POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

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

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;
}
```

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;
}
```

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)**

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();
}
```

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();
}
```

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 };
```

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

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
```

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)**

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])
```

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)**

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)**

Le professeur: Michel Dagenais