ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2014)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Mercredi le 17 décembre 2014

HEURE: 13h30 à 16h00

DUREE: 2H30

NOTE: Toute documentation permise, calculatrice non programmable permise

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) Convertissez le programme suivant en code efficace écrit en assembleur Vectoriel MIPS. **(3 points)**

```
double a[N], b[N], c[N], d[N];
int i;

for(i = 0 ; i < N ; i++)
  if((a[i] * b[i]) != 0) d[i] = (a[i] + b[i]) * c[i];
```

```
; charger les valeurs et adresses dans des registres
        LD      R1, N
        LD      R2, a
        LD      R3, b
        LD      R4, c
        LD      R5, d
        L.D     F0, #0
; nombre d'opérations vectorielles de 64 éléments
        DSRL    R6, R1, #6
; nombre d'opérations restant à la fin
        ANDI    R7, R1, #63
        BEQZ    R6, end ; moins de 64 éléments
loop:   LV      V1, R2
        LV      V2, R3
        LV      V3, R4
; d est chargé car l'ancienne valeur reste si a[i]*b[i] == 0
        LV      V4, R5
        MULVV.D V5, V1, V2
        SNEVS   V5, F0
        ADDVV.D V5, V1, V2
        MULVV.D V4, V3, V5
        CVM
        SV      V4, R5
        ADDI    R2, R2, #64 * 8
        ADDI    R3, R3, #64 * 8
        ADDI    R4, R4, #64 * 8
        ADDI    R5, R5, #64 * 8
        ADDI    R6, R6, #-1
        BNEZ    R6, loop
end:    MTC1    VLR, R7
        LV      V1, R2
        LV      V2, R3
        LV      V3, R4
```

```
LV      V4, R5
MULVV.D V5, V1, V2
SNEVS   V5, F0
ADDVV.D V5, V1, V2
MULVV.D V4, V3, V5
CVM
SV      V4, R5
```

b) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: i) addition et soustraction point flottant, ii) multiplication point flottant, iii) division point flottant, iv) opérations entières, v) opérations logiques, et vi) rangement et chargement. Ces unités en pipeline permettent de produire une nouvelle valeur à chaque cycle. Les instructions scalaires (e.g. L.D) prennent un cycle d'exécution chacune. Dans ces unités en pipeline, les additions et soustractions ont une profondeur de pipeline de 7, les multiplications de 12, les divisions de 18 et les chargements et rangements de 10. Calculez le temps requis pour l'exécution de la séquence d'instructions suivante. **(1 point)**

```
L.D       F0, Ra
L.D       F1, Rb
LV        V1, Rc
MULVS.D   V1, V1, F0
LV        V2, Rd
MULVS.D   V2, V2, F1
ADDVV     V3, V1, V2
MULVS.D   V3, V3, F0
SV        V3, Re
```

Les deux premières instructions prennent 1 cycle chacune. Les deux instructions suivantes peuvent se chaîner en 10 + 12 + 64 = 86 cycles. On ne peut chaîner le LV qui suit car l'unité de load / store est déjà occupée. Les 3 instructions suivantes peuvent aussi se chaîner en 10 + 12 + 7 + 64 = 93 cycles. Il faut arrêter là car l'instruction MULVS.D qui suit requiert l'unité de multiplication déjà occupée par le MULVS.D précédent. Les deux dernières instructions font une dernière chaîne et prennent 12 + 10 + 64 = 86 cycles. Le total est donc 2 + 86 + 93 + 86 = 267 cycles.

c) Les GPU sont constitués de plusieurs processeurs SIMD, chacun étant constitué de plusieurs éléments simples de calcul (thread processor). Est-ce que différents processeurs SIMD peuvent travailler sur différents programmes ou fonctions? Est-ce que différents éléments simples de calcul d'un même processeur SIMD peuvent travailler sur différents programmes ou fonctions? Expliquez. **(1 point)**

Différents processeurs SIMD peuvent effectivement travailler sur différents programmes ou fonctions. Par contre, et c'est la définition même de l'acronyme SIMD, différents éléments de calcul d'un processeur SIMD travaillent tous sur la même instruction en même temps. En cas d'exécution conditionnelle (if) chaque élément de calcul passera par les deux branches de la condition mais ses opérations seront désactivées lorsqu'il sera dans la branche pour laquelle son élément de donnée ne satisfera pas la condition.

Question 2 (5 points)

a) Le programme suivant calcule PI par la méthode de Monte-Carlo. La fonction srand48 initialise le générateur de nombres aléatoires alors que la fonction drand48 fournit un nombre point flottant (double) entre 0 et 1. Ainsi, ce programme génère un point dans le carré -1 à 1 (longueur de 2), en x et y, et détermine si ce point est à l'intérieur du cercle de rayon 1, ce qui devrait être le cas dans une proportion de PI (aire du cercle) / 4 (aire totale du carré). En faisant la somme du nombre de fois où c'est à l'intérieur, divisé par le nombre d'essais, on doit approximer PI / 4 et donc PI en multipliant ce résultat par 4. On vous demande de convertir ce programme en fonction kernel OpenCL qui fait un travail équivalent. Vous pouvez supposer que les fonctions srand48 et drand48 et les points flottants *double* sont disponibles en OpenCL. Vous pouvez laisser certaines opérations simples être faites par le programme qui appelle cette fonction, par exemple le calcul final de pi. Expliquez quels sont les paramètres d'entrée et de sortie de votre fonction kernel et comment vous suggérez de grouper le travail en *work item* et en groupes. **(3 points)**

```
int i, inside, nb = 100000000;
double x, y, pi;
long sum_pi = 0;

srand48(0);
for(i = 0; i < nb; i++) {
  x = drand48() + drand48() - 1.0;
  y = drand48() + drand48() - 1.0;
  inside = sqrt(x*x + y*y) <= 1 ? 1 : 0;
  sum_pi += inside;
}
pi = (sum_pi * (double)4.0) / nb;
```

Il faut faire 100M calculs. Dans ce cas-ci, il n'y a pas de dépendance entre les différents calculs, il faut simplement faire une réduction à la fin pour grouper tous les résultats. On a donc intérêt à faire beaucoup de travail dans chaque work item, ce qui est plus efficace, tout en s'assurant d'avoir assez de work item pour bien occuper tous les éléments de calcul. Souvent, on recommande d'avoir plusieurs items par élément de calcul de sorte que, lorsqu'un item est bloqué par une faute de cache, un autre peut progresser. Une carte typique contient 1000 ou 2000 éléments de calcul. Avec 10000 work item, chacun sera bien occupé. On peut donc demander à chaque work item de faire 10000 itérations. Ainsi, le kernel ne requiert pas de donnée en entrée mais doit accumuler son résultat en sortie. Chacun des 10000 work item fait un atomic_add dans la variable local_sum_pi partagée par le groupe. Ensuite, le premier élément du groupe (get_local_id(0) == 0) ajoute cette valeur avec un atomic_add à la somme globale. Il ne restera plus au programme principal qu'à multiplier la somme globale par 4 et la diviser par 100 000 000.

```
/* A la toute fin du calcul, sum_pi devra être divisé par le total et m
```

```
__kernel void calcule_pi(__global long *sum_pi)
{
  int i, inside, nb = 10000;
  double x, y;
  long tmp_sum_pi = 0;
  __local local_sum_pi = 0;

  /* chaque thread doit avoir un seed différent sinon le calcul sera id
  srand48(get_global_id(0));
  for(i = 0; i < nb; i++) {
    x = drand48() + drand48() - 1.0;
    y = drand48() + drand48() - 1.0;
    inside = sqrt(x*x + y*y) <= 1 ? 1 : 0;
    tmp_sum_pi += inside;
  }
  atomic_add(local_sum_pi, tmp_sum_pi);
  barrier(CLK_LOCAL_MEM_FENCE);
  if(get_local_id(0) == 0) atomic_add(*sum_pi, local_sum_pi);
}
```

b) Dans la fonction OpenCL qui suit, on veut faire une copie locale du vecteur a, local_a, afin d'accélérer la suite. Cette copie est répartie entre les *work item* du groupe, de sorte que chaque item copie sa part du vecteur (1024 / local_size éléments). Une fois cette copie terminée, le gros des calculs, chaque calcul pouvant utiliser tous les éléments du vecteur local_a, doit commencer. Le chargé de laboratoire vous suggère qu'il faut ajouter un énoncé de synchronisation après la copie locale et avant les vrais calculs. Est-ce vrai? Quel énoncé devriez-vous mettre? Quel est son effet? Que serait le problème qui pourrait se produire sans cet énoncé de synchronisation? **(1 point)**

```
__kernel void compute(__global float *a, __global float *b) {
  int local_id = get_local_id(0);
  int local_size = get_local_size(0);
  __local float local_a[1024];

  /* Copie locale: chaque work item copie localement
     1024 / local_size éléments */
  for(i = local_id; i <= 1024; i += local_size) {
    local_a[i] = a[i];
  }

  /* Vrais calculs: une fois la copie de a complète,
     le calcul commence */
  ...
```

Puisque chaque élément de calcul copie une partie différente du vecteur, et que chacun peut avoir à accéder tous les éléments, il faut s'assurer que la copie est terminée dans tous les

éléments avant de continuer. Ceci peut être fait avec une barrière locale, (pour tous les éléments de calcul du même processeur SIMD), barrier(CLK_LOCAL_MEM_FENCE). Cette barrière s'assure aussi que la mémoire cache entre les différents éléments de calcul devienne cohérente en propageant les modifications. A défaut de mettre une telle barrière, certains éléments de calcul pourraient commencer leur calcul en lisant des valeurs de local_a qui n'ont pas encore été copiées ou pour lesquelles la copie n'a pas été propagée à l'élément de calcul.

c) Dans le travail pratique 2, vous avez comparé la performance de OpenMP et de OpenCL pour différents problèmes (images de différentes tailles). Pour quel genre de problème utiliseriez-vous plutôt OpenMP ou plutôt OpenCL? Donnez un exemple. **(1 point)**

OpenMP donne accès à un nombre plus limité de processeurs mais requiert relativement peu de temps de mise en marche ou de transfert de données. Habituellement, les thread ont déjà été créés et sont en attente de travail. Sur un GPU avec OpenCL, il faut copier les données vers la carte, démarrer le calcul, et récupérer les données. Le surcoût est donc plus important. Par contre, le GPU donne accès à des milliers d'éléments de calcul qui peuvent opérer en parallèle. Pour de grandes images, le gain en parallélisme d'utiliser le GPU sera donc particulièrement avantageux. En règle générale, plus le problème est facile à paralléliser à grande échelle et demande relativement peu de communication, plus le GPU avec OpenCL sera avantageux.

Question 3 (5 points)

a) La fonction suivante fait l'intégration numérique par la méthode trapézoïdale d'une fonction f. Ecrivez un programme MPI qui appelle cette fonction afin de réaliser cette intégration en parallèle sur un grand intervalle (qui sera séparé en petits intervalles à traiter sur chaque noeud). Le début et la fin de l'intervalle à traiter ainsi que l'incrément (step) vous seront fournis dans des variables globales auxquelles votre programme peut faire référence: `begin`, `end`, `step`. Votre programme doit imprimer le résultat final sur le noeud 0. **(2 points)**

```
float integre_trapeze(float begin, float end, float step) {
  float resultat = 0;
  float x;
  int i;

  resultat = (f(begin) + f(end))/2.0;
  x = begin;
  for (x = begin + step; x < end; x += step) {
    resultat += f(x);
  }
  resultat *= h;
  return resultat;
}
```

Il suffit de diviser l'intervalle en autant de sous-intervalles que nous avons de noeuds, et ensuite chaque noeud s'occupe de son sous-intervalle. Cette information peut être recalculée par chaque noeud et n'a pas à être communiquée. Finalement, il faut faire une réduction pour accumuler les intégrales calculées pour chaque sous-intervalle.

```
float begin = 0.0, end = 100.0, step = .00001;

int main (int argc, char *argv[])
{
  int size, rank, nb_interval;
  float interval_size, start_interval, result, global_result;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  interval_size = (end - begin) / size;
  start_interval = begin + rank * interval_size;

  result = integre_trapeze(start_interval, start_interval + interval_si

  MPI_Reduce(&result, &global_result, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COM

  if (rank == 0) { printf("Result = %f\n", global_result);
}
```

b) Le programme suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez le contenu des deux lignes (in et o1) imprimées sur chaque noeud? **(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 + 2 * 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();
}

0: in={0, 1, 4, 9};
0: o1={0, 4, 16, 36};
1: in={1, 4, 9, 16};
1: o1={0, 4, 16, 36};
2: in={4, 9, 16, 25};
2: o1={0, 4, 16, 36};
3: in={9, 16, 25, 36};
3: o1={0, 4, 16, 36};
```

c) Dans le cadre du travail pratique 3, à la question 3.4, vous avez tracé un graphique de l'accélération selon le nombre de processeurs, avec une courbe pour chaque taille d'image. Décrivez l'allure de ces différentes courbes. Est-ce que l'accélération en fonction du nombre de processeurs croit plus vite pour les petites ou les grandes images? Comment expliquez-vous cela? **(1 point)**

Etant donné le coût initial de démarrage, et de communication pour envoyer les données et recevoir les résultats, il est beaucoup plus efficace de traiter des grosses images. Pour de petites images, le facteur d'accélération sature assez rapidement lorsque le nombre de processeurs et noeuds augmente (e.g., 30 processeurs). Par contre, avec des plus grosses images, un facteur d'accélération intéressant est obtenu jusqu'à un peu plus d'une dizaine de noeuds (e.g., 128 processeurs).

Question 4 (5 points)

a) L'outil Helgrind de Valgrind permet de détecter les courses, un des problèmes les plus importants dans les programmes parallèles multithread. Donnez un exemple d'accès à une variable partagée par plus d'un thread en décrivant le cas où il y a une course et le cas où des primitives de synchronisation empêchent une course. Expliquez comment Helgrind peut distinguer les deux et détecter le cas où la course est présente. **(2 points)**

Lorsque plusieurs thread accèdent une même variable sans synchronisation, une trace des accès mémoire et des verrous montrera des accès à cette même case mémoire par deux thread sans qu'un verrou associé n'ait changé de propriétaire (relâchement par le premier thread et prise par le suivant qui veut accéder la variable). Supposons le cas d'une structure de donnée qui décrive les paramètres d'une imprimante. La fonction qui modifie cette structure de donnée prend un verrou, modifie quelques paramètres, et relâche le verrou. Si un autre thread accède à ces paramètres sans prendre de verrou et tombe en plein pendant que cette structure se fait modifier, il pourrait lire divers paramètres dont certains ont été modifiés et d'autres pas et se retrouver avec des valeurs incohérentes entre elles. S'il utilise un verrou pour ces accès, ceci ne pourra pas se produire. Dans le cas sans verrou, même si la lecture

survient à un moment où les valeurs ne sont pas en train d'être modifiées, la trace montrera néanmoins que la même case mémoire est accédée par un second thread sans qu'un verrou n'ait été pris. La détection peut donc se faire sans avoir à tomber juste au moment rare où la course cause une erreur. Helgrind vérifie aussi pour chaque variable partagée quels sont les verrous possédés par le thread qui accède la variable. Il prend l'intersection entre les verrous possédés au moment de l'accès par les différents threads. A la fin de l'exécution, l'intersection contient le verrou associé à cet élément de donnée. Si l'intersection est vide, un des thread a fait l'accès sans détenir le bon verrou.

b) L'outil gcov peut indiquer le nombre de fois que chaque bloc de code est exécuté. L'outil gprof fournit un profil du temps d'exécution pour chaque section du programme. Les deux peuvent donc donner une bonne indication des points chauds d'un programme. Quels sont les avantages de gprof par rapport à gcov pour évaluer le temps passé dans chaque section d'un programme? **(1 point)**

Pour étudier le temps passé dans chaque section d'un programme, gprof présente deux avantages. Premièrement, gprof impose un surcoût moins important puisqu'il demande un échantillon à chaque miliseconde, ce qui est presque négligeable, de même qu'une valeur à chaque entrée dans une fonction, ce qui est tout de même beaucoup moins coûteux qu'à chaque entrée dans un bloc linéaire comme gcov. Deuxièmement, l'échantillonnage est basé sur le temps consommé et est donc plus précis puisqu'il tient compte du coût relatif de chaque instruction, contrairement à gcov qui ne fait que tenir un décompte des instructions exécutées.

c) Les outils OProfile ou perf peuvent, tout comme gprof, fournir un profile du temps d'exécution d'un programme. Quels sont les avantages de ces programmes par rapport à gprof? En quoi sont-ils plus versatiles? **(1 point)**

Les outils comme OProfile et perf font de l'échantillonnage tout comme gprof mais sont plus versatiles car le temps (compteur de cycle) n'est qu'une des nombreuses métriques qu'ils peuvent employer. Ils peuvent ainsi produire un profil des fautes en cache, des blocages dans le pipeline et de nombreux autres paramètres.

d) La virtualisation permet, entre autres, de partager une machine physique entre plusieurs machines virtuelles. Ce partage est-il utile pour le calcul parallèle? En quoi les machines virtuelles peuvent-elles être utiles pour le calcul parallèle? **(1 point)**

Le calcul parallèle est généralement constitué de processus qui font un usage intensif du CPU. Il y a donc peu d'intérêt à consolider quelques machines sur une seule machine physique par le biais de la virtualisation; il n'y a pas de temps mort et donc peu d'économie à faire en récupérant du temps mort comme avec des serveurs souvent peu occupés. Par contre, la virtualisation permet aussi de découpler le logiciel du matériel et donc aide la portabilité. Avec la virtualisation, l'usager peut fournir une image configurée selon ses besoins (version du système d'exploitation et des librairies). Ceci peut présenter un avantage suffisant pour compenser la perte de performance associée à la couche de virtualisation.

Le professeur: Michel Dagenais