POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2022)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Dimanche le 18 décembre 2022

HEURE: 9h30 à 12h00

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) Considérez le code OpenCL suivant contenant deux fonctions kernel, `kernel1` et `kernel2`, qui sont exécutées à la suite l'une de l'autre. On fixe pour les deux la taille du problème global = 1024, group = 64 et n = 1024. Donnez le contenu des 8 premiers éléments du vecteur `x` après l'exécution de `kernel1`, puis leur valeur après l'exécution de `kernel2`. La constante `M_PI_F` est la valeur de pi (3,14...) et on néglige ici les erreurs d'arrondi de ces calculs numériques à virgule flottante. **(2 points)**

```
__kernel void kernel1(__global float* x, const size_t n) {
    const int id = get_global_id(0);
    x[id] = sin(M_PI_F * (float) id / 2.);
}

__kernel void kernel2(__global float* x, const size_t n) {
    const int window = 4;
    const int id = get_global_id(0);

    float sum = 0.;
    for(int j = 0; j < window; ++j) {
       sum += x[(id + j) % n];
    }
    x[id] = sum;
}
```

Le premier kernel remplit le vecteur x avec sin(0), sin(pi/2), sin(pi), sin(3pi/2), ... On obtient alors `x = {0, 1, 0, -1, 0, 1, 0, -1, ...}`.

Le second kernel fait une somme glissante par 4 des éléments de x. Comme le sinus s'annule, on obtient `x = {0, 0, 0, 0, 0, 0, 0, 0, ...}`.

b) En mathématiques appliquées, l'interpolation est une technique permettant d'estimer des valeurs après discrétisation (comme par exemple un échantillonnage) et est trivialement parallélisable. Le kernel `interpolate` implémente une interpolation bilinéaire (linéaire en deux dimensions) pour doubler la taille d'une image carrée. La taille du problème, en deux dimensions, est égale à la taille de l'image. Proposez une modification pour améliorer sensiblement les performances du kernel. **(2 points)**

```
__kernel void interpolate(__global unsigned char* in,
                          __global unsigned char* out) {
    const int i = get_global_id(0);
    const int j = get_global_id(1);

    const int dim = get_global_size(0) * get_global_size(1);
    const int dim_x = get_global_size(0);

    const int base     = j +     i * dim_x;
    const int base_out = j * 2 + i * dim_x * 4;

    unsigned char x00 = in[base];
```

```
    unsigned char x01 = in[(base + 1) % dim];
    unsigned char x10 = in[(base + dim_x) % dim];
    unsigned char x11 = in[(base + 1 + dim_x) % dim];

    out[base_out] = x00;
    out[base_out + 1] = (x00 + x01) / 2;
    out[base_out + dim_x * 2] = (x00 + x10) / 2;
    out[base_out + 1 + dim_x * 2] = (x00 + x01 + x10 + x11) / 4;
}
```

Pour une utilisation efficace des transferts entre la mémoire globale et le cache, il faut favoriser le regroupement des accès entre les multiples coeurs d'un même processeur SIMD. Pour cela, il faut que les items adjacents selon get_global_id(0) accèdent des mots adjacents en mémoire. Ici, deux items qui varient de 1 selon cette dimension auront une différence de 1 en i et seront distants de dim_x en entrée et de 4 dim_x en sortie. Il serait préférable d'intervertir les indices, de manière à ce que j soit selon get_global_id(0). Ainsi, deux coeurs adjacents varieront de 1 en j et accèderont des mots adjacents en mémoire en entrée. L'accélération obtenue sur un test donne plus de 40x.

```
    const int j = get_global_id(0);
    const int i = get_global_id(1);
```

c) Dans le langage OpenCL, la fonction `barrier()` peut prendre soit `CLK_LOCAL_MEM_FENCE` soit `CLK_GLOBAL_MEM_FENCE` comme argument. Que fait cette fonction? Quelle est la différence de comportement entre ces deux valeurs possibles d'argument? **(1 point)**

Cette fonction est une barrière de synchronisation (rendez-vous) pour tous les `work item` du `work group`, i.e. on attend que tous les `work item` du `work group` soient rendus à ce point avant de les laisser poursuivre. L'argument CLK_LOCAL_MEM_FENCE introduit une barrière mémoire pour synchroniser les mises à jour du `work item` versus la mémoire locale du `work group`. L'argument CLK_GLOBAL_MEM_FENCE introduit plutôt une barrière mémoire pour synchroniser les mises à jour du `work item` versus la mémoire globale. Plusieurs font l'erreur de penser que GLOBAL s'applique au rendez-vous, plutôt qu'à la barrière mémoire, et introduit une barrière de synchronisation globale (i.e. attendre après tous les `work item` de tous les `work group`). Cela ne peut pas être, car les `work group` sont souvent beaucoup plus nombreux que ce qui peut être exécuté en même temps sur un GPU, et ils sont gardés en queue et exécutés au fur et à mesure que d'autres sont terminés et que de la place devient disponible sur les processeurs SIMD. Pour cette raison, la nouvelle spécification OpenCL dit maintenant "the built-in function barrier has been renamed work_group_barrier; for backward compatibility, barrier is also supported", ce qui devrait enlever cette confusion.

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 i, rank, size, prev, next, val, newval;
```

```
  MPI_Status s[2]; MPI_Request r[2];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if(rank == 0) prev = size - 1; else prev = rank - 1;
  if(rank == (size - 1)) next = 0; else next = rank + 1;
  val = rank * rank;
  printf("Rank %d, prev %d, next %d, size %d\n",rank,prev,next,size);

  for(i = 0; i < 5; i++) {
    MPI_Irecv(&newval, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &r[0]);
    MPI_Isend(&val, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &r[1]);
    MPI_Waitall(2, r, s);
    val = newval;
  }
  printf("Rank %d, value %d\n", rank, val);
  MPI_Finalize();
}
```

```
Rank 0, prev 3, next 1, size 4
Rank 0, value 9
Rank 1, prev 0, next 2, size 4
Rank 1, value 0
Rank 2, prev 1, next 3, size 4
Rank 2, value 1
Rank 3, prev 2, next 0, size 4
Rank 3, value 4
```

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 size, rank, i, in[4], o1[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+rank*rank; o1[i] = 0; }
  MPI_Allgather(in + rank, 1, MPI_INT, o1, 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]);
  MPI_Finalize();
}
```

```
1: in={1, 1, 3, 7};
```

```
3: in={9, 7, 7, 9};
3: o1={0, 1, 4, 9};
0: in={0, 1, 4, 9};
0: o1={0, 1, 4, 9};
2: in={4, 3, 4, 7};
2: o1={0, 1, 4, 9};
1: o1={0, 1, 4, 9};
```

c) On veut créer un type qui correspond à `Struct Element` pour envoyer avec MPI. Pour ce faire, il faut préciser la position des champs de cette structure à la fonction `MPI_Type_create_struct`. Votre partenaire propose le vecteur `{0, 8, 24}`. Est-ce correct? Comment expliquez-vous ces valeurs 0, 8 et 24? Est-ce qu'il y a une manière plus simple de spécifier la position des champs qui ne demande pas de tels calculs au programmeur? **(1 point)**

```
struct Element { unsigned n; double v[2]; char c; } st;

MPI_Aint ElementFieldPosition[3] = { 0, 8, 24 };
```

Le premier champ commence au début de la structure (0). Le champ double doit être aligné sur un multiple de 8 octets (64 bits) et commence donc à 8 octets. Le troisième champ est après les deux mots de 8 octets de v[2] et se trouve donc à 24 octets. Il serait plus facile de demander au compilateur de calculer la position des champs avec le mot clé `offsetof`.

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 fil d'exécution, 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	12	1	
f1	11	4	main: 4
f2	5	6	main: 4, f1: 2
f3	4	7	f1: 3, f2: 4
f4	9	5	f1: 3, f2: 2
f5	2	8	f2: 4, f4: 4
f6	3	4	f3: 4

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 (f5 et f6) où le temps self+childs est le temps de chaque

fonction elle-même (self). La fonction f6 n'a qu'un seul appelant et son temps self+childs pourra donc être imputé directement à f3. Le temps de f5 (2ms) doit être imputé 4/8 à f2 (1ms) et 4/8 à f4 (1ms). Le temps de f4 (10ms) doit être imputé 3/5 à f1 (6ms) et 2/5 à f2 (4ms). Le temps de f3 (7ms) doit être imputé 3/7 à f1 (3ms) et 4/7 à f2 (4ms). Le temps de f2 (14ms) doit être imputé 4/6 à main (9.33ms) et 2/6 à f1 (4.66ms). Finalement, le temps de f1 (24.66ms) doit être imputé entièrement à main. Le résultat complet est fourni dans le tableau qui suit.

```
Fonction  self  self+childs
main      12ms  12ms + 9.33ms (f2) + 24.66ms (f1) = 46ms
f1        11ms  11ms + 6ms (f4) + 3ms (f3) + 4.66ms (f2) = 24.66ms
f2         5ms  5ms + 1ms (f5) + 4ms (f4) + 4ms (f3) = 14ms
f3         4ms  4ms + 3ms (f6) = 7ms
f4         9ms  9ms + 1ms (f5) = 10ms
f5         2ms  2ms
f6         3ms  3ms
```

b) Un outil de mesure comme Heap Profile mémorise le contenu de la pile d'appels (la fonction courante et celles dans le chemin d'appel), ainsi que le nombre d'octets alloués, à chaque appel pour allouer de la mémoire (e.g. malloc et new). Voici les échantillons récoltés. Calculez le nombre d'octets alloués dans chaque fonction elle-même (self) et dans chaque fonction elle-même plus ses enfants (self+childs). **(2 points)**

```
(main), 64
(main, f1, f2, f4, f6), 128
(main, f2), 32
(main, f2, f3), 16
(main, f2, f3, f5), 28
(main, f3, f5, f6), 48
(main, f1, f6), 8
(main, f4, f5, f6), 36
```

L'exécution complète a alloué 360 octets. Pour le compte d'octets self de chaque fonction, on somme le nombre d'octets, pour tous les appels d'allocation où elle arrive en début de pile. Pour le compte d'octets self + child de chaque fonction, on somme le nombre d'octets, pour tous les appels d'allocation où elle est présente dans la pile d'appels.

```
fonction   self    self+child

main       64   64+128+32+16+28+48+8+36=360
f1          0   128+8=136
f2         32   128+32+16+28=204
f3         16   16+28+48=92
f4          0   128+36=164
f5         28   28+48+36=112
f6         128+48+8+36=220  128+48+8+36=220
```

c) Expliquez comment un outil comme Address Sanitizer peut détecter diverses erreurs d'accès en mémoire. Quelle instrumentation utilise-t-il, quelle information mémorise-t-il, et quelles vérifications effectue-t-il? **(1 point)**

L'outil Address Sanitizer instrumente les appels aux fonctions d'allocation et de libération de mémoire pour mémoriser l'information sur les cases mémoire allouées versus celles qui ne le sont pas. Il vérifie aussi que les cases mémoire libérées étaient effectivement préalablement allouées. Il instrumente toutes les écritures en mémoire afin de mémoriser les cases qui ont été initialisées (écrites) et vérifier que les écritures se font sur des cases allouées. Finalement, il instrumente les lectures en mémoire pour vérifier que les cases lues sont accessibles et ont été initialisées.

Question 4 (5 points)

a) Vous proposez de réaliser un programme parallèle pour solutionner un système d'équations linéaires, $b = Ax$, par la méthode itérative de Jacobi. Soit A la matrice des coefficients, b le vecteur des valeurs, et x le vecteur des variables, on peut définir D la matrice diagonale de A, et O la matrice en enlevant la diagonale de A ($O = A - D$). La solution à l'itération $k+1$ est donnée en fonction de la solution à l'itération k par $x_{k+1} = D^{-1}(b - Ox_k)$. On vous demande de démontrer comment fonctionne la méthode itérative de Jacobi en l'appliquant manuellement sur les données suivantes, avec le vecteur x de solution initiale (i.e. pour x_0) tel que spécifié. Donnez la valeur obtenue pour le vecteur x après la première itération (x_1). **(2 points)**

```
A = np.array([
    [1, 2, 4],
    [3, 5, 7],
    [5,10,15]
])
b = np.array([7, 17, 27])
x = np.array([1, 1, 1])
```

Il suffit de créer les matrices D et O et d'appliquer l'équation.

```
[0, 2, 4] [1]   [ 6]  | [ 7]   [ 6]   [ 1]
[3, 0, 7] [1] = [10]  | [17] - [10] = [ 7]
[5,10, 0] [1]   [15]  | [27]   [15]   [12]

[1   0    0]   [ 1]   [  1]
[0 1/5    0] x [ 7] = [1.4]
[0   0 1/15]   [12]   [0.8]
```

Si on continue sur plusieurs itérations, on réalise que cet algorithme ne converge malheureusement pas pour cette matrice.

b) Lorsqu'on fait un tri par fusion conventionnel de n éléments sur un seul coeur, on trie dans une première étape les éléments 2 par 2, on fusionne ensuite 2 par 2 les groupes de 2 en des groupes de 4, puis 2 par 2 les groupes de 4 en groupes de 8... jusqu'à n'avoir qu'un seul groupe fusionné qui contient les n éléments. Pour réaliser cela, il faut $log_2(n)$ étapes, et à chaque étape on traite tous les n éléments une

fois. Le nombre total de traitements d'éléments est donc $nlog_2(n)$ et le nombre d'unités de temps requis est aussi de l'ordre de $nlog_2(n)$. Si on fait maintenant ce même tri par fusion, mais en parallèle sur 32 coeurs, combien d'unités de temps seront requises? Quel sera le taux d'utilisation global de nos 32 coeurs? Le taux d'utilisation global pour m coeurs est la somme du nombre de coeurs utilisés (parmi les m) pour chaque unité de temps, sur le temps total multiplié par m. Par exemple, si on utilise 8 coeurs sur 8 pendant 2 unités de temps et 5 coeurs sur 8 pendant 3 unités de temps, le taux d'utilisation global serait de $(8coeurs \times 2cycles + 5coeurs \times 3cycles)/(8coeurs \times (3+2)cycles) = 0.775$. **(2 points)**

Avec le tri par fusion en parallèle, à la première étape, les m coeurs sont actifs, chacun traitant n/m éléments en $n/m \times log_2(n/m)$ étapes pour le tri initial. Avec $m = 32$, cela donne $n/32 \times log_2(n/32) = n/32 \times (log_2(n) - 5) = n/32log_2(n) - 5n/32$. A l'étape suivante, $m/2$ sont actifs, chacun traitant $2n/m$ éléments en une étape de fusion. A la dernière étape, un seul coeur est actif sur n éléments. Le nombre d'unités de temps est donc de $n/32log_2(n/32) - 5n/32 + n/16 + n/8 + n/4 + n/2 + n = (nlog_2(n) + 57n)/32$. Le taux d'utilisation est de $(32 \times (n/32log_2(n) - 5n/32) + 16 \times n/16 + 8 \times n/8 + 4 \times n/4 + 2 \times n/2 + 1 \times n)/((nlog_2(n) + 57n)/32 \times 32) = (nlog_2(n) - 5n + n + n + n + n + n)/(nlog_2(n) + 57n) = log_2(n)/(log_2(n) + 57)$.

c) On vous demande de mettre une matrice M à la puissance 150. Expliquez comment vous pourriez effectuer ce calcul efficacement en parallèle sur un ordinateur multi-coeurs à mémoire partagée. Expliquez brièvement comment vous décomposeriez le problème en opérations sur chaque coeur, pas le détail de la programmation. **(1 point)**

Plutôt que de multiplier 150 fois la matrice M, on peut composer les opérations en calculant $M^2 = M \times M$, $M^4 = M^2 \times M^2$.... Ainsi, on pourrait calculer pour la matrice M les puissances de 2, 4, 8, 16, 32, 64 et 128 en 7 multiplications de matrices. On pourrait finalement obtenir $M^{150} = M^{128} \times M^{16} \times M^4 \times M^2$ avec 3 autres multiplications, pour un total de 10. La multiplication de matrices se calcule bien en parallèle sur m processeurs en décomposant la matrice de n par n en m sous-matrices de $n/\sqrt{m}$ par $n/\sqrt{m}$, pour un facteur d'accélération qui s'approche de m. Chaque fil d'exécution sur un coeur s'occupe alors de calculer les n^2/m éléments de sa sous-matrice en multipliant ensemble les n éléments de chacune des n rangées et colonnes correspondantes, ce qui demande de l'ordre de n^3/m opérations.

Le professeur: Michel Dagenais