Introduction à OpenACC

Amina Guermouche

Introduction

- OpenACC est un standard GPU open source
- OpenACC n'est pas limité aux GPUs NVIDIA
- Utilisation simple grâce à des directive :
  - C/C++ : `#pragma acc kernels [clauses]`
  - Fortran : `!$acc directive [clauses]` et `!$acc end directive`

  $\Rightarrow$ Semblable à OpenMP
- Compilation :
  - `nvc -acc toto.c -o toto`
  - Plus d'information à la compilation avec :
    `nvc -acc -Minfo=accel toto.c -o toto`

Un peu de terminologie pour commencer

- Worker : L'unité de base (comme le thread)
- Gangs (groups) : Groupe de worker. Les gangs sont indépendants. Les gangs sont équivalents aux blocks.

Fonctionnement de base

```
main()
{
  // code sequentiel
#pragma acc kernels
  {
   //code parallele sur GPU
        (automatiquement)
  }
  // code sequentiel
}
```

- La directive dit au compilateur de générer du code GPU
- Le compilateur détermine le transfert des données

Fonctionnement de base

```
int main(int argc, char **argv){
  int N=1000;
  float a = 3.0f;
  float x[N], y[N];
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }
#pragma acc kernels
  for (int i = 0; i < N; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
main:
     10, Generating implicit copy(y[:1000]) [if not already present]
         Generating implicit copyin(x[:1000]) [if not already present]
     12, Loop is parallelizable
         Generating Tesla code
         12, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Ce n'est pas si simple!!!!

- Le code n'est parallélisé que s'il n'y a pas de dépendance des données

```
int main(int argc, char **argv){
  int N=1000;
  float a = 3.0f;
  float x[N];
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
  }
#pragma acc kernels
  for (int i = 0; i < N - 1; ++i) {
    x[i] = a * x[i+1];
  }
}
```

```
main:
     10, Generating implicit allocate(x[:1001]) [if not already present]
         Generating implicit copyin(x[1:1000]) [if not already present]
         Generating implicit copyout(x[:1000]) [if not already present]
     12, Loop carried dependence of x prevents parallelization
         Loop carried backward dependence of x prevents vectorization
         Accelerator serial kernel generated
         Generating Tesla code
         12, #pragma acc loop seq
     12, Loop carried backward dependence of x prevents vectorization
```

Ce n'est pas si simple!!!!

- Le code n'est parallélisé que s'il n'y a pas de dépendance des données
- Le code est séquentiel

```
int main(int argc, char **argv){
  int N=1000;
  float a = 3.0f;
  float x[N];
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
  }
#pragma acc kernels
  for (int i = 0; i < N - 1; ++i) {
    x[i] = a * x[i+1];
  }
}
```

```
main:
     10, Generating implicit allocate(x[:1001]) [if not already present]
         Generating implicit copyin(x[1:1000]) [if not already present]
         Generating implicit copyout(x[:1000]) [if not already present]
     12, Loop carried dependence of x prevents parallelization
         Loop carried backward dependence of x prevents vectorization
         Accelerator serial kernel generated
         Generating Tesla code
         12, #pragma acc loop seq
     12, Loop carried backward dependence of x prevents vectorization
```

Les dépendances

```
int main(int argc, char **argv){
  int N=1000;
  float a = 3.0f;
  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }
#pragma acc kernels
  for (int i = 0; i < N; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
    14, Complex loop carried dependence of x-> prevents parallelization
        Loop carried dependence of y-> prevents parallelization
        Loop carried backward dependence of y-> prevents vectorization
        Accelerator serial kernel generated
        Generating Tesla code
        14, #pragma acc loop seq
```

Les dépendances : L'aliasing

- Deux pointeurs qui semblent différents, peuvent pointer sur un espace dans un même tableau (c'est le fonctionnement en C/C++, le GPU n'y est pour rien).

```
void compute(double *a, double *b, double *c) {
  for (i=1; i<N; i++) {
    a[i] = b[i] + c[i];
  }
}
void main(){
  compute(p, p-1, q);//a et b sont s'overlap donc dans
      le tableau
}
```

source : https://cvw.cac.cornell.edu/vector/coding_aliasing

Les dépendances : L'aliasing

- L'utilisation du mot clé `restrict` permet d'indiquer que le pointeur y sera le seul à accéder au tableau (c'est une promesse faite par le programmeur au compilateur)

```
int main(int argc, char **argv){
  int N=1000;
  float a = 3.0f;
  float *x = (float*)malloc(N * sizeof(float));
  float * restrict y = (float*)malloc(N * sizeof(float)
      );
  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }
#pragma acc kernels
  for (int i = 0; i < N; ++i) {
    y[i] = a * x[i] + y[i];
  }
}
```

La parallélisation avec `parallel`

```
#pragma acc parallel
    for(int i = 0; i < N; i++)
      y[i] = a * x[i] + y[i];
```

- `parallel` dit au compilateur de créer une région parallèle. Mais contrairement à `kernel`, tous les threads exécutent le même code (sans partager le travail).

```
#pragma acc parallel loop
    for(int i = 0; i < N; i++)
      y[i] = a * x[i] + y[i];
```

- Il est nécessaire d'ajouter le mot clé `loop` (ou `for`) pour partager le travail

Les différentes parallélisations

```
#pragma acc kernels
  {

   for(i = 0; i < n; i++)
     a[i] = x * (i + 1);

   for(i = 0; i < n; i++)
     b[i] = y * a[i];
  }
```

- 2 kernels sont automatiquement lancés
- Il y a une barrière implicite entre les deux kernels

```
#pragma acc parallel
  {
#pragma acc loop
     for(i = 0; i < n; i++)
       a[i] = x * (i + 1);
#pragma acc loop
     for(i = 0; i < n; i++)
     b[i] = y * a[i]
  }
```

- 1 seul kernel
- Il n'y a pas de barrière entre les 2 boucles

`kernels` *vs* `parallel`

`kernels`

- Plus implicite
- Le compilateur a plus de liberté pour détecter le parallélisme

`parallel`

- Plus explicite
- Nécessite une analyse du code (par l'utilisateur) pour détecter le parallélisme
- Facilité de passer d'un code OpenMP

Exercice 7-1

La réduction

- La ligne 38 du fichier d'origine montre une variable à laquelle tous les threads vont accéder.
- Le compilateur est assez malin pour détecter qu'il s'agit d'une réduction, et la génère lui-même (comme vous le montre la sortie de votre ligne de compilation).
- La réduction avec `loop` n'est pas automatique (contrairement à `kernel`). Pour la déclarer explicitement :

```
#pragma parllel loop reduction(max:dt) // indique
    qu'il y a une reduction sur la variable dt,
    avec max comme operation effectuee
```

Les copies mémoires implicites

- Les performances avec OpenACC sont moins bonnes qu'en séquentiel.
- ☹ Beaucoup de temps est passé dans la mémoire (0.2s pour le kernel contre 3s pour la copie vers le GPU)

```
Accelerator Kernel Timing data
/autofs/unityaccount/cremi/amguermouche/coursCuda/openACC/exo8/laplace_acc.c
  main  NVIDIA  devicenum=0
    time(us): 17,127,250
    29: compute region reached 3372 times
        30: kernel launched 3372 times
            grid: [7813]  block: [128]
             device time(us): total=142,103 max=52 min=41 avg=42
            elapsed time(us): total=220,046 max=722 min=58 avg=65
    29: data region reached 6744 times
        29: data copyin transfers: 3372
             device time(us): total=3,422,100 max=1,042 min=978 avg=1,014
        34: data copyout transfers: 3372
             device time(us): total=3,227,134 max=1,134 min=880 avg=957
    38: compute region reached 3372 times
        39: kernel launched 3372 times
            grid: [7813]  block: [128]
             device time(us): total=203,516 max=75 min=59 avg=60
            elapsed time(us): total=268,661 max=239 min=75 avg=79
        39: reduction kernel launched 3372 times
            grid: [1]  block: [256]
             device time(us): total=36,657 max=14 min=10 avg=10
            elapsed time(us): total=77,796 max=233 min=21 avg=23
    38: data region reached 6744 times
        38: data copyin transfers: 10116
             device time(us): total=6,826,763 max=1,046 min=6 avg=674
        43: data copyout transfers: 6744
             device time(us): total=3,268,977 max=1,135 min=5 avg=484
```

Les copies mémoires implicites

- Les performances avec OpenACC sont moins bonnes qu'en séquentiel.
- Il y a des copies implicites avant et après chaque section parallèle (indiquées à la compilation)
- Remarque : Avec `kernels`, les données sont copiées une seule fois par région parallèle (même si celle-ci contient plusieurs boucles). Les données restent sur le GPU pendant toute la section parallèle. Avec `parallel loop`, le compilateur peut faire des copies entre deux boucles successives.

```
#pragma acc kernels
  for(i = 1; i <= ROWS; i++) {
    for(j = 1; j <= COLUMNS; j++) {
      A[i][j] = 0.25 * (A_last[i+1][j] + A_last[i-1][j] +
                        A_last[i][j+1] + A_last[i][j-1]);
      } // A_last doit etre copie sur le GPU. A doit etre copie depuis
          le GPU
    }
   dt = 0.0;
#pragma acc kernels
  for(i = 1; i <= ROWS; i++){
    for(j = 1; j <= COLUMNS; j++){
      dt = fmax( fabs(A[i][j]-A_last[i][j]), dt); //A et A_last doivent
           etre copies sur le GPU
      A_last[i][j] = A[i][j]; // A_last doit etre copie sur le CPU
      }
    }
```

La gestion de la mémoire avec OpenACC

- copy : La mémoire est allouée sur le GPU. Les données sont transférée du host vers le GPU à l'entrée de la réunion parallèle, et copiées vers le CPU à la sortie de la région parallèle.
- copyin : La mémoire est allouée sur le GPU. Les données sont transférées du host vers le device à l'entrée de la région parallèle.
- copyout : La mémoire est allouée sur le GPU. Les données sont transférées du GPU vers le host à la sortie de la région parallèle.
- create : La mémoire est allouée sur le GPU. Aucune donnée n'est copiée.
- present : La donnée est déjà présente sur le GPU

La gestion de la mémoire avec OpenACC

- Syntaxe : `#pragma acc data copy (a) copyin(b), copyout(c) create(d) present(d)`
- Le compilateur peut déterminer la taille du tableau. Dans ce cas pas la peine de la préciser.
- Si l'on veut spécifier ce qu'on veut manipuler, la syntaxe est la suivante : `copy (a[start:count])`

Comment bien définir les régions des données

```
int main(int argc, char** argv
    ){
  float A[1000];


#pragma acc kernels // A est
    copie sur le GPU
  for(int i = 1; i < 1000; i
      ++){
    A[i] = 1.0;
  }// A est copie sur le host

  A[10] = 2.0; //s'execute sur
       le Host

 printf(''A[10] = %f", A[10]);
```

Affiche : A[10] = 2.0

```
int main(int argc, char** argv
    ){
  float A[1000];
#pragma acc data copy(A)
 {
#pragma acc kernels // A est
    copie sur le GPU
  for(int i = 1; i < 1000; i
      ++){
    A[i] = 1.0;
  }// A est copie sur le host

  A[10] = 2.0; //s'execute sur
       le Host
 }// A est copie sur le CPU
 printf(''A[10] = %f", A[10]);
```

Affiche : A[10] = 1.0

Exercice 7-2

Performances avec une meilleure gestion de la mémoire

Il n'y a plus que 4 copies ! ! !

```
Accelerator Kernel Timing data
/autofs/unityaccount/cremi/amguermouche/coursCuda/openACC/exo8/laplace_acc_Q2.c
  main  NVIDIA  devicenum=0
    time(us): 953,221
    28: data region reached 2 times
        28: data copyin transfers: 1
             device time(us): total=649 max=649 min=649 avg=649
        51: data copyout transfers: 1
             device time(us): total=702 max=702 min=702 avg=702
    31: compute region reached 3372 times
        32: kernel launched 3372 times
            grid: [7813]  block: [128]
             device time(us): total=362,112 max=129 min=105 avg=107
            elapsed time(us): total=397,360 max=172 min=115 avg=117
    40: compute region reached 3372 times
        41: kernel launched 3372 times
            grid: [7813]  block: [128]
             device time(us): total=533,471 max=187 min=155 avg=158
            elapsed time(us): total=568,686 max=234 min=165 avg=168
        41: reduction kernel launched 3372 times
            grid: [1]  block: [256]
             device time(us): total=35,779 max=13 min=10 avg=10
            elapsed time(us): total=66,456 max=95 min=18 avg=19
    40: data region reached 6744 times
        40: data copyin transfers: 3372
             device time(us): total=6,858 max=6 min=1 avg=2
        45: data copyout transfers: 3372
             device time(us): total=13,650 max=12 min=3 avg=4
```

Quand gérer la mémoire ?

- Le compilateur agit de façon conservative. C'est à vous de lui forcer la main avec la gestion de données comme vous la voyez.
- Il faut bien analyser le code pour savoir quand la donnée doit être transférée, et voir si c'est mieux que le comportement par défaut.
- Il est possible de forcer la mise à jour des données (pour que le code soit encore plus asynchrone, mais avec un contrôle) :
  - `#pragma acc update host(Temperature)`

OpenACC vs CUDA

- Faible modification du code (et similarité avec OpenMP)

$\Rightarrow$ Facilité de lecture du code pour des non initiés à CUDA

- L'architecture reste cachée (plus besoin de connaître le nombre de blocks, de threads, ...)