Université Bordeaux Master Informatique IN8W16
Examen du mardi 29 avril 2014 Durée : 1h30 – Sans document

Programmation Multicœur et GPU

Barème indicatif : 3 - 3 - 1 - 4 - 2 - 5 - 3 - 4

Optimisation de code séquentiel

Le code suivant place dans `B` la transposée de la matrice carrée `A`.

```
int A[DIM][DIM];
int B[DIM][DIM];
...
for (int i = 0; i < DIM; i++)
   for (int j = 0; j < DIM; j++)
      B[j][i] = A[i][j];
```

On suppose que le cœur dispose d'un cache de 128 ko et dont les lignes font 64 octets (il y a donc 2048 entrées de 64 octets). Un entier est codé sur 4 octets, une ligne de cache contient donc 16 entiers. Ce cache est géré par la politique LRU (évincement de la ligne de cache la moins récemment utilisée). On s'intéresse au nombre de défauts de cache provoqués par l'exécution de cet algorithme à froid (aucune donnée n'est présente dans le cache). On note ce nombre *Défauts(DIM)* pour un tableau d'entiers de dimensions DIM × DIM.

Question 1 Donner une estimation de Défauts(128), Défauts(256) et Défauts(4096). Justifier soigneusement [1].

Question 2 Quelle technique/principe mettre en œuvre pour optimiser cet algorithme ? Illustrer votre réponse en proposant un algorithme de transposition optimal du point de vue cache pour le cas DIM = 4096.

Sommes préfixées en parallèle

Le code suivant place dans le tableau `Prefixe` la somme préfixée du tableau `A` :

```
int A[DIM];
int Prefixe[DIM];
...
Prefixe[0] = A[0];
for (int i = 1; i < DIM; i++)
  Prefixe[i] += Prefixe[i-1]+A[i];
```

Question 3 Montrer sur le cas `A[] = {0,1,2,3,4,5,6,7,8,9,10,11}` pourquoi cette boucle ne peut être parallélisée trivialement à l'aide d'une simple directive OpenMP `parallel for` (considérer deux threads).

Il est cependant possible de paralléliser ce calcul en modifiant l'algorithme de la façon suivante. On définit un tableau annexe dont la dimension est égale au nombre de threads et on distribue les indices aux threads par bloc. Dans une première phase chaque thread place dans le tableau annexe la somme des éléments du bloc de A dont il a la charge. Ensuite le maître calcule directement la somme préfixée du tableau annexe. Enfin les threads utilisent le tableau annexe pour calculer le tableau `Prefixe`.

1. Une matrice carrée de dimensions 128 pèse 64 ko et occupe donc 1k entrées ; une matrice carrée de dimensions 256 pèse 256 ko et occupe $256 \times 256/16 = 256 \times 16 = 4$k entrées ; une matrice carrée de dimensions 4096 pèse 64Mo et occupe 1M entrées.

Exemple : considérons le cas `A[] = {0,1,2,3,4,5,6,7,8,9,10,11}` et quatre threads. Le traitement tableau est réparti entre les threads et chaque thread calcule la somme de sa partie :

thread	bloc attribué	somme du bloc
0	`{0,1,2}`	3
1	`{3,4,5}`	12
2	`{6,7,8}`	21
3	`{9,10,11}`	30

La valeur du tableau annexe est donc `[3,12,21,30]` et le thread maître calcule la somme préfixée de ce tableau : `[3,15,36,66]`. Chaque thread utilise alors le tableau annexe pour obtenir la somme des éléments précédents son bloc pour réaliser le calcul :

thread	valeur utilisée	calcul effectué
0	0	`{0+0,0+1,1+2}`
1	3	`{3+3,6+4,10+5}`
2	15	`{15+6,21+7,28+8}`
3	36	`{36+9,45+10,55+11}`

Question 4 Écrire le code OpenMP correspondant à cet algorithme. On pourra considérer que le nombre de threads vaut `nbthreads` et utiliser la fonction `omp_get_thread_num()` qui retourne le numéro du thread. Bien faire attention au problème du *false sharing*.

Question 5 Estimer, sous forme d'une formule, l'accélération maximale prédite par la loi d'Amdhal. Quelle est cette valeur pour 10 threads et 1 000 000 éléments ?

Question 6 Proposer une parallélisation MPI de cet algorithme. On se contentera d'écrire le code maître en supposant toutefois que celui-ci calcule la première tranche. On trouvera en annexe un florilège de l'API MPI.

OpenCL

On s'intéresse au rendu 3D de particules dans une simulation calculant des interactions entre particules en OpenCL sur accélérateurs graphiques. Comme dans le cadre du projet, les coordonnées des atomes manipulés par la simulation sont stockées dans un tampon `pos_buffer` alloué sur le GPU (Fig. 1).

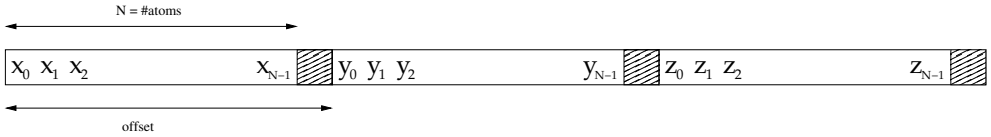

FIGURE 1 – Pour améliorer la performance des accès mémoire sur les accélérateurs, le tableau `pos_buffer` est agencé de la manière illustrée ci-dessus : une première tranche contient les coordonnées x, une seconde contient les y et la troisième contient les z. Chaque tranche occupe `offset` nombres de type *float* en mémoire.

Pour afficher les atomes en leur donnant l'aspect d'une sphère 3D, on décide d'utiliser une fonctionnalité OpenGL qui permet de dessiner une sphère en utilisant un maillage de triangles (voir Fig. 2) dont les coordonnées des sommets doivent être stockées dans un tampon alloué sur le GPU. On notera que les coordonnées d'un sommet (x, y, z) d'un triangle doivent être rangées consécutivement dans ce tampon. On appelera `vertex_buffer` ce tampon. Le nombre de sommets nécessaire pour afficher un atome est égal à `vertices`. En notant N le nombre d'atomes manipulés par la simulation, le tableau `vertex_buffer` contient donc $N \times$`vertices`$\times 3$ coordonnées.
Pour faciliter la réactualisation de ces coordonnées à chaque fois que les atomes bougent (c'est-à-dire que le contenu du tableau `pos_buffer` change), on utilise un troisième tableau `model_buffer` dans lequel on a pré-calculé les coordonnées des sommets d'un atomes qui se trouverait à la position $(0, 0, 0)$. Ce

tableau est constant et alloué sur le GPU à l'initialisation de l'application. Sa taille est de `vertices`×3 coordonnées. À chaque fois que l'on doit rafraîchir l'affichage, il faut donc mettre à jour le tableau `vertex_buffer` en calculant pour chaque atome les coordonnées de ses `vertices` sommets : ceux-ci s'obtiennent en récupérant les coordonnées de chaque sommet du modèle et en lui ajoutant la position de l'atome (i.e. en faisant une translation).

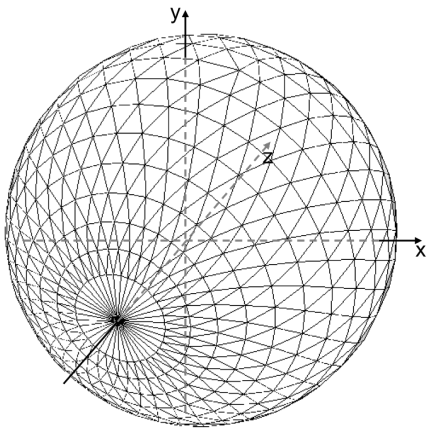

FIGURE 2 – Les atomes sont représentés par un maillage de triangles à `vertices` sommets. Les coordonnées relatives de chacun des sommets (par rapport à un repère orthonormé ayant pour origine le centre de l'atome) sont précalculées et enregistrées dans un tampon `model_buffer`.

Question 7
Voici une proposition de code pour noyau OpenCL `refresh` qui effectue la mise-à-jour des coordonnées contenue dans `vertex_buffer`. Ce noyau est exécuté en lançant un thread par élément (float) de `vertex_buffer`.

```
__kernel
void refresh(__global float *vertex_buffer,
             __global float *pos_buffer,
             __global float *model_buffer,
             unsigned vertices,
             unsigned offset)
{
    unsigned index = get_global_id(0);
    unsigned coord = index % 3;
    unsigned tpa = 3 * vertices;
    unsigned idx = index / tpa;

    vertex_buffer[index] = model_buffer[index % tpa] +
                           pos_buffer[idx + coord * offset];
}
```

Expliquez le rôle des différentes variables locales, et indiquez quels sont les accès mémoire qui risquent de pénaliser les performances. Expliquez.

Question 8 Donnez une version utilisant de la mémoire partagée entre threads. On supposera que l'on peut former des *workgroups* de taille 1024 au maximum. Par ailleurs, on a la garantie que le nombre de sommets par sphère (`vertices`) n'exécède pas 256.

ANNEXES

1 - Envionnement

Initialiser MPI et quitter MPI :

int MPI_Init(int *argc,char*** argv)

int MPI_Finalize(void)

Quitter brutalement MPI :

int MPI_Abort(MPI_Comm comm)

Savoir si un processus à fait un MPI_Init :

int MPI_Initialized(int *flag)

Récupérer la chaine de caractères associée au code d'erreur err:

int MPI_Error_string(int errcode,char *chaine,int *taille_chaine)

2 - Communications point à point bloquantes

Envoyer un message à un processus :

int MPI_[R,S,B]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)

Recevoir un message d'un processus :

int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Status *status)

Envoyer et recevoir un message :

int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype sendtype,int dest,int sendtag,void *recvbuf,int recvcount,MPI_Datatype recvtype,int source,int recvtag,MPI_Comm comm,MPI_Status *status)

Compter le nombre d'éléments reçus :

int MPI_Get_count(MPI_status *status,MPI_Datatype datatype,int *count)

Tester l'arrivée d'un message :

int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status)

3 - Communications point à point non bloquantes

Commencer à envoyer un message :

int MPI_I[r,s,b]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request * request)

Commencer à recevoir un message :

int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request)

Compléter une opération non bloquante :

int MPI_Wait(MPI_Request *request,MPI_Status *status)

Tester une opération non bloquante :

int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status)

Libérer une requête avant de la réutiliser :

int MPI_Request_free(MPI_Request *request)

MPI_Test checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

int MPI_Testany(int count,MPI_Request *array_of_requests,int *index, int *flag,MPI_Status *status)

int MPI_Testall(int count,MPI_Request *array_of_requests, int *flag,MPI_Status *array_of_statuses)

int MPI_Testsome(int incount,MPI_Request *array_of_requests,
int *outcount,int *array_of_indices, MPI_Status *array_of_statuses)

int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag, MPI_Status *status)

int MPI_Cancel(MPI_Request *request);

MPI_Waitall(), MPI_Cancel MPI_Iprobe(), MPI_Test_cancelled()

4 - Communications persistantes

Décrire un schéma persistant :

int MPI_[R,S,B]send_init(void *sendbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)

int MPI_Recv_init(void *recbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)

Démarrer une communication persistante :

int MPI_Start(MPI_Request *request)

Autres fonctions

MPI_Startall(), MPI_Request_free()

5 - Communications collectives

Barrière :

int MPI_Barrier(MPI_Comm comm)

Diffusion générale d'un message :

int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root,MPI_Comm comm)

Collecte de données :

int MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)

Diffusion sélective d'un message :

int MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)

Collecte de données et rediffusion :

int MPI_Alltoall(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm)

Calcul d'une réduction :

int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,int root,MPI_Comm comm)

Calcul d'une réduction et rediffusion du résultat :

int MPI_Allreduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,MPI_Comm comm)

Opérateurs de MPI_Reduce et MPI_Allreduce :

MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR

6 - Constantes

Jokers :

MPI_ANY_TAG, MPI_ANY_SOURCE

Datatypes élémentaires :

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE,
MPI_UNSIGNED, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT ,MPI_UNSIGNED_LONG,
MPI_LOGICAL, MPI_BYTE, MPI_PACKED

Constantes réservées :

MPI_PROC_NULL, MPI_UNDEFINED

Communicateurs reservés :

MPI_COMM_WORLD, MPI_COMM_SELF

7 – MPI_Status

status.source status.tag status.error status.length status.size status.byte