Examen de rattrapage du cours d'Architecture « partie GPU » - mention ASI

Mars-Avril 2023

Stéphane Vialle

Cet examen consiste en :

- Un développement de codes CUDA d'un produit de matrices carrées denses de **nombres complexes**
- Une analyse justifiée de la coalescence des algorithmes possibles et implantés
- Un calcul du nombre d'opérations flottantes réalisées par le programme et une mesure du nombre de « flops » obtenu pour quelques tailles de matrices.
- Une présentation des codes développés et des réponses aux questions précédentes lors d'un ORAL sur le campus de Metz AVANT la fin février.

Question 1 : choix d'une stratégie de stockage des données en mémoire

Les matrices carrées GPU_A, GPU_B et GPU_C de nombres complexes (avec GPU_C = GPU_A × GPU_B) peuvent être stockées de différentes manières sur le GPU, mais on se focalise sur deux possibilités :

```
T_real GPU_A[2][SIZE][SIZE];
```

Un tableau de deux matrices : l'une pour les parties réelles, l'autre pour les parties imaginaires.

ou :

```
T_real GPU_A[SIZE][SIZE][2];
```

Une matrice de couple de valeurs (partie réelle et partie imaginaire)

Question : Quelle solution vous parait mener aux meilleures performances pour une implantation sur GPU ? Justifiez votre réponse.

Rmq : cette question préliminaire a pour but de vous mener vers la bonne implantation. Mais il vous sera toujours possible d'implanter rapidement les deux versions dans les questions suivantes et de vérifier votre raisonnement.

Question 2 : développement d'une version CPU de référence

Travail demandé : Repartez du code de produit de matrices carrées denses et réelles des premiers TP du cours et adaptez le pour traiter des matrices complexes :

- Recopiez le code de départ sur votre compte de travail au DCE :

```
cp  ~gpu_vialle/GPU-RS/MatrixProduct-Reel-enonce.zip  .
```

- Adaptez les tableaux A, B, C … et les « symboles » GPU_A, GPU_B et GPU_C
- Adaptez leurs initialisations : initialisez les parties réelles et imaginaires de manière déterministe (n'utilisez pas de valeurs aléatoires) et essayez de mettre des valeurs différentes dans chaque partie réelle et dans chaque partie imaginaire.
- Adaptez l'affichage des 3 valeurs numériques de la contra-diagonale pour afficher leurs parties réelles et imaginaires.
- Adaptez le calcul du nombre d'opérations flottantes, dont on déduit les Gflops en le divisant par le temps d'exécution (détaillez votre raisonnement et vos calculs).

- Adaptez le noyau de calcul sur CPU pour multiplier des nombres complexes.
- Adaptez toute autre partie du code qui le nécessiterait.

Votre code conservera le type **`T_real`** afin de pouvoir être compilé facilement en **`float`** ou en **`double`**.

Question 3 : développement d'un noyau de calcul sur GPU sans la *shared memory*

Travail demandé : Développez un « *kernel* » GPU utilisant une grille 2D de blocs 2D, et fonctionnant pour toute taille de matrices (donc un *kernel* qui traite les « *boundaries* »), sur le modèle de celui développé en TP dans le cours.

Démarche de validation du code :

- Compilez le code en « double » avec des tailles de blocs de 32×32 et vérifiez l'égalité des calculs sur CPU et sur GPU pour des matrices de 1024×1024 (erreur habituellement inférieure à 1E-14 avec des « double »).
- Recommencez avec des matrices de 1027×1027.

Démarche d'analyse de performances :

- Compilez le code en « float » avec des tailles de blocs de 32×32 et mesurez ses performances sur GPU pour des matrices de 1024×1024, 2048×2028, 4096×4096.
- Les vitesses de calcul sont-elles stables entre deux exécutions du même problème ?
- Les vitesses de calcul sont-elles identiques pour les 3 tailles de matrices (tracez une courbe) ?

Question 4 : développement d'un noyau de calcul sur GPU avec la *shared memory*

Travail demandé : Développez un « *kernel* » GPU utilisant une grille 2D de blocs 2D et la ***shared memory***, fonctionnant pour toute taille de matrices, sur le modèle de celui développé en TP dans le cours.

Démarche de validation du code :

- Compilez le code en **`float`** avec des tailles de blocs de 32×32 et validez son fonctionnement en comparant ses résultats numériques à ceux du kernel de la question 3 sur des matrices de 1024×1024
- Recommencez avec des matrices de 1027×1027.

Démarche d'analyse de performances :

- Compilez le code en « float » avec des tailles de blocs de 32×32 et mesurez ses performances sur GPU pour des matrices de 1024×1024, 2048×2028, 4096×4096.
- Les vitesses de calcul sont-elles stables entre deux exécutions du même problème ?
- Les vitesses de calcul sont-elles identiques pour les 3 tailles de matrices (tracez une courbe) ?
- Calculez les accélérations par rapport à la version de la question 3 pour les trois tailles de matrices (tracez une courbe).

Rmq : mettre au point tout d'abord un kernel fonctionnant uniquement pour des tailles de matrices multiples de la taille des blocs, donc ne nécessitant pas de traitement des *boundaries*, peut-être une étape utile.