ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

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

CORRIGÉ DE L'EXAMEN FINAL

DATE: Mercredi le 18 décembre 2013

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 reçoit dans un vecteur *in* des informations lues de capteurs. Il doit effectuer une conversion et ajouter une calibration *cal* spécifique à chaque capteur. Convertissez ce programme en code efficace écrit en assembleur Vectoriel MIPS. **(3 points)**

```
double in[N], out[N], cal[N];
int i;

for(i = 0 ; i < N ; i++) out[i] = ((in[i] + cal[i]) * 1.8 + 32);

; charger les valeurs et adresses dans des registres
        LD      R1, n
        LD      R2, in
        LD      R3, cal
        LD      R4, out
        L.D     F0, #1.8
        L.D     F1, #32
; nombre d'opérations vectorielles de 64 éléments
        DSRL    R5, R1, #6
; nombre d'opérations restant à la fin
        ANDI    R6, R1, #63
        BEQZ    R5, end ; moins de 64 éléments
loop:   LV      V1, R2
        LV      V2, R3
        ADDV.D  V3, V1, V2
        MULVS.D V3, V3, F0
        ADDVS.D V3, V3, F1
        SV      V3, R4
        ADDI    R2, R2, #64 * 8
        ADDI    R3, R3, #64 * 8
        ADDI    R4, R4, #64 * 8
        ADDI    R5, R5, #-1
        BNEZ    R5, loop
end:    MTC1    VLR, R6
loop:   LV      V1, R2
        LV      V2, R3
        ADDV.D  V3, V1, V2
        MULVS.D V3, V3, F0
        ADDVS.D V3, V3, F1
        SV      V3, R4
```

b) Le Xeon Phi peut traiter 512 bits en parallèle, sous la forme de 16 valeurs point flottant de 32 bits, ou 8 de 64 bits. Son pipeline d'exécution comporte 12 étages mais permet

de produire un résultat de 512 bits à chaque cycle. Comment cela se compare-t-il avec l'architecture vectorielle MIPS étudiée, en termes de matériel requis, de performance et de bande passante requise venant de la mémoire? **(1 point)**

Le Xeon Phi requiert beaucoup plus de matériel, avec l'équivalent de 8 ALU point flottant de 64 bits pour chaque processeur, contre seulement 1 pour le VMIPS. Par contre, il peut produire 8 calculs point flottant de 64 bits à chaque cycle, contre seulement 1 pour le VMIPS. Le coût de remplir le pipeline au début d'un calcul est un facteur à ne pas négliger pour le nombre réel de calculs point flottant par cycle, mais il nous manque d'information pour comparer les deux. Il est probable que le Xeon Phi puisse plus facilement enchaîner les opérations point flottant sans avoir à bloquer et attendre plusieurs cycles de latence pour remplir le pipeline. Sur la plupart des architectures vectorielles, la bande passante vers la mémoire demeure ultimement le goulot d'étranglement. Pour tirer parti de son matériel de calcul plus performant que le VMIPS, le Xeon Phi devrait avoir une bande passante vers la mémoire accrue en proportion.

c) Vous désirez mettre sur votre liste de cadeaux un ordinateur pour le calcul parallèle. Vous hésitez entre un co-processeur de type GPU (e.g. AMD ou NVidia) ou un Intel Xeon Phi. Quels sont les principales différences architecturales et avantages de chacun? **(1 point)**

Les GPU sont composés de nombreux processeurs SIMD (e.g. 16), chacun venant avec plusieurs éléments de calcul (e.g. 16). Ces processeurs SIMD ont une architecture particulière, optimisée pour le traitement graphique, et leur répertoire d'instruction n'est pas visible par le programmeur et peut changer d'une génération à l'autre. Les GPU se vendent en grande quantité, étant donné leur attrait pour l'affichage des jeux. Leur prix est donc très compétitif. Cependant, ils sont découplés du processeur principal et utilisent un modèle de programmation différent. Il est donc plus difficile de les mettre en oeuvre, de les programmer, d'étudier leur performance et de les déboguer. Le Xeon Phi offre un matériel comparable en terme de puissance de calcul, avec 60 processeurs contenant chacun l'équivalent de 16 ALU 32 bits, pour un total de 960 unités de calcul. Néanmoins, cela demeure une extension naturelle à l'architecture usuelle des processeurs centraux des ordinateurs. Il s'agit d'une unité centrale de traitement de 60 coeurs qui accepte le répertoire d'instruction X86_64 avec des instructions vectorielles ajoutées. Il est donc plus facile de le mettre en oeuvre en réutilisant les outils existants.

Question 2 (5 points)

a) On veut générer un histogramme pour la valeur des pixels dans une grande image. 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. **(3 points)**

```
void Histogram(unsigned char image[4096][4096],
    unsigned int histogram[256])
{
  int i, j;

  for(i = 0; i < 4096; i++) {
    for(j = 0; j < 4096; j++) {
      histogram[image[i][j]]++;
    }
  }
}
```

Le calcul à effectuer sur cette grande quantité de données est relativement simple mais le résultat final combiné de tous ces calculs constitue un point de contention. Heureusement, il est possible de calculer un histogramme sur un sous-ensemble de l'image et ensuite de combiner ces histogrammes à l'aide d'une réduction. Une solution assez simple est d'avoir comme unité de travail le calcul d'un histogramme sur un grand nombre de points, par exemple une rangée de 4096 pixels. Ensuite, par le biais d'opérations atomiques locales, chaque unité de travail termine en combinant son histogramme avec un seul appartenant au groupe de travail. Finalement, cet histogramme est reporté sur celui global en donnant une section à transférer pour chaque unité de travail. Cette fonction de type kernel est appelée avec une seule dimension (le nombre de rangées) et une taille de 4096, chaque unité de travail prenant en compte une rangée complète. La taille du groupe de travail (nombre de rangées traitées dans un même groupe) est laissée à 0, pour être choisie par OpenCL à l'exécution.

```
__kernel void Histogram(__global unsigned char image[4096][4096],
    __global unsigned int histogram[256])
{
  int i;
  int row = get_global_id(0), group_size = get_local_size(0);
  int chunk_size = 256 / group_size + 1;
  int start_chunk = get_local_id(0) * chunk_size;
  int end_chunk = min(256, start_chunk + chunk_size);
  unsigned int tmp_histogram[256];
  __local unsigned int local_histogram[256];

  for(i = 0; i < 256; i++) tmp_histogram[i] = 0;
  for(i = 0; i < 4096 ; i++) tmp_histogram[image[row][i]]++;
  for(i = start_chunk; i < end_chunk; i++) local_histogram[i] = 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  for(i = 0; i < 256; i++)
    atomic_add(local_histogram[i], tmp_histogram[i]);
  barrier(CLK_LOCAL_MEM_FENCE);
  for(i = start_chunk; i < end_chunk; i++)
```

```
    atomic_add(histogram[i], local_histogram[i]);
}
```

b) Que fait la fonction `barrier(CLK_LOCAL_MEM_FENCE)`? Donnez un exemple de cas où son utilisation serait requise? **(1 point)**

Cette fonction effectue un rendez-vous entre tous les fils d'exécution d'un groupe de travail. De plus, elle constitue une barrième mémoire garantissant que tout accès effectué après verra le résultat de tout accès effectué avant. Cette fonction est requise lorsque plusieurs éléments de calcul travaillent sur différents morceaux d'un résultat et qu'ensuite ils doivent consulter les autres morceaux produits par les autres éléments de calcul. Ceci assure en effet que chaque élément a terminé ses calculs et que toutes les modifications ont été propagées vers la mémoire locale (partagée par le groupe de travail).

c) Dans le travail pratique 2, vous deviez calculer une image avec le programme sinoscope en utilisant d'abord OpenMP et ensuite OpenCL. À partir de quelle superficie d'image valait-il mieux utiliser OpenCL et la carte graphique plutôt que le processeur avec OpenMP à 8 fils? Expliquez pourquoi. **(1 point)**

L'utilisation de OpenCL requiert un certain temps de démarrage pour initialiser le contexte et la carte GPU, compiler le code OpenCL, transmettre le programme et les données à la carte et récupérer le résultat. Si le temps de calcul avec OpenMP est du même ordre de grandeur que ce temps de démarrage pour le GPU, ce n'est pas intéressant d'utiliser OpenCL. Par contre, lorsque l'image est plus grande, dans les milliers ou dizaines de milliers de pixels, le temps de calcul augmente rapidement sur les 8 processeurs avec OpenMP. La vitesse de calcul plus grande du GPU, étant donné ses centaines d'ALU, rend alors l'utilisation d'OpenCL avantageuse.

Question 3 (5 points)

a) Le programme suivant doit s'exécuter en 21 processus sur autant de noeuds. Il décompose un vecteur en morceaux de taille semblable et demande à chaque processus d'additionner un nombre à chaque élément dans son morceau, en plus de faire la somme de ses éléments. Pour chacun des 21 processus, dites combien d'éléments font partie du morceau de vecteur à traiter (`mysize` tel qu'imprimé sur chaque noeud)? Le programme bloque avant la fin et reste en attente. Quel est le problème? **(2 points)**

```
#define SIZE 100000000
float  v[SIZE];

int main (int argc, char *argv[])
{
  int i, j, rank, size, offset, chunksize, mysize, extra;
  double mysum = 0.0, sum = 0.0;
```

```
  MPI_Status s;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  chunksize = SIZE / size;
  extra = SIZE % size;
  if(rank == 0) {
    for(i=0; i < SIZE; i++) {v[i] =  i; sum += v[i]; }
    offset = chunksize;
    for (i = 1; i < size; i++) {
      mysize = i < (size - extra) ? chunksize : chunksize + 1;
      MPI_Send(&offset, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
      MPI_Send(&mysize, 1, MPI_INT, i, 2, MPI_COMM_WORLD);
      MPI_Send(&v[offset], mysize, MPI_FLOAT, i, 3, MPI_COMM_WORLD);
      offset = offset + mysize;
    }
    offset = 0; mysize = chunksize;
  }
  else {
    MPI_Recv(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &s);
    MPI_Recv(&mysize, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, &s);
    MPI_Recv(&v[offset], mysize, MPI_FLOAT, 0, 3, MPI_COMM_WORLD, &s);
    printf("%d: mysize = %d\n", rank, mysize);
  }

  for(i = offset; i < offset + mysize; i++) {
    v[i] = v[i] + i; mysum = mysum + v[i];
  }
  MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  if(rank == 0) {
    for(i = 1; i < size; i++) {
      MPI_Recv(&offset, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &s);
      MPI_Recv(&mysize, 1, MPI_INT, i, 2, MPI_COMM_WORLD, &s);
      MPI_Recv(&v[offset], mysize, MPI_FLOAT, i, 2, MPI_COMM_WORLD, &s);
    }
    printf("Sum = %f\n",sum);
  } else {
    MPI_Send(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(&mysize, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&v[offset], mysize, MPI_FLOAT, 0, 3, MPI_COMM_WORLD);
  }
MPI_Finalize();
}
```

La taille de 100 000 000 divisée par 21 donne 4761904, reste 16. Les 5 noeuds de 0 à 4 inclusivement traiteront 4761904 éléments, alors que les 16 derniers, de 5 à 20 inclusivement, en traiteront un de plus, 4761905.

Au moment de retourner le résultat, le troisième élément, le contenu du vecteur v*, est envoyé avec une valeur de* `tag` *de 3, alors que le processus 0 fait un MPI_Recv avec un* `tag` *de 2. Il attend donc un message avec cette valeur 2 qui n'arrivera jamais et bloque ainsi.*

b) Le programme suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez le contenu des deux lignes (o1 et o2) imprimées sur chaque noeud? **(2 points)**

```
int main (int argc, char *argv[])
{
  int size, rank, i, in[4], o1[4], o2[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 + rank * rank; o1[i] = o2[i] = 0; }
  MPI_Allgather(in, 1, MPI_INT, o1, 1, MPI_INT, MPI_COMM_WORLD);
  MPI_Alltoall(in, 1, MPI_INT, o2, 1, MPI_INT, MPI_COMM_WORLD);
  printf("%d: o1={%d, %d, %d, %d};\n", rank, o1[0], o1[1], o1[2], o1[3]);
  printf("%d: o2={%d, %d, %d, %d};\n", rank, o2[0], o2[1], o2[2], o2[3]);
  MPI_Finalize();
}
```

Les processus 0 à 3 ont respectivement dans in: { 0, 1, 2, 3 }, { 1, 2, 3, 4 }, { 4, 5, 6, 7 } et { 9, 10, 11, 12 }. Conséquemment, Allgather placera dans o1 les premières valeurs de chaque in. Le contenu de o1, { 0, 1, 4, 9 }, sera le même pour tous les processus. Alltoall placera dans o2 le premier élément de chaque in sur le processus 0, les second éléments sur le processus 1... ce qui donnera pour les processus 0 à 3 respectivement dans o2: { 0, 1, 4, 9 }, { 1, 2, 5, 10 }, { 2, 3, 6, 11 } et { 3, 4, 7, 12 }.

c) Dans le cadre du travail pratique 3, vous avez mesuré la performance des fonctions de communication de type bloquant (blocking), avec tampon (buffered) et asynchrone (async). Expliquez comment fonctionne chaque type, et décrivez et expliquez les résultats obtenus pour ces types de communication dans le cadre du travail pratique 3? **(1 point)**

Les appels bloquants ne retournent que lorsque les données sont parties et que le tampon d'envoi passé en paramètre peut être réutilisé. Ce temps, bloqué en attente, aurait pu être utilisé pour continuer des calculs pendant que le DMA copie les données de ce tampon vers la carte réseau. Les appels avec tampon (buffered) requièrent une copie supplémentaire vers un tampon temporaire, un coût additionnel en temps et en mémoire; il n'y a pas d'attente mais un surcoût léger en temps d'exécution actif

et en mémoire. Les appels asynchrones sont normalement les plus efficaces. Il n'y a pas d'attente ni de copie mais seulement le coût d'un appel pour vérifier lorsque la communication est terminée et le tampon peut être réutilisé. La programmation avec les appels asynchrones peut cependant être légèrement plus complexe.

Dans le travail pratique, l'application utilisait une chaîne d'appels bloquants qui attendaient les uns après les autres. C'était donc très lent à cause de la sérialisation qui en résultait. Les appels avec tampon donnaient un meilleur résultat et les appels asynchrones, évitant une copie, donnaient le meilleur résultat.

Question 4 (5 points)

a) Vous voulez évaluer le temps d'exécution pris par les différentes parties d'un programme afin de l'optimiser. Trois options s'offrent à vous: l'outil de profilage `gprof` qui instrumente les entrées de fonction et échantillonne le compteur de programme à chaque miliseconde, l'outil de trace `lttng` avec l'entrée et la sortie de chaque fonction qui ont été instrumentées, et l'outil `Perftools CPU profile` qui échantillonne le contenu de la pile d'exécution à chaque miliseconde. Comparez ces trois options en termes de surcoût ajouté au programme et d'informations obtenues (complétude, fiabilité, précision...). **(2 points)**

L'outil gprof ajoute environ 5% de temps pour compter le nombre de fois que chaque fonction est appelée et par qui, et .5% de temps pour acculumer les échantillons de compteur de programme à chaque miliseconde. Un grand tampon est requis pour l'histogramme des échantillons mais il est relativement peu rempli et plusieurs pages ne sont probablement jamais allouées. Les temps passés dans chaque fonction sont assez précis, même si une erreur associée à l'échantillonnage est toujours possible. Les temps imputés aux parents (temps passé dans une fonction et celles appelées) ne sont pas fiables. En effet, ils reposent sur l'hypothèse que chaque appel, indépendamment de sa provenance, a la même durée en moyenne. Il est très possible que les appels de différentes origines pour une même fonction soient en moyenne assez différents. Le nombre d'appels de chaque fonction, obtenu par instrumentation, est exact.

L'outil lttng, qui instrumente chaque entrée et sortie de fonction, fournira une information très précise et détaillée. Chaque appel pourra être analysé séparément et il demeure possible de créer toutes sortes de statistiques (temps moyen par appel, selon la provenance, temps passé dans chaque fonction ou chaque fonction et celles appelées...). Par contre, le surcoût en temps d'exécution sera considérable puisque les entrées et sorties de fonctions sont très nombreuses, un facteur de ralentissement de 2 ou plus, selon la longueur moyenne des fonctions. Un grand espace sera aussi requis pour écrire la trace sur disque.

L'outil CPU profile prend un peu plus de temps que .5%, puisqu'il échantillonne toute la pile plutôt que simplement l'adresse courante, mais moins que les 5% requis pour instrumenter toutes les entrées de fonction. De plus, il est facile de jouer avec ce pourcentage en ajustant la fréquence d'échantillonnage. Cette technique ne permet

pas de voir tous les appels effectués et ne fournit donc pas un décompte exact des appels, ou même un graphe complet des appels. Par contre, les valeurs de temps passé dans une fonction et celles appelées est plus fiable, autant que celles pour le temps passé dans chaque fonction elle-même. Cet outil représente un rapport surcoût versus information extraite particulièrement avantageux.

b) Comment l'outil lockdep peut-il détecter la possibilité d'un interblocage, même si celui-ci ne s'est pas produit pendant l'exécution sous analyse? **(1 point)**

Les interblocages interviennent lorsqu'il y a un cycle dans le graphe de dépendance entre les fils d'exécution et les mutex. Une excellente stratégie pour éviter toute possibilité de tel cycle est d'imposer un ordre entre les verrous pour leur acquisition par chaque fil. L'outil lockdep note tout au long de l'exécution d'un programme l'ordre dans lequel les verrous sont acquis par chaque fil d'exécution. Chaque fois qu'un verrou est pris par un fil, un lien dirigé est ajouté entre ce verrou et le précédent acquis et encore détenu par ce fil. Ces liens s'ajoutent et un graphe est formé reliant les différents verrous. Si, à la fin de l'exécution, il existe un cycle dans le graphe, il n'y a pas d'ordre qui ait été toujours respecté pour l'acquisition des verrous. En conséquence, même si un interblocage ne s'est pas produit, ceci donne une indication qu'un interblocage aurait pu se produire avec un ordonnancement différent des actions effectuées par les différents fils d'exécution.

c) Quel outil permet efficacement d'estimer les fautes de cache, causées par les différentes sections d'un programme en exécution, avec un surcoût ajouté au programme extrêmement faible? Expliquez? **(1 point)**

Les compteurs de performance peuvent générer une interruption à chaque n fautes de cache. Il est alors possible de noter l'adresse qui a causé la faute de cache. Ce coût est faible et peut être ajusté en variant la valeur de n. Les outils comme Oprofile et Perf utilisent les compteurs de performance de cette manière.

d) En calcul parallèle, l'efficacité du traitement est un facteur souvent primordial. Néanmoins, l'utilisation de virtualisation, par exemple avec OpenStack, est de plus en plus populaire pour le calcul parallèle, en dépit de son surcoût qui peut facilement atteindre 10%. Quel en est l'intérêt qui compense pour ce surcoût? **(1 point)**

Pour les organisations où il existe de nombreux utilisateurs potentiels et occasionnels, le temps requis pour adapter le logiciel à l'ordinateur parallèle est un frein à l'utilisation d'une infrastructure de calcul partagée. De la même manière, si un utilisateur peut avoir à utiliser différents calculateurs parallèles, (pour le prototype, pour la recherche, pour la production, pour des calculs de plus grande dimension), la portabilité de son environnement de travail devient une considération importante. En encapsulant sa configuration dans une image OpenStack, plus un fichier de configuration OpenStack, l'usager peut obtenir une grande portabilité. Ainsi, un service informatique pour le calcul parallèle peut, avec OpenStack, perdre 10% de sa puissance de calcul avec la virtualisation mais gagner en taux d'occupation de ses infrastructures, attirant plus de clients. De plus, le service informatique et ses clients sauvent beaucoup de temps en effort d'adaptation de leurs configurations logicielles.

Un système comme OpenStack permet de consolider plusieurs machines virtuelles peu occupées sur une seule machine physique. Cependant, ceci est peu utile pour le calcul parallèle, où chaque noeud est normalement occupé à presque 100%.

Le professeur: Michel Dagenais