Systèmes distribués et grilles : TP CUDA

Thibaut EHLINGER, Benjamin HERB *Université de Strasbourg*

19 septembre 2016

Dans ce TP nous devions exécuter des produits matriciels, qui sont des opérations mathématiques très gourmandes, avec des complexité algorithmique de l'ordre de $\mathcal{O}(n^3)$. Afin de mettre pleinement à disposition les capacités de nos machines, nous avons donc exécuté les calculs cette fois sur la carte graphique de nos ordinateurs, grâce à la technologie CUDA (Compute Unified Device Architecture). Comme pour le TP précédent, il s'agit d'un produit de matrices 4096×4096. Il est à noter que sur les cartes graphiques des machines de la salle J4 , des **gtx680, architecture Kepler**, nous n'avions pas le droit de lancer des calculs trop longs. Pour des raisons de sécurité, sans doute, il nous était impossible de réaliser des calculs dépassant les 11-12 secondes. Néanmoins cela ne nous a pas empêchés d'obtenir des résultats concluants, étant donné que de toute manière nous cherchions à optimiser le temps de calcul.

Contexte : des dilemmes de pavage

L'intérêt du calcul distribué sur une carte graphique est de mettre à profit la puissance de calcul d'une GPU. En revanche, les GPU sont optimisées pour les calculs dits *SIMD*, pour *single instruction - multiple data*, ce qui veut dire qu'il est très simple pour une carte graphique de réaliser moult fois la même instruction sur un grand jeu de données diversifiées. Le calcul matriciel correspond très bien à ce genre de calculs.

Une GPU s'organise en blocs de *threads*, aussi appelés *warps*. Une première contrainte est que le nombre de *threads* par bloc ne peut dépasser 1024. On est aussi limité pour le nombre total de blocs, mais cette limite ne nous a pas concerné pour notre TP, celle-ci allant de 2^{16} à $2^{31} - 1$.

Le dilemme exprimé dans ce TP.

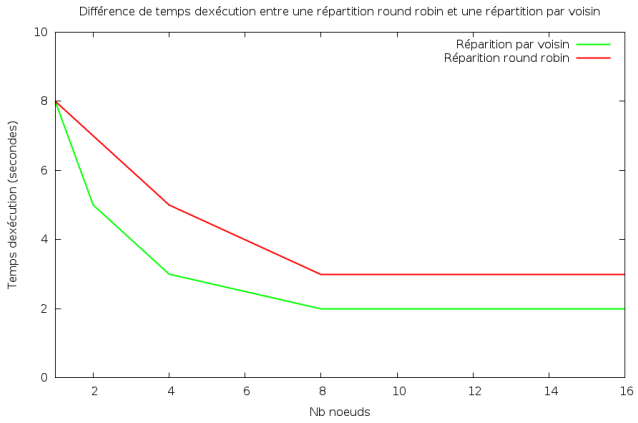

Figure 1 – *Comparaison du temps d'exécution en round robin contre voisins pour 1 et 2 threads. Les performances sont évidemment les mêmes pour un seul nœud.*

Conditions d'expérimentation

Les mesures de l'expérimentation n'ont été effectuées qu'une seule fois. Il est donc claire que ces résultats sont qualitatifs et non quantitatifs. Ce rapport sert surtout à mettre en avant un raisonnement. Les conclusions sont à prendre avec précautions et à revérifier expérimentalement sur plus d'essais.
Nous nous sommes basés sur le temps d'exécution de la boucle pour les mesures de temps.

Kernel 1 contre kernel 0

Round robin contre une répartition voisine des tâches

Initialisation MPI

```
1 #include <stdio.h>
  #define N 10
3
  /* Initialisation of processor coordinates */
5
  void ProcessorInit(void){
7   MPI_Comm_size(MPI_COMM_WORLD,&NbPE);
      MPI_Comm_rank(MPI_COMM_WORLD,&Me);
9 }
```

Calcul de l'offset pour un processus `Me` donné

```
/* Compute the current step offset, in the MPI program, to access
     right C lines */ OffsetStepLigC = ((Me + step) * LOCAL_SIZE)
    % SIZE;
```

Boucle principale

```
void ComputationAndCirculation()
{
 unsigned long step;

 for(step=0;step<NbPE;step++) {
  OneLocalProduct(step);
  OneStepCirculation(step);
 }
}
/* Elementary circulation of A and B.
                        */
void OneStepCirculation(unsigned long step)
{
 MPI_Status   status;

 MPI_Sendrecv_replace(A_Slice, SIZE*LOCAL_SIZE, MPI_DOUBLE, (Me
    -1+NbPE)%NbPE, 0, (Me+1)%NbPE, 0, MPI_COMM_WORLD, &status);
}
```