TP2 de programmation CUDA
(utilisation des registres et du cache générique et optimisation par *shared memory*)

Par Marek Felsoci et Aurélien Rausch

15 février 2019

Introduction

Le but de ce travail pratique était d'étendre un programme de multiplication de matrices carrées, en utilisant la technologie CUDA, de façon à ce que les calculs soient effectués par le processeur de la carte graphique (GPU) plutôt que par le processeur central (CPU). Notre implémentation comprend plusieurs noyaux de calcul dont nous avons étudié les performances pour différentes dimensions de blocs de threads CUDA.

Pour nos expérimentations, nous avons considéré deux matrices de taille 4096×4096 composées de nombres à virgule flottante. Ce document rend compte des résultats obtenus au cours d'une série de mesures de performances pratiquées sur un ordinateur de la salle J4 (`pcj411`) disposant d'une carte graphique *nVidia* GeForce GTX 680.

1 Méthode d'expérimentation

Trois noyaux de calcul ont été expérimentés et mesurés avec, pour chacun d'eux, des dimensions de grille différentes. Nous cherchons ici à mesurer le temps d'exécution (en secondes) ainsi que le nombre d'opérations à virgule flottante (flops) nécessaires à la multiplication de matrices.

À partir des résultats de l'exécution utilisant uniquement le processeur central (CPU) de l'ordinateur nous avons pu déterminer un speed-up pour chaque exécution du calcul effectué à l'aide de la carte graphique.

Aussi, et pour nous assurer de la cohérence des données récoltées, chaque expérimentation a été répétée une dizaine de fois à l'aide d'un script BASH afin de pouvoir établir, pour chaque résultat, une valeur minimale, maximale, moyenne et médiane.

Voici la liste des noyaux de calcul que nous avons expérimentés :

- **Noyau K1** : découpage en blocs de threads à UNE dimension ; utilisation des **registres** et du **cache générique** uniquement
- **Noyau K2** : découpage en blocs de threads à DEUX dimensions ; utilisation des **registres** et du **cache générique** uniquement
- **Noyau K4** : découpage en blocs de threads à DEUX dimensions ; optimisation en utilisant la **mémoire partagée**

2 Détails de l'implémentation

La première étape de l'extension du programme initial consistait en l'implémentation des fonctions assurant le transfert des données des matrices opérandes vers des symboles GPU (voir le listage 1) puis la récupération des données du résultat de calcul depuis les symboles GPU (voir le listage 2). La macro `SIZE` indique la taille des dimensions des matrices, soit 4096 dans le cas de nos expérimentations.

Listage 1 – Fonction de transfert de données vers GPU

```
void gpuSetDataOnGPU(void) {
  // Set GPU_A symbol
  CHECK_CUDA_SUCCESS(cudaMemcpyToSymbol(
     GPU_A, &A[0][0],
     (SIZE * SIZE) * sizeof(T_real),
     0, cudaMemcpyHostToDevice
   ), "Transfer␣A␣-->␣GPU_A");

  // Set GPU_B symbol
  CHECK_CUDA_SUCCESS(cudaMemcpyToSymbol(
     GPU_B, &B[0][0],
     (SIZE * SIZE) * sizeof(T_real),
     0, cudaMemcpyHostToDevice
   ), "Transfer␣B␣-->␣GPU_B");
}
```

Listage 2 – Fonction de transfert de données depuis GPU

```
void gpuGetResultOnCPU(void) {
  // Get GPU_C symbol
  CHECK_CUDA_SUCCESS(cudaMemcpyFromSymbol(
      &C[0][0], GPU_C,
      (SIZE * SIZE) * sizeof(T_real),
      0, cudaMemcpyDeviceToHost
    ), "Transfer␣GPU_C␣-->␣C");
}
```

Dans un second temps, nous avons procédé à l'implémentation de différents noyaux de calcul rappelés dans la section 1.

2.1 Noyau K1

Dans notre premier noyau de calcul, nous utilisons des blocs de threads CUDA à une dimension pour calculer les portions de la matrice résultat. Ce choix se traduit par la configuration des dimensions de la grille de calcul présente dans le listage 3 et par l'implémentation dans le listage 4.

Listage 3 – Configuration des dimensions de la grille de calcul pour *K1*

```
...
dim3 Dg, Db;
...
// Configuration des dimensions de la grille
Db.x = BLOCK_SIZE_X_K0;
Db.y = 1; Db.z = 1;
Dg.x = !(SIZE % BLOCK_SIZE_X_K0) ? SIZE /
    BLOCK_SIZE_X_K0 : SIZE / BLOCK_SIZE_X_K0 +
    1;
Dg.y = SIZE; Dg.z = 1;
// Appel de la fonction de calcul (version 1 du
    noyau)
MatrixProductKernel_v1<<<Dg,Db>>>();
```

Listage 4 – Implémentation du noyau *K1*

```
__global__ void MatrixProductKernel_v1(void) {
  // Index computations
  int lig = blockIdx.y; T_real res = 0.0;
  int col = threadIdx.x + blockIdx.x *
      BLOCK_SIZE_X_K0;
  // Matrix product computation
  if(col < SIZE) {
   for (int i = 0; i < SIZE; i++) {
     res += GPU_A[lig][i] * GPU_B[i][col];
   }
   GPU_C[lig][col] = res;
  }
}
```

En fait, le nombre de threads par bloc correspond à la taille du bloc. Le nombre de blocs par grille est égal au nombre de blocs nécessaires pour couvrir l'intégralité de la matrice résultat. Si la taille de cette dernière n'est pas divisible par la taille du bloc, il faut compter un bloc supplémentaire pour traiter les cases de matrices restantes.

Par ailleurs, comme nous avons entrepris d'effectuer le pavage de la matrice résultat avec des bloc de threads à une dimension, les dimensions y et z de la grille sont égales à 1.

2.2 Noyau K2

Le deuxième noyau utilise des blocs de threads à 2 dimensions pour effectuer le calcul. La configuration dimensionnelle de la grille correspondante se trouve dans le listage 5 et l'implémentation dans le listage 6.

Listage 5 – Configuration des dimensions de la grille de calcul pour *K2*

```
...
dim3 Dg, Db;
...
// Configuration des dimensions de la grille
Db.x = BLOCK_SIZE_X_K1;
Db.y = BLOCK_SIZE_Y_K1;
Db.z = 1;
Dg.x = !(SIZE % BLOCK_SIZE_X_K1) ? SIZE /
    BLOCK_SIZE_X_K1 : SIZE / BLOCK_SIZE_X_K1 +
    1;
Dg.y = !(SIZE % BLOCK_SIZE_Y_K1) ? SIZE /
    BLOCK_SIZE_Y_K1 : SIZE / BLOCK_SIZE_Y_K1 +
    1;
Dg.z = 1;
// Appel de la fonction de calcul (version 1 du
    noyau)
MatrixProductKernel_v2<<<Dg,Db>>>();
```

Listage 6 – Implémentation du noyau *K2*

```
__global__ void MatrixProductKernel_v2(void) {
  // Index computations
  int lig = threadIdx.y + blockIdx.y *
      BLOCK_SIZE_Y_K1;
  int col = threadIdx.x + blockIdx.x *
      BLOCK_SIZE_X_K1;
  T_real res = 0.0;
  // Matrix product computation
  if(col < SIZE && lig < SIZE) {
   for (int i = 0; i < SIZE; i++) {
     res += GPU_A[lig][i] * GPU_B[i][col];
   }
   GPU_C[lig][col] = res;
  }
}
```

Cette fois-ci, le nombre de threads par bloc en dimension y devient supérieur à 1. De même pour le nombre de blocs par grille. Ainsi le programme effectuera un pavage de la matrice résultat en 2D dimensions.

2.3 Noyau K4

En dernier, nous avons tenté d'optimiser le calcul en nous servant de la mémoire partagée pour cacher les valeurs qui sont utilisées plusieurs fois durant le calcul et ainsi éviter leur chargement répétitif depuis les registres et le cache général. Dans ce cas, la configuration des dimensions de la grille de calcul ne change guère par rapport à celle de *K2* (voir le listage 5). Par contre, l'implémentation de cette version du noyau devient nettement plus complexe (voir le listage 7).

Listage 7 – Implémentation du noyau *K4*

```
__global__ void MatrixProductKernel_v4(void) {
  int lig = threadIdx.y + blockIdx.y *
      BLOCK_SIZE_XY_K3;
  int col = threadIdx.x + blockIdx.x *
      BLOCK_SIZE_XY_K3;
  __shared__ T_real data_A[BLOCK_SIZE_XY_K3][
      BLOCK_SIZE_XY_K3];
  __shared__ T_real data_B[BLOCK_SIZE_XY_K3][
      BLOCK_SIZE_XY_K3];
```

```
  T_real res = .0;
  int limit = (SIZE % BLOCK_SIZE_XY_K3) ? SIZE /
       BLOCK_SIZE_XY_K3 + 1 : SIZE /
      BLOCK_SIZE_XY_K3;

  for (int e = 0; e < limit; e++) {
    if (lig < SIZE && e * BLOCK_SIZE_XY_K3 +
        threadIdx.x < SIZE)
      data_A[threadIdx.y][threadIdx.x] = GPU_A[
          lig][e * BLOCK_SIZE_XY_K3 + threadIdx.
          x];
    else
      data_A[threadIdx.y][threadIdx.x] = .0;
    if (col < SIZE && e * BLOCK_SIZE_XY_K3 +
        threadIdx.y < SIZE)
      data_B[threadIdx.y][threadIdx.x] = GPU_B[e
          * BLOCK_SIZE_XY_K3 + threadIdx.y][col
          ];
    else
      data_B[threadIdx.y][threadIdx.x] = .0;

    __syncthreads();
    if (lig < SIZE && col < SIZE)
      for (int i = 0; i < BLOCK_SIZE_XY_K3; i++)
        res += data_A[threadIdx.y][i] * data_B[i
            ][threadIdx.x];
    __syncthreads();
  }
  if (lig < SIZE && col < SIZE)
    GPU_C[lig][col] = res;
}
```

Dans cette version, nous créons deux tableaux partagés qui contiendront les portions des matrices opérandes nécessaires pour calculer le produit partiel. Comme ces portions sont réutilisées plusieurs fois durant le calcul il convient de les cacher dans la mémoire partagée et y accélérer ainsi l'accès depuis les threads afin d'améliorer les performances du programme.

Avant de commencer le calcul, chaque thread remplit la portion des tableaux partagés qui lui correspond. Finalement, et après la synchronisation, les threads peuvent effectuer le calcul. Cette séquence est répétée autant fois qu'il y a de blocs dans la grille de calcul.

3 Analyse des résultats

3.1 GPU versus CPU

De manière générale, en considérant tous les noyaux de calcul testés, plus la taille absolue (largeur × hauteur) de la grille de calcul augmente, plus la performance du programme croît.

Nous pouvons bien observer ce comportement notamment en analysant les résultats des tests de performances des noyaux *K1* et *K4*. Cependant, dans ces deux cas, soit il s'agit d'utiliser des blocs de threads à seulement une dimension ou alors le nombre de threads par bloc et le nombre de blocs par grille sont équivalents. Alors que dans le cas du noyaux *K2*, utilisant des blocs de threads 2D, nous constatons que les meilleures performances ont été obtenues pour la configuration avec 16 threads par bloc et 64 blocs par grille.

3.2 Comparaison des noyaux

En comparant les figures 1 et 1, nous pouvons voir que le noyau *K2* présente de meilleures performances vis à vis du noyau *K1*. En effet, puisque *K2* utilise des blocs de threads à deux dimensions, plusieurs calculs peuvent être effectués en parallèle.

Cependant le gain en performance le plus marquant offre le noyau *K4* (figure 2) pour lequel nous nous sommes servis de la mémoire partagée de la carte graphique pour optimiser les temps d'accès mémoire. Même en gardant une grille de dimensions plus petites (32×32) comparée aux dimensions expérimentées pour *K2*, la diminution du nombre d'accès à la mémoire globale permet d'obtenir un gain d'environ 40% par rapport au meilleur résultat obtenu avec *K2*.

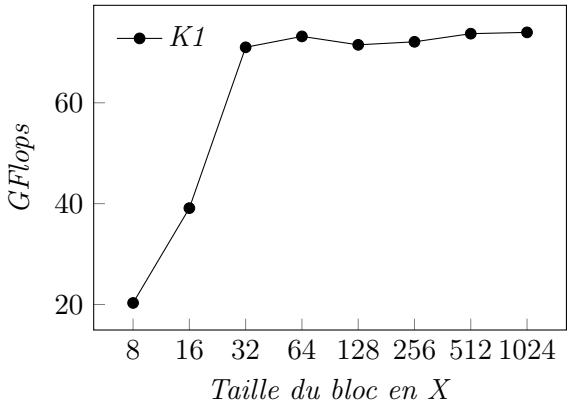

FIGURE 1 – Mesures de performance pour le noyau *K1*

Nous observons sur la figure 1 l'évolution de la performance du noyau de calcul *K1* jusqu'à une taille de blocs de 32 pour se stabiliser autour de 73 GFlops. Cela s'explique en partie par le fait que le nombre maximum de blocs résidents par multiprocesseur supportés par CUDA est de 32. Au-delà de cette valeur, il n'est plus possible d'observer de gain important pour le noyau *K1*.

Comme formulé précédemment, le noyau de calcul *K4* présente les meilleurs résultats (figure 2). L'utilisation de mémoire partagée influe grandement sur la performance du noyau mais est plus complexe à mettre ne oeuvre. En effet, il peut être nécessaire de concevoir un algorithme de cache dédié en fonction du problème (calculs) et des données à traiter. De plus, la gestion d'une mémoire partagée nécessite de mettre en place une synchronisation entre les différents threads. Néanmoins et dans la plupart des cas, la

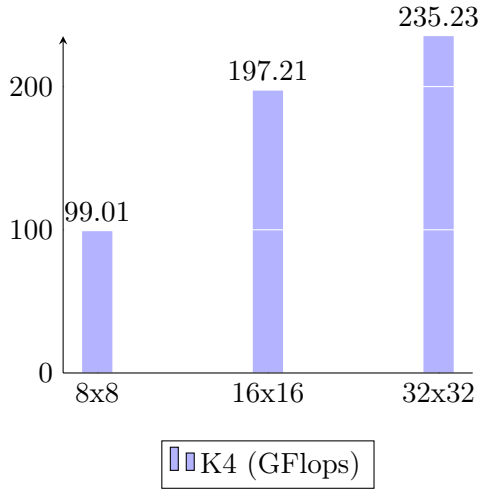

FIGURE 2 – Mesures de performance pour le noyau *K4*

mémoire partagée permet aux différents processeurs graphiques d'échanger entre eux des données utiles au calcul, et ce, sans devoir passer par la mémoire globale.

		Taille du bloc en X								
		2	4	8	16	32	64	128	256	512
Taille du bloc en Y	2	0	0	38.95	75.98	103.11	73.44	117.68	118.45	117.98
	4	0	0	73.75	106.77	109.16	117.7	119.13	119.14	0
	8	34.19	67.94	88.44	111.02	118.8	119.16	119.13	0	0
	16	46.85	71.47	93.95	156.12	119.08	119.12	0	0	0
	32	44.77	68.11	106.47	164.29	118.76	0	0	0	0
	64	23.79	46.49	114.61	166.42	0	0	0	0	0
	128	19.76	48.49	128.62	0	0	0	0	0	0
	256	18.68	63.25	0	0	0	0	0	0	0
	512	18.2	0	0	0	0	0	0	0	0

TABLE 1 – Mesures de performance pour le noyau *K2*

Le noyau de calcul *K2* offre des performances variables selon la taille de blocs utilisée (tableau 1). En effet, on peut observer que le gain en GFlops est catastrophique lorsque la taille de blocs en X est inférieur à une certaine valeur. Pour mieux comprendre ce phénomène, il faut s'intéresser au fonctionnement de la mémoire, et notamment de la mémoire cache dans le contexte de l'accès à celle-ci selon le principe de localité spatiale. Dans le cas d'un tableau où l'accès à ses données est ordonné par ligne (et non par colonne), il convient de s'assurer que l'accès à celui-ci respecte au mieux la contrainte de localité mémoire qui lui est propre. Ainsi, si la valeur de X est trop petite, il n'y a pas suffisamment de données contiguës pour chaque thread qui, arrivé à la fin de son fragment de données rencontrera un *cache-miss*. Il faut alors s'assurer que chaque fragment soit suffisamment long pour avoir une proportion plus importante de *cache-hit* pour éviter de devoir recourir à une lecture des données dans la mémoire globale, plus longue que le cache *L1* ou *L2*.

4 Conclusion

En dépit de sa complexité, il semble que le noyau de calcul *K4* ait obtenu le meilleur gain de performance. Aussi, cela nous permet de constater que la gestion et la manière d'accéder aux données en mémoire ont un impacte sur les performances. En effet, l'utilisation de mémoire partagée par le noyau *K4* a montré des résultats significatifs tout comme la façon de répartir les blocs pour le noyau *K2* impacte sur la localité spatiale des données en mémoire.