ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2016)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Jeudi le 22 décembre 2016

HEURE: 9h30 à 12h00

DUREE: 2H30

NOTE: Toute documentation permise, calculatrice non programmable permise

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) Un programme en C a été converti en assembleur Vectoriel MIPS. Le code source en C a été égaré. On vous demande de donner la section de programme en C la plus simple possible à laquelle pourrait correspondre le programme Vectoriel MIPS qui suit. **(2 points)**

```
        LD      R1, N
        LD      R2, a
        LD      R3, b
        LD      R4, c
        ANDI    R5, R1, #63
        DSRL    R6, R1, #6
        ADDI    R6, R6, #1
        MTC1    VLR, R5
loop:   LV      V1, R2
        LV      V2, R3
        SLTV.D  V1, V2
        ADDV.D  V3, V1, V2
        CVM
        SGEV.D  V1, V2
        MULV.D  V3, V1, V2
        CVM
        SV      R4, V3
        MTC1    VLR, #64
        ADDI    R2, R2, #64 * 8
        ADDI    R3, R3, #64 * 8
        ADDI    R4, R4, #64 * 8
        ADDI    R6, R6, #-1
        BNEZ    R6, loop
```

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

for(i = 0 ; i < N ; i++) {
  if(a[i] < b[i]) c[i] = a[i] + b[i];
  else c[i] = a[i] * b[i];
}
```

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 8, les multiplications de 16, les divisions de 24 et les chargements et rangements de 12. Calculez le temps requis pour l'exécution de la séquence d'instructions suivante. **(2 points)**

```
1   LV       V3, R3
2   LV       V4, R4
3   SUBV.D   V5, V2, V1
4   SUBV.D   V6, V3, V4
5   ADDV.D   V1, V5, V6
6   SUBV.D   V3, V5, V6
7   DIVVS.D  V2, V1, F1
8   DIVVS.D  V4, V3, F1
9   SV       V2, R5
10  SV       V4, R6
```

Les opérations vectorielles peuvent être chaînées lorsque qu'elles sont consécutives et utilisent toutes des unités différentes. C'est le cas des lignes 2 et 3, 6 et 7, et 8 et 9. Le total est de 576 cycles.

```
1   LV       V3, R3       ; 12 + 64
2   LV       V4, R4       ; 12 +
3   SUBV.D   V5, V2, V1   ; 8 + 64
4   SUBV.D   V6, V3, V4   ; 8 + 64
5   ADDV.D   V1, V5, V6   ; 8 + 64
6   SUBV.D   V3, V5, V6   ; 8 +
7   DIVVS.D  V2, V1, F1   ; 24 + 64
8   DIVVS.D  V4, V3, F1   ; 24 +
9   SV       V2, R5       ; 12 + 64
10  SV       V4, R6       ; 12 + 64
```

c) Plusieurs nouveaux systèmes avec CPU et GPU intégrés, de compagnies comme AMD et Intel, offrent une mémoire virtuelle partagée entre les CPU et GPU. Quel est l'impact de ce changement? Donnez au moins deux conséquences, nouvelles possibilités ou opérations qui ne sont plus nécessaires. **(1 point)**

Ceci a un impact important. Les copies ne sont plus requises entre la mémoire centrale et une mémoire séparée sur la carte graphique. La latence pour soumettre un calcul au GPU s'en trouve diminué et il est même possible de partager la mémoire pendant le calcul, par exemple avec des opérations atomiques sur des variables accédées simultanément par le programme sur le GPU et un programme sur le CPU. Autre conséquence, il est maintenant facile d'utiliser des pointeurs dans les structures de données car les adresses virtuelles sont partagées entre le CPU et le GPU.

Question 2 (5 points)

a) La fonction OpenCL suivante est exécutée pour 1023 *work items*. Le vecteur d'entiers `v` contient (1, 2, 3, 4, 5..., 1024). Quelle est la valeur des paramètres `s` et `d` à la fin de l'exécution de cette

fonction OpenCL. **(2 points)**

```
__kernel void sumo(__global long *s, __global long *d,
    __global const long *v)
{
  __local long local_s = 0;
  __local long local_d = 0;
  int i = get_global_id(0);
  atomic_add(local_s, v[i]);
  atomic_add(local_d, v[i+1] - v[i]);
  barrier(CLK_LOCAL_MEM_FENCE);
  if(get_local_id(0) == 0) {
    atomic_add(*s, local_s);
    atomic_add(*d, local_d);
  }
}
```

Ce programme calcule dans s la somme des éléments de 1 à 1023 dans le vecteur, ce qui donne 1023 éléments dont la valeur moyenne est de (1 + 1023) / 2 = 512, soit un total de $512 \times 1023 = 523776$*. Dans d on fait la somme des différences entre les éléments consécutifs. La différence est toujours de 1, le total est donc de* $1 \times 1023 = 1023$*.*

b) Une fonction OpenCL fait un traitement simple sur les éléments d'une matrice. On vous propose deux versions. Laquelle sera la plus performante? Pourquoi? On vous suggère de changer pour avoir chaque *work item* qui traite une rangée complète de la matrice. Quel impact cela aurait-il sur le nombre de *work item*? Selon quel critère doit-on choisir d'avoir plus de *work item* plus courts, ou moins de *work item* mais plus longs? **(2 points)**

```
__kernel void MatrixA(const __global float* input,
    uint matrix_width, uint matrix_height,
    __global float* output)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  int pos = j * matrix_width + i;
  if(input[pos] < 0) output[pos] = 0;
  else output[pos] = input[pos];
}

__kernel void MatrixB(const __global float* input,
    uint matrix_width, uint matrix_height,
    __global float* output)
{
  int j = get_global_id(0);
  int i = get_global_id(1);
```

```
  int pos = j * matrix_width + i;
  if(input[pos] < 0) output[pos] = 0;
  else output[pos] = input[pos];
}
```

Dans la version A, les work item consécutifs (prochaine valeur de get_global_id(0)) accèdent des cases mémoires consécutives. Ces accès consécutifs seront automatiquement regroupés et se feront donc beaucoup plus efficacement avec la mémoire globale. La version A sera donc significativement plus rapide. Pour optimiser un calcul, on cherche à avoir assez de work item pour utiliser tous les coeurs de calcul en parallèle (e.g., environ 8 thread par coeur de calcul pour être occupé même en cas de fautes de cache). Une fois ce nombre de work item atteint, il est préférable d'avoir des work item plus longs plutôt que plus nombreux, pour diminuer le temps requis pour transmettre et initialiser les work item (augmenter la fraction du temps utile, par rapport au temps de démarrage). Ici, chaque work item fait très peu de choses et il serait intéressant de lui en faire faire plus, à condition qu'il y ait assez de travail pour occuper tous les coeurs de calcul.

c) Sur les nouvelles architectures avec CPU et GPU intégrés, de compagnies comme AMD et Intel, on trouve maintenant des queues de commandes en mode usager (user-level queues) pour donner du travail au GPU. Quel est l'impact de cette nouvelle fonctionallité sur le coût d'exécution d'une commande, comme l'exécution d'une fonction (kernel en OpenCL)? **(1 point)**

Pour soumettre une nouvelle commande au GPU, il n'est plus nécessaire de faire un appel système. On sauve ainsi plusieurs centaines de ns. Ceci coupe d'autant le surcoût associé au lancement d'une commande et peut rendre efficace le fait de déléguer plus de travail au GPU, par exemple des courtes commandes qui autrement n'auraient pas été assez longues pour compenser le surcoût associé à leur lancement sur GPU.

Question 3 (5 points)

a) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Quelle est la sortie produite par ce programme à l'écran? **(2 points)**

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int i, j, rank, size, chunk, start, end, tmp, res;
  int v[] = {11, 9, 7, 5, 3, 1, 10, 8, 6, 4, 2, 0};
  MPI_Status status;
  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++) {
    for(j = i; j < end; j++) {
      if(v[j] < v[i]) {
        tmp = v[i]; v[i] = v[j]; v[j] = tmp;
      }
    }
  }

  printf("Node %d: %d\n", rank, v[start]);
  MPI_Reduce(v+start,&res,1,MPI_INT,MPI_MIN,0,MPI_COMM_WORLD);
  if(rank == 0) printf("Result: %d\n", res);
  MPI_Finalize();
}
```

```
Node 0: 7
Node 1: 1
Node 2: 6
Node 3: 0
Result: 0
```

b) Le programme suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez le contenu des trois lignes imprimées sur chaque noeud. **(2 points)**

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
  int size, rank, i, i1[4], i2[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++) i1[i] = rank * i + i*i;
  MPI_Alltoall(i1,1,MPI_INT,i2,1,MPI_INT,MPI_COMM_WORLD);
  for(i = 0; i < 4; i++) i2[i] = i2[i] * rank;
  MPI_Alltoall(i2,1,MPI_INT,o1,1,MPI_INT,MPI_COMM_WORLD);
  printf("%d: i1={%d,%d,%d,%d}\n",rank,i1[0],i1[1],i1[2],i1[3]);
```

```
  printf("%d: i2={%d,%d,%d,%d}\n",rank,i2[0],i2[1],i2[2],i2[3]);
  printf("%d: o1={%d,%d,%d,%d}\n",rank,o1[0],o1[1],o1[2],o1[3]);
  MPI_Finalize();
}
```

```
0: i1={0,1,4,9}
1: i1={0,2,6,12}
1: i2={1,2,3,4}
1: o1={0,2,12,36}
2: i1={0,3,8,15}
2: i2={8,12,16,20}
2: o1={0,3,16,45}
3: i1={0,4,10,18}
3: i2={27,36,45,54}
3: o1={0,4,20,54}
0: i2={0,0,0,0}
0: o1={0,1,8,27}
```

c) Expliquez la différence entre les fonctions MPI_Send, MPI_Bsend et MPI_Isend. **(1 point)**

La fonction Send bloque jusqu'à ce que la tampon d'envoi redevienne disponible, soit i) parce que le message a été envoyé ou ii) parce qu'il a été copié dans un autre tampon. La fonction Bsend prend une copie des données à envoyer et retourne le contrôle au programme, ce qui correspond au cas ii) de Send. La fonction Isend retourne immédiatement, même si le tampon n'est pas encore disponible (i.e. il est peut-être encore requis pour l'envoi); il faut vérifier l'état de la requête avant de le réutiliser, mais au moins en attendant on peut progresser sur autre chose.

Question 4 (5 points)

a) Vous compilez un programme avec les options de profilage de gprof et l'exécutez ensuite. L'information suivante est produite pendant l'exécution. Sur 12 échantillons du compteur de programme récoltés à intervalle régulier d'une seconde, 6 sont dans la fonction C, 3 dans D, 2 dans A et 1 dans B. Le nombre d'appels de chaque fonction (avec la provenance des appels entre parenthèses) est `main`: 1 (pas d'appelant), A:1 (main:1), B:2 (A:2), C:3 (A:3), D:6 (B:4, C:2), E:2 (A:1, B:1). Calculez pour chaque fonction le temps passé dans la fonction elle-même et le temps passé dans la fonction elle-même plus ses enfants, comme le ferait l'outil `gprof`. **(2 points)**

Le temps passé dans chaque fonction elle-même (soi) est donné par le nombre des échantillons, chacun comptant pour 1s. On a ainsi 6 secondes dans C, 3s dans D, 2s dans A, 1s dans B et 0s dans les autres. Pour le temps passé dans chaque fonction incluant les fonctions appelées (soi + appelés), il faut imputer le temps des fonctions appelées aux fonctions appelantes au prorata des appels. On commence par les fonctions feuilles où le temps "soi + appelés" est le temps de chaque fonction elle-même, soit D: 3s, E: 0s. Ce temps est imputé 4/6=2s à B et 2/6=1s à C. On

peut donc calculer le temps "soi + appelés" pour B et C et continuer ainsi en remontant jusqu'au programme principal.

```
Fonction  self  self+childs
main         0s          12s
A            2s          12s (1/1*12s=12s de main)
B            1s           3s (2/2*2s=3s de A)
C            6s           7s (3/3*7s=7s de A)
D            3s           3s (4/6*3s=2s de B et 2/6*3s=1s de C)
E            0s           0s
```

b) Pour chacun des 4 cas suivants, suggérez un outil qui serait particulièrement approprié pour découvrir ou cerner le problème. Expliquez comment cet outil permet de détecter ou cerner chaque problème. i) Un programme de calcul à paralléliser prend trop de temps en raison de certaines fonctions qui ne sont pas assez optimisées ou qui pourraient être parallélisées, on veut identifier ces fonctions. ii) Un programme fait par un étudiant de première année est bourré de problèmes comme l'accès à des variables non initialisées, des débordements de vecteurs et des libérations prématurées de mémoire. iii) Un programme s'exécute avec une bonne performance en moyenne et produit un résultat correct mais présente, à de rares occasions, des latences trop élevées pour accomplir certaines tâches. iv) Un programme présente tout probablement des problèmes de faux partage en mémoire cache, et on veut identifier à quel endroit ceci se produit dans le programme. **(2 points)**

Pour avoir une bonne idée du temps passé dans chaque fonction au total i), l'outil gprof ou des profileurs comme Oprofile ou perf fonctionnent très bien. A chaque intervalle de temps, la position courante du compteur de programme est échantillonnée, ce qui donne la fonction en train de s'exécuter. Toute fonction qui consomme beaucoup de temps CPU générera beaucoup d'échantillons et sera facilement identifiée. De plus, le surcoût à l'exécution est faible. Pour trouver les problèmes d'accès mémoire incorrects ii), les outils Memcheck ou Address Sanitizer sont très efficaces, avec Memcheck possiblement plus complet mais plus coûteux en temps à utiliser. Ces outils instrumentent les allocations et libérations de mémoire (pour identifier les zones valides à accéder) ainsi que les lectures et écritures. Pour chaque écriture on note que le contenu devient initialisé et on vérifie que la mémoire est allouée, et pour chaque lecture on vérifie que le contenu a été initialisé et que la mémoire est allouée. Pour des problèmes intermittents comme ceux de latence iii), un traceur comme ftrace ou LTTng, possiblement couplé à un détecteur de latence pour activer la sauvegarde de la trace, est généralement le meilleur choix. On peut ainsi savoir qu'est-ce qui s'est passé à quel moment et trouver les événements qui indiquent où et quand a lieu une latence trop élevée. Par exemple, on peut avoir des événements pour l'entrée et la sortie de fonctions importantes et des appels systèmes, et pour l'ordonnancement des processus. Ceci permet normalement de comprendre ce qui s'est passé pendant la latence problématique. Pour trouver les problèmes de fautes de cache et faux partage iv), les outils de profilage basés sur les compteurs de performance comme Oprofile ou perf sont excellents et peu coûteux en temps. En demandant une interruption à toutes les 100000 fautes de cache, par exemple, et en échantillonnant l'adresse du code et de la variable visés, on peut rapidement voir les sections de code, ou les variables, où surviennent un grand nombre de fautes de cache.

c) Pour le deuxième travail pratique, vous avez repris un algorithme sériel afin de le paralléliser avec OpenMP. Après avoir simplement ajouté des directives OpenMP parallel for, il y avait des sources de bruits dans la sortie générée. Quelle en était la cause? Est-ce que ces problèmes affectaient aussi la performance? **(1 point)**

Une variable partagée causait des problèmes lorsque plusieurs fils d'exécution se mettaient à l'accéder en parallèle avec OpenMP. Ceci causait à la fois un problème de corruption non déterministe (bruit dans les données) et un problème de performance en raison du ping pong d'invalidations et de fautes entre les cache de plusieurs processeurs, dont les fils accédaient en parallèle la même variable.

Le professeur: Michel Dagenais