Calcul parallèle avancé

ENSIMAG 3A - IF

17 février 2014

Chapitre 3 - Patterns de programmation

Agenda

1. Quelques exemples
2. Premiers patterns
3. Ameliorations des patterns

Exemple 1 : Calcul matriciel

- Tous les threads d'un bloc doivent participer : chaque thread charge un élément de M et un élément de N des blocs de la matrice.
- On associe les éléments chargés à chaque thread en faisant en sorte que les accès soient contigües.
- Il est nécessaire de synchroniser les threads pour être sur que tous éléments sont bien chargés et que tous les éléments sont bien utilisés.
- `__syncthreads()` permet de synchroniser les threads d'un même bloc.

```
//Charger un bloc matriciel
int tx = threadIdx.x
int ty = threadIdx.y
// Accede au bloc 0 avec un indexage 2D :
...M[Row][tx]
...N[ty][Col]
// Accede au bloc 1 avec un indexage 2D :
...M[Row][1*TILE_WIDTH+tx]
...N[1*TILE_WIDTH+ty][Col]
// Accede aux elements avec un indexage 1D
...M[Row*Width + m*TILE_WIDTH + tx]
...N[(m*TILE_WIDTH+ty) * Width + Col]
```

Exemple 1 : Multiplication par blocs

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int
    Width)
{
     __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
     __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
     int bx = blockIdx.x; int by = blockIdx.y;
     int tx = threadIdx.x; int ty = threadIdx.y;
     // Position de l'element de P sur lequel on travail
     int Col = bx * TILE_WIDTH + tx;
     float Pvalue = 0;
     // Boucle sur l'ensemble les blocs de M et N necessaire pour
     // calculer un element de
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {
          // Chargement collaboratif en memoire partagee
          ds_M[ty][tx] = d_M[Row*Width + m*TILE_WIDTH+tx];
          ds_N[ty][tx] = d_N[(m*TILE_WIDTH+ty)*Width+Col]; 11.
              __syncthreads();
          for (int k = 0; k < TILE_WIDTH; ++k)
               Pvalue += ds_M[ty][k] * ds_N[k][tx];
          __synchthreads();
     }
     d_P[Row*Width+Col] = Pvalue;
}
```

Exemple 1 : Découpage

Afin de déterminer exactement le matériel, on peut récupérer les informations de façon dynamique

- Nombres d'accélérateurs dans le système

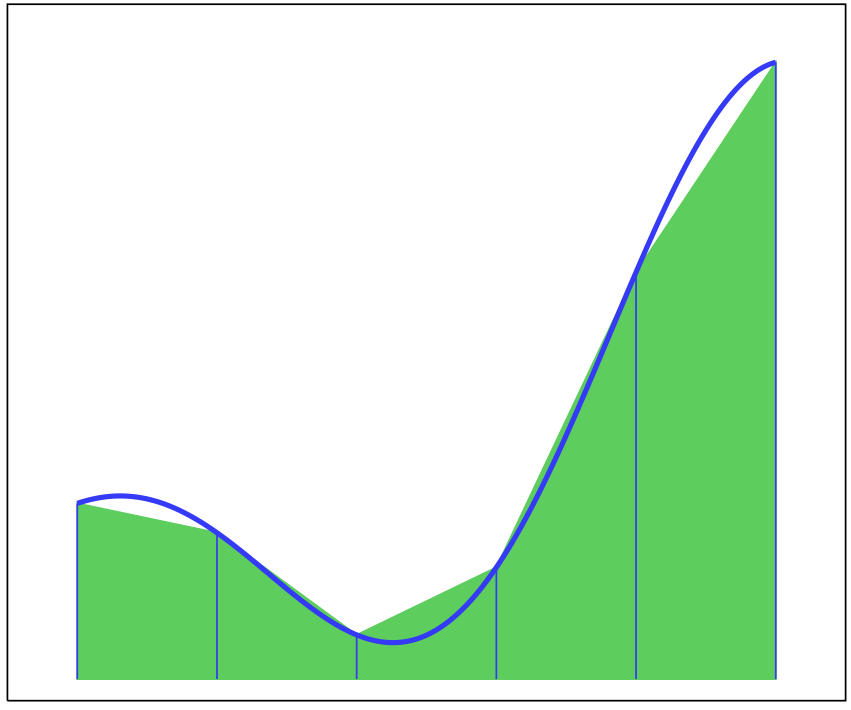
```
int dev_count;
cudaGetDeviceCount(&dev_count);
```

- Propriétés

```
cudaDeviceProp dev_prop;
for (i = 0; i < dev_count; i++) { cudaGetDeviceProperties( &dev_prop, i)
    ;
// on peut decider si l'accelerateurs a des ressources suffisantes.
}
```

- `cudaDeviceProp` est une structure C

Example 2 : Calcul Integral



La formule du trapèze s'écrit $\forall f \in C^2([a;b]), \exists \xi \in [a;b]$

$$I = h\left(f(a) + 2\sum_{k=1}^{n-1} f(a+kh) + f(b)\right) - (b-a)\frac{h^2}{12} f^{(2)}(\xi)$$

Example 2 : Pseudo-code

```
Input b, a, n
h = (b-a)/n
I = (f(a) + f(b))/2.0;
for (i = 0; i<= n-1; i++)
{
  x_i = x_i + h
  I += f(x_i)
}
I = h*I
```

Example 2 : Pseudo-code parallèle

```
Find b, a, n                                                    1
h = (b-a)/n                                                     2
local_n = n/n_p                                                 3
local_a = a + id * local_n*h                                    4
local_b = local_a + local_n*h                                   5
local_I = Trap(local_a, local_b, local_n)                       6
                                                                7
If (id == 0)                                                    8
{                                                               9
  I = local_I;                                                  10
  for (i = 1; i<= n_p; i++)                                     11
  {                                                             12
    I += local_I;                                               13
  }                                                             14
  Display I;                                                    15
}                                                               16
```

Example 3 : Tri Bitonic

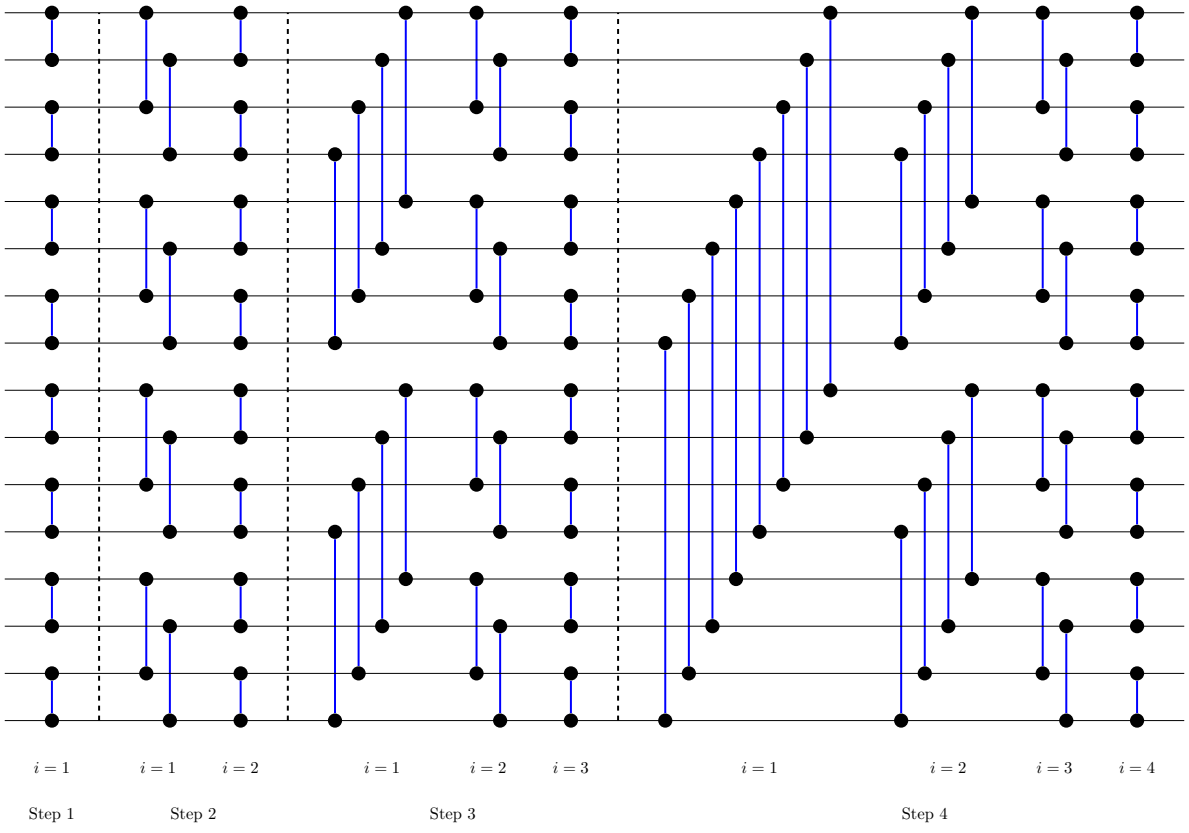
- Une suite bitonic est une suite de nombre $a_0, a_1, \ldots, a_{n-1}$ qui augmente de façon monotone en valeur.which monotonically, atteint un maximum et décroit de manière monotone.

$$a_0 < a_1 < \ldots < a_{i-1} < a_i > a_{i+1} > \ldots > a_{n-2} > a_{n-1}$$

  pour somme valeur de i. Une suite est également considérée comme bitonic si la relation précédente est atteinte par permutation circulaire des nombres.
- Une suite de 2 éléments est bitonic.
- 4 éléments : (a_0, a_1, a_2, a_3)
  - Trier (a_0, a_1) tel que $a_0 \leq a_1$
  - Trier (a_2, a_3) tel que $a_2 \geq a_3$
- 8 éléments : $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$
  - Réaliser un trie bitonic pour (a_0, a_1, a_2, a_3) et (a_4, a_5, a_6, a_7).
  - Faire une séparation bitonic pour faire en sorte que (a_0, a_1, a_2, a_3) soit croissante.
  - Faire une séparation bitonic pour faire en sorte que (a_4, a_5, a_6, a_7) soit décroissante.

Example 3 : Idée Séquentielle

- Pour les listes de longueur $l = 2, 4, 8, \ldots 2^m = n$
- Utiliser le tric bitonic afin de créer des groupes successifs de l éléments alternativement croissants et décroissants.

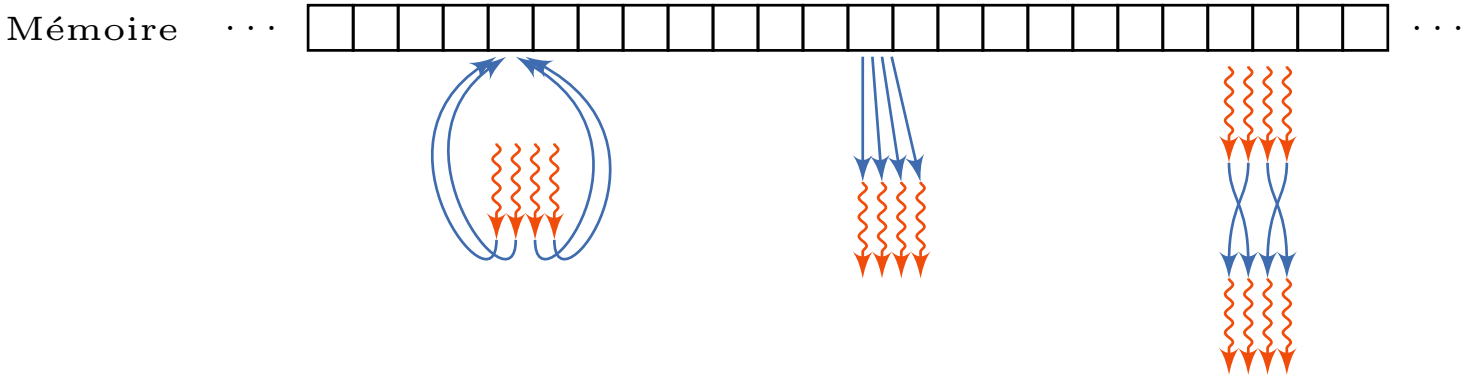


Agenda

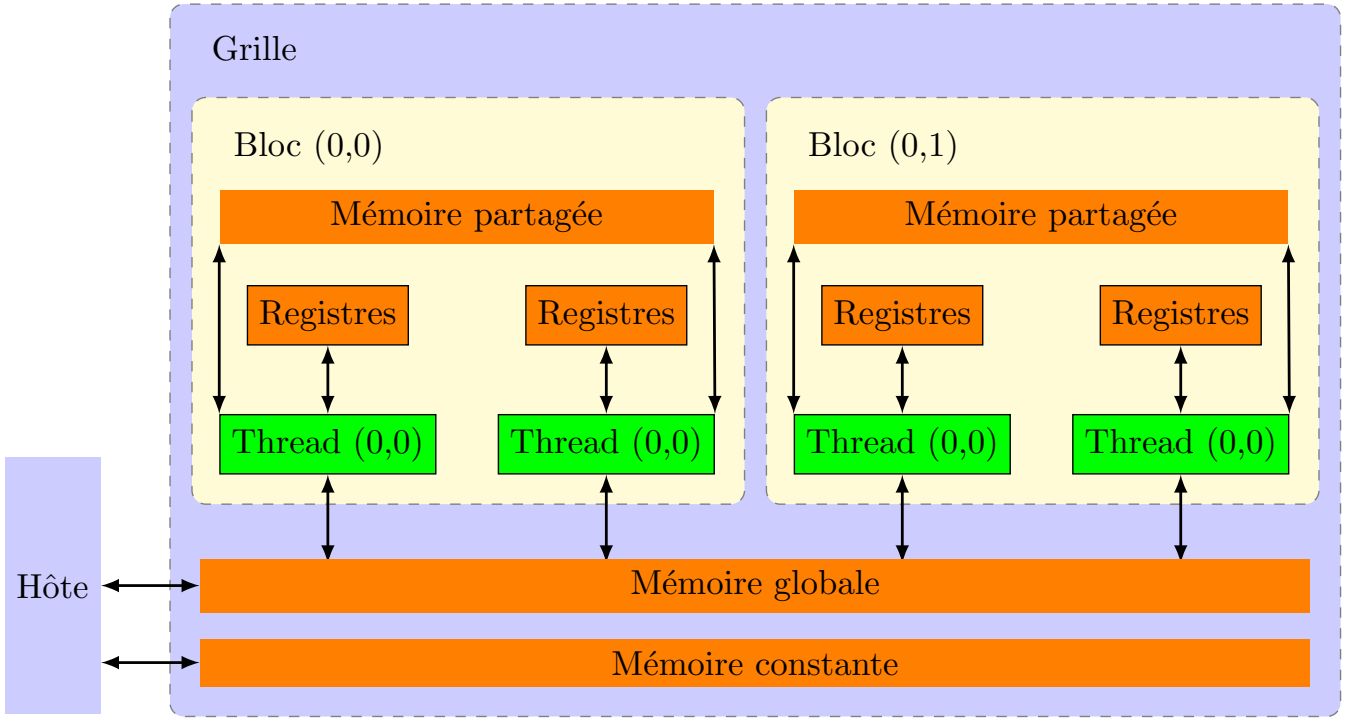
Principe

Idée :
Un accélérateur est efficace si un très grand nombre de threads résolvent un problème en travaillant ensemble.

Le point clé est la communication entre les threads.



Les différents niveaux



- Registres : accès très haut débit en lecture et écriture (1 cycle) par les threads.
- Mémoire partagée : accès haut débit en lecture et écriture (5 cycles) par les blocs
- Mémoire globale : accès bas débit en lecture et écriture (500 cycles) par la grille
- Mémoire constante : accès haut débit en lecture seule (5 cycles)

CUDA : les garanties

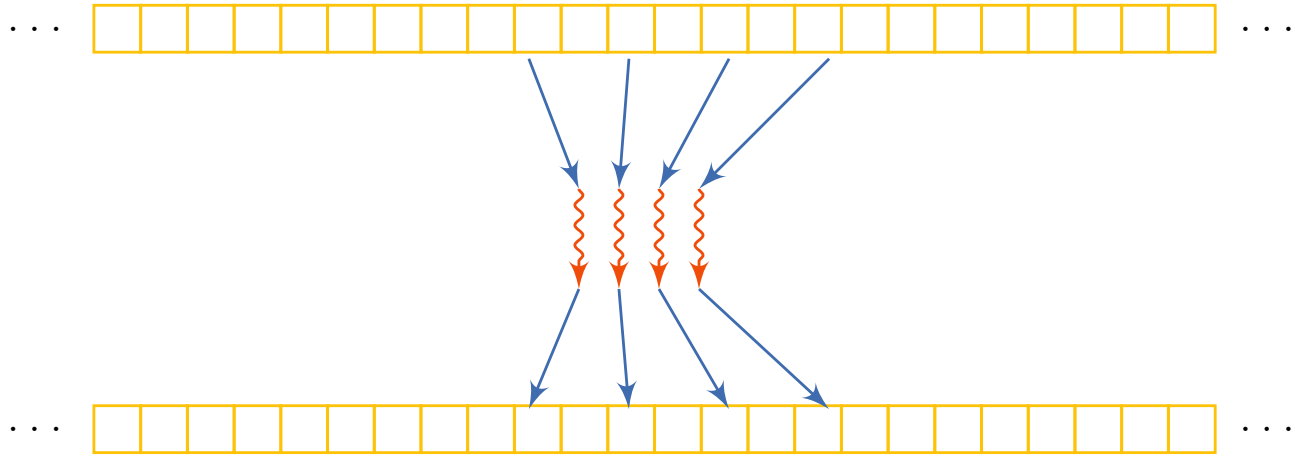
- Pas d'hypothèses sur l'affectation des blocs sur les les unités de calculs.
- Pas de communication entre les blocs.
- Les threads, les blocs doivent se terminer avant de pouvoir exécuter les noyaux suivants.
- Tous les threads d'un blocs s'exécutent sur la même unité de calcul au même moment.
- Tous les blocs d'un noyau doivent être terminer avant que n'importe quel bloc du noyau suivant puisse être exécuter.

Identification des patterns de base

```
#include <stdio.h>

#define NUM_BLOCKS 16
#define BLOCK_WIDTH 1

__global__ void identifierBloc()
{
  printf("Je suis un thread dans le bloc %d\n", blockId.x);
}

int main(int argc, char **argv)
{
  // Lancement du noyau
  identifierBloc<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

  // Force printf a afficher le resultat
  cudaDeviceSynchronize();

  printf("Terminer\n");

  return 0;
}
```

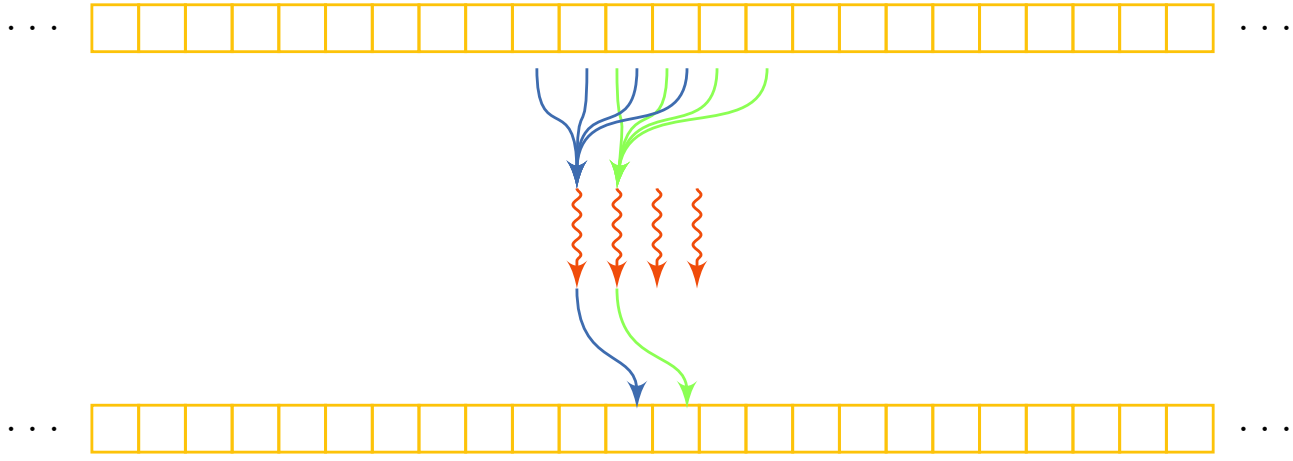
Map pattern

Les threads lisent et écrivent les données à partir d'emplacements mémoires spécifiques distincts.

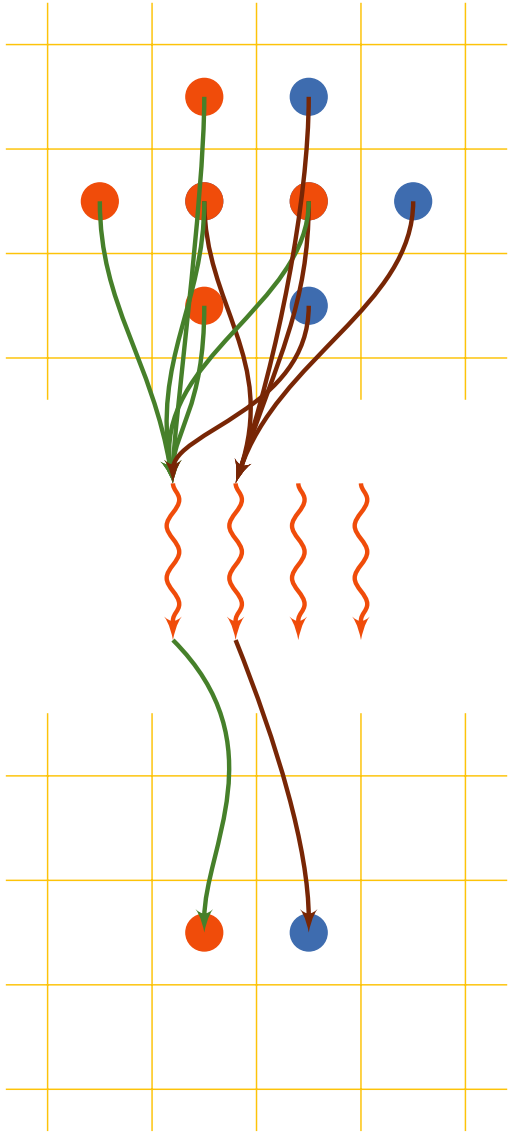


Gather pattern

Les threads lisent les données à partir d'emplacements mémoires spécifiques et distincts, réalisent une opération sur ces données et écrivent le résultat sur un unique emplacement.

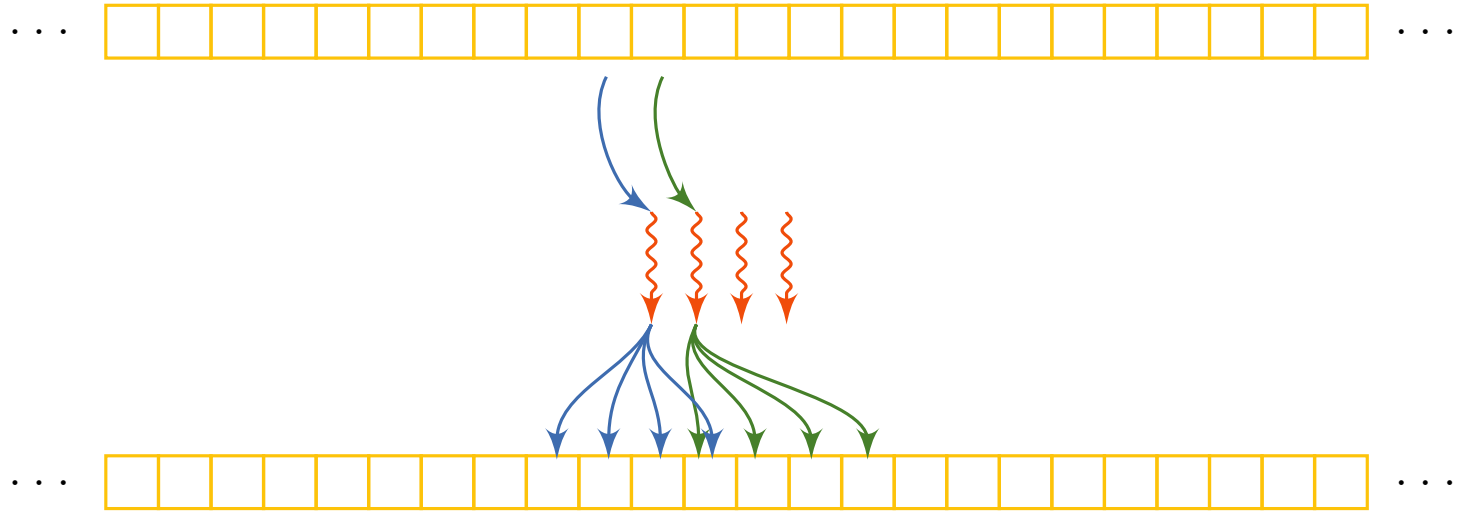


Gather pattern

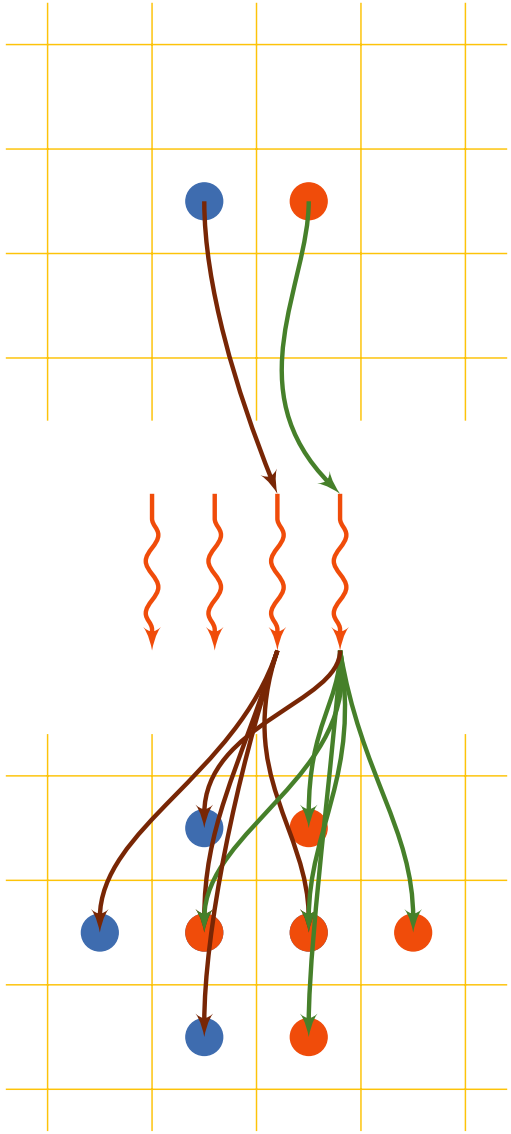


Scatter pattern

Les threads calculent l'emplacement mémoire dans lequel la résultat sera écrit.

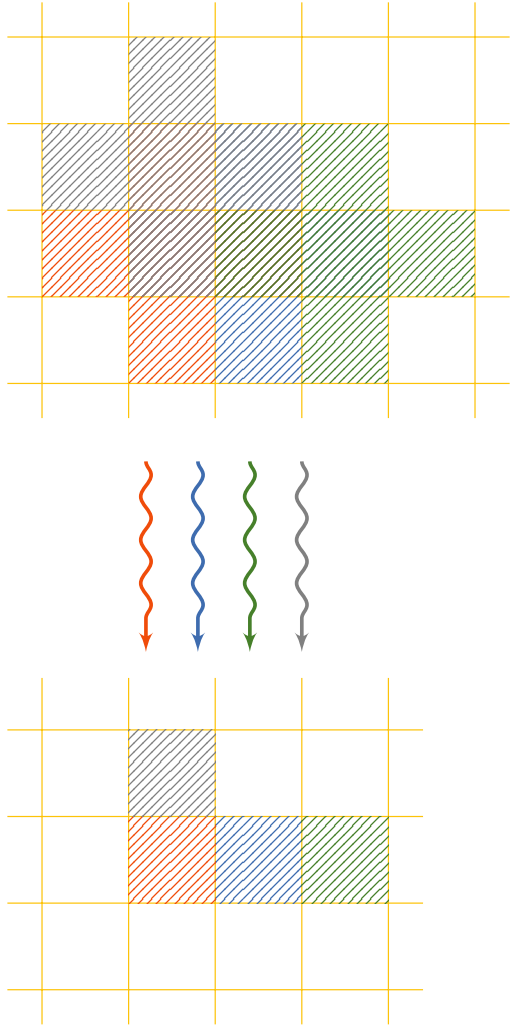


Scatter pattern (2)



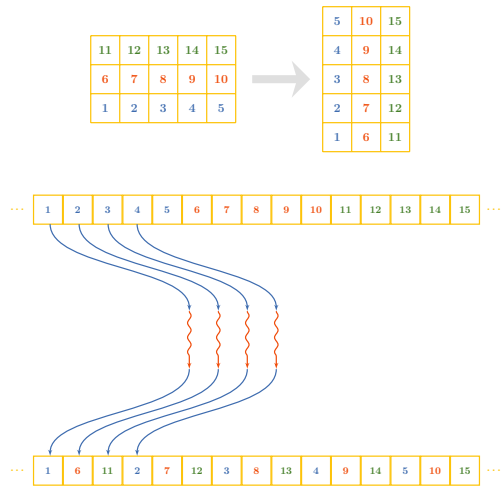
Motif pattern

Les threads mettent à chaque élément d'un tableau en utilisant les éléments voisins en utilisant un même motif.



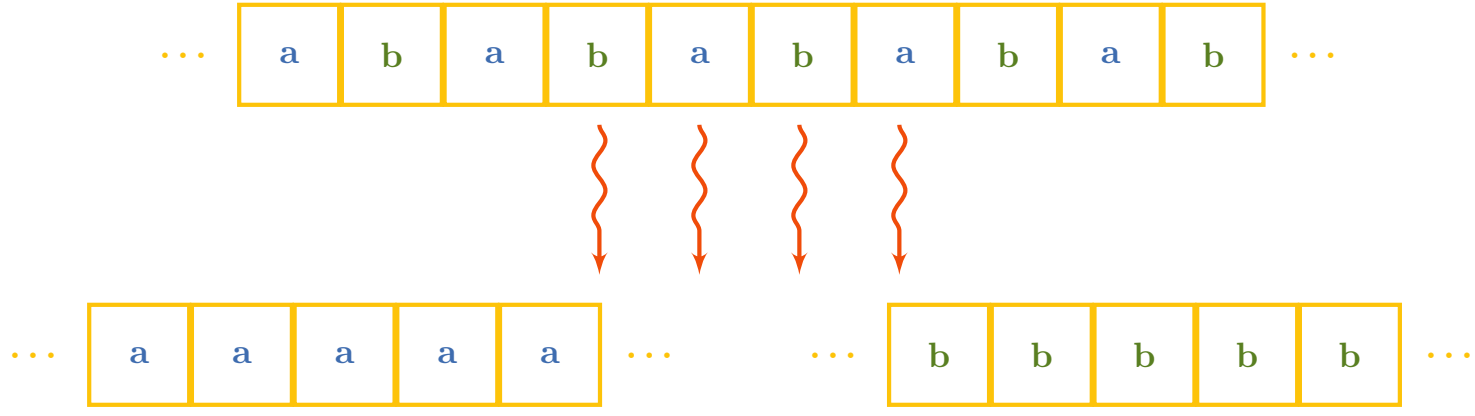
Transposition pattern

Chaque thread lis un élément du tableau et le réécrit de manière épars, selon la taille de la colonne.



Transposition pattern (2)

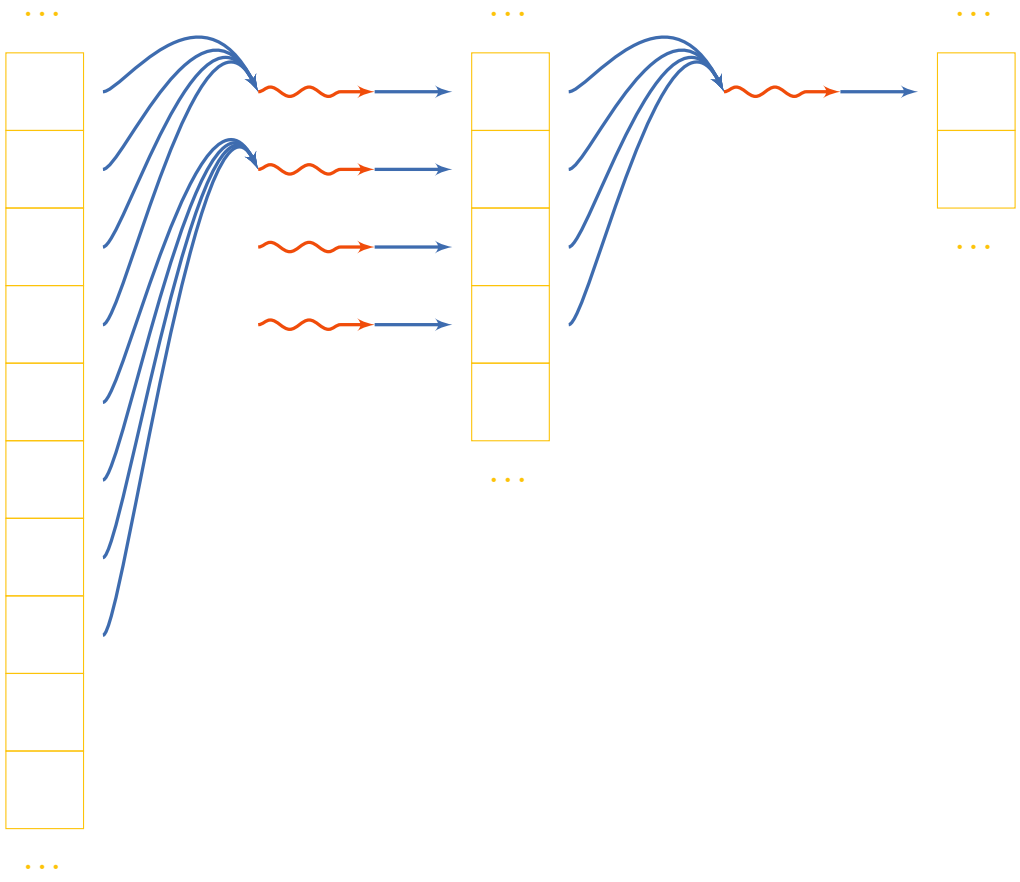
Le concept de transposition peut également à des tableaux de structures : on obtient une opération qui transforme un tableau de structures en structure de tableaux.



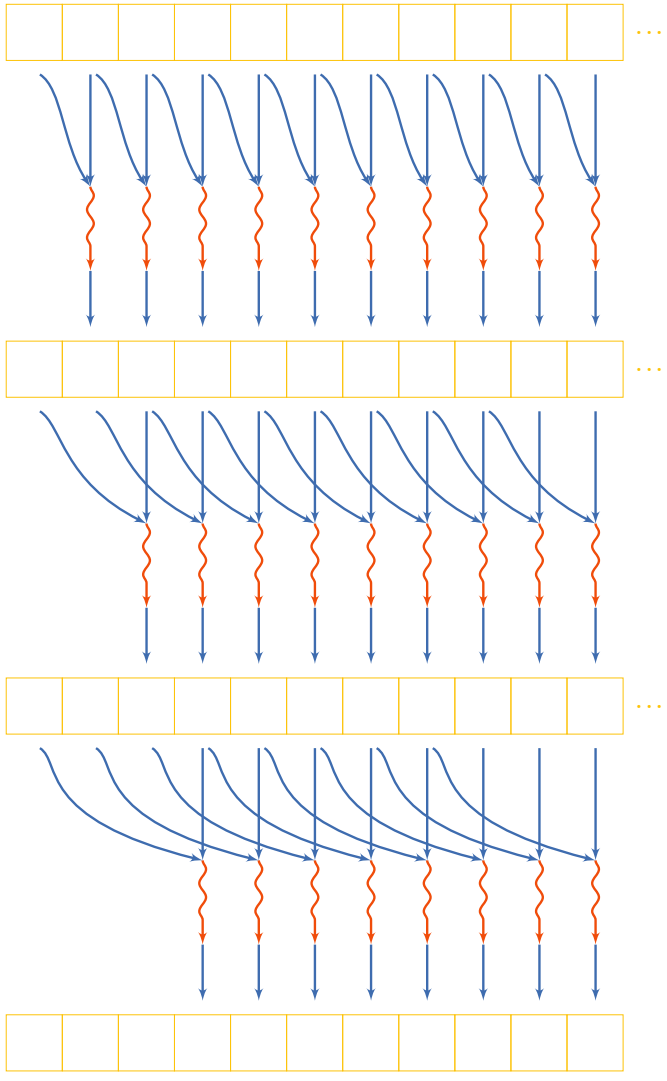
Identification des patterns de base

```
float out[], in[];
int i= threadId.x;
int j= threadId.y;

const float pi = 3.1415;

out[i] = pi * in[i];
out[i + j*128] = in[j + i*128];

if (i %2 )
{
  out[i-1] += pi * in[i]; out[i+1] += pi*in[i];
  out[i] = (in[i] + in[i-1] + in[i+1]) * pi/3.0f;
}
```

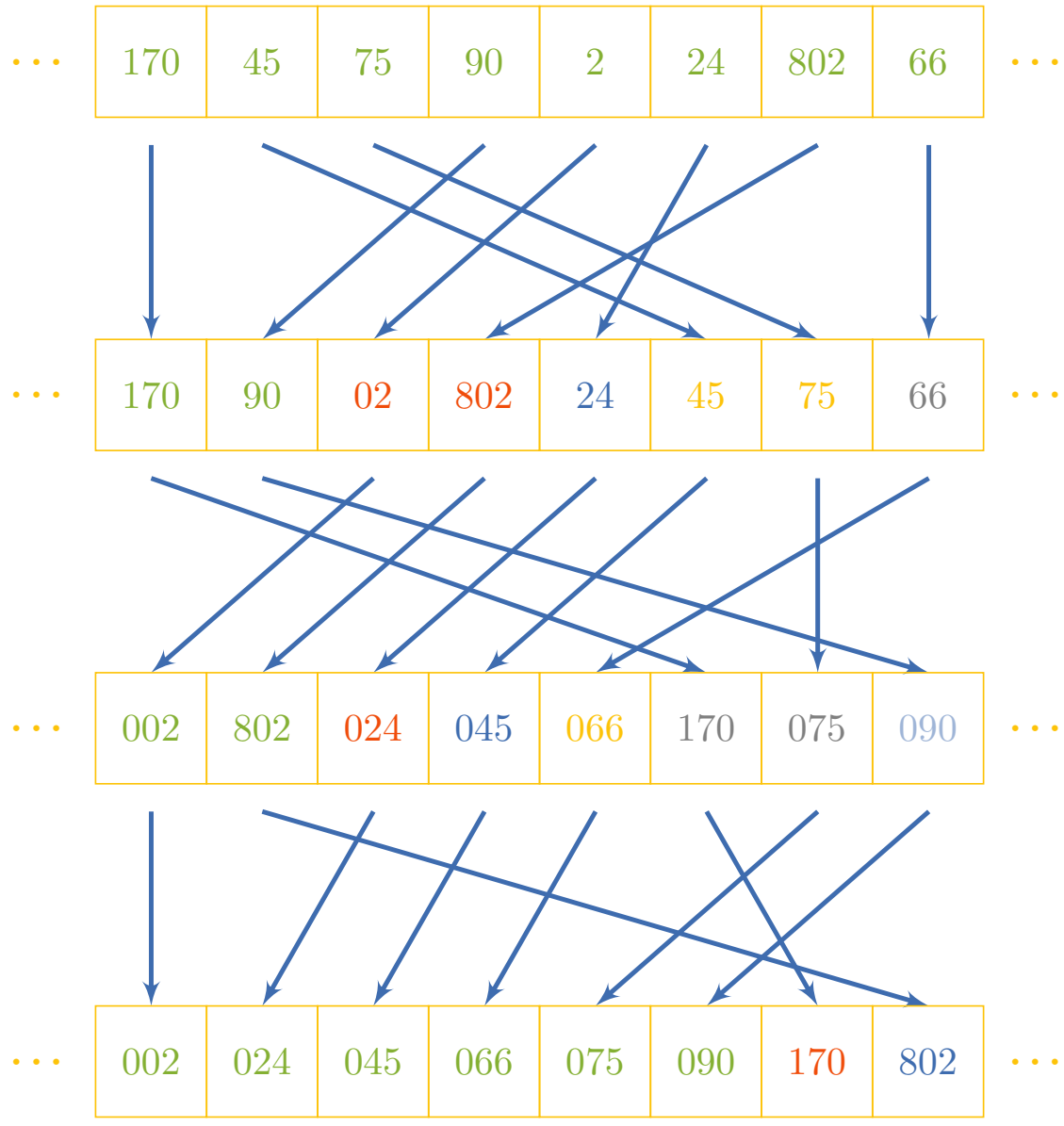
Réduction



Scan



Trie


170 45 75 90 2 24 802 66
170 90 02 802 24 45 75 66
002 802 024 045 066 170 075 090
002 024 045 066 075 090 170 802

Synchronisation

- Les threads peuvent accéder aux résultats des autres en utilisant la mémoire partagée ou la mémoire globale.
- Les threads peuvent donc travailler ensemble.
- A travers les patterns, il est nécessaire de faire particulièrement attention lorsqu'un thread a besoin de lire une donnée pendant qu'un autre l'écrit.
- Il est nécessaire de synchroniser les threads.

Exemple de barrière

```
int idx = threadIdx.x;
__shared__ int array[128];

array[idx] = threadIdx.x;
if(idx <127)
  array[idx] = array[idx+1];
```

Barrière (2)

```
__global__ void foo()
{
  __shared__ int s[1024];
  int i = threadIdx.x;

  s[i] = s[i-1];

  if(i%2) s[i] = s[i-1];

  s[i] = (s[i-1] -2* s[i] + s[i+1]) /2.0;
  printf("s[%d] = %f \n",i,s[i]);
}
```

Ecrire du code efficace

1. Maximiser l'intensité arithmétique

$$A_I = \frac{op}{memoire}$$

   - On peut maximiser le nombre d'opérations par thread
   - On peut minimiser le temps passer dans les accès mémoires : on déplace les données fréquemment utilisées dans les mémoires à accès rapides.
   - On réalise des accès coalescer.

2. Eviter les divergences.

Agenda

Quelques exemples