 1$
- Thread 2 $Old = 0$
- $Mem[x] = 2$ après la séquence d'exécution

Scénario #3 du minutage des threads

Temps	Thread 1	Thread 2
1	(0) $Old \leftarrow Mem[x]$	
2	(1) $New \leftarrow Old + 1$	
3		(1) $Old \leftarrow Mem[x]$
4	(1) $Mem[x] \leftarrow New$	
5		(1) $New \leftarrow Old + 1$
6		(1) $Mem[x] \leftarrow New$

- Thread 1 $Old = 0$
- Thread 2 $Old = 0$
- $Mem[x] = 1$ après la séquence d'exécution

Scénario #4 du minutage des threads

Temps	Thread 1	Thread 2
1		(0) $Old \leftarrow Mem[x]$
2		(1) $New \leftarrow Old + 1$
3	(0) $Old \leftarrow Mem[x]$	
4		(1) $Mem[x] \leftarrow New$
5	(1) $New \leftarrow Old + 1$	
6	(1) $Mem[x] \leftarrow New$	

- Thread 1 $Old = 0$
- Thread 2 $Old = 0$
- $Mem[x] = 1$ après la séquence d'exécution

Le but des opérations atomiques: s'assurer de résultats exactes

```
thread1:  Old ← Mem[x]
          New ← Old + 1
          Mem[x] ← New

                                thread2:  Old ← Mem[x]
                                          New ← Old + 1
                                          Mem[x] ← New
```

OU

```
                                thread2:  Old ← Mem[x]
                                          New ← Old + 1
                                          Mem[x] ← New

thread1:  Old ← Mem[x]
          New ← Old + 1
          Mem[x] ← New
```

Les concepts de base des opérations atomiques

- L'opération Lecture-Modifier-Ecrire est effectuée à une adresse mémoire par une unique instruction matériel
  - Lire l'ancienne valeur, calculer la nouvelle valeur, et écrire la nouvelle valeur dans l'adresse mémoire désignée
- Le matériel garantit qu'aucun autre thread puisse effectuer une autre opération Lecture-Modifier-Ecrire à la même adresse jusqu'à la fin de l'opération atomique
  - Tous autres threads qui tentent d'effectuer une opération atomique sur la même adresse seront retenus dans une file d'attente
  - Tous les threads qui effectuent une opération atomique à la même adresse seront donc sérialisés

Les opérations atomiques en CUDA

- Les opérations atomiques sont accessible en appellant des fonctions spécifiques qui sont traduites en instructions uniques (aussi appellées *intrinsic functions* ou *intrinsics*)
  - Atomic *add*, *sub*, *inc*, *dec*, *min*, *max*, *exch* (échange), *CAS* (compare et échange)
  - Lire la documentation de CUDA pour plus de détails

- Fonction atomique d'ajout : `int atomicAdd(int* address, int val);`
- Lit un mot *old* de 32-bit à l'adresse pointée par *address* en mémoire globale ou partagée, calcule $(old + val)$ et écrit le résultat en mémoire à la même adresse. La fonction retourne *old*.

D'autres fonctions atomiques d'ajout en CUDA

- Ajout atomique de nombre entiers 32-bit non signés
  `unsigned int atomicAdd(unsigned int* adress, unsigned int val);`
- Ajout atomique de nombre entiers 64-bit non signés
  `unsigned long long int atomicAdd(unsigned long long int* adress, un`
- Ajout atomique de nombre flottants simple précision
  `float atomicAdd(float* adress, float val);`

Un exemple simple d'un noyau d'histogramme sur du texte (1/2)

- Le noyau reçoit un pointer sur un buffer en entrée
- Chaque thread est en charge de calculer un sous ensemble du buffer en entrée

```
__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;
  }
}
```

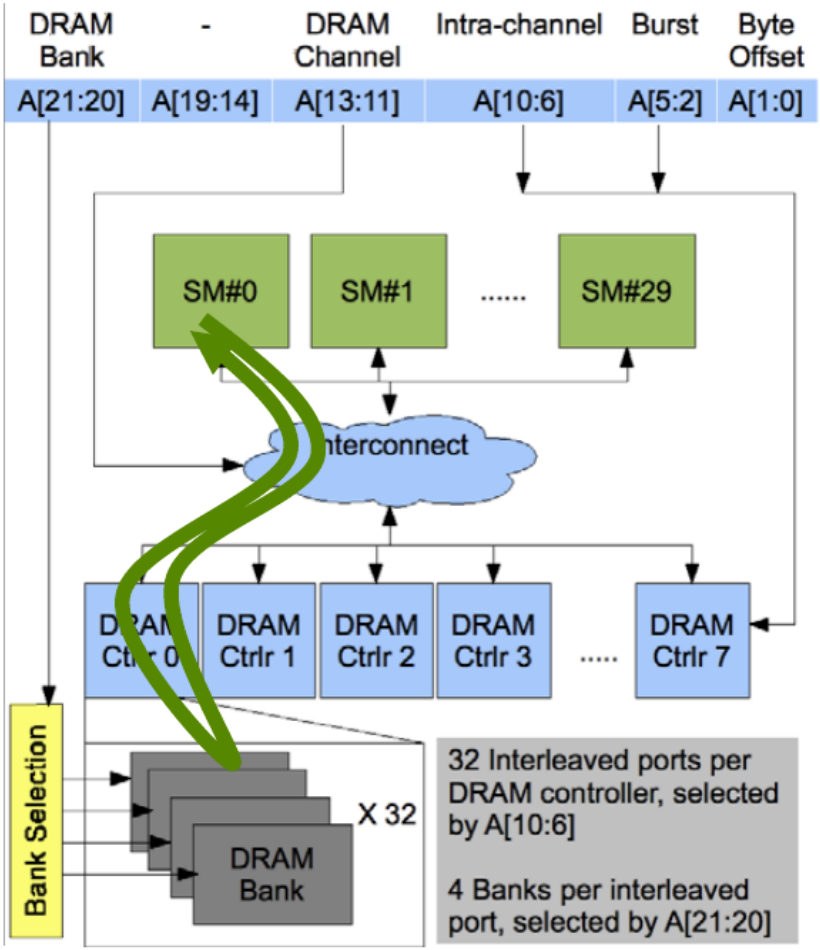
Un exemple simple d'un noyau d'histogramme sur du texte (2/2)

- Le noyau reçoit un pointer sur un buffer en entrée
- Chaque thread est en charge de calculer un sous ensemble du buffer en entrée

```
__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;
  }
```

Les opérations atomiques en mémoire globale (DRAM) (1/2)

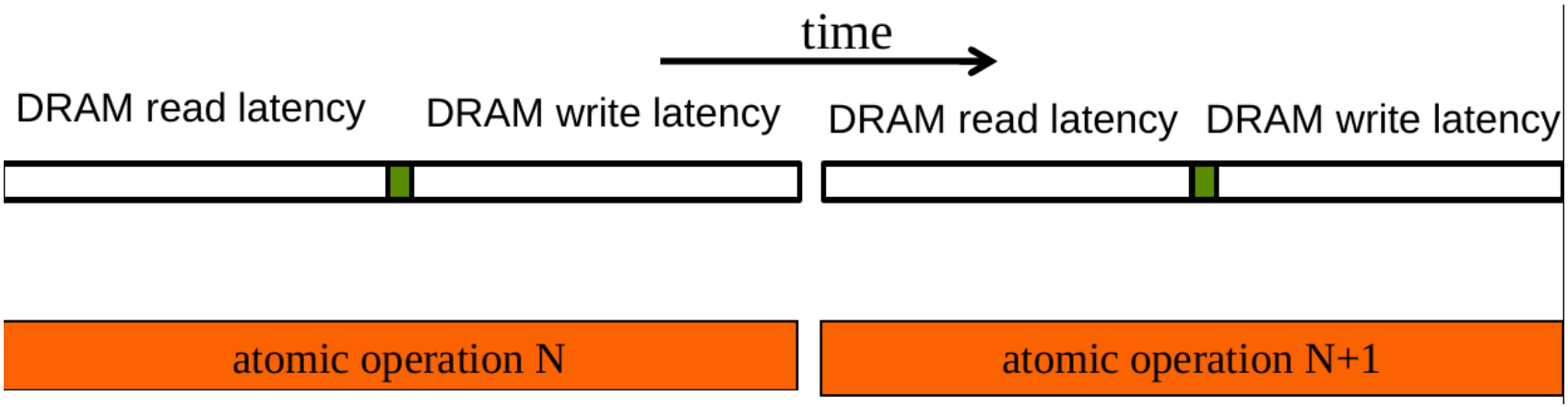
- Une opération atomique sur une adresse DRAM commence par une lecture qui a une latence de quelques centaines de cycles
- Une opération atomique finit par une écriture sur la même adresses avec également une latence de quelques centaines de cycles
- Durant ce temps, personne d'autre ne peut accéder à cette adresse



Les opérations atomiques en mémoire globale (DRAM) (2/2)

Chaque opération Lecture-Modifier-Ecrire paye 2 latences d'accés à la mémoire

- Toutes les opérations atomiques sur la même variable (adresse DRAM) sont sérialisées



La latence détermine la bande passante

- La bande passante des opérations atomiques sur la même adresse mémoire en DRAM peut être définie comme la vitesse à laquelle chaque application peut exécuter une opération atomique
- La vitesse des opérations atomiques sur une adresse est limitée par la latence totale de la séquence Lecture-Modifier-Ecrire, en général, plus de 1000 cycles pour les adresses en mémoire globale (DRAM)
- Cela signifie que si plusieurs threads essayent d'effectuer des opérations atomiques sur la même adresse (contention), la bande passante mémoire est réduite à être inférieur à $\frac{1}{1000}$ de la bande passante maximun d'un seul canal mémoire !

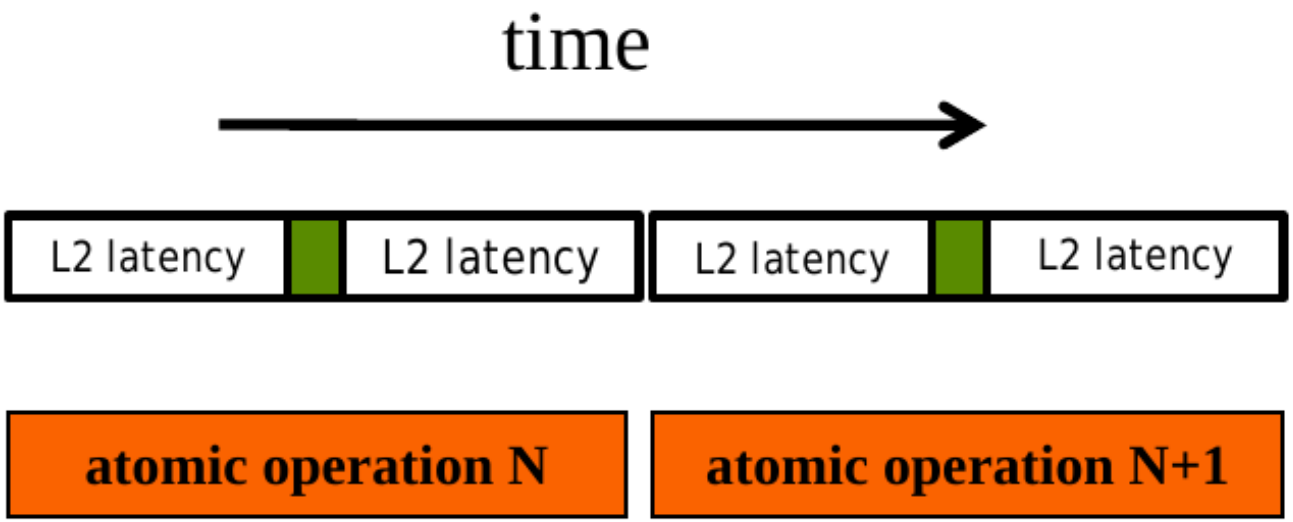
Exemple similaire : Passer à la caisse dans un supermarché

- Certains clients réalisent qu'ils ont oublié un article après qu'ils aient commencé à passer en caisse
- Ils courent dans les allées et récupérent l'article manquant pendant que la file attend
  - La vitesse de passage en caisse est réduite drastiquement à cause de la longue latence de la course dans les allées du supermarché et revenir en caisse
- Imaginer un magasin où chaque client commence à faire la queue avant qu'ils aient commencé à prendre des articles
  - La vitesse de passage en caisse sera de 1 sur le temps total de course de chaque client

Amélioration du matériel: Opération atomique dans le cache L2

Opération atomique en cache L2

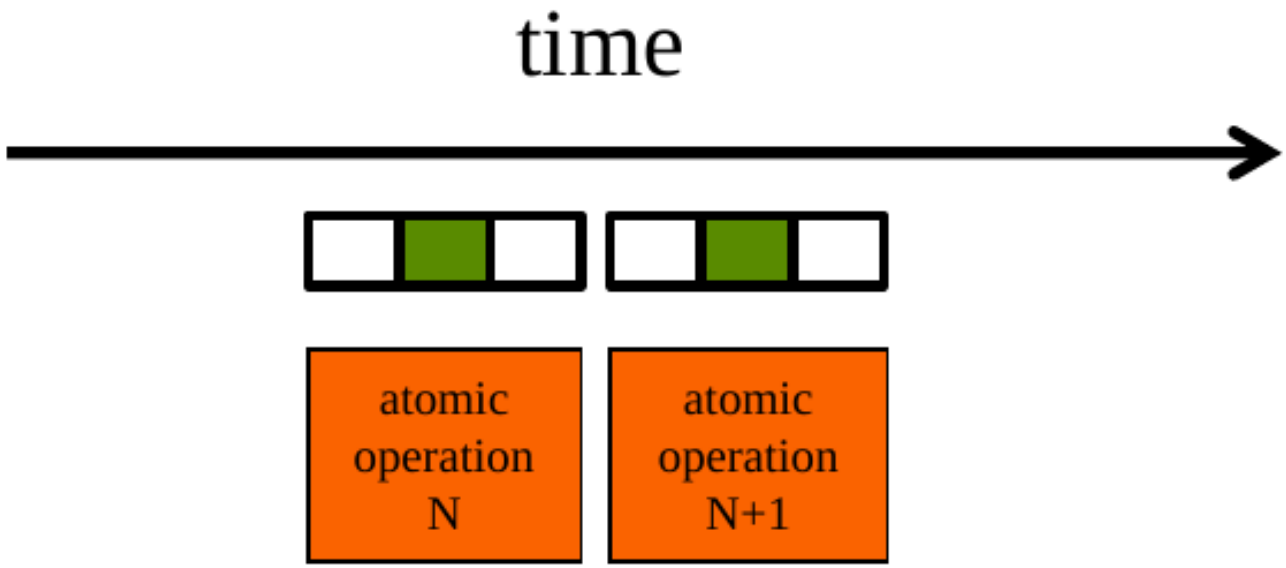
- Latence moyenne, environ $\frac{1}{10}$ de celle de la DRAM
- Partagée entre tous les blocs
- Coût en terme de développement/algorithmique presque nul par rapport à la mémoire globale



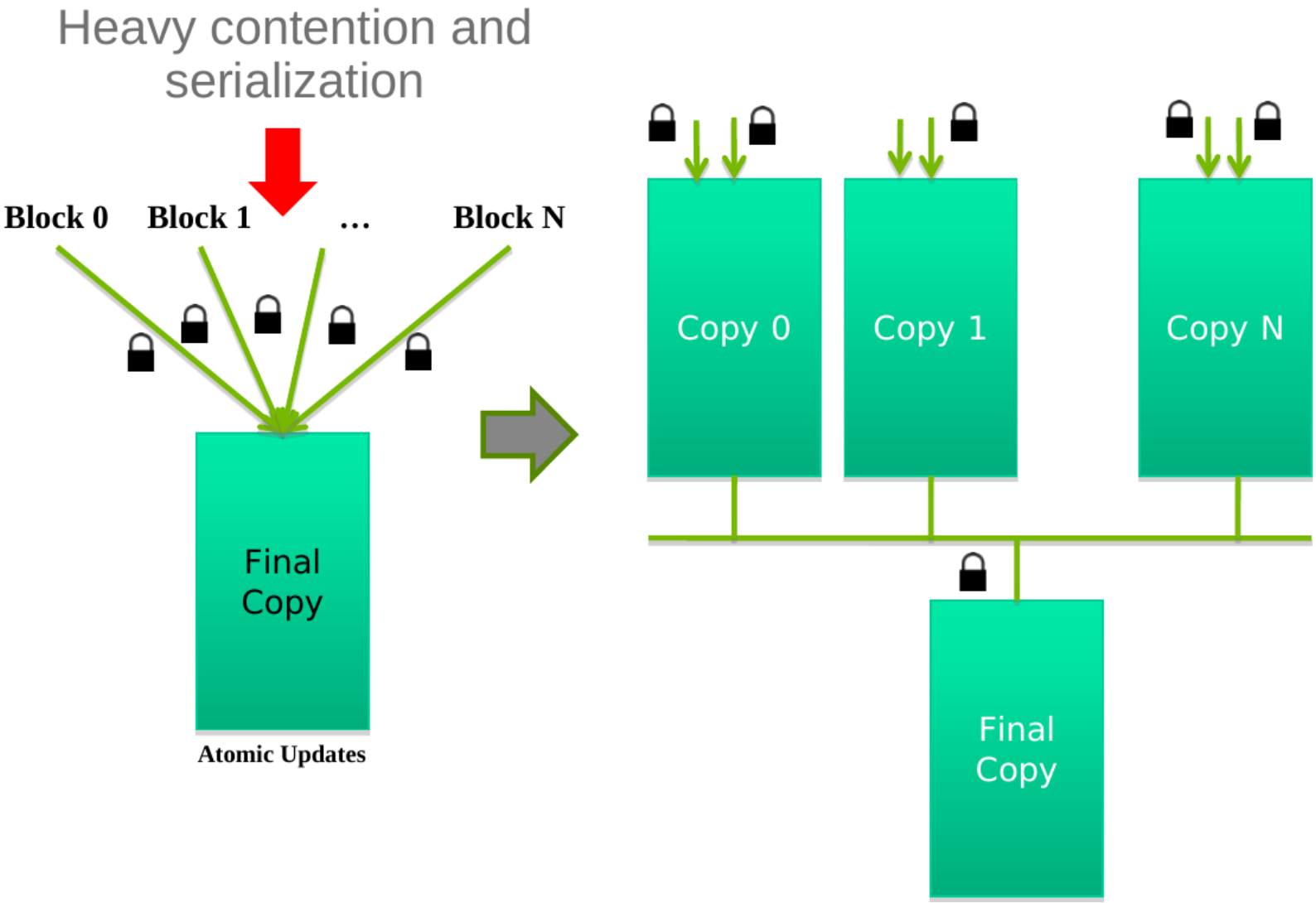
Amélioration du matériel: Opération atomique dans la mémoire partagée

Opération atomique en mémoire partagée

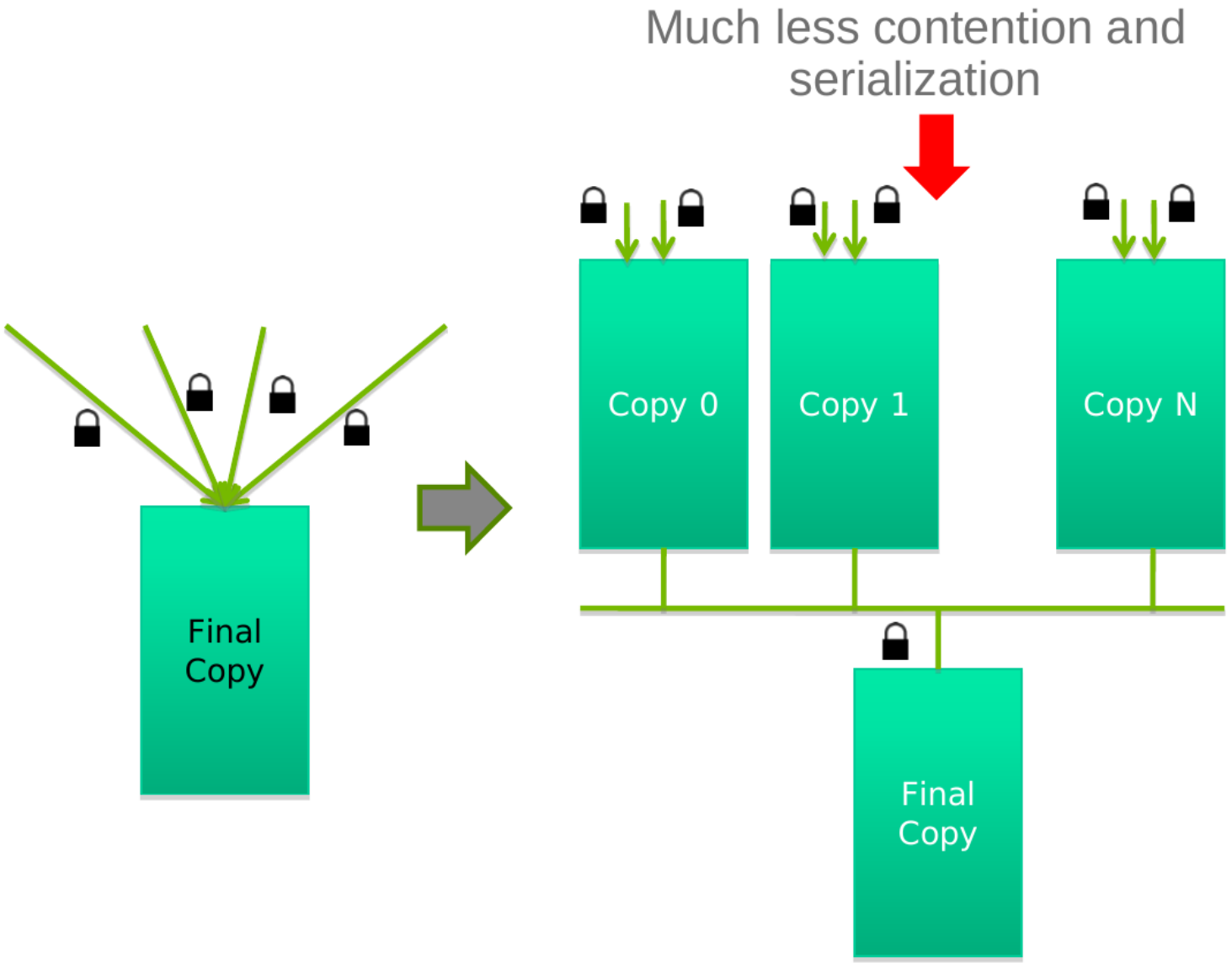
- Très faible latence
- Privé à chaque bloc de threads
- Peu nécessité un changement d'algorithme



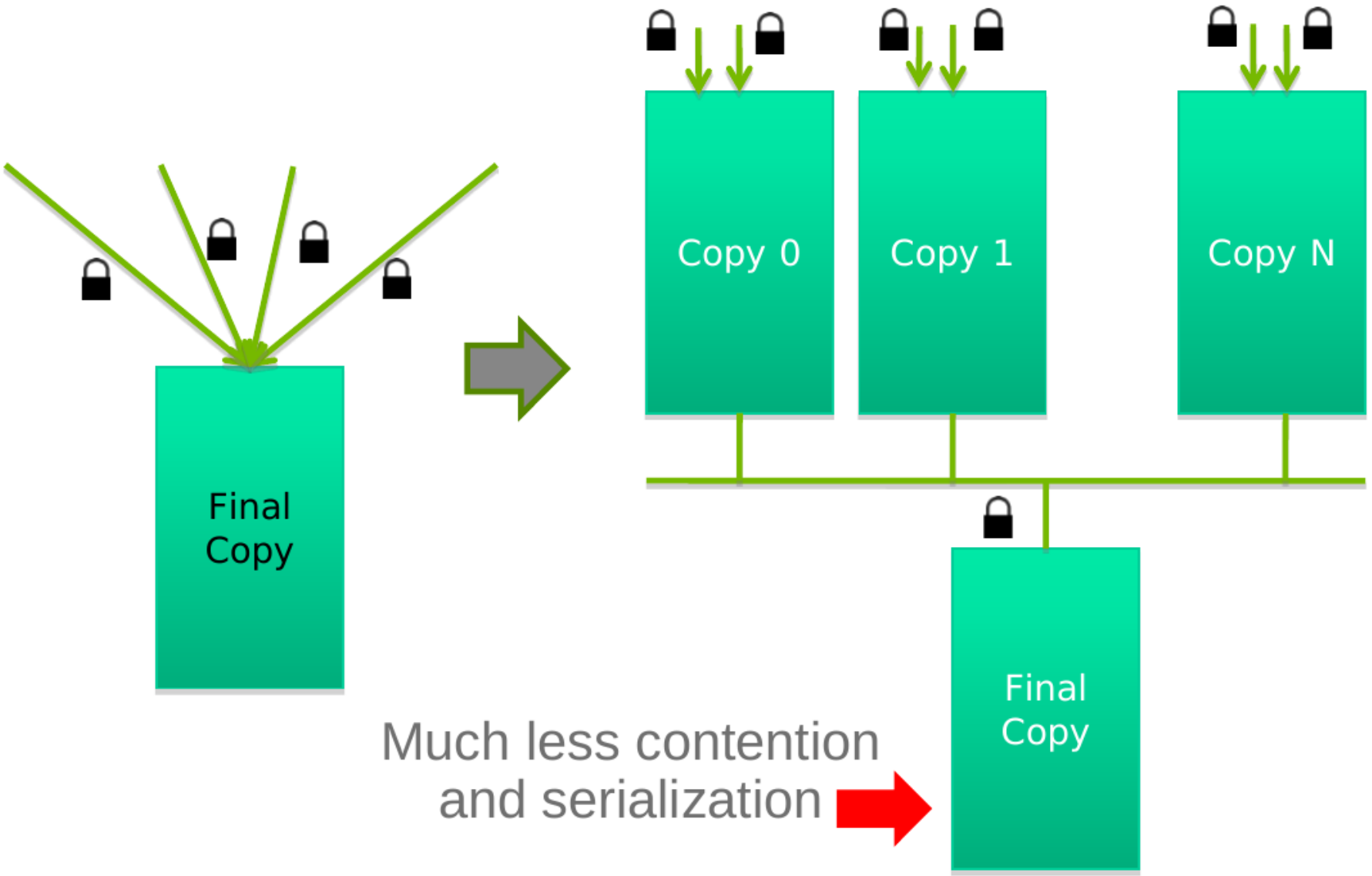
Privatisation (1/3)



Privatisation (2/3)



Privatisation (3/3)



Coût et bénéfice de la privatisation

- Coût
  - sur-coût à la création et initialisation des copies privées
  - sur-coût pour l'accumulation du contenu des différentes copies privées dans la copie finale
- Bénéfice
  - Beaucoup moins de contention et de sérialisation lors de l'accés aux copies privées et à la copie finale
  - La performance totale peut souvent être améliorée de plus de 10x

Opérations atomiques en mémoire partagée pour les histogrammes

- Chaque sous ensemble de threads sont dans le même bloc
- Beaucoup plus de bande passante que pour les opérations atomiques en DRAM (100x) ou L2 (10x)
- Beaucoup moins de contentions : seulement les threads d'un même bloc peuvent accéder à la variable en mémoire partagée
- C'est un des cas majeurs d'utilisation de la mémoire partagée

Opération atomique en mémoire partagée nécessite la privatisation

Créer des copies privées du tableau *histo*[] pour chaque bloc de threads

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
  __shared__ unsigned int histo_private[7];
```

Opération atomique en mémoire partagée nécessite la privatisation

Initialisation les compteurs de caractéristiques dans la copie privée de *histo*[]

```
__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();
```

Construire un histogramme privé

```
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;
}
```

Construire l'histogramme finale

```
    // wait for all other threads in the block to finish
    __syncthreads();
    if (threadIdx.x < 7) {
      atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x]
    }
}
```

Plus d'information sur la privatisation

- La privatisation est une technique puissante et fréquemment utilisée pour la parallélisation d'applications (au delà de CUDA)
- Les opérations doivent être associative et communatative
  - L'opération ajout (*add*) de l'histogramme est associative et communtative
  - Pas de privatisation si l'opération ne respecte pas ces critères
- La taille de l'histogramme privatisée doit être petite *i.e.* il tient dans la mémoire partagée
- Que faire si l'histogramme est trop gros pour être privatisé
  - Des fois, il est possible de partiellement privatiser l'histogramme de sortie et utiliser des gammes pour tester si il faut utiliser la mémoire globale ou celle partagée

La convolution comme un filtre

Souvent effectué comme filtre qui transforme les valeurs de signaux et/ou pixels en valeurs plus adaptées

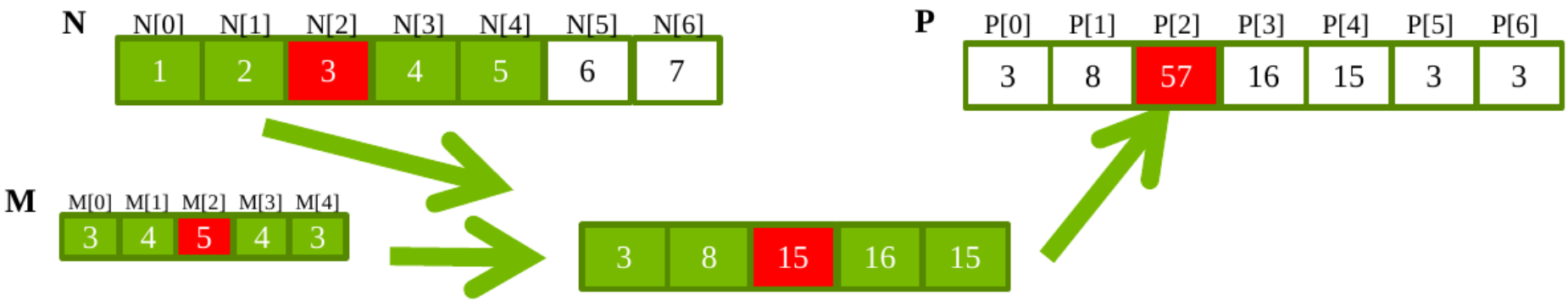
- Certaines filtres filtrent les valeurs d'un signal pour faire ressortir la tendance générale
- D'autres comme les filtres gaussiens peuvent être utilisés pour raffiner les contours ou des angles d'objets dans des images

Définition computationelle de la convolution

- Une opération sur un tableau où chaque élément des données en sortie est un somme pondérée d'un ensemble d'éléments voisins des données en entrée

- La pondération utilisée dans le calcul de la somme pondérée est définie comme un tableau de masques en entrée, il est généralement appellé noyau de convolution (*convolution kernel*)
  - Ces tableaux de masques seront nommés masques de convolution pour éviter les confusions
  - Les valeurs du motifs du tableau de masques définit le type de filtrage à effectuer
  - Notre exemple de floutage d'image est un cas spécial où le masque est le même pour tous les éléments avec la même valeur et qu'il est programmé en dur dans le code source

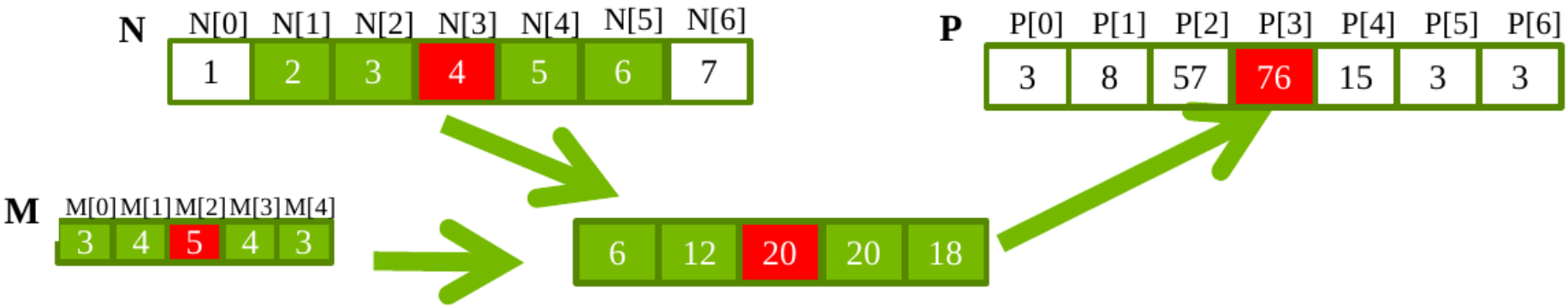
Exemple de convolution en 1D



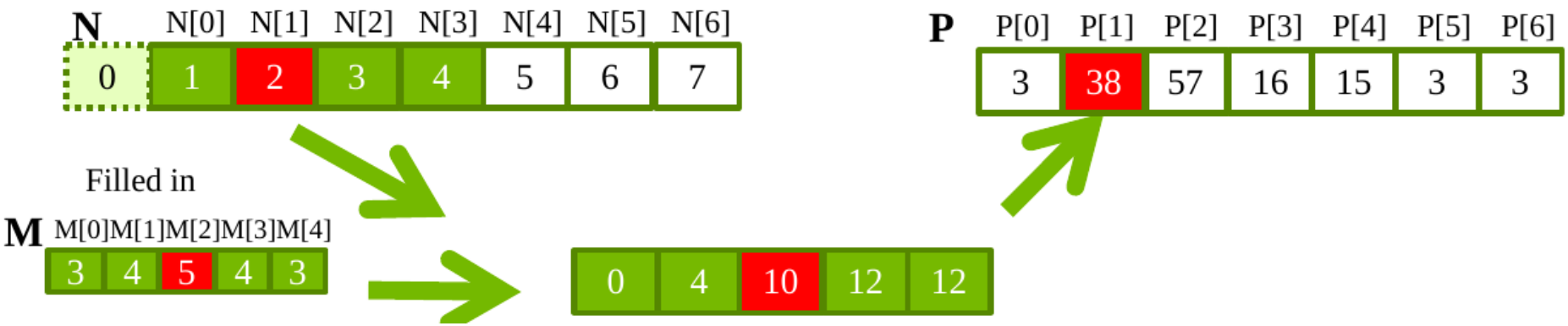
- Souvent utilisé pour le traitement du signal audio
  - La taille du masque est souvent un nombre impair pour avoir une symétrie autour de l'élément central (*e.g.*, 5)
- La figure représente le calcul de l'élément $P[2]$

$$P[2] = N[0]*M[0]+N[1]*M[1]+N[2]*M[2]+N[3]*M[3]+N[4]*M[4]$$

Calcul de l'élément $P[3]$



Les conditions de borne pour la convolution



Le calcul des éléments en sortie proche des bornes (début et fin) du tableau nécessite de traiter avec des éléments “fantomes“

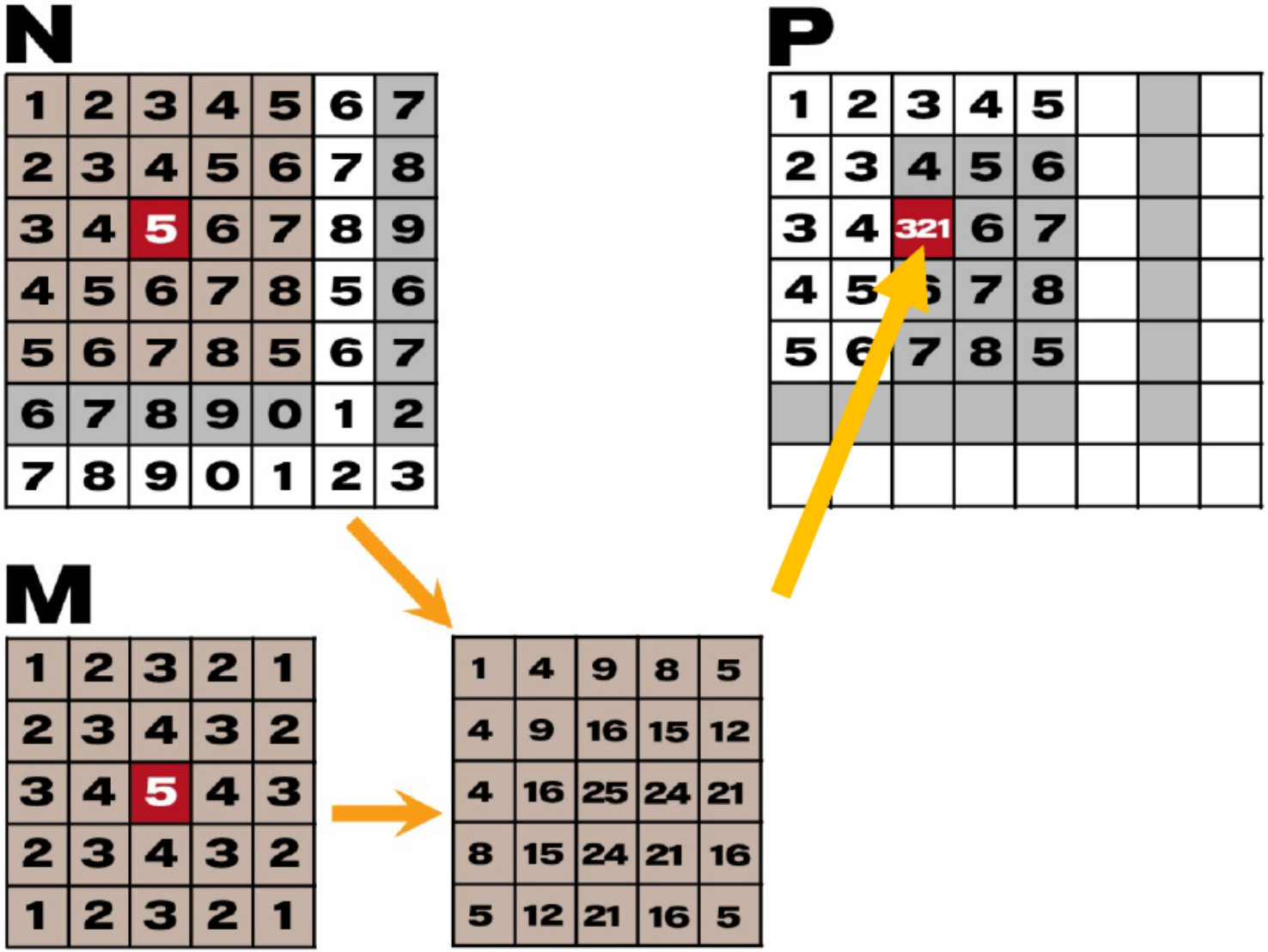
- Plusieurs politiques possible : 0, replication des valeurs en borne, etc.

Noyau de calcul à 1D avec gestion des conditions de bornes

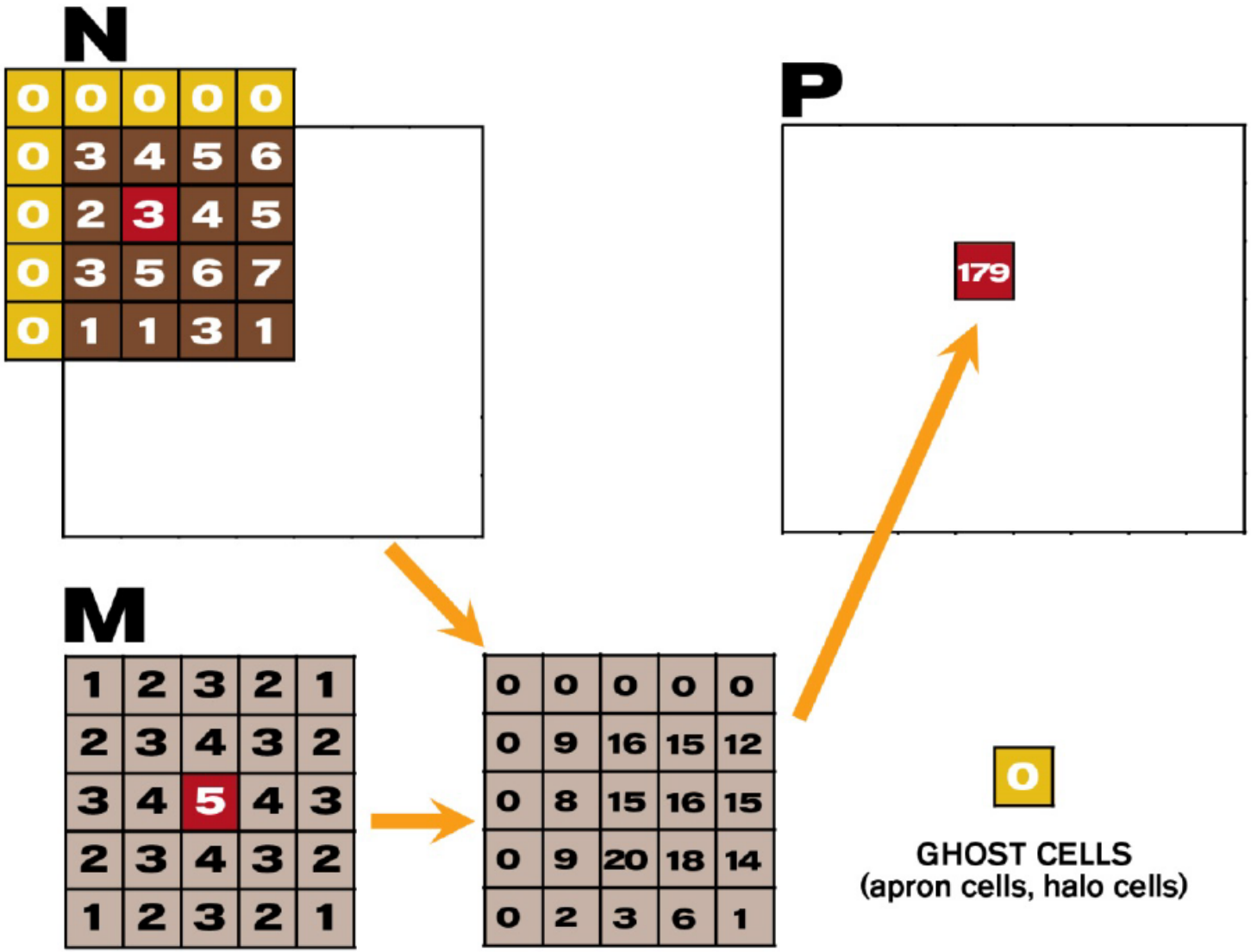
Ce noyau force à 0 tous les éléments en dehors de la gamme valide

```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
                                            float *P, int Mask_Width, int Width)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int N_start_point = i      (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
}
```

Convolution en 2D



Convolution en 2D: Cellules fantomes

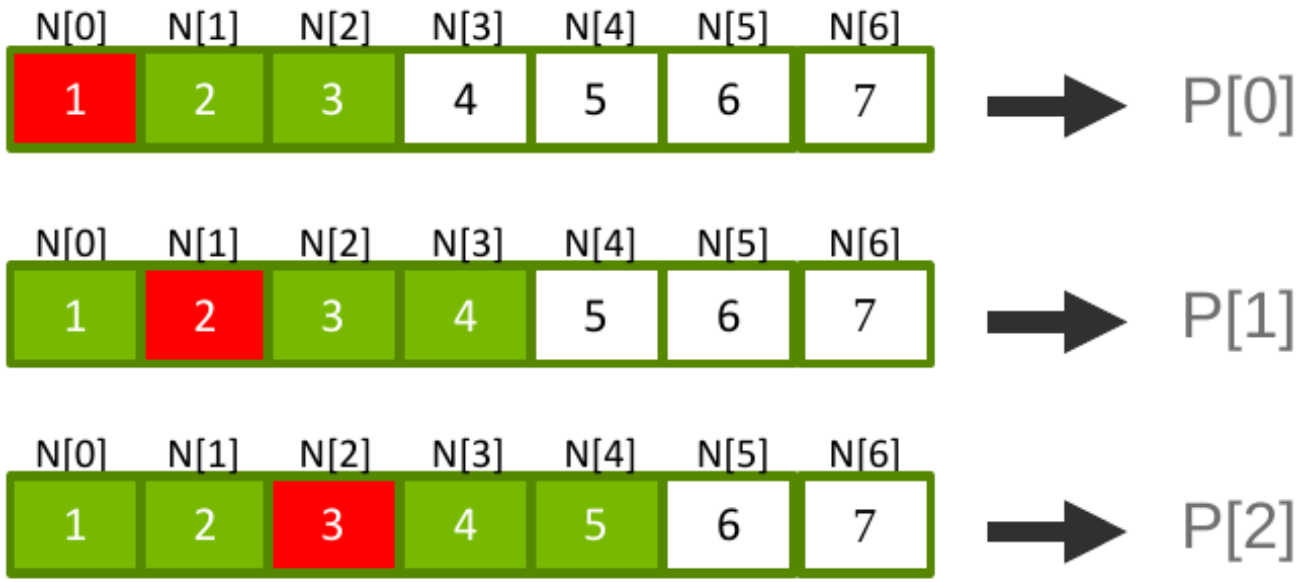


Noyau de calcul à 2D

```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned ch
        unsigned char * out, int maskwidth, int w, int h) {
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  if (Col < w && Row < h) {
    int pixVal = 0;
    N_start_col = Col       (maskwidth/2);
    N_start_row = Row       (maskwidth/2);
    // Get the of the surrounding box
    for(int j = 0; j < maskwidth; ++j) {
      for(int k = 0; k < maskwidth; ++k) {
        int curRow = N_Start_row + j;
        int curCol = N_start_col + k;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1
                                  && curCol < w) {
          pixVal += in[curRow * w + curCol] *
                                  mask[j*maskwidth+k];
        }
```

Convolution et Tiling

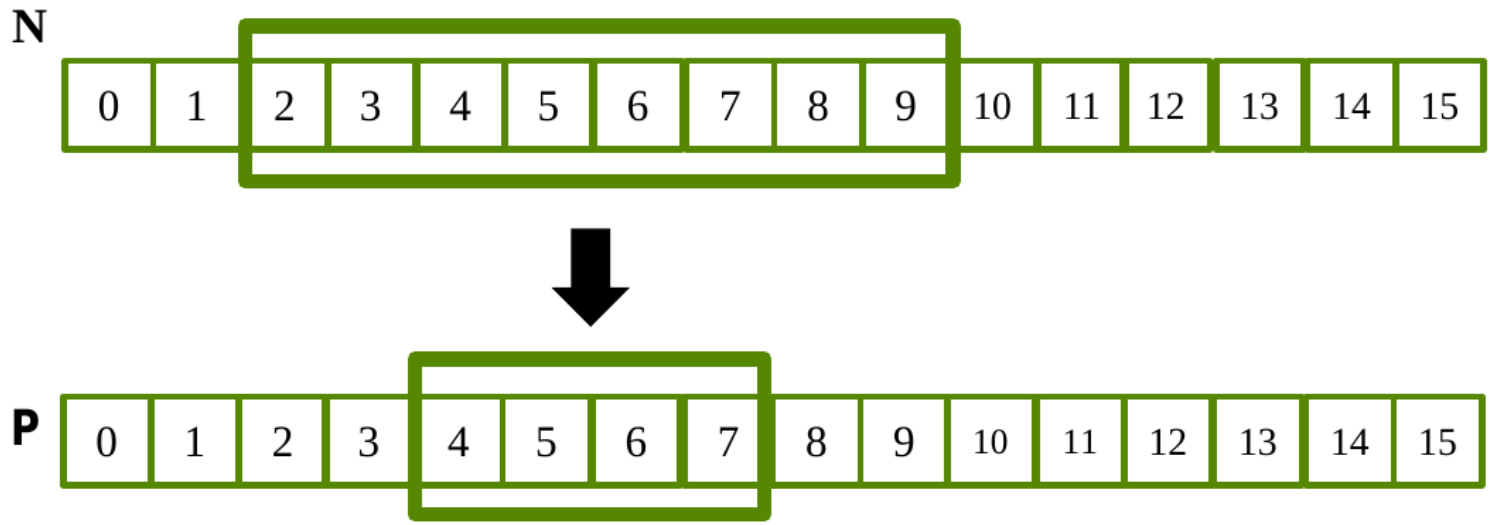
- Le calcul d'élements adjacents impliquent des éléments en entrée qui sont partagés
  - Par exemple, $N[2]$ est utilisé pour le calcul de $P[0]$, $P[1]$, $P[2]$, $P[3]$ et $P[4]$ pour un masque de convolution 1D de taille 5
- Il serait possible de charger tous les éléments requis par tous les threads d'un bloc dans la mémoire partagée pour réduire les accés à la mémoire globale



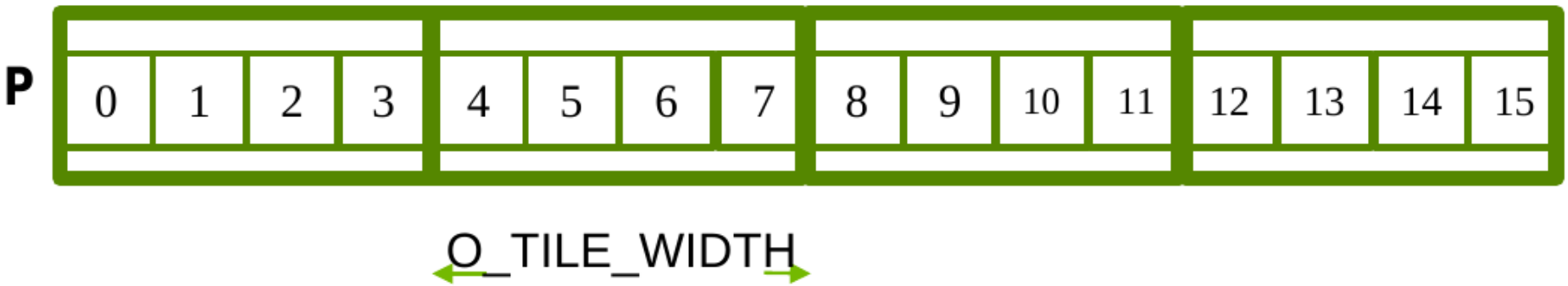
Quelles données en entrée

Supposons que nous souhaitons que chaque bloc calcule T éléments en sortie

- $T + Mask\_Width - 1$ éléments en entrée sont nécessaire pour calculer T éléments en sortie
- $T + Mask\_Width - 1$ est généralement par un multiple de T (sauf si T est petit)
- T est généralement significativement plus large que $Mask\_Width$

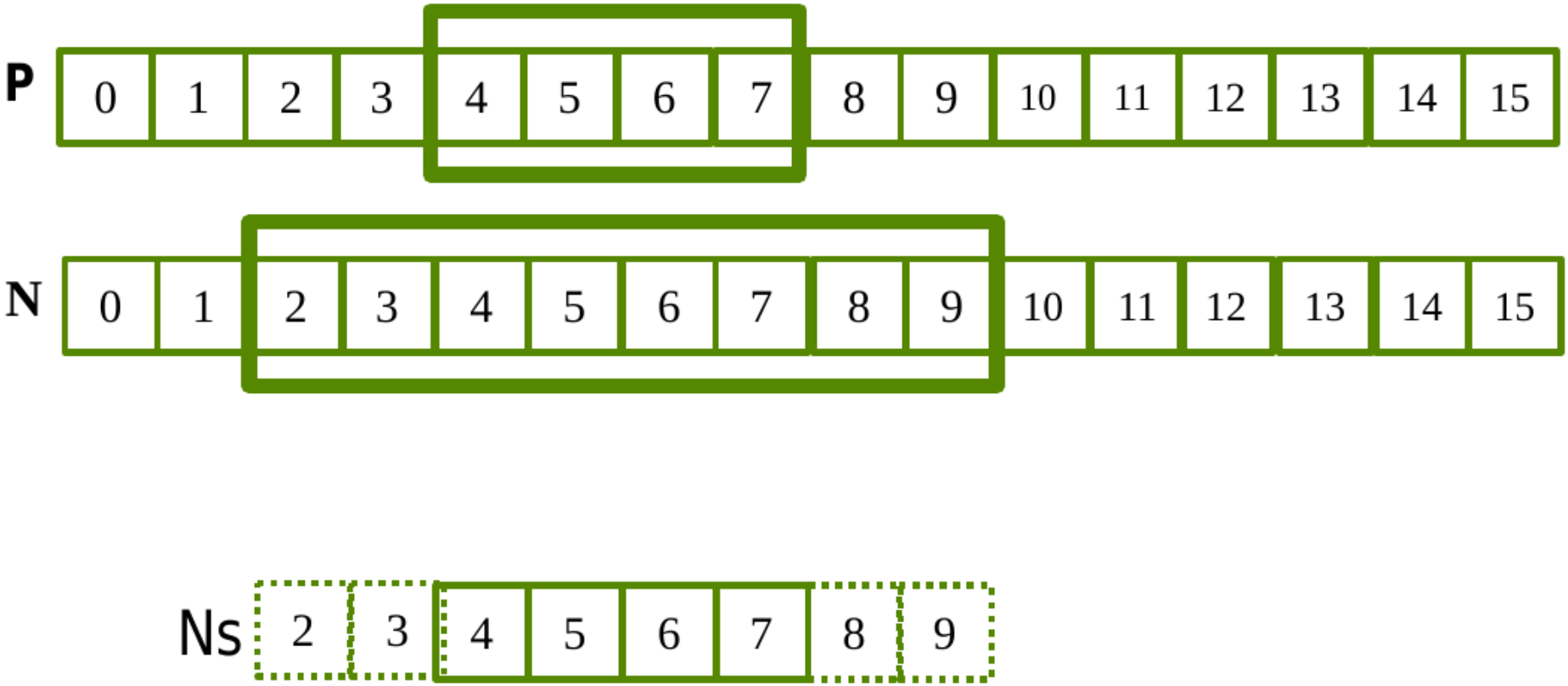


Définition d'un carré de sortie



- Chaque bloc de threads calcul un carreau de sortie
- Chaque carreau de sortie est de largeur *O_TILE_WIDTH*
- Dans la figure, *O_TILE_WIDTH* vaut 4

Définition d'un carré de sortie



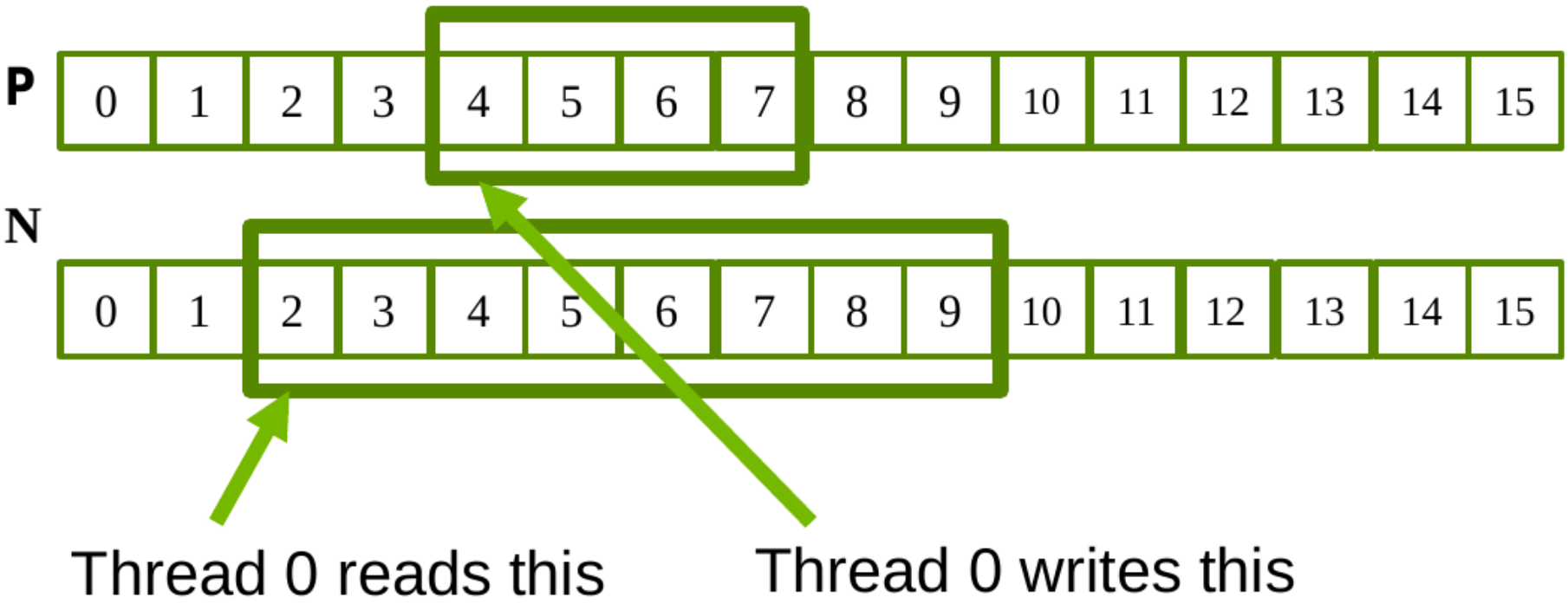
Chaque carreau d'entrée contient toutes les valeurs nécessaire pour calculer le carreau de sortie correspondant

2 méthodes de conception

- Méthode 1 : La taille de chaque bloc de threads correspond à la taille d'un carreau de sortie
  - Tous les threads participent au calcul des éléments de sortie
  - *blockDim.x* doit donc être de 4 dans notre exemple
  - Certains threads doivent charger plus d'un élement en entrée dans la mémoire partagée

- Méthode 2 : La taille de chaque bloc de threads correspond à la taille d'un carreau d'entrée
  - Certains threads ne participent pas au calcul des élements en sortie
  - *blocDim.x* doit donc être de 8 dans notre exemple
  - Chaque thread charge un seul élément en entrée dans la mémoire partagée

Exemple du fonctionnement pour un thread



- Pour chaque thread, $index_i = index_o - n$
- Avec $n = \frac{Mask\_Width}{2}$
- $n = 2$ dans cette exemple

Tous les threads participent au chargement des carreaux en entrée

```
float output = 0.0f;
if((index_i >= 0) && (index_i < Width)) {
  Ns[tx] = N[index_i];
} else {
  Ns[tx] = 0.0f;
}
```

Certains threads ne participent pas au calcul de la sortie

```
if (threadIdx.x < O_TILE_WIDTH){
  output = 0.0f;
  for(j = 0; j < Mask_Width; j++) {
    output += M[j] * Ns[j+threadIdx.x];
  }
  P[index_o] = output;
}
```

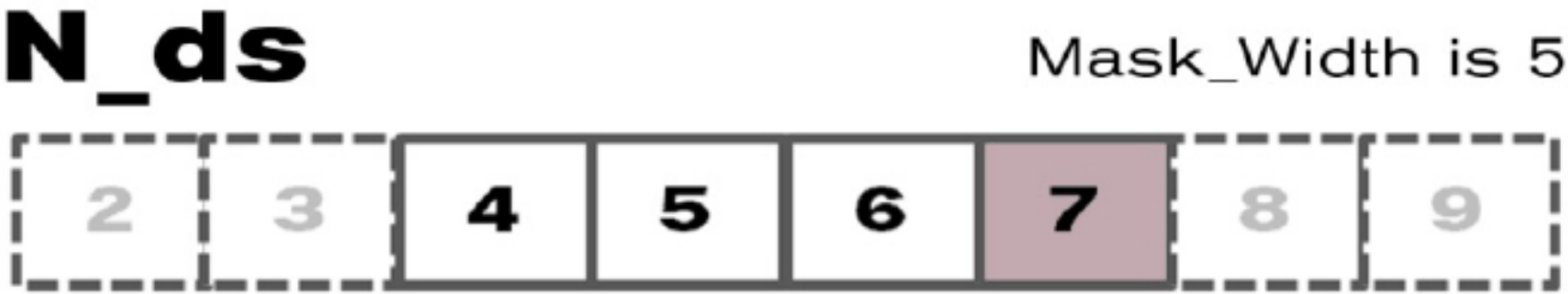
- $index\_o = blockIdx.x * O\_TILE\_WIDTH + threadIdx.x$
- Seulement les threads entre 0 et $O\_TILE\_WIDTH - 1$ participent au calcul de la sortie

Définir la taille de bloc

```
#define O_TILE_WIDTH 1020
#define BLOCK_WIDTH (O_TILE_WIDTH + 4)
dim3 dimBlock(BLOCK_WIDTH,1, 1);
dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)
```

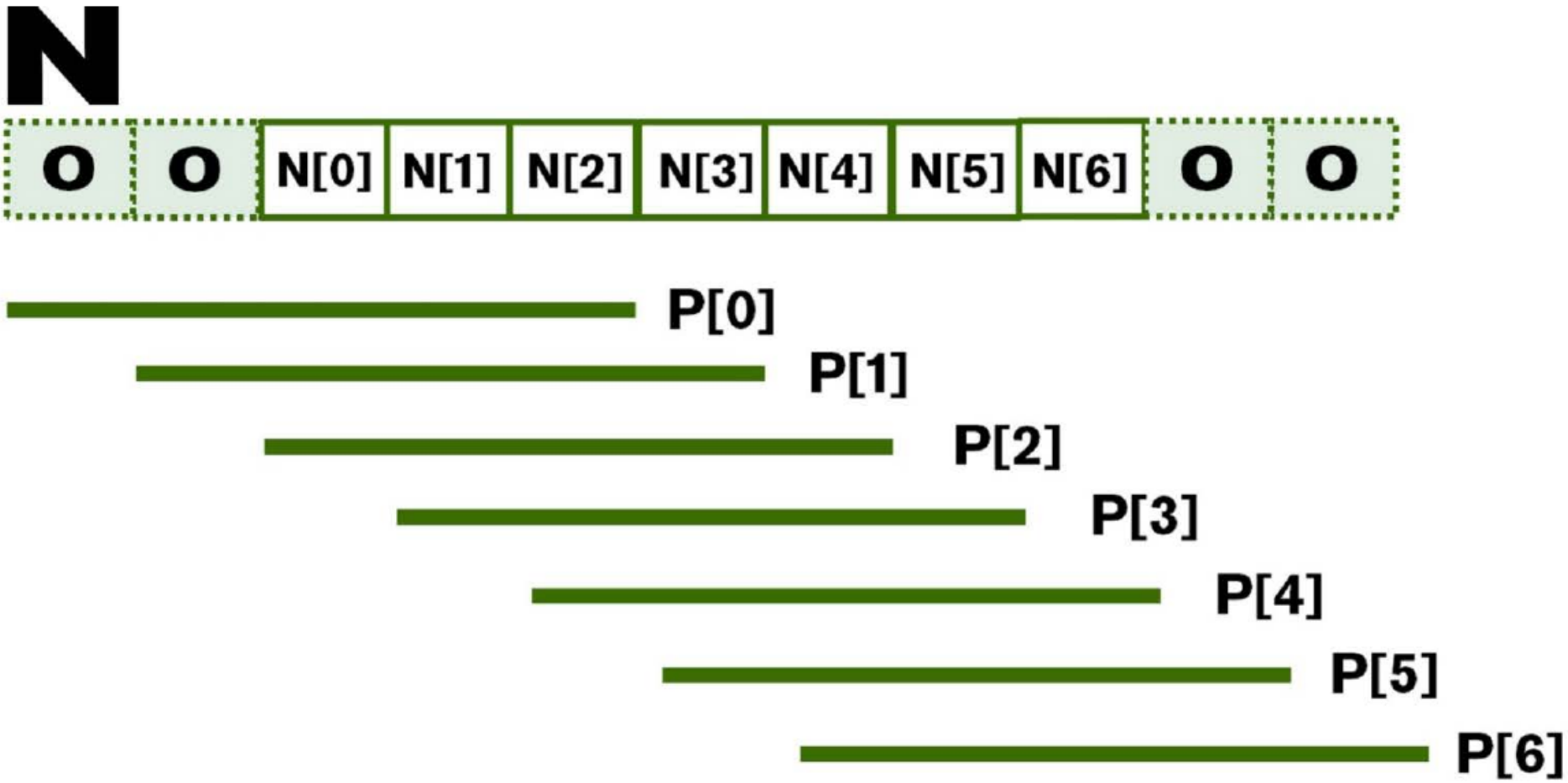
- La taille de *Mask_Width* est 5 dans notre exemple
- En général, la taille du bloc devrait être *largeur_carreau_sortie* + (*Mask_Width*) − 1

Réutilisation des données en mémoire partagée



- Element 2 est utilisé par le thread 4 (1x)
- Element 3 est utilisé par le thread 4, 5 (2x)
- Element 4 est utilisé par le thread 4, 5, 6 (3x)
- Element 5 est utilisé par le thread 4, 5, 6, 7 (4x)
- Element 6 est utilisé par le thread 4, 5, 6, 7 (4x)
- Element 7 est utilisé par le thread 5, 6, 7 (3x)
- Element 8 est utilisé par le thread 6, 7 (2x)
- Element 9 est utilisé par le thread 7 (1x)

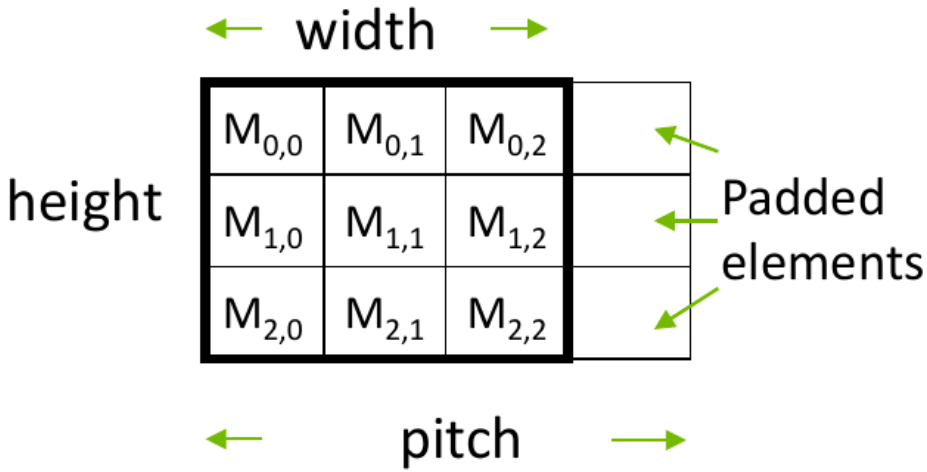
Cellules fantomes



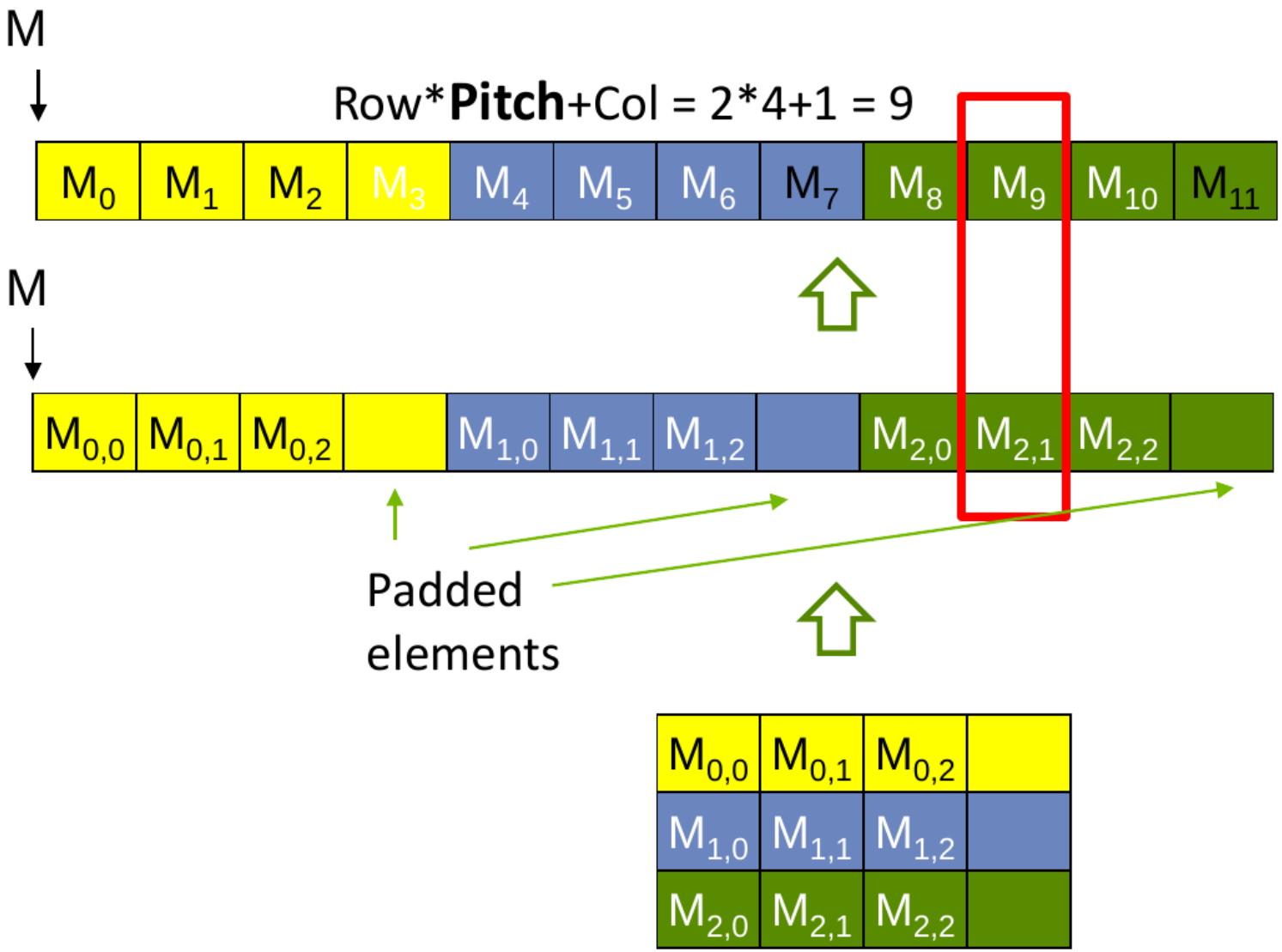
Image/matrice 2D avec remplissage (padding) automatique

- Il est des fois préférable de remplir chaque ligne d'une matrice avec un multiple des bursts DRAM
  - Pour chaque ligne commence au début d'un burst DRAM
  - Pour cela, il faut ajouter des colonnes
  - C'est généralement fait automatiquement lors de l'allocation
  - Le putch peut être différent suivant le matériel

- Exemple: une matrice 3x3 rempli dans une matrice 3x4
  - Hauteur = 3
  - Largeur = 3
  - Pitch = 4



Exemple d'agencement mémoire avec pitch



Le type matrice d'image

La structure *wbImage* (des TPs) reprend ce principe

```
typedef struct {
  int width;
  int height;
  int pitch;
  int channels;
  float* data;
} * wbImage_t;
```

Définir la taille de bloc

```
#define O_TILE_WIDTH 12
#define BLOCK_WIDTH (O_TILE_WIDTH + 4)
dim3 dimBlock(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 dimGrid((wbImage_getWidth(N)-1)/O_TILE_WIDTH+1,
             (wbImage_getHeight(N)-1)/O_TILE_WIDTH+1, 1);
```

En général, *BLOCK_WIDTH* est défini à
$O\_TILE\_WIDTH + (MASK\_WIDTH - 1)$

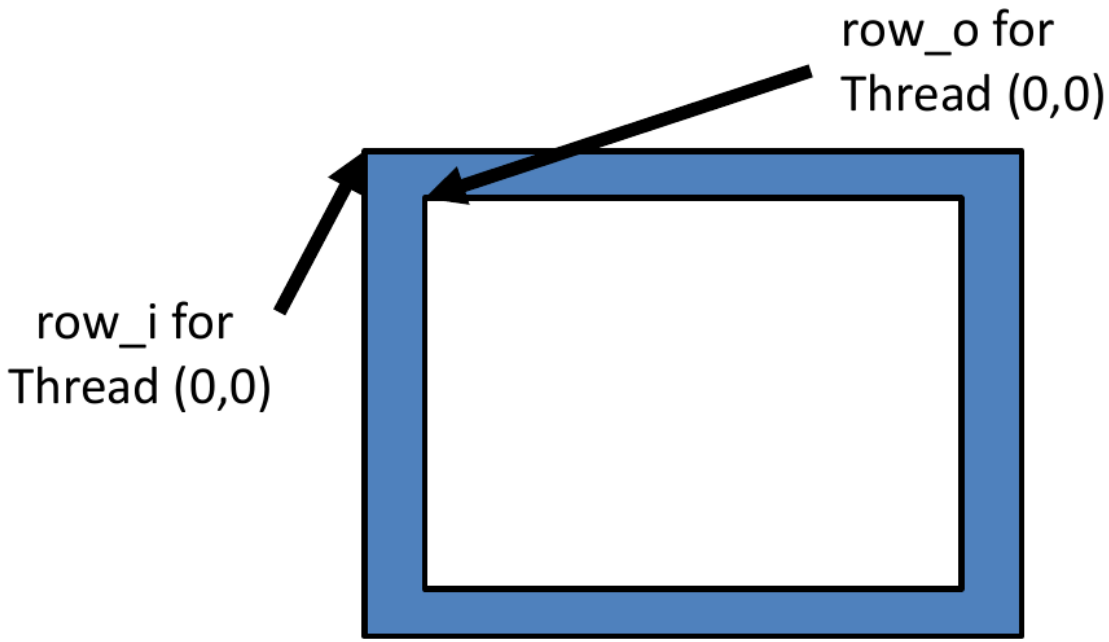
Utiliser la mémoire constante et le cache pour le masque

- Le masque est utilisé par tous les threads et n'est pas modifié par le noyau de convolution
  - Tous les threads dans un warp accédent à la même adresse à chaque moment
- CUDA fournit de la mémoire constante dont le contenu est mis en cache
  - Les valeurs en cache sont diffusées à tous les threads dans un warp
  - Cela permet d'augmenter la bande passante mémoire sans consommer de la mémoire partagée
- Il faut utiliser const __restrict__ pour informer le compilateur qu'une variable est éligible pour la mémoire constante:

```
__global__ void convolution_2D_kernel(float *P,
        float *N, height, width, channels,
        const float __restrict__ *M) {
```

Calculer les indexes d'entrée et ceux de sortie

```
int tx = threadIdx.x;
int ty = threadIdx.y;
int row_o = blockIdx.y*O_TILE_WIDTH + ty;
int col_o = blockIdx.x*O_TILE_WIDTH + tx;
int row_i = row_o - 2;
int col_i = col_o - 2;
```



Faire attention aux gammes

```
if((row_i >= 0) && (row_i < height) &&
    (col_i >= 0) && (col_i < width)) {
  Ns[ty][tx] = data[row_i * width + col_i];
} else {
  Ns[ty][tx] = 0.0f;
}
```

Attention, ici on suppose que *pitch* est égale à *width*

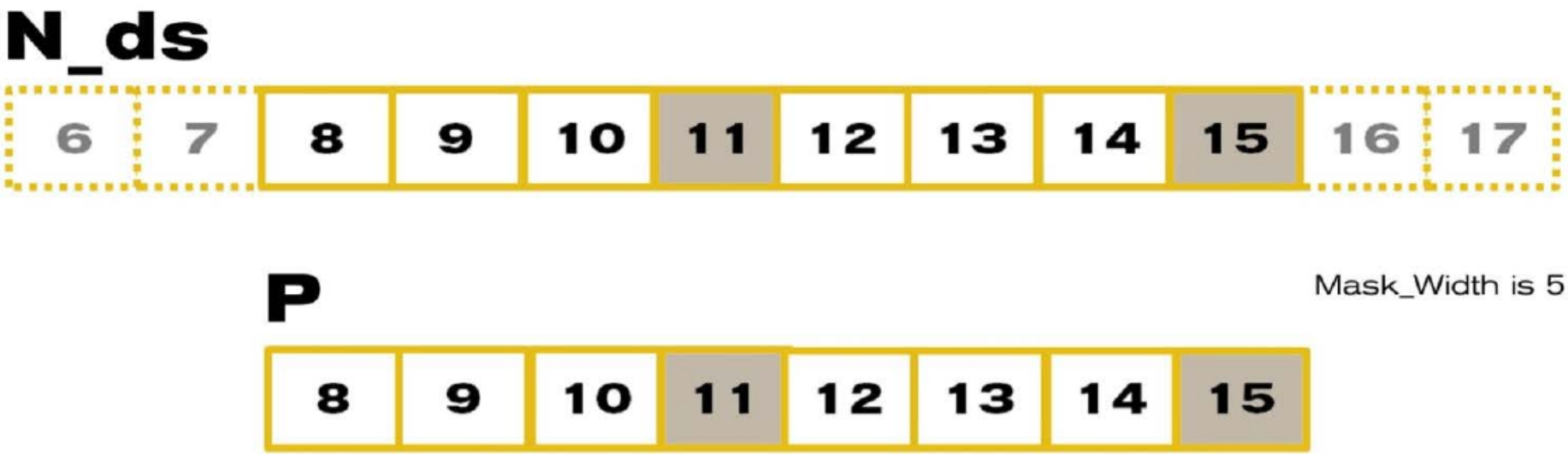
Certains threads ne participent pas au calcul de la sortie

```
float output = 0.0f;
if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
  for(i = 0; i < MASK_WIDTH; i++) {
    for(j = 0; j < MASK_WIDTH; j++) {
      output += M[i][j] * Ns[i+ty][j+tx];
    }
  }
}
```

Certains threads n'écrirent pas de sortie

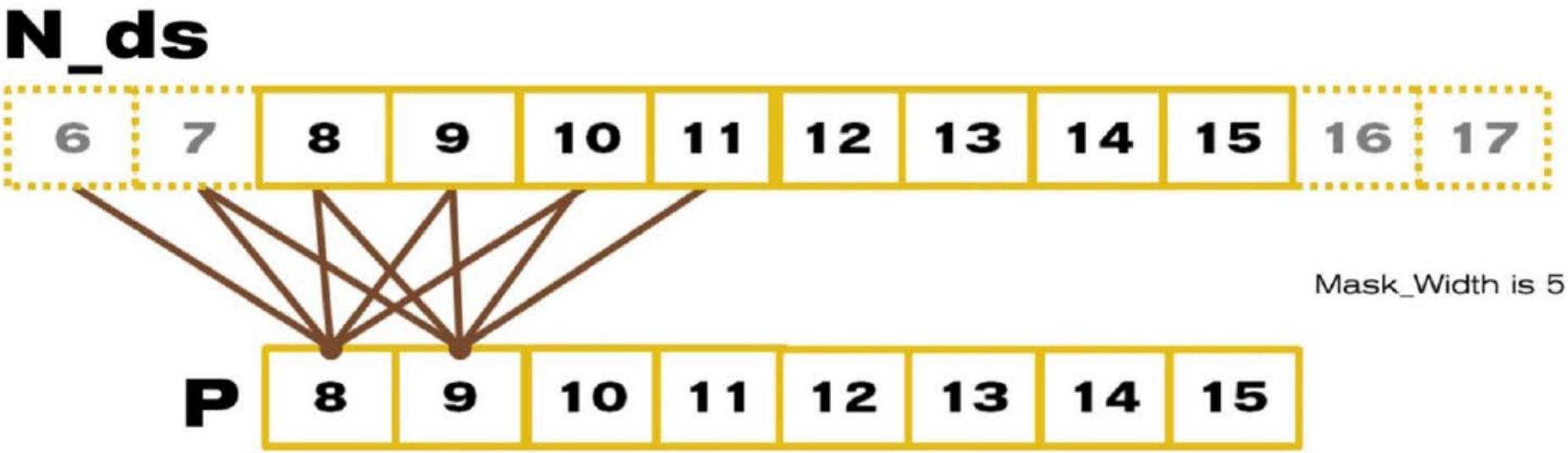
```
if(row_o < height && col_o < width)
  data[row_o*width + col_o] = output;
```

Convolution avec des carreaux de 8 éléments



Pour $Mask\_Width = 5$, on charge $8 + 5 - 1 = 12$ éléments

Chaque élément de P utilise 5 éléments de N



- P[8] utilise N[6], N[7], N[8], N[9], N[10]
- P[9] utilise N[7], N[8], N[9], N[10], N[11]
- P[10] utilise N[8], N[9], N[10], N[11], N[12]
- ...
- P[14] utilise N[12], N[13], N[14], N[15], N[16]
- P[15] utilise N[13], N[14], N[15], N[16], N[17]

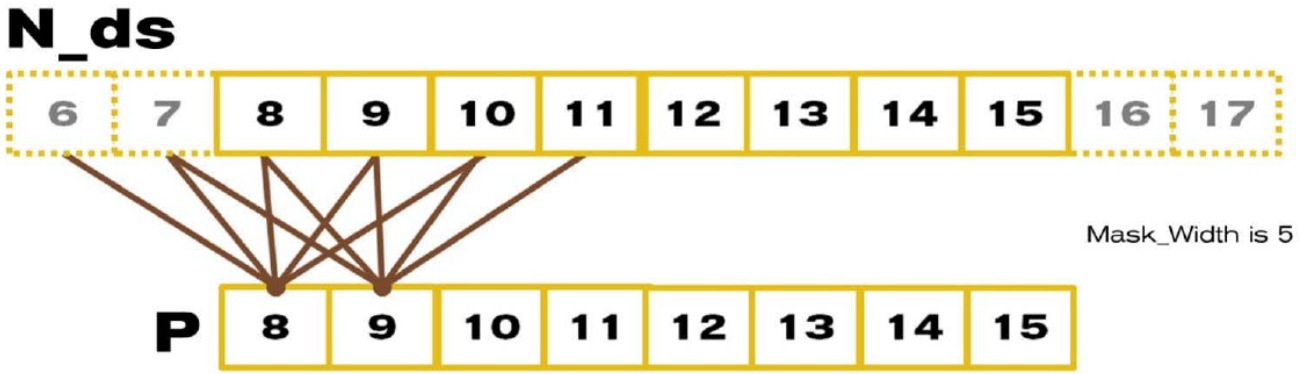
Une manière simple de calculer le bénéfice du tiling

- $8 + 5 - 1 = 12$ éléments chargés
- $8 * 5$ accés à la mémoire globale sont remplacé par des accés à la mémoire partagée
- Cela donne une réduction de la consommation de bande passante de $40/12 = 3.3$

En général, pour un convolution tilisé en 1D

- $O\_TILE\_WIDTH + MASK\_WIDTH - 1$ éléments chargés pour chaque carreau en entrée
- $O\_TILE\_WIDTH + MASK\_WIDTH$ accés à la mémoire globale sont remplacés par des accés à la mémoire partagée
- Cela donne le facteur de reduction suivante : $(O\_TILE\_WIDTH * MASK\_WIDTH)/(O\_TILE\_WIDTH + MASK\_WIDTH - 1)$

Une autre manière de regarder la réutilisation (1/2)



- N[6] est utilisé par P[8] (1X)
- N[7] est utilisé par P[8], P[9] (2X)
- N[8] est utilisé par P[8], P[9], P[10] (3X)
- N[9] est utilisé par P[8], P[9], P[10], P[11] (4X)
- N10 est utilisé par P[8], P[9], P[10], P[11], P[12] (5X)
- ... (5X)
- N[14] est utilisé par P[12], P[13], P[14], P[15] (4X)
- N[15] est utilisé par P[13], P[14], P[15] (3X)

Une autre manière de regarder la réutilisation (1/2)

Le nombre total d'accès à la mémoire globale (au 8+5-1=12 éléments de N sont remplacés par des accés à la mémoire partagée :

$1 + 2 + 3 + 4 + 5 * (8 - 5 + 1) + 4 + 3 + 2 + 1$

$= 10 + 20 + 10$

$= 40$

La réduction est donc de : $40/12 = 3.3$

En général, pour 1D

Le nombre total d'accés à la mémoire globale pour un carreau d'entrée peut calculer comme suit:

$$
\begin{aligned}
&1 + 2 + \ldots + \text{MASK\_WIDTH} - 1 + \text{MASK\_WIDTH} * (\text{O\_TILE\_WIDTH} - \\
&\qquad \text{MASK\_WIDTH} + 1) + \text{MASK\_WIDTH} - 1 + \ldots + 2 + 1 \\
&= \text{MASK\_WIDTH} * (\text{MASK\_WIDTH} - 1) + \text{MASK\_WIDTH} * \\
&\qquad (\text{O\_TILE\_WIDTH} - \text{MASK\_WIDTH} + 1) \\
&= \text{MASK\_WIDTH} * \text{O\_TILE\_WIDTH}
\end{aligned}
$$

Pour un total de $\text{O\_TILE\_WIDTH} + \text{MASK\_WIDTH} - 1$ éléments dans le carreau d'entrée

Exemple de réduction de bande passante pour 1D

Le ratio de réduction est : $\frac{MASK\_WIDTH*(O\_TILE\_WIDTH)}{O\_TILE\_WIDTH+MASK\_WIDTH-1}$

O_TILE_WIDTH	16	32	64	128	256
$MASK\_WIDTH = 5$	4.0	4.4	4.7	4.9	4.9
$MASK\_WIDTH = 9$	6.0	7.2	8.0	8.5	8.7

Pour la convolution tilisée en 2D

- $(O\_TILE\_WIDTH + MASK\_WIDTH - 1)^2$ éléments d'entrée ont besoin d'être chargés en mémoire partagée
- Le calcul de chaque élément en sortie nécessite l'accés à $(MASK\_WIDTH)^2$ éléments en entrée
- $(O\_TILE\_WIDTH)^2 * (MASK\_WIDTH)^2$ d'accés à la mémoire globale sont convertis en accés à la mémoire partagée
- Le ratio de réduction est donc de : $\frac{O\_TILE\_WIDTH^2 * MASK\_WIDTH^2}{(O\_TILE\_WIDTH + MASK\_WIDTH - 1)^2}$

Exemple de réduction de bande passante pour 2D

Le ratio de réduction est : $\frac{O\_TILE\_WIDTH^2 * MASK\_WIDTH^2}{(O\_TILE\_WIDTH + MASK\_WIDTH - 1)^2}$

O_TILE_WIDTH	8	16	32	64
$MASK\_WIDTH = 5$	11.1	16	19.7	22.1
$MASK\_WIDTH = 9$	20.3	36	51.8	64

- La taille de carreau a un impact significatif sur le ratio de réduction de la bande passante mémoire
- C'est un argument pour des tailles de mémoire partagée plus importantes

Partionnnement et Résumé

- Une stratégie classique pour traiter un grand volume de données
  - Il n'y a pas pré-requis sur l'ordre de traitement des éléments dans un ensemble de données (associativité et commutativité)
  - Partionnement de l'ensemble de données en petits morceaux
  - Chaque thread traite un morceau
  - Utilisation d'un arbre de réduction pour résumer et sommer les résultats de chaque morceau dans la réponse finale
- Les boites à outils Google et Hadoop MapReduce implémentent ce type de stratégie
- Dans cette partie du cours, on se focalise sur la partie arbre de réduction

La réduction permet la mise en place d'autres stratégies

- La réduction est également nécessaire pour nettoyer après des transformations utilisées classiquement en calcul parallèle

- Privatisation
  - Plusieurs threads écrivent dans une adresse en sortie
  - Replication de l'adresse en sortie pour que chaque thread est sa propre adresse de sortie (privatisation)
  - Utilisation de l'arbre de réduction pour combiner les valeurs des adresses privées dans la sortie globale

Qu'est ce qu'une opération de réduction ?

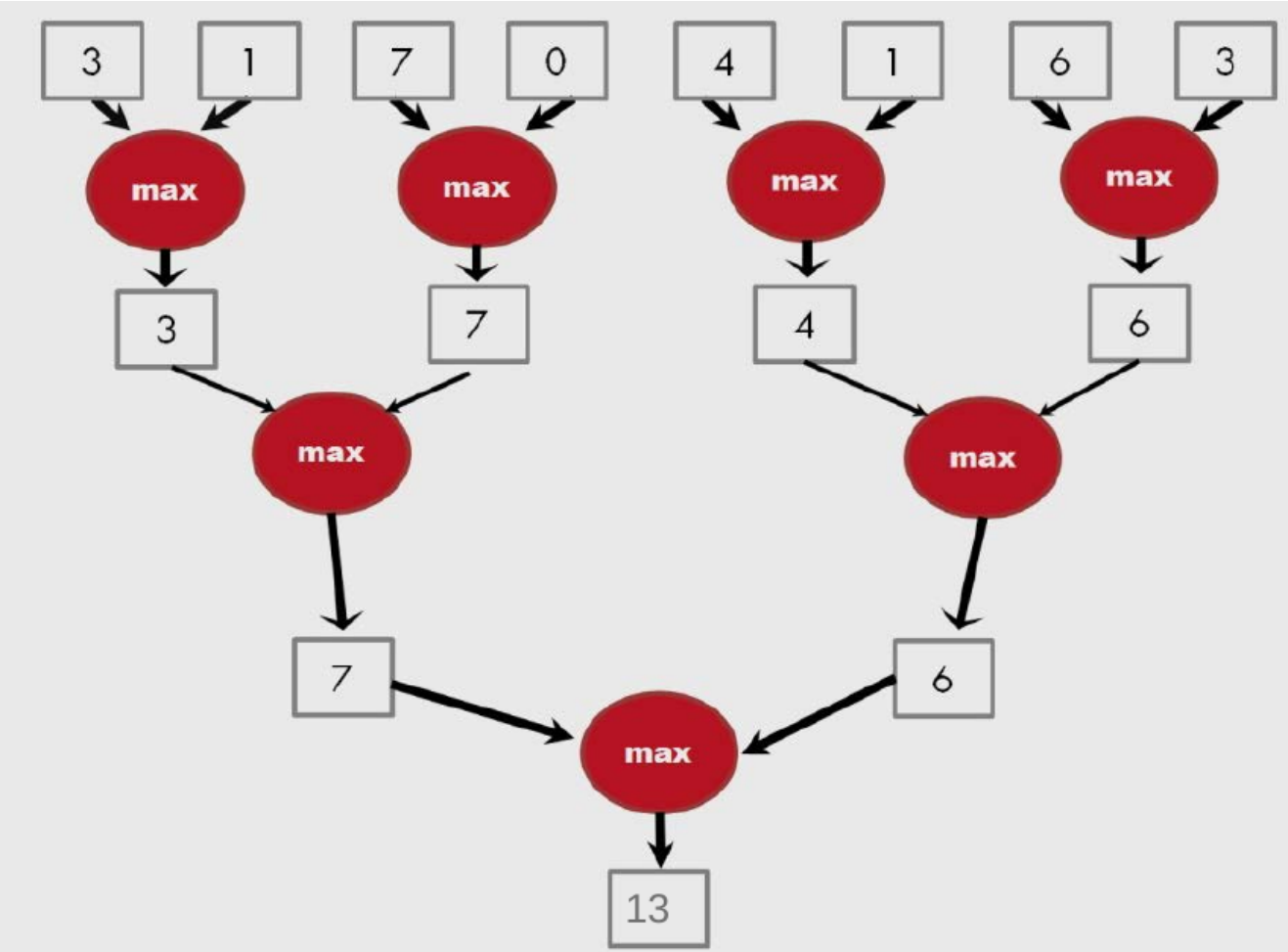
- Permet de résumer un ensemble de valeurs en entrée dans une valeur en sortie en utilisant une opération de réduction
  - Max
  - Min
  - Somme
  - Produit
  - ...
- Il est souvent possible de fournir une fonction définie par l'utlisateur tant que l'opération
  - est associative et communtative
  - a une valeur neutre bien définie (par exemple, 0 pour la somme)
  - Par exemple, l'utilisateur fournit un version personnalisée de la fonction *max* pour des ensemble de données avec des coordonnées en 3D où la valeur d'un point pour une coordonnée 3D est égale à la distance depuis l'origine
- Souvenez vous des opérations collectives en MPI

Une réduction efficace en séquentielle O(N)

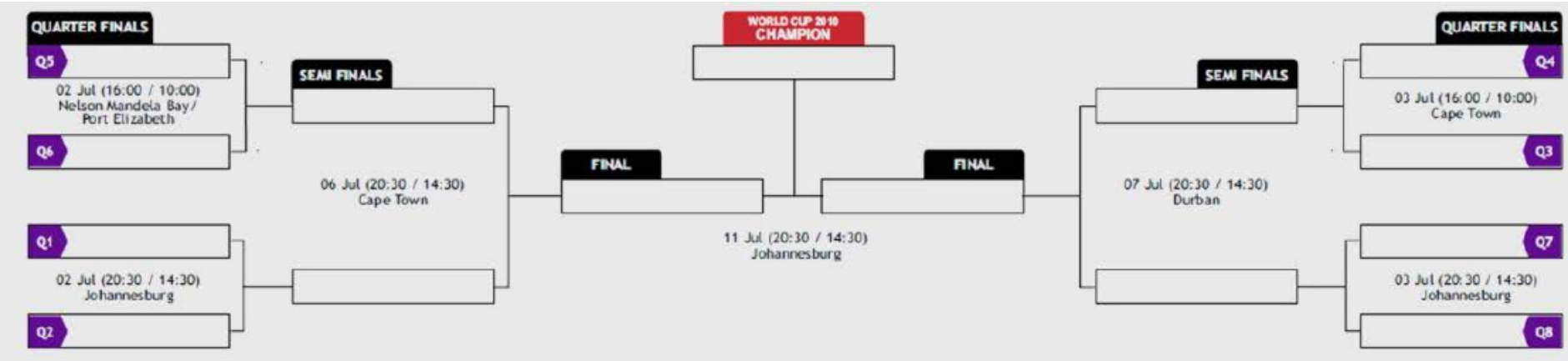
- Initialisation du résultat avec la valeur neutre correspondant à l'opération de réduction utilisée
  - La plus petite valeur possible pour l'opération *max*
  - La plus grande pour l'opération *min*
  - 0 pour la somme
  - 1 pour le produit

- Parcourir tous les éléments en entrée et effectuer l'opération de réduction entre la valeur en entrée courante et celle du résultat
  - N opérations de réduction sont effectuées pour N valeurs en entrée
  - Chaque valeur en entrée est traitée une seule et unique fois *i.e.*, un algorithme en O(N)
  - C'est un algorithme qui est dit efficace en terme de calcul

Un arbre de réduction en paralléle effectue $N - 1$ opérations en $log(N)$ étapes



Une compétition est un arbre de réduction avec une opération *max*



Analyse de complexité

- Pour N valeurs en entrée, l'arbre de réduction effectue
  - $\frac{1}{2} * N + \frac{1}{4} * N + \frac{1}{8} * N + ... + 1 * N = (1 - \frac{1}{N}) * N = N - 1$ opérations
  - $log(N)$ étapes *e.g.* 1,000,000 valeurs en entrée sont traitées en 20 étapes (si il y a suffisament de ressources !)
  - Parallélisme moyen $\frac{N-1}{log(N)}$
    - Pour $N = 1,000,000$, le parallélisme moyen est de 50,000
    - Mais le pic de ressources nécessaire est de 500,000
    - Ce n'est pas efficace en terme d'utilisation de ressources

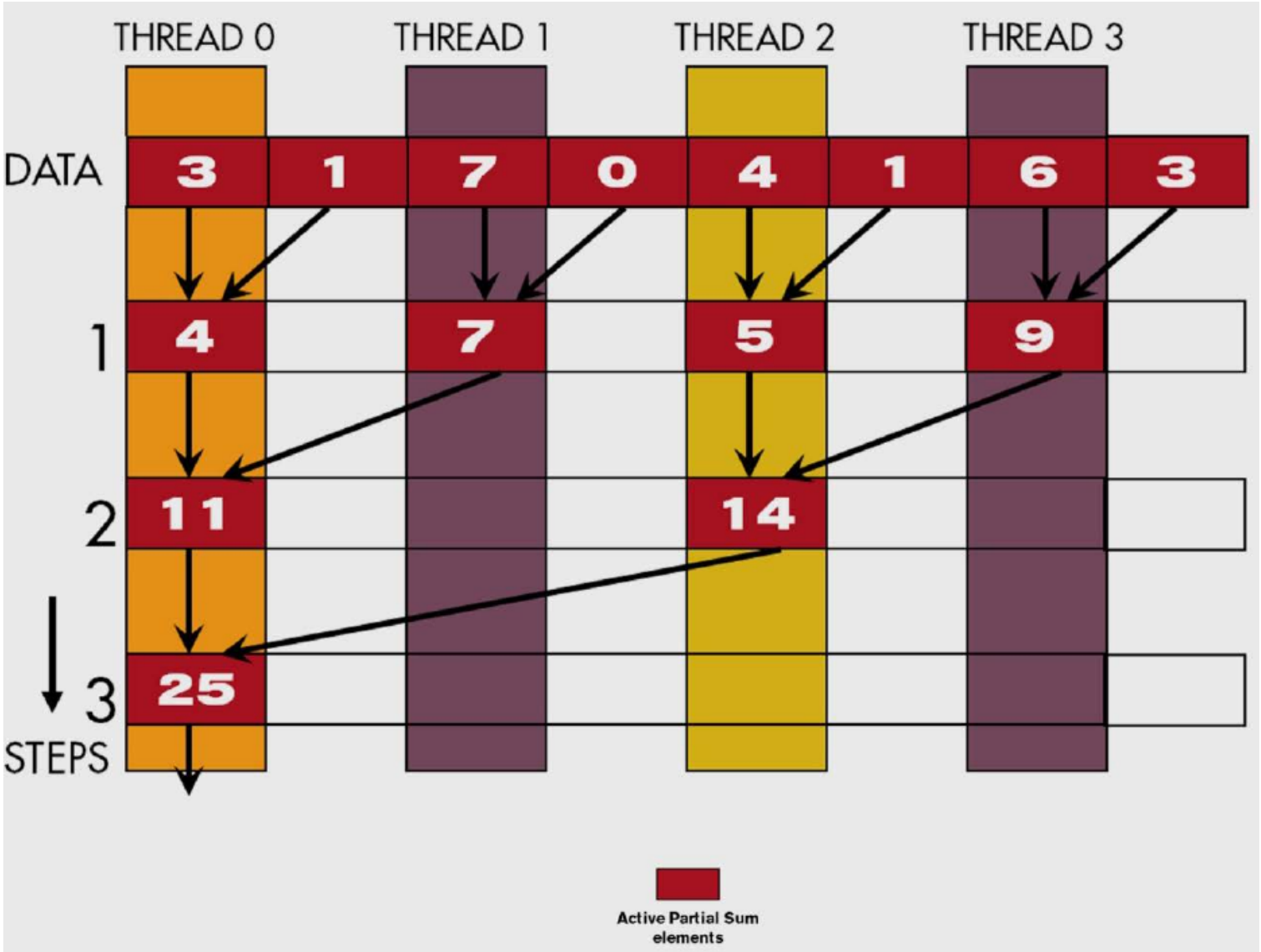
- Qu'est ce qu'un algorithme paralléle efficace
  - La quantité de travail fait est comparable à celui de l'algorithme séquentiel efficace
  - La plupart des algorithmes parallèle ne le sont pas

Opération de somme en paralléle

- Implémentation en paralléle
  - Chaque thread ajoute 2 valeurs à chaque étape
  - Récursivement division par 2 du nombre de threads
  - Cela prend $log(n)$ étapes pour n éléments et nécessite $\frac{n}{2}$ threads

- Faire une réduction avec utilisation de la mémoire partagée
  - Les données en entrée sont dans la mémoire globale
  - La mémoire partagée est utilisée pour stocker les résultats partiels
  - Au début, le tableau de somme partiel est simplement le tableau d'entrée
  - Chaque étape amène le tableau de somme partiel plus proche du résultat final
  - La somme finale sera stocké dans la case 0 du tableau de somme partiel
  - Les interactions avec la mémoire globale sont réduites puisque les lectures/écritures des sommes partiels sont fait en mémoire partagée
  - La taille de bloc limite que n soit plus petit ou égal à 2,048

Exemple d'opération de somme en paralléle



Une version naive de la répartition des données entre threads

- Chaque thread est responsable d'un nombre pair d'adresses dans le tableau de sommes partielles
- A chaque étape, la moitié des threads ne sont plus nécessaire
- Une des 2 entrées est toujours dans la gamme d'adresses dont le thread à la responsabilité
- A chaque étape, une des entrées vient d'une distance toujours plus lointaine

Une conception simple des blocs de threads

- Chaque bloc de threads est en charge de $2 * blockDim.x$ éléments d'entrée
- Chaque thread charge 2 éléments dans la mémoire partagée

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
partialSum[t] = input[start + t];
partialSum[blockDim+t] = input[start + blockDim.x+t];
```

Les étapes de la réduction

```
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```

Pourquoi faut-il faire appel à __syncthreads() ?

Les étapes de la réduction

```
for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```

Pourquoi faut-il faire appel à __syncthreads() ?

__syncthreads() est nécessaire pour s'assurer que tous les éléments d'une étape de la somme partielle ont été calculés avant de passer à l'étape suivante

Retour à la vision globale

- A la fin de l'exécution du noyau, le thread 0 de chaque bloc écrit la somme du bloc de threads (stockée dans *partialSum*[0] dans la case d'un tableau en utilisant *blockIdx.x* comme index
- Il peut y avoir un grand nombre de ces sommes partielles si la taille des données en entrée est très grande. Il est ensuite possible de lancer un second noyau pour finir la somme
- Si il y a seulement un petit nombre de sommes partielles, il peut être plus rentable de récupérer sur l'hôte le tableau de sommes partielles pour calculer la somme finale sur l'hôte
- Une alternative est que le thread 0 de chaque bloc peut utiliser les opérations atomiques pour accumuler les sommes partielles dans une somme globale

Quelques observations sur le noyau naif

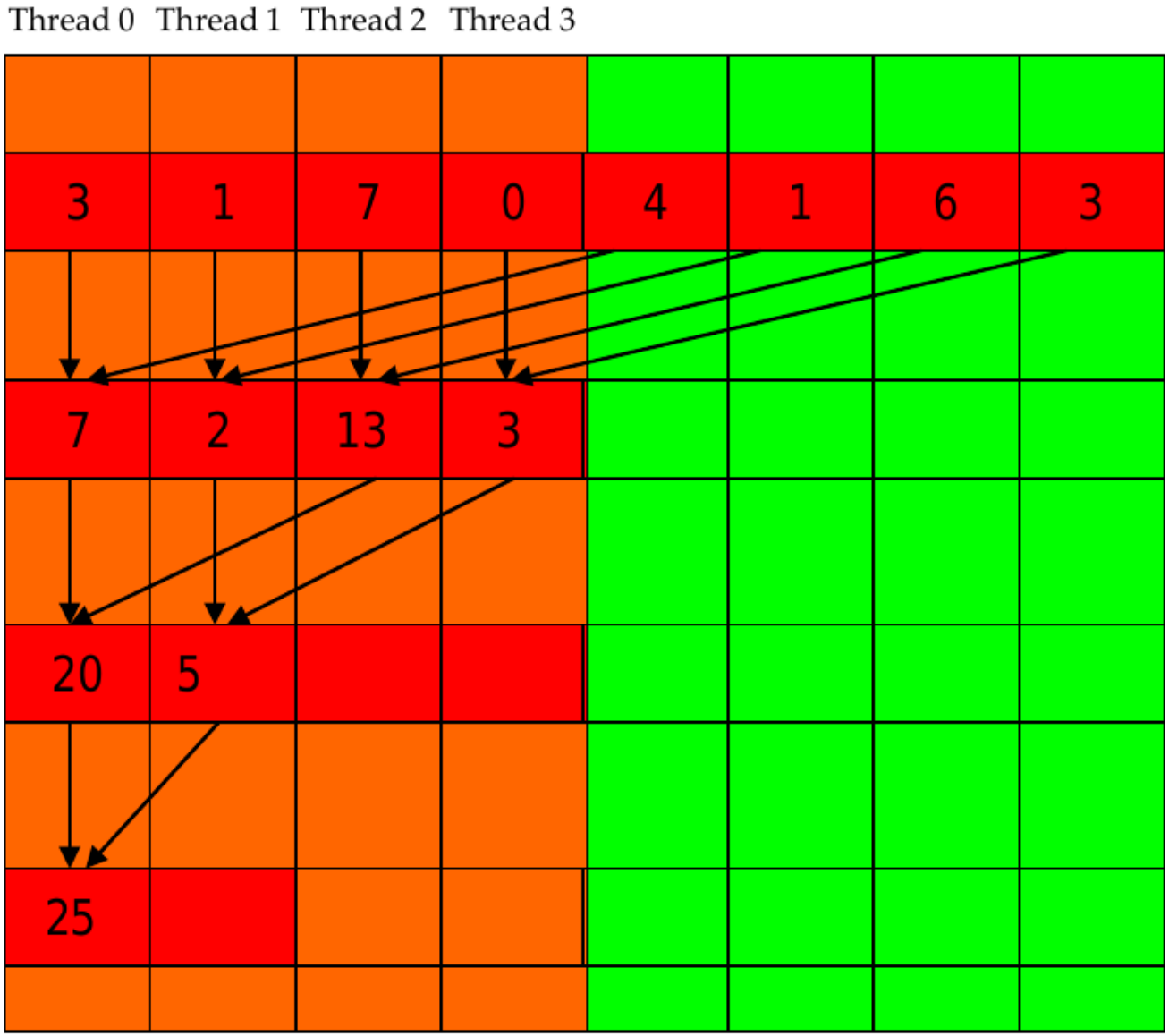
- A chaque itération, 2 chemins dans le flux de contrôle seront traversés séquentiellement pour chaque warp
  - Les threads qui effectuent une opération et ceux qui n'en font pas
  - Les threads qui ne font pas d'opération consomment malgré tout des ressources

- La moitié ou moins des threads exécuteront une opération après la première étape
  - Les threads avec un nombre impair sont désactivé après la première étape
  - Après la 5ème étape, des warps entiers dans un bloc vont échouer au test conditionnel, mauvaise utilisation des ressources mais pas de divergence
  - Cela est de pire en pire, lorsqu'il y a plus de 6 étapes, chaque warp actif a seulement un thread productif et cela jusqu'à ce que tous les warps d'un bloc finissent

Importance d'un bon usage des indexes

- Pour certains algorithmes, il est possible de modifier l'utilisation des indexes pour améliorer les comportements de divergence
- Toujours rassembler les sommes partielles dans les adresses au début du tableau *partialSum*[]
- Garder les threads actifs consécutifs (en terme d'indexes)

Exemple avec 4 threads



Un meilleur noyau de réduction

```
for (unsigned int stride = blockDim.x; stride > 0; stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

Analyse du nouveau noyau

Pour un bloc de threads de 1024

- Pas de divergence dans les 5 premières étapes
  - 1024, 512, 256, 128, 64, 32 threads consécutifs sont actifs à chaque étape
  - Tous les threads dans chaque warp sont ou actif ou inactif
- Seulement l'étape 5 (la dernière) aura de la divergence

Définition du Scan inclusif ou Somme préfixé

Définition : L'opération `scan inclusif` prend un opérateur binaire associatif $\oplus$ (plus cercle) et un tableau de n éléments
$[x_0, x_1, ...x_n]$
et retourne un tableau $[x_0, x_0 \oplus x_1, ...x_0 \oplus x_1 \oplus ... \oplus x_n]$

Exemple : si $\oplus$ est l'addition, alors l'opération `scan` sur le tableau devra retourner
3 1 7 0 4 1 6 3,
3 4 11 11 15 16 22 25.

Exemple d'application d'un scan inclusif

- Supposons que nous avons un sandwich de 100 cm pour nourir 10 personnes
- Nous savons combien de centimètres veulent chaque personne: $[3, 5, 2, 7, 28, 4, 3, 0, 8, 1]$
- Comment pouvons nous couper rapidement le sandwich ?
- Combien il en restera ?

- Méthode 1 : couper les sections séquentiellement : d'abord 3cm, puis 5cm puis 2cm ...

- Méthode 2 : calculer la somme préfixé
  $[3, 8, 10, 17, 45, 49, 52, 52, 60, 61]$ (39cm restants)

Cas d'utilisation typique du scan

- Le scan est un bloc de construction simple et pratique pour le parallélisme
  - Convertir des récurrences séquentiellement :
    for (j=1;j<n;j++) out[j] = out[j−1] + f(j);
  - En parallèle : forall (j) temp[j] = f(j) ; scan (out,temp);

Utiliser par plein d'algorithmes parallèles

- Tri par base
- Tri rapide
- Comparaison de caractères
- Analyse lexicale
- Flux de compression
- Evaluation polynomial
- Résoudre des récurrences
- Opération sur les arbres
- Histogrammes
- ...

Cas d'utilisation pratique

- Assigner des places dans un camping
- Assigner des places sur un marché
- Assigner des places dans des champs
- Allouer de la mémoire à des threads parallèles
- Allouer des buffers mémoires pour ces canaux de communications
- ...

Un scan inclusif pour l'addition en séquentiel

Pour une séquence $[x_0, x_1, x_2, ...]$
Calculer un résultat $[y_0, y_1, y_2, ...]$

Tel que
$y_0 = x_0$
$y_1 = x_0 + x_1$
$y_2 = x_0 + x_1 + x_2$

En utilisant la définition récursive
$y_i = y_{i-1} + x_i$

Une implémentation efficace en C

```
y[0] = x[0];
for (i = 1; i < Max_i; i++) y[i] = y[i-1] + x[i];
```

Efficace d'un point de vue computationelle:

- N additions nécessaires pour N éléments : $O(N)$
- Seulement légérement plus couteaux qu'une réduction séquentielle !

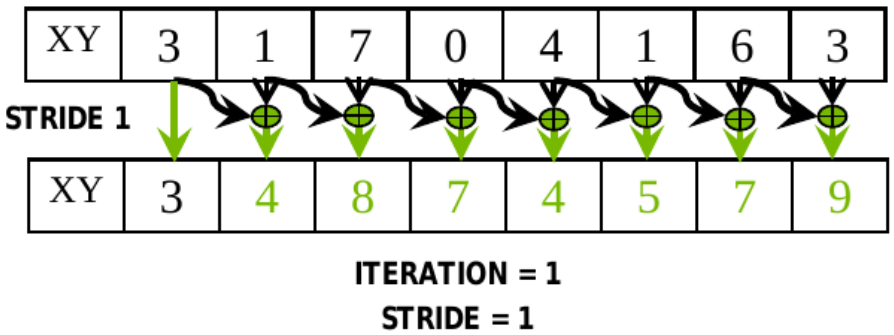
Un algorithme naif pour le scan inclusif en parallèle

- Assigner un thread pour calculer chaque élément de y
- Faire que tous les threads additionne tous les éléments de x nécessaire pour calculer l'élément de y $y_0 = x_0$
  $y_1 = x_0 + x_1$
  $y_2 = x_0 + x_1 + x_2$

La programmation paralléle s'est facile tant qu'on ne se soucie pas des performances !

Un meilleur algorithme parallèle de scan

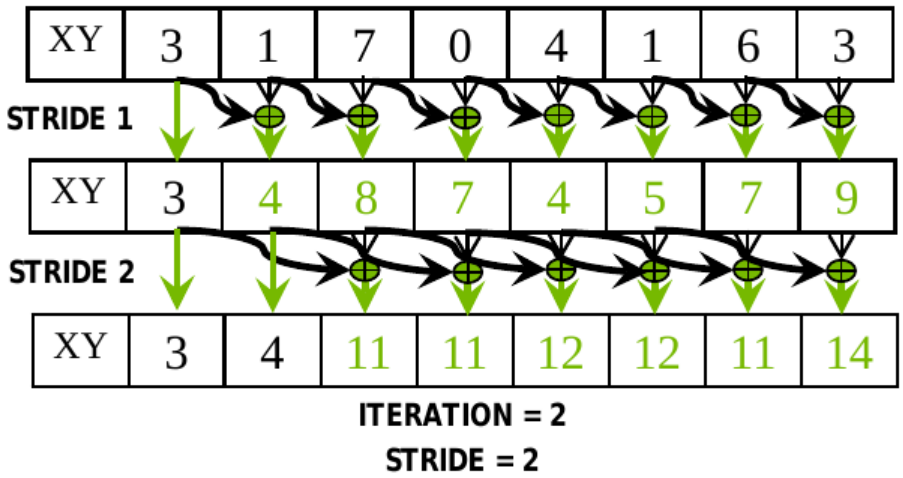
1. Lire l'entrée depuis la mémoire globale vers la mémoire partagée
2. Répéter $log(n)$ fois : parcourir de 1 à $n-1$: double parcours pour chaque itération



- Les threads actifs de *stride* à $n-1$ ($n - stride$ threads)
- Le thread j ajoute les éléments j et $j - stride$ depuis la mémoire partagée et écrit le résultat dans la case j en mémoire partagée
- Nécessite une barrière de synchronisation, une fois que la lecture est effectuée et avant que l'écriture le soit

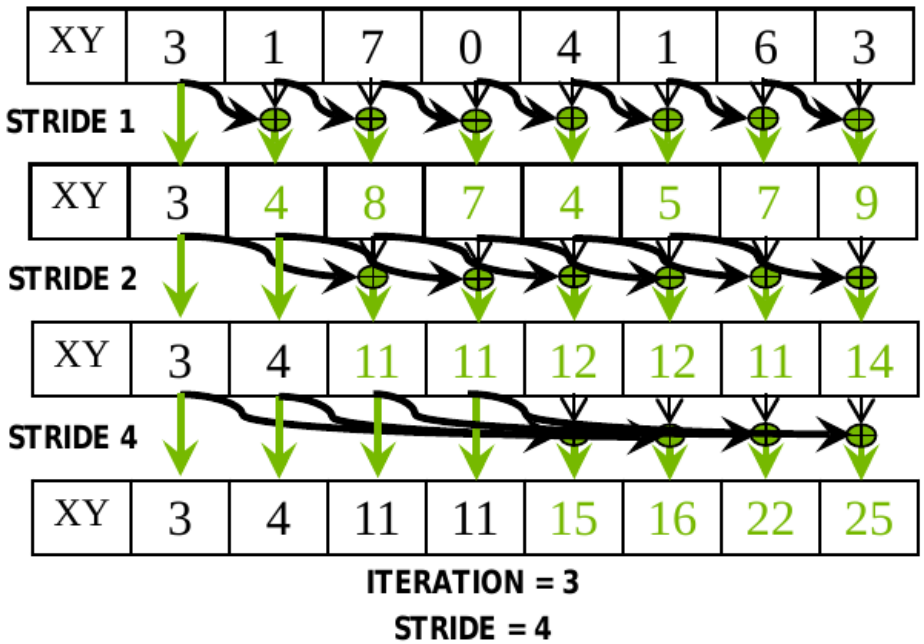
Un meilleur algorithme parallèle de scan

1. Lire l'entrée depuis la mémoire globale vers la mémoire partagée
2. Répéter $log(n)$ fois : parcourir de 1 à $n-1$: double parcours pour chaque itération



Un meilleur algorithme parallèle de scan

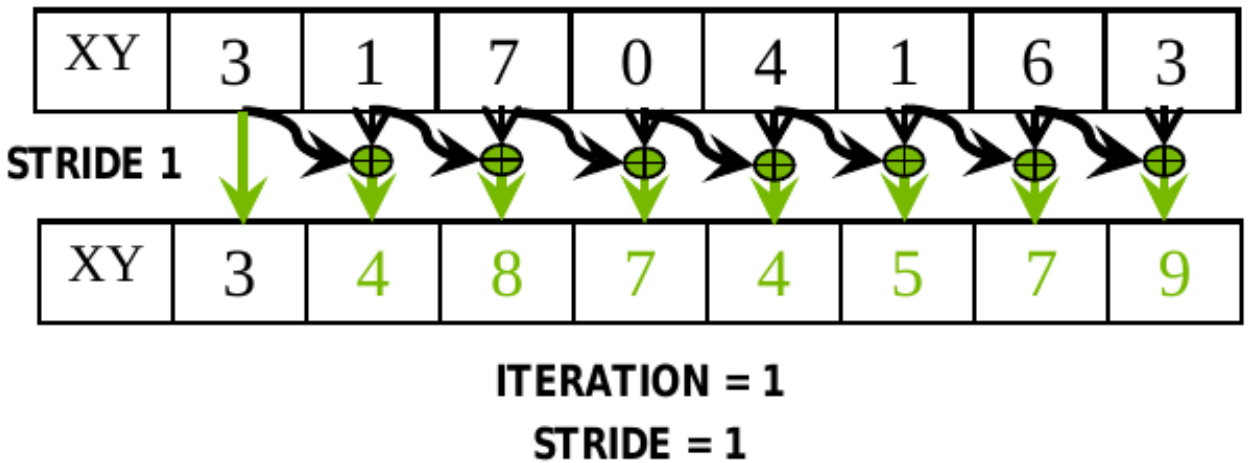
1. Lire l'entrée depuis la mémoire globale vers la mémoire partagée
2. Répéter $log(n)$ fois : parcourir de 1 à $n-1$: double parcours pour chaque itération
3. Ecrire les résultats depuis la mémoire partagée vers la mémoire globale



Gérer les dépendances

Durant toutes les itérations, chaque thread peut écraser l'entrée d'un autre thread

- La barrière de synchronisation garantit que toutes les entrées ont été écrites
- Tous les threads sécurisent leurs entrées qui peuvent être écrasées par un autre thread
- La barrière de synchronisation est nécessaire pour garantir que tous les threads ont des entrées sécurisées
- Tous les threads font une addition et écrivent une sortie



Une version inefficace du noyau scan

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y,
  __shared__ float XY[SECTION_SIZE];
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < InputSize) {XY[threadIdx.x] = X[i];}
  // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride <= threadIdx.x; stride *=
    __syncthreads();
    float in1 = XY[threadIdx.x - stride];
    __syncthreads();
    XY[threadIdx.x] += in1;
  }
  __ syncthreads();
  if (i < InputSize) {Y[i] = XY[threadIdx.x];}
}
```

Considérons l'efficacité de ce code

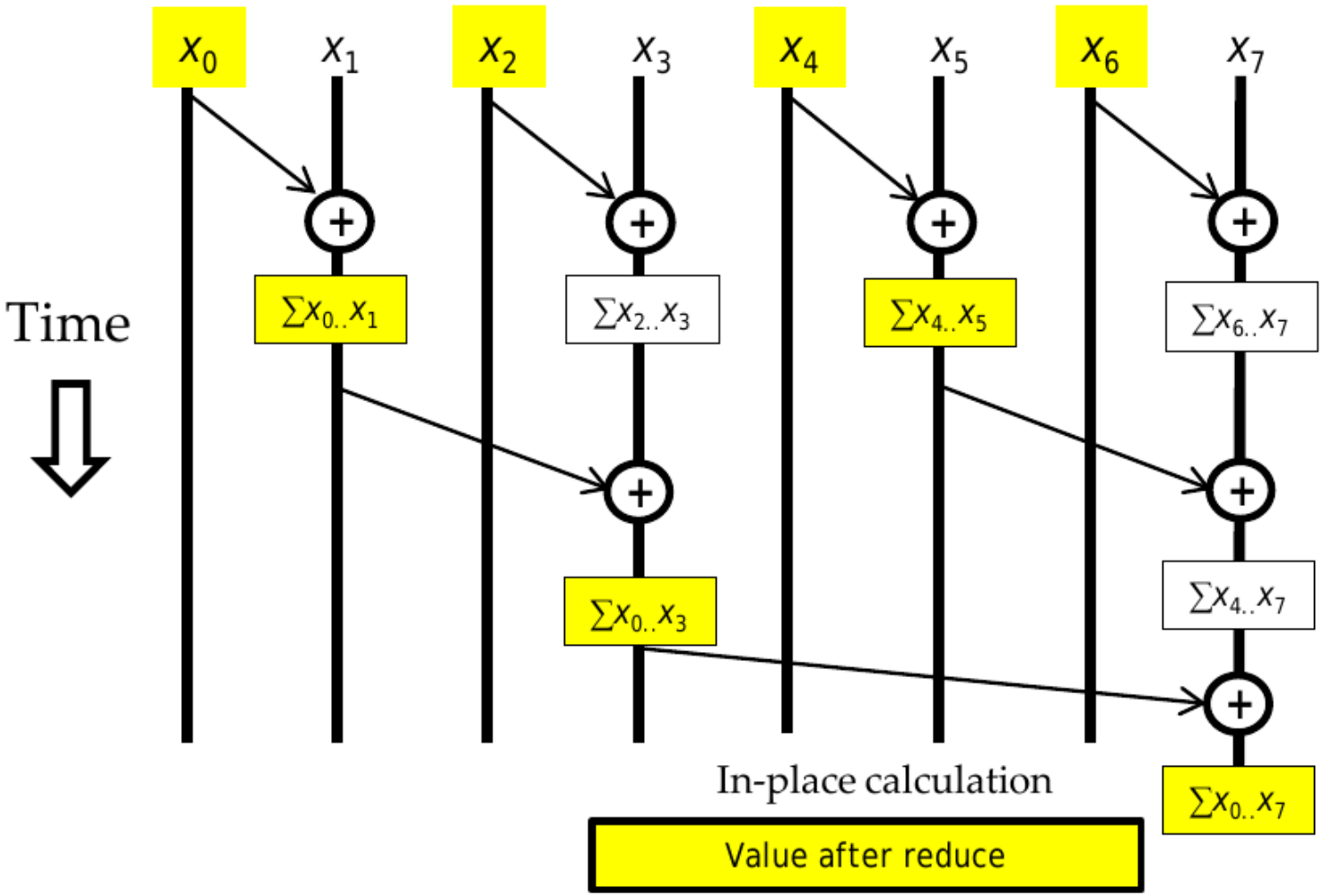
- Le scan exécute $log(n)$ itérations en paralléle
  - Les itérations font respectivement $(n-1), (n-2), (n-4), ..., (n-\frac{n}{2})$ opérations addition
  - Pour un total d'opération addition : $n * log(n) \rightarrow O(n * log(n))$
- Cet algorithme de scan n'est pas efficace
  - L'algorithme séquentiel de scan effectue n additions
  - Un facteur $log(n)$ peut coûter cher : 10x pour 1024 éléments !!!
- Un algorithme paralléle peut être plus lent que son équivalent séquentiel quand les ressources de calcul sont saturées par une faible efficacité !

Améliorer l'efficacité

- Des arbres équilibrés
  - Construire des arbres binaires équilibrés pour les données en entrée et le parcourir vers et depuis la racine
  - L'arbre n'est pas réellement la structure de données utilisée mais un concept pour déterminer ce que font chaque thread à chaque étape

- Pour `scan`
  - Traverser l'arbre depuis les feuilles jusqu'à la racine en construisant des sommes partielles à chaque noeud interne de l'arbre
  - La racine contient la somme de toutes les feuilles
  - Traverser dans le sens invrse l'arbre pour construire les résultats à partir des sommes partielles

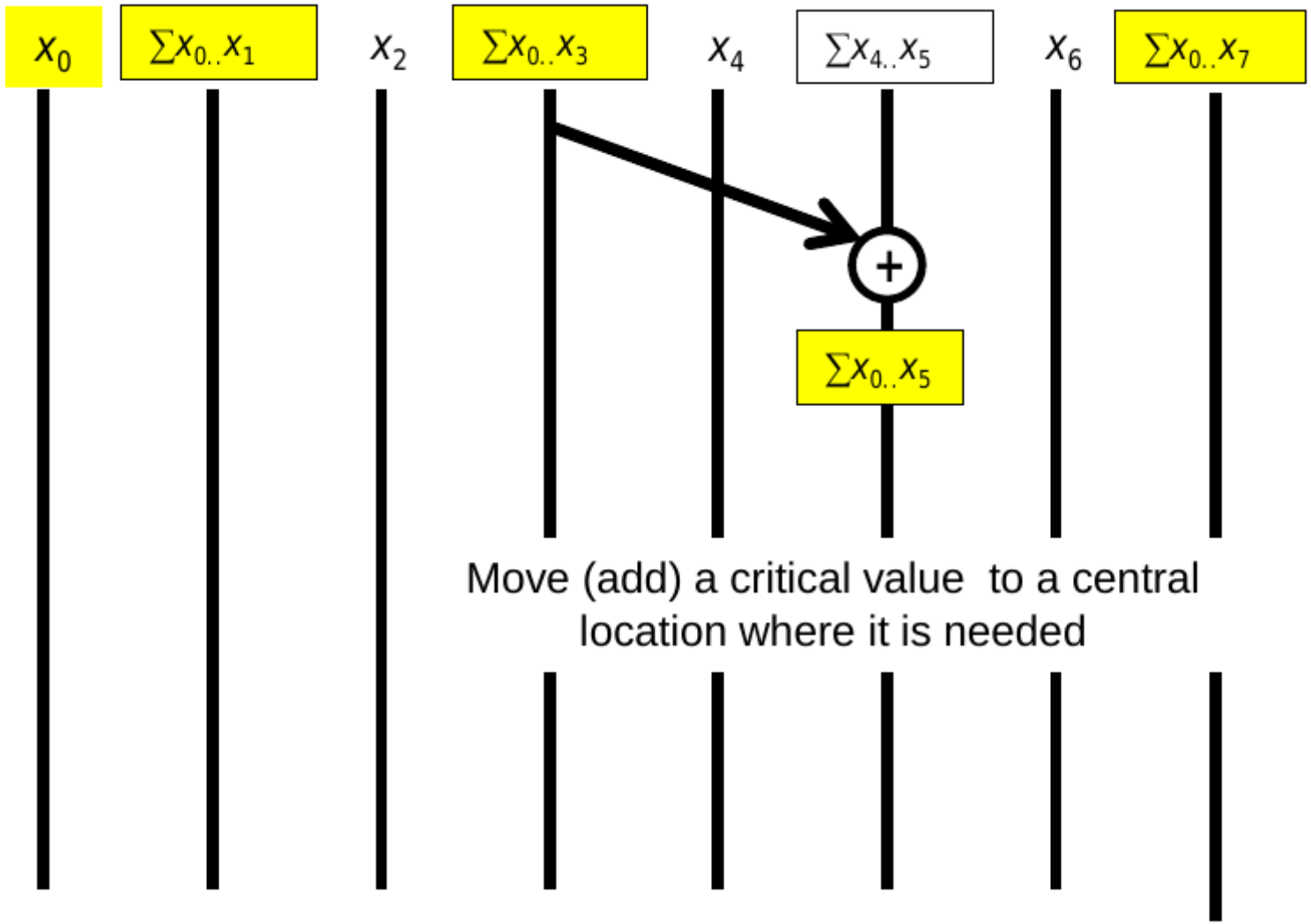
Paralléle scan : La phase de réduction



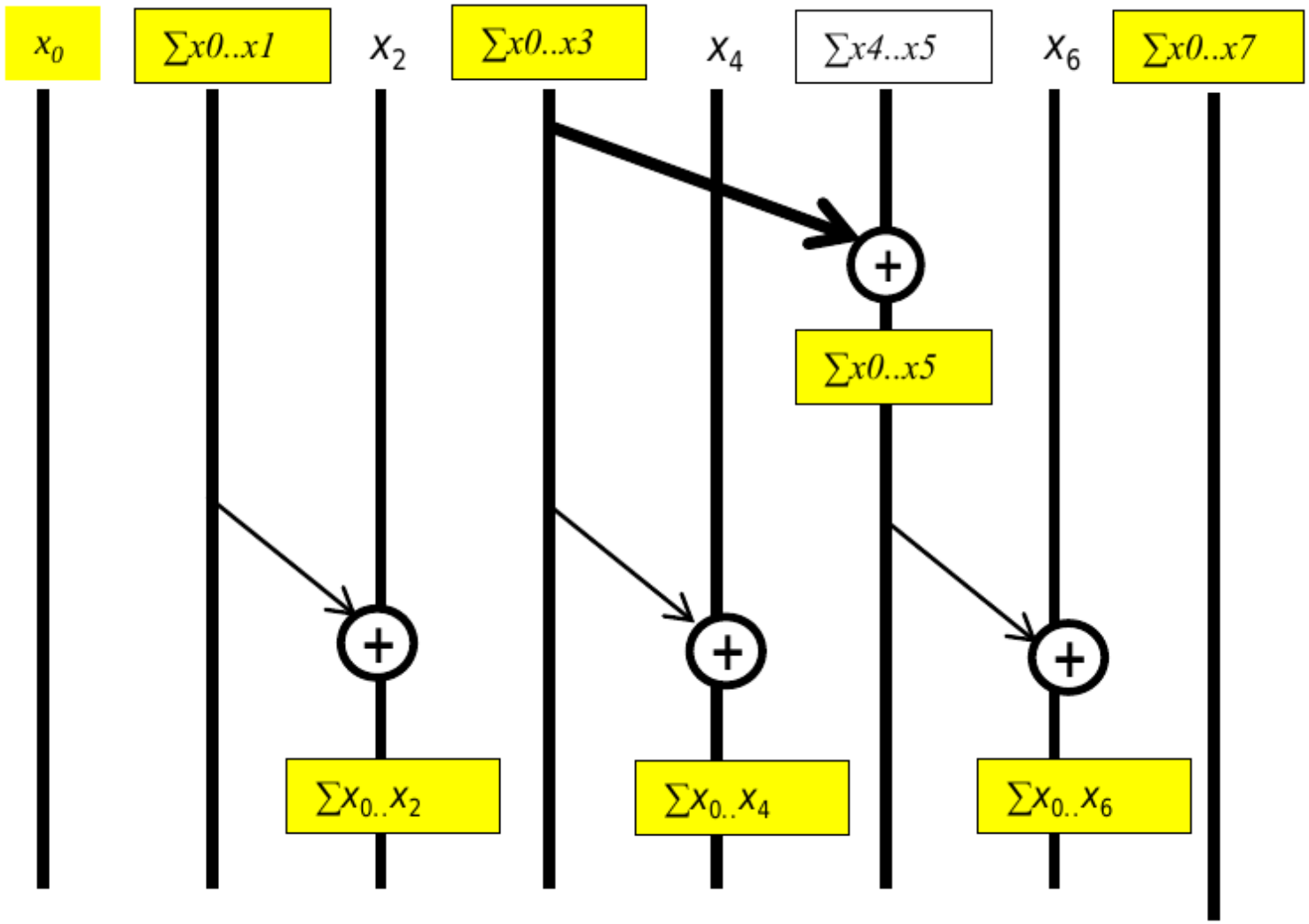
Code du noyau: Phase de réduction

```
// XY[2*BLOCK_SIZE] is in shared memory
for (unsigned int stride = 1;stride <= BLOCK_SIZE; stride *= 2)
{
  int index = (threadIdx.x+1)*stride*2 - 1;

  if(index < 2*BLOCK_SIZE)
    XY[index] += XY[index-stride];

  __syncthreads();
}
```

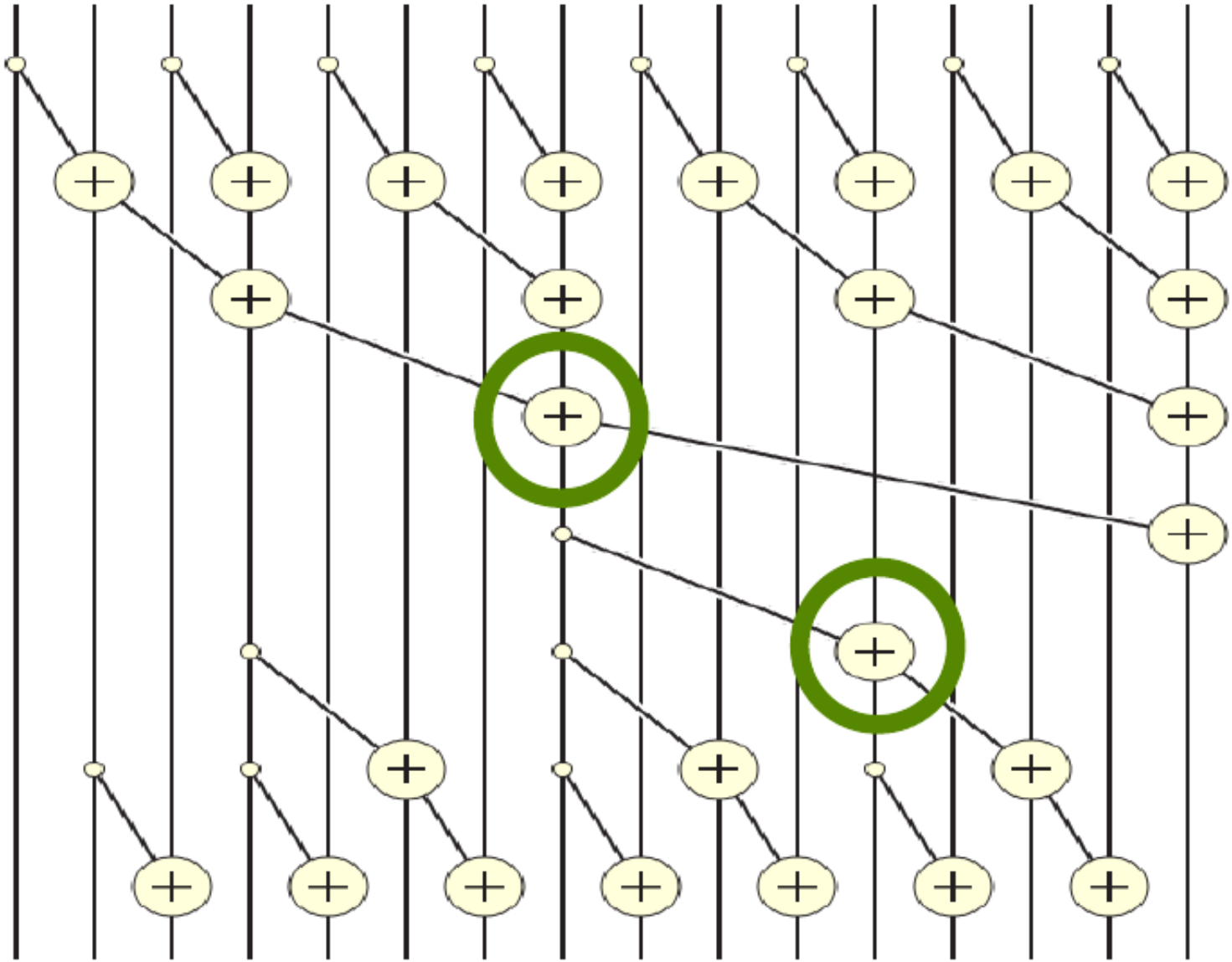
Paralléle scan : après la réduction



Paralléle scan : après la réduction



Vue d'ensemble



Code du noyau: après la phase de réduction

```
for (unsigned int stride = BLOCK_SIZE/2; stride > 0; stride /= 2
  __syncthreads();
  int index = (threadIdx.x+1)*stride*2 - 1;
  if(index+stride < 2*BLOCK_SIZE) {
    XY[index + stride] += XY[index];
  }
}

__syncthreads();

if (i < InputSize) Y[i] = XY[threadIdx.x];
```

Analyse de l'efficacité du noyau efficace

- Le noyau efficace exécute $log(n)$ iterations paralléles pour l'étape de reduction
  - Les itérations effectuent respectivement $\frac{n}{2}, \frac{n}{4}, ..., 1$ opérations d'addition
  - Au total : $(n-1) \rightarrow O(n)$
- Le noyau efficace exécute $log(n) - 1$ itérations paralléles pour l'étape post-réduction
  - Les itérations effectuent respectivement $2-1, 4-1, ..., \frac{n}{2-1}$ opérations d'addition
  - Au total : $(n-2) - (log(n) - 1) \rightarrow O(n)$
- Les 2 phases effectuent jusqu'à mais pas plus $2 * (n-1)$ opérations d'addition
- Le nombre total d'addition n'est pas plus de 2 fois ce qui est effectué par l'algorithme séquentiel efficace
  - Le bénéfice d'une exécution paralléle peut largement surmonter de ce doublement du travail quand il y a suffisament de ressources de calcul

Quelques compromis

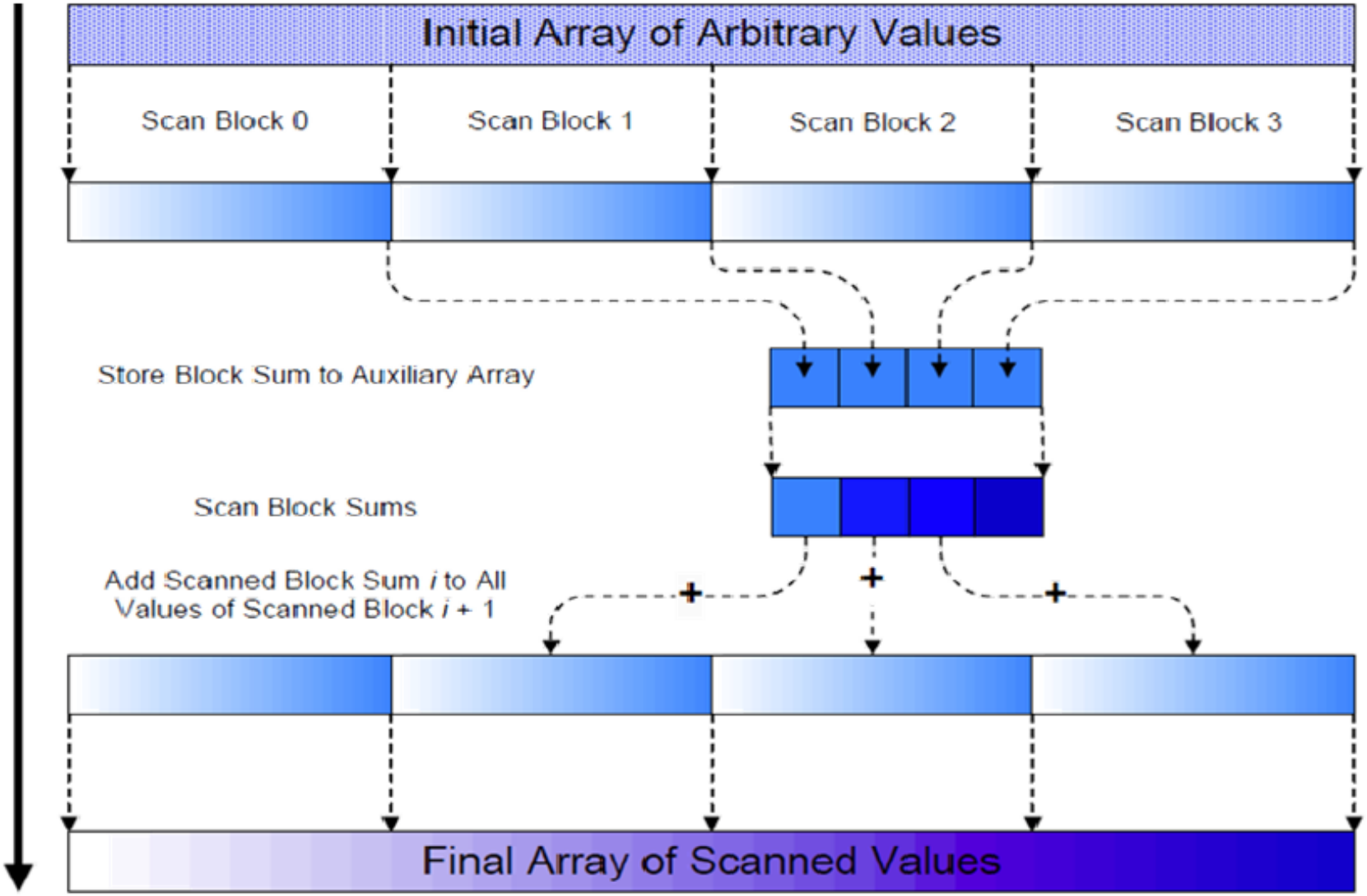
- Le noyau `scan` efficace est normalement plus souhaitable
  - Une meilleur efficacité énergétique
  - Moins de ressources sont nécessaire pour son exécution
- Malgré tout, la version inefficace du noyau pourrait être meilleur dans l'absolu en terme de performance grâce à sa phase unique
  - Il faut dans ce cas suffisament de ressources

Traitement de larges tableaux en entrée

- En se basant sur le noyau `scan` efficace
- Avoir chaque section de $2 * blockDim.x$ éléments assignée à un bloc
  - Effecuter un scan paralléle sur chaque section
- Chaque bloc écrit la somme de sa section dans un tableau de somme $Sum[]$ en utilisant son indexe $blockIdx.x$
- Exécuter le noyau `scan` sur le tableau $Sum[]$
- Ajouter les valeurs du tableau $Sum[]$ à tous les éléments des sections correspondantes

- L'adaptation du noyau inefficace se fait de manière similaire

Schéma de l'exécution complète de `scan`



Définition du scan exclusif

Définition : L'opération `scan exclusif` prend un opérateur binaire associatif $\oplus$ (plus cercle) et un tableau de n éléments
$[x_0, x_1, ...x_n]$
et retourne un tableau $[0, x_0, x_0 \oplus x_1, ...x_0 \oplus x_1 \oplus ... \oplus x_{n-2}]$

Exemple : si $\oplus$ est l'addition, alors l'opération `scan` sur le tableau devra retourner
$[3, 1, 7, 0, 4, 1, 6, 3]$,
$[0, 3, 4, 11, 11, 15, 16, 22]$.

Pourquoi utiliser un scan exclusif

- Pour trouver l'adresse de début de buffer
- Les scans inclusif et exclusif peuvent facilement être déduit l'un de l'autre, c'est une question de commodité

	$[3, 1, 7, 0, 4, 1, 6, 3]$
Exclusif	$[0, 3, 4, 11, 11, 15, 16, 22]$
Inclusif	$[3, 4, 11, 11, 15, 16, 22, 25]$

Une version simple du noyau `scan exclusif`

- Adaptation de la version inefficace du noyau `scan inclusif`
- Block 0:
  - Thread 0 met un 0 dans $XY[0]$
  - Tous les autres threads mettent la valeur de $X[threadIdx.x - 1]$ dans $XY[threadIdx.x]$
- Tous les autres blocs
  - Tous les threads mettent la valeur de $X[blockIdx.x * blockDim.x + threadIdx.x - 1]$ dans$XY[threadIdx.x]$