Chapitre 5 : OpenCL

5.1 Comment faire la transposition des éléments d'une matrice b[y][x] = a[x][y] tout en évitant les accès mémoire non consécutifs?

On peut copier la matrice de la mémoire globale vers la mémoire locale en variant le second indice en premier, transposer la matrice par une fonction kernel avec les arguments en mémoire locale, et recopier le résultat vers la mémoire globale en variant encore le second indice en premier. Dans certains tests, cette manière de faire est environ 10 fois plus rapide que de transposer à partir de la mémoire globale.

5.2 Suggérez différentes optimisations pour la fonction OpenCL suivante qui effectue la multiplication matricielle Y=AX où X et Y sont des vecteurs et A est une matrice formée uniquement d'éléments non nuls dans certaines diagonales. Le nombre de ces diagonales est diags et la position de départ (dans la première rangée) de ces diagonales est fournie dans offsets. La position est -(n-1) pour la diagonale qui ne contient que l'élément de première colonne dernière rangée, 0 pour la diagonale qui va de (0,0) à (n-1,n-1), et (n-1) pour la diagonale qui ne contient que le dernier élément de la première rangée.

La matrice A est stockée de manière compacte, une rangée par diagonale non nulle. Pour accéder les éléments de la rangée 0, il faut prendre l'élément 0 de chaque diagonale, donc les éléments de la colonne 0 dans la matrice compacte contenant une diagonale par rangée.

Un work item est le calcul d'un élément dans le vecteur de résultat.

```
__kernel
void dia_spmv(__global float *A, __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
  int row = get_global_id(0);
  float accumulator = 0;
  for(int diag = 0; diag < diags; diag++) {
    int col = row + offsets[diag];
    if ((col >= 0) && (col < rows)) {
      float m = A[diag*rows + row];
      float v = x[col];
      accumulator += m * v;
    }
  }
  y[row] = accumulator;
```

```
}
```

En premier lieu, le calcul de l'indice de A est simplifié et chaque rangée de A est alignée à l'aide d'espace ajouté à la fin de chaque rangée. Le nouveau paramètre pitch_A tient compte de l'espace ajouté. Ceci procure un gain d'environ 10%.

```
__kernel

void dia_spmv(__global float *A, __const int pitch_A,
   __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
  int row = get_global_id(0);
  float accumulator = 0;
  __global float* matrix_offset = A + row;
  for(int diag = 0; diag < diags; diag++) {
    int col = row + offsets[diag];
    if ((col >= 0) && (col < rows)) {
      float m = *matrix_offset;
      float v = x[col];
      accumulator += m * v;
    }
    matrix_offset += pitch_A;
  }
  y[row] = accumulator;
}
```

Les données dans offsets sont lues au complet dans chaque work item à partir de la mémoire globale. Il est avantageux de copier ces données en mémoire locale et de les réutiliser par tous les work item dans le même groupe. La performance est augmentée de plus de 50%

```
__kernel
void dia_spmv(__global float *A, __const int pitch_A,
   __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
  int local_id = get_local_id(0);
  int offset_id = local_id;
  while ((offset_id < 256) && (offset_id < diags)) {
    l_offsets[offset_id] = offsets[offset_id];
    offset_id = offset_id + get_local_size(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  int row = (int)get_global_id(0);
```

```
  float accumulator = 0;
  __global float* matrix_offset = A + row;
  for(int diag = 0; diag < diags; diag++) {
    int col = row + l_offsets[diag];
    if ((col >= 0) && (col < rows)) {
      float m = *matrix_offset;
      float v = x[col];
      accumulator += m * v;
    }
    matrix_offset += pitch_A;
  }
  y[row] = accumulator;
}
```

Finalement, des opérations vectorielles sont utilisées, ce qui procure un gain additionnel d'environ 15%.

```
__kernel
void dia_spmv(__global float *A, __const int pitch_A,
   __const int rows,
              __const int diags, __global int *offsets,
              __global float *x, __global float *y) {
  __local int l_offsets[256];
  int local_id = get_local_id(0);
  int offset_id = local_id;
  while ((offset_id < 256) && (offset_id < diags)) {
    l_offsets[offset_id] = offsets[offset_id];
    offset_id = offset_id + get_local_size(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);
  int row = get_global_id(0) * 4;
  float4 accumulator = 0;
  __global float* matrix_offset = A + row;
  for(int diag = 0; diag < diags; diag++) {
    int col = row + l_offsets[diag];
    float4 m = vload4(0, matrix_offset);
    float4 v;
    if ((col >= 0) && (col < rows - 4)) {
      v = vload4(0, x + col);
    } else {
      int4 id = col + (int4)(0, 1, 2, 3);
      int4 in_bounds = (id >= 0) && (id < rows);
      v.x = in_bounds.x ? x[id.x] : 0;
      v.y = in_bounds.y ? x[id.y] : 0;
      v.z = in_bounds.z ? x[id.z] : 0;
```

```
      v.w = in_bounds.w ? x[id.w] : 0;
    }
    accumulator += m * v;
    matrix_offset += pitch_A;
  }
  vstore4(accumulator, 0, row + y);
}
```