ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2017)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Lundi le 11 décembre 2017

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) L'ordinateur le plus rapide du monde, selon le site top500.org, Sunway TaihuLight, contient 40 960 processeurs SW26010 de 256 coeurs de traitement à 1.45GHz plus 4 coeurs de gestion, pour 10 649 600 coeurs au total. Sa puissance de calcul, pour un banc d'essai simple très facile à paralléliser, est de 93 014 Tera Flop/s. Calculez le nombre d'opérations point flottant par coeur et par cycle. Suggérez une ou des configurations possibles et probables pour les processeurs (nombre d'unités arithmétiques parallèles, nombre de cycle par instruction) qui expliqueraient une telle performance. **(2 points)**

Avec 40960 x 256 = 10485760 coeurs de calcul et une fréquence de 1.45GHz, on se retrouve avec 10.48576 M coeurs x 1.45 GHz = 15204.352 T cycles-coeurs/s. Le nombre d'opérations point flottant par coeur et par cycle est donc de 93014 T Flop/s / 15204.352 T cycles-coeurs/s = 6.117 instruction point flottant par cycle-coeur. Il pourrait y avoir des pipelines superscalaires mais ceux-ci sont très peu utilisés dans les processeurs hautement parallèles. Le plus vraisemblable est l'utilisation d'une unité vectorielle de taille 8, utilisée à environ 75% d'efficacité, ce qui est excellent mais possible pour un programme très facile à paralléliser.

b) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: i) addition, soustraction et comparaison 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. LD, DRSL) prennent un cycle d'exécution chacune. Dans ces unités en pipeline, les additions, soustractions et comparaisons point flottant ont une profondeur de pipeline de 10, les multiplications point flottant de 15, les divisions point flottant de 22 et les chargements et rangements de 12. Calculez le temps requis pour l'exécution de la séquence d'instructions suivante. **(2 points)**

```
1  LV      V1, R2
2  LV      V2, R3
3  SLTV.D  V1, V2
4  ADDV.D  V3, V1, V2
5  SGEV.D  V1, V2
6  MULV.D  V3, V1, V2
7  SV      R4, V3
```

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, et 5, 6 et 7. Le total est de 337 cycles.

```
1  LV      V1, R2      ; 12 + 64 +
2  LV      V2, R3      ; 12 +
3  SLTV.D  V1, V2      ; 10 + 64 +
4  ADDV.D  V3, V1, V2  ; 10 + 64 +
5  SGEV.D  V1, V2      ; 10 +
6  MULV.D  V3, V1, V2  ; 15 +
7  SV      R4, V3      ; 12 + 64
```

c) Dans une grille de noeuds de calcul, comme celle du produit MRG de Red Hat, il peut arriver qu'un noeud soit surchargé ou doive être arrêté, et qu'une de ses tâches doive être déménagée sur un autre noeud. Dans d'autres cas, un noeud peut tomber en panne subitement et ses tâches en cours d'exécution sont perdues. Comment le service de gestion des tâches de la grille réagit-il, et quels mécanismes utilise-t-il, lorsqu'un noeud doit être libéré? Lorsqu'un noeud tombe en panne subitement? **(1 point)**

Lorsqu'un noeud doit être libéré, la tâche peut être suspendue pour reprendre plus tard sur le même noeud, ou un signal peut lui être envoyé afin qu'elle sauve son état intermédiaire avant de se terminer, en vue de redémarrer plus tard ailleurs avec l'état sauvé. Lorsque la technologie de grille est devenue populaire, la virtualisation n'était pas encore très répandue, ce qui aurait permis de migrer la tâche vers un autre noeud sans l'arrêter. Lorsqu'un noeud tombe en panne subitement, le service de gestion des tâches s'en rend compte éventuellement, puisqu'il monitore l'état des noeuds. Il peut alors repartir la tâche sur un autre noeud éligible. La tâche redémarrée repart normalement de zéro, le travail effectué jusqu'à ce que le noeud tombe en panne est donc perdu. Toutefois, un programmeur astucieux peut sauver régulièrement des résultats intermédiaires sur un serveur de fichiers et, au moment de démarrer le programme, part toujours des plus récents résultats intermédiaires disponibles.

Question 2 (5 points)

a) On vous demande d'écrire la fonction OpenCL PolyMax afin de trouver la valeur maximale dans un grand vecteur v qui contient 4Gi (2^{32}) entiers longs non signés. Le résultat doit être placé dans la valeur de retour globale res. Le GPU utilisé contient 64 processeurs SIMD de 64 coeurs chacun, pour un total de 4096 ALU parallèles. Ecrivez la fonction OpenCL PolyMax afin d'effectuer efficacement cette tâche. Expliquez le choix que vous aurez fait de la quantité de travail effectuée dans chaque *work item* et de la taille des *work group*. **(3 points)**

```
__kernel void PolyMax(__global const unsigned long *v,
    __global unsigned long *res)
{
// Que faire?
}
```

Afin d'occuper tous les coeurs avec plusieurs threads, par exemple 8 threads en hyper-threading, on veut avoir environ $4096 \times 8 = 32768 = 2^{15}$ work item. *Puisque nous avons* 2^{32} *éléments à traiter dans le vecteur, cela fait* $2^{32}/2^{15} = 2^{17} = 131072$ *éléments par* work item. *On peut laisser le système gérer la taille des* workgroup. *En général, une taille égale au nombre de processeurs SIMD ou un petit multiple de cette valeur est un bon choix. Il est possible d'obtenir le nombre de processeurs SIMD, et un multiple recommandé, avec les valeurs CL_DEVICE_MAX_COMPUTE_UNITS et CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE fournies par OpenCL.*

```
__kernel void PolyMax(__global const unsigned long *v,
    __global unsigned long *res)
```

```
{
  unsigned long i, start, end, val, m = 0, N = 131072;
  __local unsigned long local_res = 0;

  start = get_global_id(0) * N;
  end = start + N;
  for(i = start; i < end; i++) {
    val = v[i];
    if(val > m) m = val;
  }
  atomic_max(&local_res, m);
  barrier(CLK_LOCAL_MEM_FENCE);
  if(get_local_id(0) == 0) atomic_max(res, local_res);
}
```

Une variante intéressante est d'entrelacer les accès sur v, de sorte que les items successifs accèdent des cases successives de v, ce qui est normalement plus efficace pour les accès en cache qui sont regroupés par le matériel lorsqu'émis en même temps par les coeurs adjacents.

```
__kernel void PolyMax(__global const unsigned long *v,
    __global unsigned long *res)
{
  unsigned long i, start, end, val, m = 0, N = 131072;
  __local unsigned long local_res = 0;

  start = get_global_id(0);
  end = N * get_global_size(0);
  for(i = start; i < end; i += N) {
    val = v[i];
    if(val > m) m = val;
  }
  atomic_max(&local_res, m);
  barrier(CLK_LOCAL_MEM_FENCE);
  if(get_local_id(0) == 0) atomic_max(res, local_res);
}
```

b) Dans le travail pratique 2, vous deviez calculer une image avec le programme sinoscope en utilisant d'abord OpenMP et ensuite OpenCL. Lequel permettait d'obtenir la meilleure performance. Expliquez comment chacun des différents facteurs suivants peuvent affecter la performance relative des deux solutions: i) nombre de coeurs du CPU, ii) nombre de coeurs de la carte graphique, iii) taille de l'image, iv) mémoire virtuelle partagée ou non entre le CPU et le GPU? **(1 point)**

OpenMP roule sur les différents coeurs du CPU alors que OpenCL roule sur le GPU. i) Ainsi, plus de coeurs CPU vont favoriser OpenMP. ii) A l'inverse, plus de coeurs sur le GPU vont

favoriser OpenCL. Le problème toutefois avec OpenCL est qu'un certain temps est requis pour compiler le code et envoyer les données et le code vers le GPU. iii) Ainsi, pour les petites images, le temps de traitement sur le CPU peut être plus petit que le temps pour ce transfert servant au démarrage de la tâche sur le GPU. Dans ce cas, le GPU n'est pas compétitif. Pour les grandes images, le GPU avec ses très nombreux coeurs peut faire le traitement vraiment plus rapidement que le CPU, compensant facilement pour ce temps de démarrage. iv) Si la mémoire virtuelle est partagée entre le CPU et le GPU, il n'est pas nécessaire de copier les données vers le GPU et la pénalité pour accéder le GPU s'en trouve diminuée. Le GPU commencera donc à devenir plus efficace pour des images un peu moins grandes.

c) Un programme OpenCL appelle la fonction *clEnqueueNDRangeKernel* pour faire exécuter un kernel. Lorsque la fonction retourne *CL_SUCCESS*, est-ce que cela veut dire que le kernel s'est exécuté avec succès? Sinon, comment peut-on savoir à quel moment l'exécution du kernel s'est terminée et s'il n'y a pas eu d'erreur? **(1 point)**

Cette fonction sert à mettre la tâche en queue. Le code de retour indique que la mise en queue, et non pas l'exécution, a été effectuée avec succès. En revenant de cette fonction, la tâche peut ou non avoir débuté sur le GPU. Elle pourrait même débuter beaucoup plus tard. Pour savoir à quel moment la tâche est complétée, il faut associer un événement à cette commande au moment de l'appel à clEnqueueNDRangeKernel *en ayant eu soin de définir un* callback *pour cet événement qui sera appelé par l'environnement d'exécution quelques temps après que l'exécution de la tâche soit terminée. Il est possible de demander à l'environnement d'exécution de nous retourner dans cette fonction de rappel des informations sur le temps d'exécution de la tâche avec* clGetEventProfilingInfo.

Question 3 (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 à l'écran? Est-ce que le programme comporte une erreur? Expliquez. **(2 points)**

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

int main (int argc, char *argv[])
{
  int i, j, rank, size, dest, data, max = 0, res;
  int v[] = {2, 4, 8, 6, 12, 20, 18, 16, 32, 29, 31, 22};
  MPI_Status s;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
  if(rank == 0) {
    for(i = 0; i < sizeof(v) / sizeof(int); i++) {
      dest = (i % (size - 1)) + 1;
      MPI_Send(v + i, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
    }
  } else {
    for(i = 0; i < sizeof(v) / sizeof(int); i++) {
      dest = (i % (size - 1)) + 1;
      if(dest == rank) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &s);
        if(data > max) max = data;
      }
    }
    fprintf(stderr, "Result: rank = %d, max = %d\n",rank,max);
    MPI_Reduce(&max,&res,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
  }

  if(rank == 0) fprintf(stderr, "Reduced Result: %d\n", res);
  fprintf(stderr, "Program end, rank = %d\n", rank);
  MPI_Finalize();
}
```

Le programme effectue un MPI_Reduce, une communication globale, sur tous les noeuds sauf la racine (rang 0). Le noeud racine va donc simplement imprimer un résultat non défini et se terminer. Les autres noeuds vont envoyer leur résultat mais il ne sera pas reçu par le noeud racine, ce qui est très certainement une erreur de la part du programmeur. Nous obtenons donc la sortie suivante.

```
Reduced Result: 22029
Program end, rank = 0
Result: rank = 1, max = 29
Program end, rank = 1
Result: rank = 3, max = 32
Program end, rank = 3
Result: rank = 2, max = 31
Program end, rank = 2
```

b) Le programme suivant s'exécute sur 5 noeuds (MPI_Comm_size retourne 5). i) Donnez une sortie possible produite par ce programme à l'écran. ii) Le même résultat pourrait être obtenu avec les fonctions Send et Recv plutôt que ISend et IRecv utilisées, en quoi ISend et IRecv peuvent-elles être plus efficace pour un programme parallèle? iii) Expliquez la fonction du 5ème argument de la fonction Irecv qui est présentement à 0. Que serait le résultat si cette valeur était plutôt à 1? **(3 points)**

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

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);

  prev = rank - 1;
  next = (rank + 1) % size;
  val = rank * rank;

  for(i = 0; i < 6; 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();
}
```

i) Les sections de code pour lesquelles on demande de fournir la sortie dans une question sont toujours testées au préalable. Une erreur est si vite arrivée. Le programme fourni ici ne fait pas exception et la sortie est conforme à ce qui pourrait être attendu.

```
Rank 0, value 16
Rank 1, value 0
Rank 2, value 1
Rank 3, value 4
Rank 4, value 9
```

Toutefois, un examen minutieux du programme nous révèle que, pour le noeud 0, la valeur de la variable prev devient -1. Or, MPI stipule que la valeur de l'argument source doit être strictement entre 0 et size - 1, ou être MPI_ANY_SOURCE. Par chance, les définitions internes de l'implémentation utilisée assignent la valeur -1 à MPI_ANY_SOURCE. Ainsi, chaque noeud commence avec son rang au carré. Il transmet ensuite cette valeur au suivant (rang + 1 avec retour à 0 pour le dernier noeud). Cette opération est répétée 6 fois. Puisque nous avons 5 noeuds, les valeurs font un tour complet + 1, elles sont donc décalées d'une position au final.

Si l'implémentation avait utilisé une valeur autre que -1 pour MPI_ANY_SOURCE, le comportement aurait été différent. Par défaut, lorsqu'une erreur se présente, le handler *fait terminer tous les processus du programme avec un message d'erreur.*

```
*** An error occurred in MPI_Irecv
*** reported by process [2926575617,0]
*** on communicator MPI_COMM_WORLD
*** MPI_ERR_RANK: invalid rank
*** MPI_ERRORS_ARE_FATAL (processes in this communicator will now abor
***    and potentially your MPI job)
```

Si le handler *utilisé est plutôt MPI_ERRORS_RETURN, la fonction MPI_Irecv aurait retourné le code d'erreur MPI_ERR_RANK et la variable newval serait restée non initialisée (possiblement 0). Cette valeur non initialisée aurait ensuite été propagée aux 4 autres noeuds, ce qui aurait pu donner la sortie suivante.*

```
Rank 0, value 0
Rank 1, value 0
Rank 2, value 0
Rank 3, value 0
Rank 4, value 0
```

Ces 3 réponses alternatives étaient acceptables.

ii) Si on utilise les fonctions Send et Recv, il faudra alterner l'ordre entre les deux (Send / Recv) d'un noeud au suivant pour ne pas avoir d'interblocage. De manière plus générale, les versions asynchrones (immédiates) peuvent être plus efficaces car elles donnent la possibilité de faire autre chose pendant que les communications s'effectuent, donc plus de travail en parallèle plutôt que d'être bloqué en attente des communications.

iii) Le cinquième argument, tag *est une étiquette qui permet de bien apparier les envois et réception correspondants. Si les envois ISend sont fait avec une valeur de 0 et les réceptions IRecv sont changées pour la valeur 1, cela ne fonctionnera plus puisque la réception attendra une étiquette qui ne viendra jamais.*

Question 4 (5 points)

a) Un fil d'éxécution d'un programme s'exécute et, à chaque 1ms, le fil est interrompu très brièvement pour prendre un échantillon avec le compteur de programme et le contenu de la pile d'appel. Ces échantillons sont éventuellement copiés vers un fichier. Chaque échantillon contient donc l'adresse d'exécution courante ainsi que l'adresse dans toutes les fonctions appelantes. A la fin de l'exécution, ces échantillons sont traités pour convertir chacune de ces adresses en identificateur de la fonction dans laquelle elle se trouve. Pour une exécution donnée, voici les échantillons recueillis (les échantillons sont séparés par des ";" avec pour chacun à gauche la

fonction associée au compteur de programme et à droite les fonctions retrouvées dans l'ordre en remontant la pile d'appel. Calculez pour chaque fonction retrouvée dans les échantillons (main, A, B, C, D, E, F) le temps passé dans la fonction elle-même (self) et le temps passé dans la fonction et ses enfants (self+childs). **(2 points)**

```
D C A main; D C A main; E C A main; F D B main;
D B main; D B main; C A main; C B main; main
```

Pour chaque échantillon, on ajoute 1ms de temps d'exécution self *à la fonction la plus à gauche (où était le compteur de programme) et 1ms de temps d'exécution* self+childs *à chaque fonction dans l'échantillon (fonction où était le compteur de programme, et fonctions appelantes retrouvées dans la pile d'appel).*

Fonction	self	self+childs
main	1 = 1ms	1+1+1+1+1+1+1+1+1 = 9ms
A		1+1+1+1 = 4ms
B		1+1+1+1 = 4ms
C	1+1 = 2ms	1+1+1+1+1 = 5ms
D	1+1+1+1 = 4ms	1+1+1+1+1 = 5ms
E	1 = 1ms	1 = 1ms
F	1 = 1ms	1 = 1ms

b) Un programme s'exécute à raison de 1G instruction / seconde (ou 1000 MIPS). Les blocs de base du programme ont une longueur moyenne de 10 instructions. Chaque appel de fonction exécute en moyenne environ 100 instructions. La fonctions mcount, utilisée pour compter le nombre d'appels et leur provenance pour une fonction, nécessite l'exécution de 8 instructions. La prise d'un échantillon du compteur de programme demande 1000 instructions et s'exécute à chaque 1ms si cela est activé. i) Quel sera le surcoût d'utiliser un outil comme gcov qui ajoute l'exécution d'une instruction pour chaque bloc de base? ii) Quel sera le surcoût d'utiliser un outil comme oprofile qui échantillonne le compteur de programme à chaque 1ms? iii) Quel sera le surcoût d'un outil comme gprof qui compte le nombre d'appels de chaque fonction et leur provenance, en plus d'échantillonner le compteur de cycle à chaque 1ms? **(2 points)**

i) Pour gcov, on ajoute une instruction à chaque 10 instructions, pour un surcoût de 1/10 ou 10%.

ii) Pour oprofile, on ajoute 1000 instructions à chaque 1ms, soit 1000000 instructions / seconde ou 1 MIPS. Le programme pourra donc exécuter 1000 MIPS - 1 MIPS = 999 MIPS pour des instructions utiles. Le surcoût est de 1 MIPS / 999 MIPS ou 0.1%.

iii) Avec gprof, on a 8 instructions ajoutées à chaque 100 instructions. En 1s, on se retrouve donc avec 999 MI dont 100 sur 108 sont des instructions utiles, soit 999 MI x 100 / 108 = 925 MI. Le surcoût est donc de 75 MI / 925 MI = 8.1%.

c) Donnez un exemple de problème de performance qui ne pourrait être diagnostiqué facilement avec gprof mais qui pourrait aisément l'être avec oprofile. **(1 point)**

L'outil gprof est limité à un échantillonnage basé sur le temps, il ne peut donc que regarder le temps passé dans chaque fonction. Avec un échantillonnage basé sur les compteurs de performance, le profil généré par oprofile permet de cibler différents paramètres comme les fautes de cache, ou les blocages dans le pipeline associés aux conflits de données ou aux sauts par exemple. Ainsi, oprofile peut rapidement identifier une section du code qui fait des accès inefficaces au niveau de la cache, alors que gprof ne pourrait que montrer le temps passé dans les différentes fonctions, sans savoir si ce temps est relié à des fautes de cache.

Le professeur: Michel Dagenais