Equipe 11

Jean LE BERRE

Simon ROBIDAS

TP4 OpenCL/OpenACC

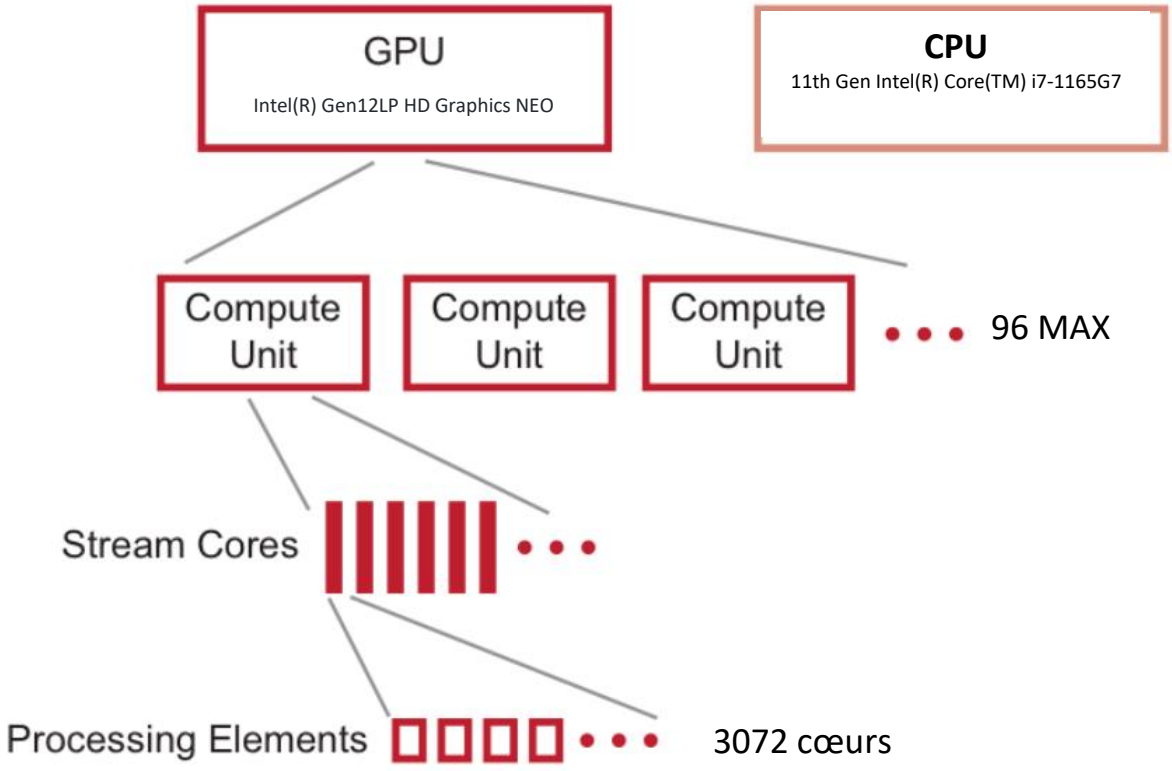
07/04/2022

Introduction

Le but de ce TP est d'inverser une matrice de taille n par l'algorithme de Gauss-Jordan en utilisant OpenCL puis OpenACC dans un second temps. On part du programme disponible sur le dépôt Git suivant : GitHub - ghazpar/PPD-H22: GIF-4104/7104 - Travaux pratiques, qui propose une forme de donnée Matrix et implémente la méthode séquentiellement. Il nous reste à coder la fonction invertParallel en utilisant MPI pour partager le problème entre différents processus.

Notre Matériel

Processeur : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
GPU : Intel(R) Gen12LP HD Graphics NEO
Nombre de cœurs physiques : 4
Nombre de cœurs logiques : 8
Mémoire RAM : 16,0 Go (15,7 Go utilisable)
OS : Ubuntu 20.04.3 LTS



Max work item dimensions	3
Max work item sizes	512x512x512
Max work group size	512
Local memory size	65536 (64KiB)

La GPU comprend 96*36 =3072 cœurs.

OpenCL

Notre Travail

Nous avons tout d'abord repris l'initialisation de OpenCL disponible dans le cours pour qu'OpenCL puisse reconnaitre son environnement.

Nous avons ensuite inséré invertParallel, voici le pseudo-code de la partie qui nous intéresse :

```
Variable :

lA = Matrice à inverser
lAI = Matrice à inverser | matrice identitée
n = nombre de lignes

Pseudo-Code :

Créer un kernel à partir de la fonction inverse_pass

Remplir un clBuffer pour cols_buffer
Remplir un clBuffer pour rows_buffer

Pour k allant de 0 à n-1, Faire :
    Trouver le max de la colonne k et son index
    Normaliser rowPivot
    Echanger la ligne du pivot avec celle de l`index max

    Remplir le clBuffer avec les données de la matrice lAI
    Remplir le clBuffer avec les données de rowPivot
    Remplir le clBuffer avec k

    Pousser les arguments (IAI, rowPivot, k, rows, cols) dans le kernel

    Lancer le kernel

    Récupérer les données de la matrice dans IAI

lA <- la matrice inverse, la partie droite de lAI
```

Avec la fonction du kernel semblable à :

```
__kernel void inverse_pass( __global double *data,
                            __global  double *pivot,
                            __constant  long *k,
                            __constant  long *cols,
                            __constant  long *rows) {

      int idx = get_global_id(0);

      if(idx < *rows) {
            if (idx != *k) {
                  double lValue = data[idx * *cols + *k];
                  for (int j = 0; j < *cols; j++) {
                        data[idx * *cols + j] -= pivot[j] * lValue;
                  }
            }
      }
}
```

Resultat

On a lancé 5 fois le programme avec n=1500. Voici les résultats :

./tp4_opencl 1500					
	Tseq	Tpar	Speed-Up	Efficacité	p
Mean	34,6358	12,76872	2,723319201	0,000886209	3073

Avec p, le nombre de cœurs utilisés, composé des 96*32 cœurs de la carte graphique.

D'après nos observations le « speed-up » est médiocre et l'efficacité est encore plus terrible.

OpenACC

Notre Travail

Voici le pseudo-code de la fonction invertParallel :

Variable :

```
lA = Matrice à inverser
lAI = Matrice à inverser | matrice identitée
data_lAI = tableau 1D des valeurs de lAI
n = nombre de lignes
```

Pseudo-Code :

```
Pour k allant de 0 à n-1, Faire :
    #pragma acc parallel loop copyin(data_lAI) reduction(max:value){
        Trouver le max de la colonne k
    }
    Trouver l`index du max

    #pragma acc parallel loop copy(data_lAI) copyout(rowPivot[0:cols]){
        Normaliser rowPivot
    }
    Echanger la ligne du pivot avec celle de l`index max

    #pragma acc parallel loop copy(data_lAI) copyin(rowPivot[0:cols])
gang worker
        Pour i allant de 0 à n-1, Faire :
#pragma acc loop vector
            Si k différent de i, Faire :
                alpha <- lAI[i][k]/rowPivot[k]
                lAI[i] <- lAI[i] - alpha*rowPivot
            Fin Si
        Fin Pour
    }
}
lA <- la matrice inverse, la partie droite de lAI
```

Resultat

On a lancé 5 fois le programme avec n=1500. Voici les résultats :

./tp4_openacc 1500					
	Tseq	Tpar	Speed-Up	Efficacité	p
Moyenne	30,83396	20,1354	1,613513223	0,00052506	3073

Encore une fois, l'efficacité est médiocre.

Quelle taille de workgroup avez-vous choisi et pourquoi?

On a défini la taille des workgroup comment étant la valeur de `CL_KERNEL_WORK_GROUP_SIZE` qui est dans notre cas 256. On a également testé plusieurs autres valeurs 16, 32, 64, cependant on n'a pas réussi à remarquer une amélioration significative.

Explicitez notamment ce que vous avez fait pour minimiser les transferts de données entre le «host» et le «device»?

Malheureusement, on n'a pas été capable de minimiser les transferts de données entre le « host » et le « device ». C'est probablement la plus grande raison qui fait que notre programme est très lent. Pour pouvoir minimiser les transferts de données il aurait fallu changer notre stratégie et faire la totalité des traitements dans les « kernel » GPU. Afin de conserver la mémoire à travers les exécutions, il aurait fallu faire appel à la clause « present » de OpenACC.

En effet, on a calculé, pour opencl par exemple, que les échanges entre « host » et « device » mettait 10 secondes sur les 14 secondes d'exécution.

Explicitez les niveaux de granularité que vous avez utilisé («gang», «worker» et «vector»)

Dans notre solution OpenACC nous avons employés 3 boucles parallélisées. Nous avons divisé une de celle-ci en « gang worker » et « vector ». L'objectif étant de diminuer le déplacement de mémoire. Nous voulons partager la valeur associée au traitement « k » afin de l'utiliser pour faire l'application du pivot sur chaque élément de la ligne.

Comment se compare vos deux solutions en terme de performance (OpenCL/CUDA vs OpenACC)?

En termes de performance, OpenACC et OpenCL sont tous les deux terribles, on a une légère amélioration avec OpenCL cependant, considérant ce que les « speed up » devraient être, c'est plutôt marginal.

Dans les deux cas, on fait énormément de copie de la mémoire hôte vers la mémoire de la carte graphique inutilement. Comme exprimé plutôt, autour de 70% du temps est à faire le transfert entre les mémoires. Afin de pallier cette problématique, on devrait revoir la modélisation de notre problème afin de limiter les transferts de mémoire. Pour ce faire, on devrait faire l'intégralité des traitements au niveau GPU : Trouver le pivot, Échanger la ligne pivot, Appliquer le pivot. De cette manière, même si on doit synchroniser les processus souvent, on n'a pas besoin de faire transfert de mémoire durant l'exécution de ceux-ci. Autrement dit, sauvegarder au niveau de la mémoire de chaque « workgroup » les n lignes dont il s'occupe. Puis de les renvoyer une fois tous les traitements effectués à la mémoire du « host ».

De plus dans les deux cas, le niveau de parallélisation est plutôt faible. On parallélise chacune des lignes, mais on aurait pu également faire une parallélisation sur les colonnes.

Annexes

Temps d'exécution en seconde de la version implémentée en OpenCL avec n=1500

./tp4_opencl 1500					
	Tseq	Tpar	Speed-Up	Efficacité	p
1	37,3999	14,512	2,577170617	0,00083865	3073
2	35,3485	11,9846	2,949493517	0,000959809	3073
3	35,1429	11,9402	2,943242157	0,000957775	3073
4	33,7096	12,3891	2,720907895	0,000885424	3073
5	31,5781	13,0177	2,42578182	0,000789386	3073
Mean	34,6358	12,76872	2,723319201	0,000886209	3073

Temps d'exécution en seconde de la version implémentée en OpenCL avec n variant

./tp4_openacc <n>				
n	Tpar	$n(n^2+n)$	alpha	$alpha*n(n^2+n)$
1	0,00048953	2	3,45472E-09	6,90944E-09
10	0,00083346	1100	3,45472E-09	3,80019E-06
100	0,0163678	1010000	3,45472E-09	0,00348927
500	0,50789	125250000	3,45472E-09	0,432703993
1000	3,7927	1001000000	3,45472E-09	3,458177222
1500	12,6993	3377250000	3,45472E-09	11,66746156
2000	28,2941	8004000000	3,45472E-09	27,65159889
2500	55,6609	1,5631E+10	3,45472E-09	54,00163108
3000	93,3086	2,7009E+10	3,45472E-09	93,3086

Temps d'exécution en seconde de la version implémentée en OpenACC avec n=1500

./tp4_openacc 1500					
	Tseq	Tpar	Speed-Up	Efficacité	p
1	29,2916	16,4641	1,779119417	0,00057895	3073
2	28,5153	20,9631	1,360261603	0,00044265	3073
3	31,905	16,2659	1,961465397	0,00063829	3073
4	34,8281	30,8665	1,128346265	0,00036718	3073
5	29,6298	16,1174	1,838373435	0,00059823	3073
Moyenne	30,83396	20,1354	1,613513223	0,00052506	3073

Temps d'exécution en seconde de la version implémentée en OpenACC avec n variant

./tp4_openacc <n>			
n	Tpar	$n(n^2+n)$	Alpha=9,90677E-09
1	0,00054026	2	1,9812E-08
10	0,00059546	1100	1,08966E-05
100	0,00641368	1010000	0,01000506
500	0,561868	125250000	1,2407265
1000	6,0991	1001000000	9,915906
1500	15,6794	3377250000	33,4550385
2000	45,33	8004000000	79,287624
2500	98,5443	15631250000	154,8431625
3000	267,572	27009000000	267,551154