Systèmes distribués et grilles : TP CUDA

Thibaut EHLINGER, Benjamin HERB *Université de Strasbourg*

19 septembre 2016

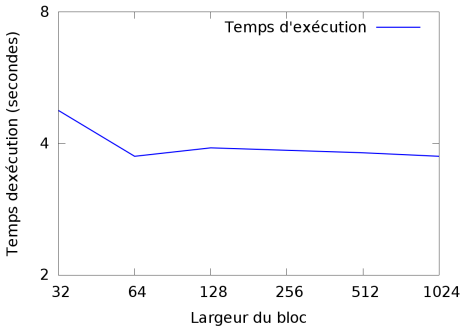
Dans ce TP nous devions exécuter des produits matriciels, qui sont des opérations mathématiques très gourmandes, avec des complexité algorithmique de l'ordre de $\mathcal{O}(n^3)$. Afin de mettre pleinement à disposition les capacités de nos machines, nous avons donc exécuté les calculs cette fois sur 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 empêché d'obtenir des résultats concluants, étant donné que de toute manière nous cherchions à optimiser le temps de calcul.

Dans ce rapport, nous expliquerons tout d'abord les problématiques de dimensionnement des blocs de *threads* et leurs conséquences. Puis, nous étudierons les différents temps et puissances de calculs obtenus en fonction du dimensionnement des blocs, sur le kernel 2. Enfin, nous traiterons les expérimentations effectuées sur le kernel 4, en nous concentrant sur la mémoire partagée : comment l'utiliser au mieux et les bienfaits de son utilisation.

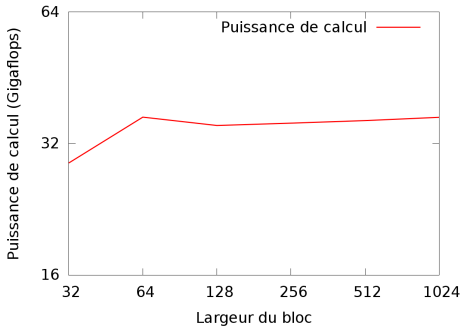
1 Contexte

1.1 Environnement d'expérimentation

Comme dit précédemment, nous travaillons ici sur une carte graphique **gtx680, architecture Kepler**. Les données ont été prélevées une seule fois, puis notées sur un fichier `Excel` prévu à cet effet. Ensuite, nous avons converti ces donnés au format `csv`, qui ont ensuite été représentées graphiquement avec `Gnuplot`. Parallèlement, nous avons aussi intégré dans ce rapport un graphique qui était déjà généré dans la feuille `Excel`.



(a) *Temps de calcul en fonction de la largeur de bloc.*



(b) *Puissance de calcul en fonction de la largeur de bloc.*

Figure 1 – *Temps et puissance de calcul en fonction de la largeur du bloc 1D.*

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 exactement à ce genre de calculs.

Une GPU s'organise en blocs de *threads*, aussi appelés *warps*. Chaque *thread* est en charge d'une toute partie du calcul. En effet ces *threads* ne sont pas faits pour se comporter comme des *threads* POSIX : ils sont conçus pour réaliser des calculs très très simples. Ici, par exemple, un *thread* est en charge d'une seule et unique case de la matrice résultat à la fois. Ce qui veut dire qu'il gère la somme de $Nb_{colonnes}(\mathcal{A}) \times (Nb_{lignes}\mathcal{B})$ produits de `double`.

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 mis en exergue dans ce TP concerne le dimensionnement de ces blocs : en effet, on peut subdiviser les matrices à multiplier entre elles en blocs de différentes tailles, ce qui a différentes conséquences. Les questions auxquelles nous essaierons de répondre sont :

— Sur une seule dimension, quelle semble être la largeur optimale ?
— Sur deux dimensions, quel(s) facteur(s) influe le plus sur les différentes puissances/durées de calcul ?

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);
}
```