TP 1 : Programmation CUDA de base

5ème année ingénieur de Polytech Paris-Sud

Stéphane Vialle, CentraleSupélec & LRI, Stéphane.Vialle@centralesupelec.fr

Laércio Lima Pilla, CNRS & LRI, pilla@lri.fr

Objectifs du TP :

Ce TP a pour objectif de pratiquer la programmation d'un GPU au sein d'un noeud de calcul CPU+GPU : il consiste à implanter un premier produit de matrices denses sur un GPU. On développera différents "kernels", et pour chacun on mesurera les performances d'un produit de matrices denses sur un GPU, en fonction de la granularité de la grille de blocs de threads. On étudiera la qualité de la "coalescence" de chaque version du kernel, pour identifier les solutions les plus intéressantes. Enfin, on comparera les performances obtenues sur GPU avec celle obtenues sur le CPU.

Plate-forme de développement :

Les machines utilisées seront celles des clusters Cameron ou Tx de CentraleSupélec :

- **Tx** : chaque machine contient un CPU Intel XEON quad-core hyperthreadés, et un GPU NVIDIA RTX2080 (architecture Turing)
- **Cameron** : chaque machine contient un CPU Intel XEON hexa-core hyperthreadés, et un GPU NVIDIA GTX1080 (architecture Pascal)

L'environnement CUDA C est disponible sur chaque machine (et donc le compilateur "**nvcc**" et les drivers pour utiliser le GPU).

Vous utiliserez les comptes de TP "hpcpps_#i", où #i est une valeur entière entre 1 et 16. Depuis votre poste de travail vous vous connecterz par ssh sur la machine ghome.metz.supelec.fr, puis vous vous connecterez à la machine term2.grid.metz.supelec.fr où vous réserverez UN noeud sur l'un des clusters à l'aide des commandes OAR.

- pour réserver UN noeuds sur **Tx** pour 5h : oarsub -p "cluster='Tx'" -l nodes=1,walltime=5:00:00 -I
- pour réserver UN noeuds sur **Cameron** pour 5h : oarsub -p "cluster='Cameron'" -l nodes=1,walltime=5:00:00 -I

Travail à effectuer :

Remarques préliminaires :

- Le squelette de programme que vous utiliserez contient un code de produit de matrices denses en OpenMP et CUDA.
- La partie OpenMP est complète, et servira de calcul de référence. **Elle est destiner à permettre de vérifier les résultats obtenus en CUDA**.
- La partie CUDA est en partie développée, mais il vous reste à compléter le fichier `gpu.cu` :
  - ajouter des instructions de transferts de données du GPU vers le CPU du même noeud, puis du CPU vers le GPU,
  - ajouter des kernels de calcul sur GPU et des définitions de grilles de blocs de threads,
  - vérifier l'exactitude des calculs, mesurer les performances et tester différentes optimisations.

- **Le squelette est compilable et contient une aide intégrée : exécutez 'make' puis './MatrixProduct -h'.**

- Pour faire vos mesures de performances vous compilerez en Simple Précision (le type "T_real" devient le type "float") avec "#-DDP" dans le Makefile. La simple précision est adaptée aux capacités des GeForce GTX1080 et TRX2080, mais il se peut que vous observiez des différences entre les calculs sur CPU et sur GPU!!

- Pour valider votre premier code vous compilerez en Double Précision (le type "T_real" devient le type "double") avec "-DDP" dans le Makefile, les résultats seront identiques sur CPU et sur GPU, mais les performances des GPU s'effondreront (car il s'agit de cartes GPU grand public non adaptées à la Double Précision).

1 - Exercice utilisant une grille "2D" de blocs et kernels "1D" :

a. **Récupérez le squellete de programme** OpenMP+CUDA. Compilez ce squelette et testez son fonctionnement sur CPU (exécutez la commande `./MatrixProduct -h` pour voir les détails de fonctionnement de l'application).
   - sur TX : **squellete**, ou bien allez le recopier sur votre compte de TP par la commande : `cp ~vialle/tmp/MatrixProduct-CUDA-CS-TX-enonce.zip .`
   - sur Cameron : **squellete**, ou bien allez le recopier sur votre compte de TP par la commande : cp ~vialle/tmp/MatrixProduct-CUDA-CS-Cameron-enonce.zip .

   Toutes les routines à compléter se trouve dans le fichier '`gpu.cu`'.
   Mais vous devrez redéfinir des tailles de matrices ou de blocs dans le fichier '`main.h`'.

   Récupérez le fichier Excel de saisi des résultats, et complétez-le au fur et à mesure du TP.

b. Complétez les routines '**gpuSetDataOnGPU**' et '**gpuGetResultOnCPU**'qui réalisent des transferts CPU/GPU. Vous devez compléter ces routines avec des appels à '**cudaMemcpyFromSymbol**' et '**cudaMemcpyToSymbol**'.

c. Implantez le **kernel K0** (et sa grille de blocs de threads) pour que :

- chaque thread calcule un élément complet de la matrice C (matrice résultat),
- **un bloc de threads soit un segment 1D selon la dimension X de threads** calculant les éléméments d'une partie d'une ligne de C,
- **les colonnes de C soient associées à la dimension X : les colonnes successives d'une ligne de C seront calculées par des threads consécutifs en X d'un bloc,**
- les lignes soient associées à la dimension Y: les lignes successives de C seront traitées par des blocs 1D différents.

Testez votre implantation sur une matrice de 1024x1024 **DOUBLE** (option -DDP active dans le Makefile), et vérifiez que vous obtenez les mêmes valeurs que sur CPU (**`MatrixProduct -t CPU -cpu-k 1 -cpu-nt 4`** sur Tx et **`-cpu-nt 6`** sur Cameron)

d. Puis implantez le **kernel K1** (et sa grille de blocs de threads) pour que votre implantation fonctionne sur une matrice de taille **non** multiple de la taille des blocs.

Testez votre kernel K1 sur une matrice de 1025x1025 **DOUBLE** (vérifiez que vous obtenez les mêmes valeurs que sur CPUs).

e. Une fois que votre kernel K1au point, mesurez les performances obtenues sur une matrice de **4096x4096 FLOAT** éléments (#-DDP dans le Makefile).

- Faites varier la taille de vos blocs 1D de threads, et mesurer les performances obtenues pour des blocs de 32 à 1024 threads.
- Puis faites varier la taille des blocs de 32 à 4 threads.
- **Est-ce que la courbe de performance obtenue semble conforme à la théorie ? pourquoi ?**

f. Comparez aux meilleurs performances obtenues sur CPU multi-coeurs en OpenMP avec un kernel de même niveau (kernel 0),

- **`MatrixProduct -t CPU -cpu-k 0 -cpu-nt 8`** sur Tx
- **`MatrixProduct -t CPU -cpu-k 0 -cpu-nt 12`** sur Cameron

Calculez le speedup GPU *vs* CPU.

2 - Etude de la coalescence :

a. **On considère le kernel K1 : avec des blocs 1D selon l'axe X, où chaque bloc traite des colonnes successives d'une même ligne de C**.

Etudier la **coalescence** :

1. des lectures de A,
2. des lectures de B,
3. des écritures dans C

b. Que serait devenu la coalescence en utilisant la transposée de A ? et la transposée de B ?

c. **On considère maintenant des blocs 1D selon l'axe X, MAIS où chaque bloc traite des lignes successives d'une même colonne de C.** Etudier à nouveau la coalescence obtenue lors des lectures de A ou de la transposée de A, de B ou de la transposée de B, et des écritures de C.

d. **Quelle est la meilleure solution ?**

e. Echangez le calcul des numéros de ligne et de colonne traitées par chaque thread, pour vous mettre dans le cas où un bloc 1D en X traite des lignes successives d'une même colonne de C. Mesurez les performances obtenues. **L'évolution des performances est-elle conforme aux prévisions ?**

3 - Exercice utilisant une grille "2D" de blocs et kernels "2D" :

a. Créez le **kernel K2** en généralisant votre kernel K1 (et sa grille de blocs) pour qu'il supporte des blocs 2D de threads.

Testez votre implantation sur une matrice de 1024x1024 éléments, puis sur une matrice de 1025x1025 éléments : vérifiez que vous obtenez les mêmes valeurs qu'avec le kernel 1.

Attention : la taille maximale d'un bloc est de 1024 threads !

b. Une fois que votre programme est au point, mesurez les performances obtenues sur une matrice de **4096x4096 FLOAT**.

Mesurez les performances obtenues avec des blocs "XxY" pour des tailles correspondant aux cases jaunes du fichier Excel. Notamment pour des tailles XxY de 8x8, 16x16, 32x32, puis de 32x8, 8x32, et 32x16 et 16x32 threads.

d. Comparez aux performances obtenues avec des blocs 1D du même nombre total de threads, et comparez aux meilleurs performances obtenues sur CPU multi-coeurs en OpenMP, calculez le speedup GPU *vs* CPU.
