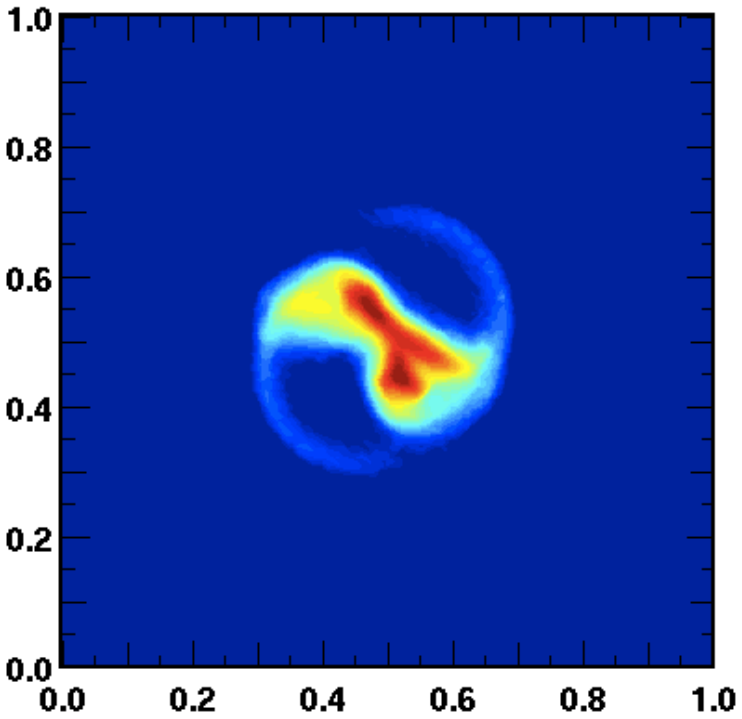
ACCELERATION GPU AVEC CUDA: APPLICATION AU CALCUL DE DIFFÉRENCE FINIE D'UN CODE PM



TRAVAUX PROPOSÉS PAR:

Dominique Aubert

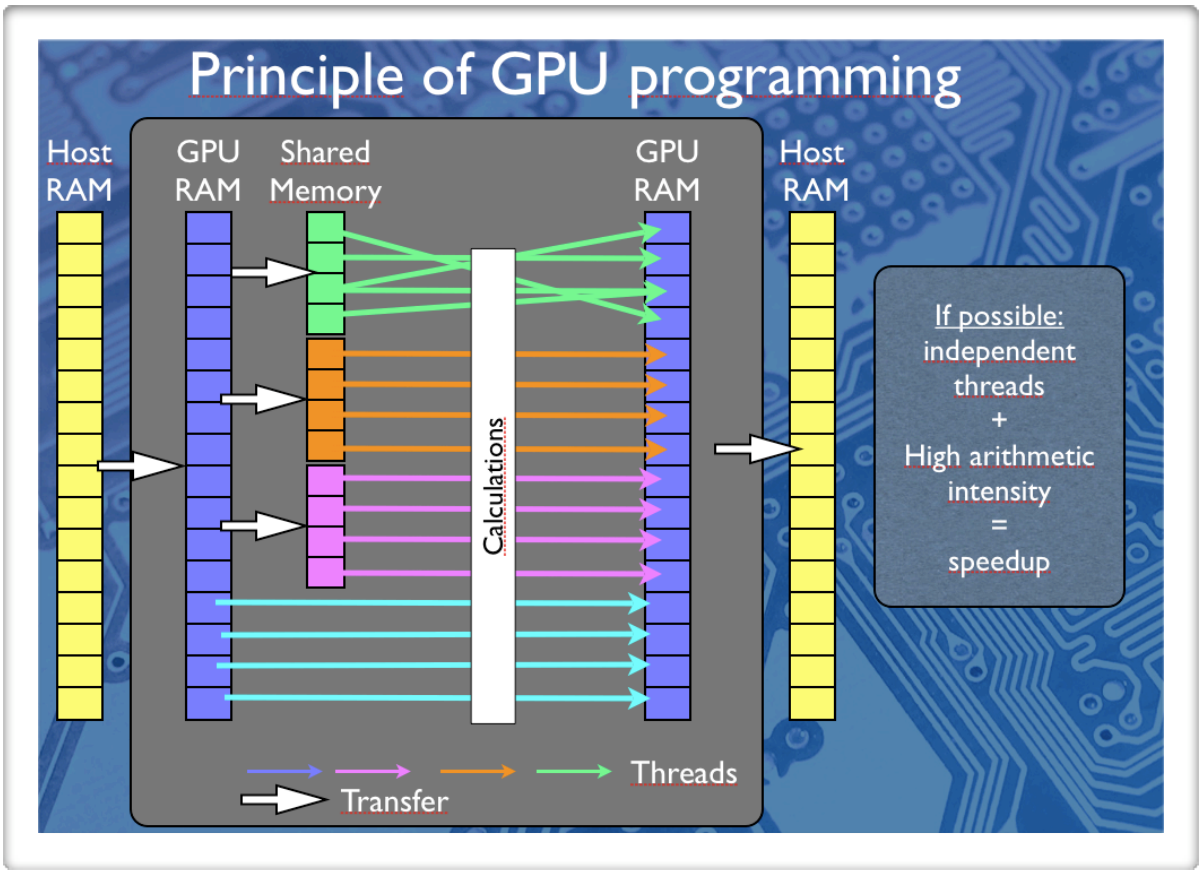
Généralités

Contexte & Objectifs

L'objectif du présent TP est de s'initer au langage de programmation CUDA, une extension du langage C permettant d'utiliser les cartes graphiques conçues par Nvidia à des fins de calculs scientifiques. Ces cartes (Graphical Processor Units, GPUs) sont capables en théorie de développer des puissances de calculs très supérieures aux CPUs standards (que ce soit en bande passante ou en nombre d'opérations flottantes).

Le TP vise à modifier l'une des étapes d'un intégrateur N-Corps de la famille Particle-Mesh pour profiter à plein des capacités de ces coprocesseurs graphiques.

Modèle de parallélisation/mémoire des GPUs



Un programme tournant sur GPU est appelé noyau ou kernel. Un kernel est invoqué par un fil d'éxécution ou thread. Un GPU peut gérer plusieurs threads de façon simultanée. Un groupe de threads est appelé block. Les blocks sont eux-même organisés en grille. Un block peut compter jusqu'à 512 threads. Une grille peut compter jusqu'à 65535 blocks.

L'objectif d'un calcul sur GPU est d'obtenir une haute intensité arithmétique sans collisions de threads. L'absence de collisions de threads permet aux calculs de se faire le plus indépendamment les uns des autres et par conséquent de limiter les communications. La haute intensité arithmétique permet de masquer les latences et les faibles taux de transferts.

Kit de Survie CUDA

Voici un résumé de ce qui est indispensable de savoir pour utiliser les cartes graphiques pour le calcul. Le schéma est toujours le suivant:

1. allouer l'espace mémoire sur la carte.
2. envoyer les données sur la carte
3. définir la configuration de la parallélisation (nombre de threads par blocks, nombre de blocks dans la grille)
4. lancer le/les kernel
5. si besoin, récupérer les données sur l'hôte.

Les commandes de bases pour ces étapes fondamentales sont:

L'allocation se fait de façon quasi identique aux pointeurs cpu standard. On déclare d'abord son pointeur, puis on enclenche l'allocation proprement dit avec cudaMalloc:
float devPtr;*
*cudaMalloc((void**)&devPtr, size);*
Notez la syntaxe particulière pour le pointeur dans le malloc.

Pour la copie vers la carte *cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice).* Pour rappatrier les données vers l'hôte la commande est identique sauf le dernier argument *cudaMemcpy(devPtr, data, size, cudaMemcpyDeviceToHost).* On peut également envisager une copie carte vers carte (pour dupliquer un tableau par exemple) avec la commande *: cudaMemcpy(devPtr, data, size, cudaMemcpyDeviceToDevice).* Si on développe l'exemple précédent, un copie vers la carte pourrait être :

```
float data[256];
int size = sizeof(data);
float* devPtr;
cudaMalloc((void**)&devPtr, size);
cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice);
```

Une configuration se déclare au travers de variables dont le type est prédéfini :

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
```

Vous noterez que l'on peut accéder aux tailles des blocs par les arguments toto.x/y/z pour les blocks qui peuvent être 3D et tata.x/y pour les grilles qui sont 2D au maximum.

On lance les kernels en explicitant la configuration parallèle dans l'appel à la fonction. Par exemple:

```
Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
```

L'exemple suivant permet d'illustrer toutes ces étapes. Il consiste en la simple somme de deux tableaux où chaque thread gère la somme entre deux éléments d'un tableau.

(0/0)	(0/1)	(0/2)	(0/3)	(0/4)	(1/0)	(1/1)	(2/2)	(3/3)	(4/4)	(2/0)	(2/1)	(2/2)	(2/3)	(2/4)

Chacun des éléments des deux tableaux (ici un seul est montré) sont chargés par chaque thread numéroté (block/# du thread au sein du block).

Exemple de Code CUDA

emacs@maggie.u-strasbg.fr

```
int main()
{
  int i;
  float *a, *b, *c;
  a=(float *)(calloc(NELEM,sizeof(float)));
  b=(float *)(calloc(NELEM,sizeof(float)));
  c=(float *)(calloc(NELEM,sizeof(float)));

  // Remplissage aléatoire des deux tableaux;
  srand(42);
  for(i=0;i<NELEM;i++)
    {
      a[i]=(float)(rand())/(float)RAND_MAX;
      b[i]=(float)(rand())/(float)RAND_MAX;
      c[i]=a[i]+b[i];
    }

  // creation des tableaux GPU correspondants

  float *da, *db, *dc;
  cudaMalloc((void**)&da,sizeof(float)*NELEM);
  cudaMalloc((void**)&db,sizeof(float)*NELEM);
  cudaMalloc((void**)&dc,sizeof(float)*NELEM);

  // envoi des données sur GPU
  cudaMemcpy(da,a,sizeof(float)*NELEM,cudaMemcpyHostToDevice);
  cudaMemcpy(db,b,sizeof(float)*NELEM,cudaMemcpyHostToDevice);

  // CONFIGURATION PARALLELE 1D

  dim3 dimBlock(NTHREADS);
  dim3 dimGrid(NELEM/NTHREADS);

  //  carte_somme<<<dimGrid,dimBlock>>>(da,db,dc);
  carte_somme_shared<<<dimGrid,dimBlock>>>(da,db,dc);

  // rappatriement des resultats (dc -> a)
  cudaMemcpy(a,dc,sizeof(float)*NELEM,cudaMemcpyDeviceToHost);


  // CONFIGURATION PARALLELE 2D

  dim3 dimBlock2D(NTHREADS_X,NTHREADS_Y);
  dim3 dimGrid2D(NELEM/(NTHREADS_X*NTHREADS_Y));

  carte_somme_2D<<<dimGrid2D,dimBlock2D>>>(da,db,dc);

  // rappatriement des resultats (dc -> b)
  cudaMemcpy(b,dc,sizeof(float)*NELEM,cudaMemcpyDeviceToHost);


  // Check
  for(i=0;i<10;i++) printf("corg= %f cgpu=%f cgpu2D=%f\n",c[i],a[i],b[i]);
  puts("...");
  for(i=NELEM-10;i<NELEM;i++) printf("corg=%f cgpu=%f cgpu2D=%f\n",c[i],a[i],b[i]);

  // destruction des tableaux GPU
  cudaFree(da);
  cudaFree(db);
  cudaFree(dc);
}
```

-:** somme.cu [(C Abbrev)]--L98--C1--Bot

Le programme fait la somme de deux tableaux 1D a et b et stocke le résultat dans C. Les valeurs sont tirées au hasard. NELEM est defini en variable globale

Les contreparties GPUs des tableaux a,b,c sont definies et alloués ainsi. A ce stade la mémoire est réservée sur la carte et assignée aux 3 pointeurs da,db,dc

cudaMemcpy est la commande de transfert.

cudaMemcpy(destination, origine, nombre d'octets, sens (ici CPU->GPU)

après transfert da et db ont le même contenu sur la carte que a & b.

dimBlock contient l'indexation des threads au sein d'un block. Ici les threads sont indéxés par un seul indice, de valeur NTHREADS-1 au maximum. **dimGrid** contient l'indexation des blocks au sein de la grille. Ici l'indexation est à nouveau 1D, avec un indice max de NELEM/NTHREADS-1.

Ces deux variables sont passées au noyau, qui lancera en tout (NTHREADS)*(NELEM/THREADS) fils d'éxécution, i.e. NELEM fils. Dans ce cas précis, chaque thread va additionner un element de tableau de a avec un element de b et le stocker dans C. On a bien NELEM tâches à faire en tout.

On récupère les resultats sur l'hôte avec un memcpy dans a.

Comme précédemment, sauf que cette fois les threads sont indéxés avec deux indices courant de 0 à NTHREADS_X-1 et 0 à NTHREADS_Y-1. Les blocks restent indexés à 1D.

On récupère les résultats dans b.

Comparaison

On libère la mémoire du GPU.

Voici les kernels correspondants.

File Edit Options Buffers Tools C Help

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>


#define NELEM 128
#define NTHREADS 64
#define NTHREADS_X 4
#define NTHREADS_Y 16

__global__ void carte_somme(float *da, float *db, float *dc)
{
  int idloc=threadIdx.x+blockIdx.x*blockDim.x; // construction de l'indice absolu
  dc[idloc]=da[idloc]+db[idloc];
}

__global__ void carte_somme_2D(float *da, float *db, float *dc)
{
  int idloc=threadIdx.x+threadIdx.y*blockDim.x+blockIdx.x*blockDim.x*blockDim.y;// construction de l'indice absolu
  dc[idloc]=da[idloc]+db[idloc];
}

__global__ void carte_somme_shared(float *da, float *db, float *dc)
{
  int idloc=threadIdx.x+blockIdx.x*blockDim.x;
  __shared__ float sa[NTHREADS],sb[NTHREADS]; // tableaux partagés par les threads

  sa[threadIdx.x]=da[idloc];
  sb[threadIdx.x]=db[idloc];
  __syncthreads();

  dc[idloc]=sa[threadIdx.x]+sb[threadIdx.x];
}
```

Les kernels sont exécutés par chaque thread, qui peuvent retrouver leur positions absolues à l'aide des variables threadIdx.x, threadIdx.y, threadIdx.z et blockIdx.x, blockIdx.y (on rappelle que les threads peuvent être indexés par 3 indices tandis que les blocks sont limités à 2). Les variables blockDim.x,y,z et gridDim.x,y permettent quant à elles de connaître les dimensions des blocks et de la grille. Notez comment dans les kernels suivants ils servent à construire des indices absolus, afin que chaque thread sache quel element il doit traiter.

Les pointeurs da,db, dc pointent sur la mémoire générale de la carte, dite globale. C'est le plus grand pool de mémoire accessible au GPU (768 Mo pour une 8800 GTX).

Le kernel carte_somme_shared fait usage de la mémoire shared. Cette mémoire est visible de tous les threads d'un même bloc (et invisible aux autres). Elle est limité à 16 Ko mais est extrêmement rapide d'accès comparée à la mémoire globale de la carte.

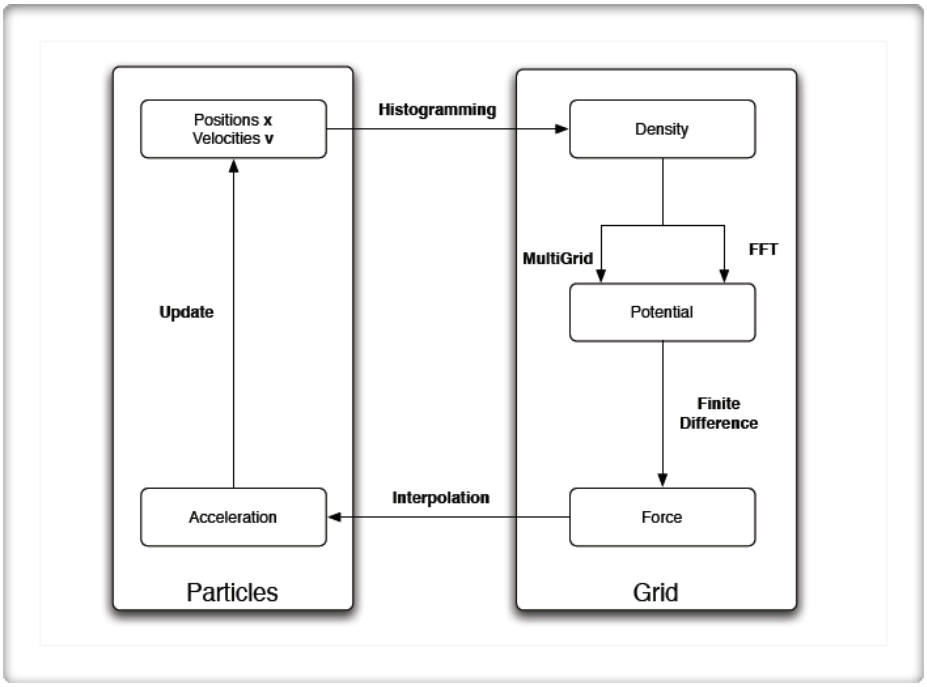
Ici, chaque thread remplit d'abord deux tableaux shared avec les élément dont il a la charge. Ces tableaux ont la taille du nombre de threads. Notez le __syncthreads qui synchronise tous les threads d'un block : il faut que tous les threads aient rempli leur tableau avant de pouvoir poursuivre.

Le résultat de la somme est directement envoyée en mémoire globale.

Travaux Pratiques

Vous avez à votre disposotion un intégrateur N-Corps Particle-Mesh (PM). Les étapes sont résumée ci-dessous et le PM qui vous est fourni réalisent l'ensemble de ces étapes (en utilisant des méthodes simples et non optimisées).

L'objectif du TP est d'accélérer la partie différence finie + mise à jour des vitesses des particules. Le code de départ effectue cette étape complètement sur le CPU, votre but étant de porter ces opérations sur la carte graphique en essayant de maximiser les gains des temps de calculs.



Principe et conventions

L'entrée dont vous disposez est le potentiel P[i][j][k] calculé en tous les points de la grille 128x128x128. La force exercée en chaque point de la grille est donnée par le gradient, approximé par:

$$fx[i][j][k]=(P[i+1][j][k]-P[i-1][j][k])/(2\ dx)\ ;$$

$$fy[i][j][k]=(P[i][j+1][k]-P[i][j-1][k])/(2\ dx)$$

$$fz[i][j][k]=(P[i][j][k+1]-P[i][j][k-1])/(2\ dx)$$

Le PM suit la convention ci-dessous, conforme à la disposition des tableaux en mémoire :

$$A[i][j][k] <=> A[i\ x\ NCELL\ x\ NCELL + j\ xNCELL + k]$$

Les conditions aux limites sont périodiques.

La vitesse d'une particule m est alors mise à jour simplement par:

$$vx[m]=vx[m]+fx[data[m]]\ DT$$

où data[m] est l'indice de la cellule où se trouve la particule m. Ce tableau est déjà calculé dans le PM.

Prise en main

1. Compilez le code pm0.cu : **nvcc -lcutil -lcufft -o pm0 pm0.cu**

2. Exécutez-le : **./pm0 -f exp128.a.bin** Cette commande indique au PM d'aller chercher les conditions intiales dans le fichier exp128.a.bin . Ces conditions initiales sont celles d'un disque exponentiel très instable. Un très forte barre doit y apparaître, qui détruit finalement le disque. Vous devez voir des sorties de ce type :

****

Time= 3.80e+00 Npdt= 380 dt=1.000000e-02

(millisec) Fx:44.130997 Fy:40.359993 Fz:41.067001 | | | Vx:47.304005 Vy:46.835007 Vz:47.912003 | | | Total:267.609009

(millisec) Fx:44.126999 Fy:40.304001 Fz:41.062012 | | | Vx:47.192997 Vy:46.809998 Vz:47.871002 | | | Total:267.367004

(millisec) Fx:44.125999 Fy:40.299004 Fz:41.067001 | | | Vx:47.220997 Vy:46.843994 Vz:47.921005 | | | Total:267.477997

3. Exécutez le code Yorick readout.i. Vous devez voir apparaître l'image début de forte spirale correspondant à la dernière sortie du code. Essayez de comprendre comment fonctionne le petit script yorick.

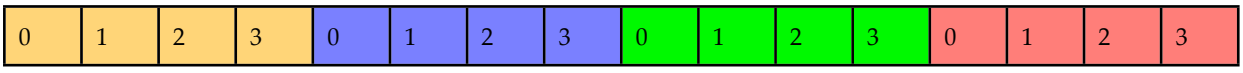
4. Parcourez le code pm0.cu et tâchez de retrouver les calculs du champ de forces dans chaque cellule ainsi que la mise à jour des vitesses des particules (routine carte_force_updatevel). Tâchez également de retrouver les commandes effectuant le transfert de données entre la carte et l'hôte. Notez qu'il existe 4 routines vides (carte_forcex,y,z et carte_updtevel) que vous devrez compléter par la suite.

Parallélisation de la mise à jour des vitesses

1. Faites une copie de pm0.cu sous le nom de pm1.cu

2. On cherche à paralléliser l'étape de mise à jour des vitesses. Pour chaque cellule, vous disposez du champ de force et pour chaque particule, vous disposez de l'indice de sa cellule (stockée dans data sur le CPU et stockée dans d_data sur le GPU). Ecrivez un kernel qui assigne à chaque thread le calcul de la vitesse d'une particule selon une direction. Notez bien que les vitesses mise à jour seront déjà présentes sur la carte à la fin du calcul, auquel cas le transfert final entre CPU et GPU n'aura plus lieu d'être. De même, on n'utilisera pas de mémoire partagée. On pourra utiliser la configuration parallèle suivante où chaque case représente la vitesse (selon x par ex.) d'une particule et où chaque couleur représente un block de threads. Le numéro indique le numéro du thread au sein du block. Par conséquent l'adresse absolue d'une case est donnée par

```
threadIdx.x+blockIdx.x*blockDim.x
```



Ici la taille d'un block vaut 4, mais libre à vous de choisir une taille plus importante. Utilisez le protoype de carte_updatevel déjà fourni, puis faites appel à cette routine dans carte_force_updatevel.

3. Vérifiez que vous retrouvez le résultat original et notez les temps d'éxecutions.

Parallélisation naïve du calcul des forces

1. Faites une copie sous le nom de pm2.cu

2. Comme précédemment, transposez le calcul des forces sur GPU, sans faire appel à de la mémoire partagée. Faites en sorte que plus aucun transfert ne soit nécessaire entre l'hôte et la carte. Pour cela, vous écrirez 3 noyaux différents : carte_forcex, carte_forcey, carte_forcez, où chaque thread calcule la valeur de la force(x,y,z) pour une cellule. Vous utiliserez ces trois noyaux dans carte_force_updatevel. On construira les blocks façons à ce que les threads d'un même block soient le long de la direction de differentiation. Par exemple le schéma suivant est le schéma de parallélisation (un block = une couleur) pour une differentiation horizontale. Dans le block bleu, le thread 1 a besoin du potentiel en (bleu,2) et en (bleu,0) pour pouvoir faire leur différence.

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

Le schéma suivant est celui d'une différentation verticale. Le thread (bleu,1) a besoin comme précédemment de (bleu,2) et (bleu,0):

3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0

3. On notera que les calculs des forces dans les trois directions font appel à des requêtes en mémoire de géométries différentes. En quoi cela impacte-t-il les performances ?

4. Vérifiez que vous obtenez le résultat attendu et notez les performances.

Optimisation de ForceZ

Le calcul de la force selon la direction z implique des requêtes mémoires coalescentes, c'est-à-dire que les threads accèdent à des zones mémoires contigues, d'où une meilleure performance que dans les autres directions. Nous allons utiliser la mémoire partagée pour améliorer encore ses performances.

1. Faites une copie pm3.cu

2. Modifiez le kernel carte_forcez de sorte que chaque thread remplisse d'abord un tableau 1D shared de valeurs du potentiel, puis fasse usage uniquement de ce tableau pour calculer la force associée.

3. Vérifez les résultats et notez le gain en temps d'exécution.

Optimisation 1 de Forcex et Forcey

Comme pour ForceZ, passez les valeurs du potentiel en mémoire partagée au lieu d'aller les chercher directement en mémoire globale. Sauvegarder le résultat dans pm3bis.cu, vérifiez sa validité et notez les gains en performances.

Optimisation 2 de Forcex et Forcey

La coalescence des accès mémoire est critique dans l'obtention de performances optimales. Considérez le schéma suivant pour la configuration des threads, pour une differentiation verticale . On cherche à stocker dans chacune des cases de la ligne du milieu la différence entre la ligne du haut et celle du bas.

1. Etape 1 : les threads stockent dans une variable locale les valeurs de la ligne du haut

1	2	3	4	5	6

2. Etape 2 : les threads stockent dans une variable locale les valeurs de la ligne du bas

1	2	3	4	5	6

3. Etape 3 : les threads stockent la différence des deux dans la ligne du milieu

1	2	3	4	5	6

Toutes ces étapes sont coalescentes. Mettez cette procédure en place pour ForceX et ForceY, vérifiez que vous obtenez le bon résultat et notez les gains en performance. Vous sauverez le résultat dans pm4.cu.