POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2021)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Lundi le 20 décembre 2021

HEURE: 13h30 à 16h00

DUREE: 2H30

NOTE: Aucune documentation permise sauf un aide-memoire, préparé par l'étudiant, qui consiste en une feuille de format lettre manuscrite recto-verso, calculatrice non programmable permise

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) Une matrice carrée `in`, dont la taille `width` est un multiple de 64, est fournie en entrée pour un calcul en OpenCL avec la fonction `tp`. La matrice de sortie `out` a la même taille que `in`. Le code pour initialiser la matrice `in` dans le programme principal est montré, de même que le code pour imprimer une partie du contenu de la matrice `out`, après l'appel de la fonction `tp` en OpenCL. La taille globale du calcul est de `width` pour chacune des deux dimensions, et la taille locale (groupe) est de 64 pour chacune des deux dimensions. Que donnera l'impression du résultat? **(2 points)**

```
// Programme principal en C
...
  for(i = 0; i < width; i++)
    for(j = 0; j < width; j++) in[i][j] = 2 * i + j;
...
// Appel du kernel tp
...
  for(i = 0; i < 3; i++) {
    for(j = 0; j < 3; j++) printf("%g ", out[i][j]);
    printf("\n");
  }
...
// Kernel en OpenCL
__kernel void tp(__global float *in, __global float *out, int width)
{
  int x = get_global_id(0), y = get_global_id(1), index;
  __local tile[64][64];

  tile[get_local_id(1)][get_local_id(0)] = in[y * width + x];
  barrier(CLK_LOCAL_MEM_FENCE);
  index = (x/64 * 64 + get_local_id(1)) * width + y/64 * 64 + get_local_id(0);
  out[index] = tile[get_local_id(0)][get_local_id(1)];
}
```

Le contenu initial de la matrice `in` est:

```
0 1 2 3 4 5 6 ...
2 3 4 5 6 7 8 ...
4 5 6 7 8 9 10 ...
6 7 8 9 10 11 12...
```

La fonction tp fait une transposition de matrice en recopiant d'abord les éléments dans une matrice locale de 64 par 64 et ensuite en écrivant le résultat transposé dans la matrice `out`. Le contenu imprimé pour `out` sera donc:

```
0 2 4
1 3 5
2 4 6
```

b) Deux alternatives, `Somme_A` et `Somme_B`, sont proposées pour une fonction. Deux différences importantes existent entre les deux, ligne 3 versus ligne 13, et lignes 6 et 8 versus lignes 16 et 17. Pour chacune des deux différences, dites quelle version (A ou B) est la plus efficace. Expliquez. **(2 points)**

```
1  __kernel void Somme_A(__global const float * entree,
2      __global float * somme, int chunk, int nb_rows)
```

```
3  { int p_somme = 0, i = get_global_id(0), j = get_global_id(1);
4
5    for(k = 0; k < chunk; k++) {
6      p_somme += entree[i + (j + k) * nb_rows];
7    }
8    atomic_add(*somme, p_somme);
9  }
10
11 __kernel void Somme_B(__global const float * entree,
12     __global float * somme, int chunk, int nb_rows)
13 { int value, i = get_global_id(1), j = get_global_id(0);
14
15   for(k = 0; k < chunk; k++) {
16     value = entree[i + (j + k) * nb_rows];
17     atomic_add(*somme, value);
18   }
19 }
```

Pour la première différence, la version A est la plus efficace. En effet, avec la version A, i est associé à global_id(0) et les coeurs adjacents d'un processeur SIMD obtiennent des valeurs consécutives pour global_id(0). Ainsi, dans la version A, l'indice calculé pour `entree` *donnera des cases consécutives en mémoire pour les coeurs adjacents. En effet, le matériel du GPU combine les accès simultanés à des cases consécutives en mémoire, par les coeurs adjacents d'un processeur SIMD, en un seul accès à un bloc de mémoire globale, ce qui est beaucoup plus efficace. Avec la version B, c'est y qui variera d'un coeur adjacent à l'autre, et ainsi l'indice calculé dans* `entree` *donnera des cases distantes de* `nb_rows` *mots pour les coeurs adjacents. Pour la seconde différence, c'est aussi la version A qui est la plus efficace, puisqu'on ne fait qu'une seule addition atomique, opération beaucoup plus coûteuse qu'une addition dans une variable privée.*

c) Les nouvelles architectures de GPU utilisent une mémoire virtuelle partagée entre le CPU et le GPU. En quoi est-ce que cela change les interactions entre le CPU et le GPU? Quelles nouvelles possibilités est-ce que cela ouvre pour les programmes qui s'exécutent sur ces GPU? **(1 point)**

Avec la mémoire virtuelle partagée entre les CPU et les GPU, les copies de données ne sont plus requises entre les deux avant et après l'exécution de calculs sur le GPU. Il est aussi maintenant possible d'utiliser des structures de données plus flexibles, par exemple avec des pointeurs, puisque ceux-ci sont utilisables autant par les programmes sur le GPU que sur le CPU, la mémoire étant partagée. En outre, il est même possible d'avoir un programme sur le CPU et un sur le GPU qui travaillent de manière concurrente sur les mêmes données, en se synchronisant avec des opérations atomiques.

Question 2 (5 points)

a) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez une sortie possible produite par ce programme? **(2 points)**

```
int main (int argc, char *argv[])
{
  int size, rank, i, in[4], o1[4], o2[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  for(i = 0; i < 4; i++) { in[i] = i * i + rank * i; o1[i] = o2[i] = 0; }
  MPI_Allgather(in + rank, 1, MPI_INT, o1, 1, MPI_INT, MPI_COMM_WORLD);
  MPI_Alltoall(in, 1, MPI_INT, o2, 1, MPI_INT, MPI_COMM_WORLD);
```

```
  printf("%d: in={%d, %d, %d, %d};\n", rank, in[0], in[1], in[2], in[3]);
  printf("%d: o1={%d, %d, %d, %d};\n", rank, o1[0], o1[1], o1[2], o1[3]);
  printf("%d: o2={%d, %d, %d, %d};\n", rank, o2[0], o2[1], o2[2], o2[3]);
  MPI_Finalize();
}
```

```
1: in={0, 2, 6, 12};
1: o1={0, 2, 8, 18};
1: o2={1, 2, 3, 4};
2: in={0, 3, 8, 15};
2: o1={0, 2, 8, 18};
2: o2={4, 6, 8, 10};
3: in={0, 4, 10, 18};
3: o1={0, 2, 8, 18};
3: o2={9, 12, 15, 18};
0: in={0, 1, 4, 9};
0: o1={0, 2, 8, 18};
0: o2={0, 0, 0, 0};
```

b) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez une sortie possible produite par ce programme? **(2 points)**

```
int main (int argc, char *argv[])
{
  int i, rank, size, chunk, start, end;
  float sum = 0, res;
  int v[] = {19, 16, 12, 8, 17, 14, 11, 5, 13, 6, 18, 15, 3, 10, 19, 9};
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  chunk = sizeof(v) / sizeof(int) / size;
  start = rank * chunk;
  end = start + chunk;
  for(i = start; i < end; i++) sum += v[i];
  sum = sum / chunk;
  printf("Node %d: %g\n", rank, sum);
  MPI_Reduce(&sum,&res,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
  res = res / size;
  if(rank == 0) printf("Result: %g\n", res );
  MPI_Finalize();
}
```

```
Node 1: 11.75
Node 2: 13
Node 0: 13.75
Node 3: 10.25
Result: 12.1875
```

c) Quelle est la différence entre les fonctions MPI_Send et MPI_Isend? **(1 point)**

La fonction MPI_Send fonctionne de manière plus conventionnelle. Une fois que l'appel retourne, ses arguments ont été utilisés et peuvent être modifiés sans problème. L'envoi peut ou non être complété (les données peuvent ne pas encore être parvenues à destination, étant en transit dans des tampons dans le système). La fonction MPI_Isend

retourne immédiatement et l'envoi est fait de manière asynchrone. Il ne faut pas modifier le contenu des arguments qui ont été passés à l'appel, puisque leur contenu n'a peut-être pas encore été copié ou envoyé. Il faut vérifier l'état de la requête associée à cet appel avant de réutiliser les arguments de l'appel.

Question 3 (5 points)

a) Vous compilez un programme avec les options de profilage de `gprof` et l'exécutez ensuite. Ce programme ne comporte qu'un seul thread, qui est interrompu par `gprof` à chaque 1ms afin de noter la fonction dans laquelle se trouve le compteur de programme. De plus, à chaque appel, `gprof` note le décompte du nombre d'appels de chaque provenance (fonction appelante). L'information donnée dans le tableau qui suit est ainsi produite pendant l'exécution. Calculez pour chaque fonction le temps passé dans la fonction elle-même (self) et le temps passé dans la fonction elle-même plus ses enfants (self+childs), comme le ferait l'outil `gprof`. **(2 points)**

Fonction	échantillons	appels	appelants
main	2	1	
f1	4	2	main: 2
f2	8	5	main: 3, f1: 2
f3	6	4	f2: 4
f4	12	6	f1: 4, f2: 2
f5	3	9	f1: 3, f3: 6
f6	5	5	f2: 5

Le temps passé dans chaque fonction elle-même (self) est donné par le nombre des échantillons, chacun comptant pour 1ms. Pour le temps passé dans chaque fonction incluant les fonctions appelées (self+childs), il faut imputer le temps des fonctions appelées aux fonctions appelantes au prorata des appels. On commence par les fonctions feuilles (f4, f5 et f6) où le temps self+childs est le temps de chaque fonction elle-même (self). Les fonctions f1, f3 et f6 n'ont qu'un seul appelant et leur temps self+childs pourra donc être imputé directement à leur seul appelant. Cependant, le temps de f5, 3ms doit être imputé 3/9 à f1 (1ms) et 6/9 à f3 (2ms). Le temps de f4, 12ms, doit être imputé 4/6 à f1 (8ms) et 2/6 à f2 (4ms). Finalement, le temps de f2, 25ms, doit être imputé 3/5 à main (15ms) et 2/5 à f1 (10ms). Le résultat complet est fourni dans le tableau qui suit.

Fonction	self	self+childs
main	2ms	2 + 25x3/5 (f2) + 23 = 40ms
f1	4ms	4 + 25x2/5 (f2) + 12x4/6 (f4) + 3x3/9 (f5) = 23ms
f2	8ms	8 + 8 (f3) + 12x2/6 (f4) + 5 (f6) = 25ms
f3	6ms	6 + 3x6/9 (f5) = 8ms
f4	12ms	12ms
f5	3ms	3ms
f6	5ms	5ms

b) Un outil de mesure de performance, comme CPU Profile, échantillonne le contenu de la pile d'appels (la fonction courante et celles dans le chemin d'appel) à un intervalle de temps régulier de 1ms. Lorsqu'un échantillon se répète, ce qui arrive souvent, un facteur multiplicatif est ajouté, plutôt que de sauver le même chemin d'appel plusieurs fois dans le fichier de sortie. Voici les échantillons récoltés, chacun suivi du nombre de fois qu'il a été rencontré. Calculez le temps passé dans chaque fonction elle-même (self) et dans chaque fonction et ses appels (self+childs). **(2 points)**

```
main: 2
main, f1: 4
main, f1, f2: 6
main, f2: 2
main, f2, f3: 3
```

```
main, f2, f3, f5: 4
main, f1, f4: 5
main, f1, f2, f6: 7
```

L'exécution complète a pris 33ms. Pour chaque fonction, on compte le nombre de fois qu'elle arrive en début de pile pour le temps self. Pour le temps self + child, on compte le nombre de fois que chaque fonction est présente dans la pile d'appel. Dans chaque cas, on tient bien sûr compte du facteur multiplicatif.

```
fonction   self    self+child

main         2ms    2+4+6+2+3+4+5+7 = 33ms
f1           4ms    4+6+5+7 = 22ms
f2    6+2 = 8ms     6+2+3+4+7 = 22ms
f3           3ms    3+4 = 7ms
f4           5ms    5ms
f5           4ms    4ms
f6           7ms    7ms
```

c) Quel serait un bon outil à suggérer pour trouver le problème dans un logiciel pour chacun des deux cas suivants: i) un problème de fautes de cache est suspecté, et ii) un résultat aléatoire est produit et on suspecte une course entre les accès à certaines variables par plus d'un thread. **(1 point)**

Un outil comme perf ou oprofile, qui se base sur les compteurs de performance de l'unité centrale de traitement, peut efficacement, avec un surcoût faible, obtenir des échantillons de où se produisent les fautes de cache. Ils produisent ainsi un profil qui montre les fonctions, et même les lignes de code, où surviennent un grand nombre de fautes de cache. Les problèmes de course sont parmi les plus difficiles à diagnostiquer. Un outil comme Valgrind Helgrind permet de les détecter lorsqu'elles se produisent, mais avec un surcoût énorme. L'outil Thread Sanitizer est ainsi souvent le plus intéressant, puisqu'il fait un travail de détection à peu près équivalent, avec un surcoût important mais moins énorme que Helgrind.

Question 4 (5 points)

a) On vous propose d'utiliser la méthode itérative de Jacobi pour résoudre un système d'équations linéaires $b = Ax$. L'équation itérative pour trouver la valeur du vecteur x à la prochaine itération est donnée par $x_{k+1} = D^{-1}(b - Ox_k)$, où D est la matrice avec la diagonale de A et O une matrice en soustrayant la diagonale de A. Si la matrice A a une dimension de n par n, combien d'additions, de soustractions, de multiplications et de divisions seront requises pour faire le calcul d'une itération, en ne comptant pas les opérations sur les valeurs qui sont nécessairement nulles et qui peuvent donc être sautées. **(2 points)**

Le calcul de O (mettre à 0 la diagonale de A) et D^{-1} (n divisions) est très simple et n'est fait qu'une seule fois, pas à chaque itération. Il n'est donc pas mentionné dans le total calculé ici. Pour la première opération Ox_k c'est une multiplication matrice vecteur qui demande $n-1$ (on peut sauter les éléments nuls de la diagonale) multiplications et additions pour chacun des n éléments du vecteur résultant. On peut argumenter que pour additionner $n-1$ nombres, il suffit de $n-2$ additions. (Toutefois, cela complexifierait le code, puisqu'il faudrait initialiser la somme avec le premier élément et sommer sur les suivants, plutôt que d'initialiser la somme à 0 et boucler sur tous les éléments). Nous avons donc $n^2 - n$ multiplications et additions. La seconde étape $(b - Ox_k)$ est une soustraction de deux vecteurs et demande n soustractions. Finalement, la dernière étape est de multiplier le vecteur résultant par la matrice diagonale D^{-1}. Ceci demande une multiplication pour chacun des éléments du vecteur (car un seul élément est non nul dans chaque rangée de la matrice diagonale), donc n multiplications. Le total est donc n^2 multiplications, $n^2 - n$ additions, et n soustractions.

b) Le programme OpenMP suivant s'exécute. Donnez une sortie possible produite par ce programme? **(2 points)**

```
void fonction_x (int *x, int n)
{ int sp[8];
  omp_set_num_threads(8);
  #pragma omp parallel
  { int i, chunk = n / 8, rank = omp_get_thread_num();
    int s = 0, start = chunk * rank, end = start + chunk;
    for(i = start; i < end; i++) { s += x[i]; x[i] = s; }
    sp[rank] = s;
    #pragma omp barrier
    #pragma omp single
    for(i = 0, s = 0; i < 8; i++) { s += sp[i]; sp[i] = s; }
    if(rank > 0 ) for(i = start; i < end; i++) x[i] += sp[rank - 1];
  }
}

int main(int argc, char **argv)
{ int v[] = {9, 6, 2, 8, 7, 4, 1, 5, 3, 6, 8, 5, 3, 0, 9, 9};
  int i, n = sizeof(v) / sizeof(int);
  fonction_x(v, n);
  for(i = 0; i < n; i++) printf("%d ", v[i]);
  printf("\n");
}

9 15 17 25 32 36 37 42 45 51 59 64 67 67 76 85
```

c) Dans le cadre du cours, trois paradigmes de programmation ont été utilisés, OpenMP, OpenCL et MPI. Pour chacun, expliquez dans quelle situation il est utile. Sont-ils mutuellement exclusifs ou peuvent-ils être utilisés de concert pour solutionner un problème? **(1 point)**

OpenMP est typiquement utilisé pour programmer un ordinateur multi-coeur à mémoire partagée, par exemple pour partager l'exécution d'une boucle d'un programme entre plusieurs threads parallèles qui roulent sur autant de coeurs du même noeud. OpenCL est typiquement utilisé pour programmer les GPU. La décomposition du travail en très nombreux work item convient bien aux très nombreux coeurs des GPU, groupés en processeurs SIMD. Finalement, MPI permet de faire communiquer de très nombreux noeuds parallèles afin de résoudre un gros calul. Ces trois langages et librairies ont été prévus pour fonctionner ensemble. Plusieurs programmes exploitent donc ainsi à l'aide de ces trois technologies un très grand nombre de noeuds, chacun avec plusieurs threads sur autant de coeurs, et avec en plus une partie des calculs délégués au GPU. La plupart des ordinateurs au sommet du Top 500 contiennent ainsi un grand nombre de noeuds avec chacun plusieurs coeurs parallèles et quelques GPU.

Le professeur: Michel Dagenais