ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Hiver 2013)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Mercredi le 5 décembre 2012

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) Un programme effectue pour chaque point la moyenne avec ses deux voisins. Convertissez ce programme en code efficace écrit en assembleur Vectoriel MIPS. Tous les éléments de a (a[0] à a[n + 1]) sont accédés mais seulement les éléments b[1] à b[n] sont écrits. **(3 points)**

```
double a[n + 2], b[n + 2];

for(i = 1 ; i <= n ; i++) b[i] = (a[i] + a[i + 1] + a[i - 1]) / 3;
```

```
; charger les valeurs et adresses dans des registres
        LD      R1, n
        LD      R2, a       ; a[0] (i - 1)
        LD      R3, b
        ADDI    R4, R2, #8 ; a[1] (i)
        ADDI    R5, R2, #16; a[2] (i + 1)
        ADDI    R3, R3, #8 ; b[1] (i)
        L.D     F0, #3
; nombre d'opérations vectorielles de 64 éléments
        DSRL    R6, R1, #6
; nombre d'opérations restant à la fin
        ANDI    R7, R6, #63
        BEQZ    R6, end ; moins de 64 éléments
loop:   LV      V1, R2
        LV      V2, R4
        LV      V3, R5
        ADDV.D  V4, V1, V2
        ADDV.D  V4, V4, V3
        DIVVS.D V4, V4, F0
        SV      V4, R3
        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, R4
        LV      V3, R5
        ADDV.D  V4, V1, V2
        ADDV.D  V4, V4, V3
        DIVVS.D V4, V4, F0
        SV      V4, R3
```

b) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: addition et soustraction point flottant, multiplication point flottant, division point flottant, opérations entières, opérations logiques, et 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 10, les divisions de 20 et les chargements et rangements de 12. L'instruction de comparaison (SGTV) est traitée par l'unité d'addition et soustraction point flottant, et prend le même nombre de cycles qu'une addition ou soustraction. Calculez le temps requis pour l'exécution de chacune des deux séquences d'instructions suivantes. **(2 points)**

```
/* Version inconditionnelle */
L.D      F1, #10
L.D      F2, #2
LV       V1, R1
MULVS.D  V1, V1, F2
SV       V1, R1

/* Version conditionnelle */
L.D      F1, #10
L.D      F2, #2
LV       V1, R1
SGTVS.D  V1, F1
MULVS.D  V1, V1, F2
CVM
SV       V1, R1
```

Les deux premières instructions prennent 1 cycle chacune, la troisième et la quatrième peuvent être chaînées en 12 + 10 + 64 cycles, la cinquième est seule (unité de rangement/chargement occupée dans la chaine précédente) et prend 12 + 64 cycles. Le total est donc: 2 + 12 + 10 + 64 + 12 + 64 = 164 cycles. Dans la version conditionnelle, le temps requis est le même, à la différence que SGVT s'ajoute à la chaîne LV/MULTV, ce qui ajoute 8 cycles, et CVM ajoute 1 cycle, pour un total de 173 cycles.

Question 2 (5 points)

a) Il faut calculer une version réduite d'une image. Pour ce faire, la moyenne d'une région de 16x16 pixels est calculée et devient le pixel correspondant de l'image réduite. Le code en C pour effectuer ce travail est fourni. Proposez une implémentation efficace en OpenCL qui effectue ce travail. Fournissez le code pour la fonction de type kernel correspondante. Expliquez par ailleurs les arguments fournis pour l'appel de cette fonction et donnez l'énoncé clEnqueueNDRangeKernel avec ses arguments qui serait appelé pour exécuter votre fonction. **(3 points)**

```
int i, j;
float image_in[1024][1024];
float image_out[64][64]

for(i = 0; i < 1024; i++) {
  for(j = 0; j < 1024; j++) {
    image_out[i/16][j/16] += image_in[i][j] / 256;
  }
}
```

Chaque item (work item) est constitué d'une portion 16x16 de l'image qui produit un pixel en sortie. En deux dimensions, la taille du problème est de 64x64. La fonction clEnqueueNDRangeKernel est appelée avec deux dimensions, une grandeur totale de 64x64 et sans taille de groupe spécifiée, laissant le système décider des valeurs optimales.

```
__kernel void Thumb_Pixel(__global float image_in[1024][1024],
    __global float* image_out[64][64])
{
  int i, j;
  int k = get_global_id(1);
  int l = get_global_id(0);
  int start_i = k * 16;
  int end_i = start_i + 16;
  int start_j = l * 16;
  int end_j = start_j + 16;
  for(i = start_i ; i < end_i ; i++)
    for(j = start_j; j < end_j ; j++)
      sum += image_in[i][j];
  image_out[k][l] = sum / 256;
}
```

b) Dans la mesure où les processeurs sur les GPGPU sont de type SIMD, qu'arrive-t-il lorsque le code OpenCL d'une fonction de type kernel, exécutée sur ces processeurs, contient des conditions (if then else) de sorte que le traitement requis peut différer d'un élément à l'autre du groupe de traitement (work group). Comment cela est-il exécuté par le processeur SIMD? Le temps de traitement est-il le plus court entre la branche if et else, le plus long, ou la somme des deux? **(1 point)**

Le processeur SIMD doit effectuer chaque opération, qu'elle soit dans le if ou le else. Il va seulement désactiver ces opérations selon le cas pour chaque élément de donnée, afin d'obtenir le résultat voulu (if ou else). En conséquence, le temps requis est déterminé par la somme des opérations dans les deux branches.

c) En OpenCL, est-il possible d'appeler la fonction malloc (ou new en C++) à l'intérieur d'une fonction de type kernel? Pourquoi? **(1 point)**

Les fonctions de type kernel ne peuvent allouer de mémoire autrement que statiquement. En effet, le nombre de registres disponible sur chaque processeur SIMD est limité et doit être connu à l'avance pour bien répartir les différents fils d'exécution.

Question 3 (5 points)

a) Dans un cours de sécurité informatique, le professeur a donné un énoncé de devoir encrypté. Heureusement, il a fourni la fonction de décryption (sans la clé) et spécifié que la clé de décryption est un entier de 64 bits dont seulement les 6 octets les moins significatifs sont utilisés. Il faut donc essayer toutes les valeurs qu'il est possible de représenter avec 6 octets. Voici le programme sériel pour ce faire mais qui prendrait plus de temps que l'échéance pour la remise du devoir. Convertissez ce programme en programme MPI efficace. Les ordinateurs utilisés ont des entiers (int) de 64 bits et la taille du message chiffré est connue statiquement (LEN). Pour simplifier le programme MPI, il suffit d'imprimer le résultat trouvé sur le noeud qui trouve la bonne clé, comme dans le programme sériel. **(3 points)**

```
int key, end;
char message_chiffre[LEN];
char *message_clair;

end = 1 << (6 * 8);
for(key = 0 ; key < end; key++) {
  decrypt(message_chiffre, LEN, key, message_clair);
  if(strncmp("Devoir 1", message_clair, 8) == 0)
      printf("Devoir decode:\n%s", message_clair);
}
```

Le noeud 0 est le coordonnateur, il divise le travail entre tous les noeuds, lui inclus.

```
char message_chiffre[LEN];
char *message_clair;
int size, rank, end, nb_keys, ks, ke;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
if(rank == 0) read_message(message_chiffre, LEN);
MPI_Bcast(message_chiffre, LEN, MPI_CHAR, 0, MPI_COMM_WORLD);

end = 1 << (6 * 8);
```

```
nb_keys = (end / size);
ks =  nb_keys * rank;
if(rank == (size - 1)) ke = end;
else ke = ks + nb_keys

for(; ks < ke; ks++) {
  decrypt(message_chiffre, LEN, ks, message_clair);
  if(strncmp("Devoir 1", message_clair, 8)
      printf("Devoir decode:\n%s", message_clair);
}

MPI_Finalize();
```

b) Dans le cadre du travail pratique 3, le nombre des instances du programme MPI était augmenté de quelques instances jusqu'au nombre total de coeurs par ordinateur multiplié par le nombre d'ordinateurs dans la grappe. Est-ce que le délai de communication varie beaucoup entre le cas de deux instances sur le même ordinateur versus deux instances sur deux ordinateurs différents? Le délai de communication signifie-t-il nécessairement un délai équivalent perdu pour le programme parallèle? **(1 point)**

Sur un même noeud, les messages peuvent s'échanger par mémoire partagée, ce qui est passablement plus rapide que par réseau. La bande passante de la mémoire est de plusieurs centaines de MO par seconde, alors que la bande passante du réseau est de 10MO/s ou 100MO/s selon que les prises sont à 100Mbit/s ou 1Gbit/s. Heureusement, avec des communications asynchrones, il est souvent possible de continuer à effectuer du traitement utile pendant les délais de réseau, en structurant bien son application.

c) La fonction suivante, rencontrée dans le cadre du travail pratique 2 contient un problème de performance sérieux. Lequel? Suggérez un correctif. **(1 point)**

```
int encode_slow_c(struct chunk *chunk)
{ int i, j, checksum = 0;
  int width = chunk->width, height = chunk->height;
  int key = chunk->key;
  char *data = chunk->data;

  #pragma omp parallel for private(i,j) reduction(+:checksum)
  for (i = 0; i < width; i++) {
    for (j = 0; j < height; j++) {
      int index = i + j * width;
      data[index] = data[index] + key;
      checksum += data[index];
    }
  }
  chunk->checksum = checksum;
```

```
  return 0;
}
```

Les accès au vecteur data dans la boucle la plus interne sont à des positions non consécutives, ce qui donne une mauvaise localité de référence et cause beaucoup plus de fautes de cache. Il suffit de remplacer les trois lignes après le pragma par:

```
for (i = 0; i < height; i++) {
  for (j = 0; j < width; j++) {
    int index = i * width + j;
```

Question 4 (5 points)

a) Les plus récents processeurs Intel bénéficient du support pour les tables de pages étendues (EPT) afin de mieux supporter la virtualisation. Comment cela fonctionne-t-il? Expliquez quelles opérations sur une machine virtuelle seront plus rapides ou plus lentes avec EPT? **(2 points)**

Les tables de pages étendues permettent de supporter par matériel la double traduction de 1) adresse logique machine virtuelle à adresse physique de machine virtuelle (= adresse logique de machine physique), 2) adresse logique de machine physique à adresse physique. Ainsi, lorsque la machine virtuelle met à jour ses tables de pages, elles sont automatiquement prises en compte par le matériel. Autrement, sans EPT, l'environnement de virtualisation doit protéger la mémoire prise par ces tables de pages des machines virtuelles afin qu'une interruption soit générée lors de leur mise à jour, permettant de reporter le changement dans les tables de la machine physique et d'émuler le bon comportement. Ainsi, avec EPT, les mises à jour de tables de pages dans les machines virtuelles sont beaucoup plus rapides. Par contre, les accès en mémoire à partir des machines virtuelles, en raison de la double traduction, sont un peu plus lents. Ceci ne touche toutefois que les accès pour lesquels le TLB ne contient pas déjà la pré-traduction.

b) Un GPGPU récent de NVidia, le GTX580, contient 16 processeurs SIMD de 32 éléments de traitement (avec un ALU) chacun, pour un total de 512 éléments de traitement. Par comparaison, le Intel Xeon Phi qui vient d'être annoncé contient 60 processeurs avec vecteur. Sur les forums, certains amateurs prétendent qu'avec 60 processeurs plutôt que 512, le Xeon Phi n'est pas compétitif du tout. Qu'en est-il? Comment peut-on comparer leur performance pour des tâches de calcul scientifique parallèle?

(1 point)

Le vecteur du Xeon Phi peut traiter en parallèle 16 mots de 32 bits. Avec 60 processeurs contenant un vecteur de 16 mots, ceci fait un total équivalent à 960 éléments de traitement, ce qui se compare avantageusement aux 512 du GPGPU. En outre, les processeurs du Xeon Phi sont cadencés à une plus haute fréquence et ne souffrent

pas des mêmes délais requis sur les GPU pour communiquer entre la mémoire hôte et la mémoire du GPU. Ainsi, le Xeon Phi a le potentiel d'être très compétitif avec le GTX580 et même avec les modèles améliorés de GPU qui sortiront en même temps.

c) L'entreprise pour laquelle vous travaillez aimerait pouvoir tirer profit des cycles inutilisés de tous les ordinateurs de bureau des employés, particulièrement lorsque ceux-ci sont absents, en réunion ou le soir. Quel logiciel pourrait faire ce travail? Est-ce que ceci constitue une grille ou un nuage? **(1 point)**

Ceci est le scénario typique d'une grille. Le logiciel MRG de Red Hat, en particulier la portion grille basée sur Condor, est un bon logiciel pour offrir un tel service. Un nuage est typiquement utilisé pour permettre un accès flexible à une grappe d'ordinateurs dédiés à être partagés.

d) Dans quelles circonstances utiliseriez-vous OpenMP ou OpenCL? Donnez un exemple. **(1 point)**

OpenMP ne peut être utilisé pour accéder à la puissance de calcul du GPGPU, OpenCL est donc le seul choix pour un GPGPU. OpenCL, bien qu'il puisse être utilisé pour un CPU multi-coeur conventionnel, est inutilement contraignant comme environnement de programmation comparé à OpenMP. Si les deux type de ressources sont disponibles, un GPGPU avec OpenCL sera intéressant s'il est facile de décomposer le problème en un très grand nombre d'unité de travail (work item) qui peuvent être traitées en parallèle de manière plus ou moins indépendante. Par exemple, sur un ordinateur sans GPU, OpenMP est préférable car plus simple d'utilisation. Pour un problème de traitement d'image, avec un grand nombre d'opérations parallèles, OpenCL sera le meilleur choix si la performance est critique et mérite un investissement plus important en programmation, et si un GPGPU est disponible.

Le professeur: Michel Dagenais