

RAPPORT DE TRAVAUX SUR L'UTILISATION DE GPU AVEC CUDA APPLIQUÉE A LA MÉTHODE DE LA PUISSANCE


GPU TECHNOLOGY CONFERENCE

TESLA V100

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink



-full GV100 chip contains 84 SMs

 **NVIDIA**

Thomas Firmin - Yann Cauchepin

IS5 - Polytech Lille - 12/01/2021

Introduction

Ce travail pratique nous permet d'étendre nos connaissances sur l'utilisation de GPU pour accélérer les algorithmes. Bien évidemment, tous les algorithmes ne peuvent pas être efficacement exécutés sur GPU et leur exploitation doit être codée de manière spécifique. Ici, on se propose d'implémenter l'algorithme de la puissance que nous connaissons bien : on peut aisément séparer les parties indépendantes du code et répartir les calculs en de nombreuses parties.

Nous disposons pour cela de deux codes en langage C :

- *sequential.c* : création d'une matrice de taille $n \times n$ et implémentation de la méthode de la puissance de façon séquentiel.
- *checker.c* : code source ayant pour but de vérifier les résultats.

Le code CUDA que nous nous proposons d'implémenter sera nommé avec le suffixe *.cu* et nous utiliserons le réseau *GRID5000* pour l'exécuter.

I - Stratégie d'implémentation GPU

La méthode de la puissance est répartie de la façon suivante (boucle de calcul) :

- Un calcul matrice vecteur.
- Une normalisation du vecteur résultat.
- Un calcul de l'écart quadratique des vecteurs résultats entre deux itérations.

Nous nous proposons donc d'implémenter le code de la façon suivante pour une exécution sur GPU.

- Nous utilisons 4 Kernels CUDA où chacun des kernels correspond à une application de fonction.
 - **A** : $\frac{n}{256}$ blocs, contenant 256 threads distribué
Répartition des lignes de la matrice à chacun des threads pour obtenir la composante du vecteur résultat correspondante pour chaque thread.
 - **B** : 1 bloc de 1 thread
Calcul la somme séquentielle des composantes du vecteur résultat et ainsi obtenir la norme euclidienne du vecteur résultat.
 - **C** : $\frac{n}{256}$ blocs distribué en 1D, contenant 256 threads distribué en 1D
Calcul de la composante normée du vecteur résultat pour chaque thread grâce à la norme totale du vecteur calculé dans le kernel B.
 - **D** : 1 bloc de 1 thread
Calcul l'écart quadratique entre le vecteur résultat actuel et celui précédent.

Quelles sont maintenant les communications entre CPU et GPU, nécessaires au bon fonctionnement du programme ?

Dans un premier temps il faut allouer l'espace mémoire nécessaire dans le GPU pour stocker la matrice, le vecteur X, le vecteur résultat, l'erreur et la norme. On utilise pour cela *cudaMalloc*. Durant la phase d'initialisation, deux communications sont nécessaires, il faut envoyer au GPU la matrice et le vecteur initial sur lesquels effectuer la méthode de la puissance. Nous passons pour cela par la fonction *cudaMemcpy* avec l'option *cudaMemcpyHostToDevice*, pour les communications CPU vers GPU.

Puis le CPU va donner l'ordre d'exécution des kernels au GPU, cela passe simplement par l'appel du CPU des kernels.

L'exécution se fait dans cet ordre :

- Multiplication matrice-vecteur
- Calcul de la norme
- Normalisation
- Calcul de l'erreur

Entre l'exécution de ces quatre étapes, aucune communication CPU-GPU n'est nécessaire. En effet, le vecteur résultat est directement stocké sur le GPU, de même pour la norme.

Cependant, à la fin du calcul de l'erreur, le CPU doit récupérer l'erreur calculée par le GPU, afin de vérifier la condition d'arrêt, et d'effectuer ou non, une nouvelle fois les quatre étapes précédentes.

Une fois l'erreur calculée, le CPU récupère le vecteur propre, et le copie dans la mémoire du GPU dédié au vecteur X (vecteur utilisé dans le produit matrice-vecteur), à l'aide de la fonction *cudaMemcpy* et des options *cudaMemcpyDeviceToHost* et *cudaMemcpyHostToDevice*.

II - Implémentation CUDA

Le code contient donc trois versions. La première version est un code simpliste de la méthode la puissance, celui-ci n'est pas optimisé à l'exécution sur GPU.

La seconde version est un code optimisée pour l'exécution sur GPU, et la dernière version utilise les routines cuBLAS.

Nous allons ici étudier la première version, qui implémente les quatre kernels décrits précédemment, nous verrons plus tard comment ceux-ci peuvent être améliorés.

III - Comparaison des performances GPU et CPU

Nous allons comparer ici différentes tailles de matrices pour comparer les performances.

| Matériel / Taille matrice | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
|---------------------------------|-----|-----|------|------|------|------|-------|
| CPU (temps d'exécution en s) | 0.0 | 0.1 | 0.3 | 1.2 | 4.7 | 23.5 | 96.3 |
| GPU (temps d'exécution en s) | 0.2 | 0.2 | 0.2 | 0.3 | 0.4 | 1.2 | 3.8 |
| Nombre d'itérations | 85 | 123 | 120 | 113 | 119 | 148 | 151 |

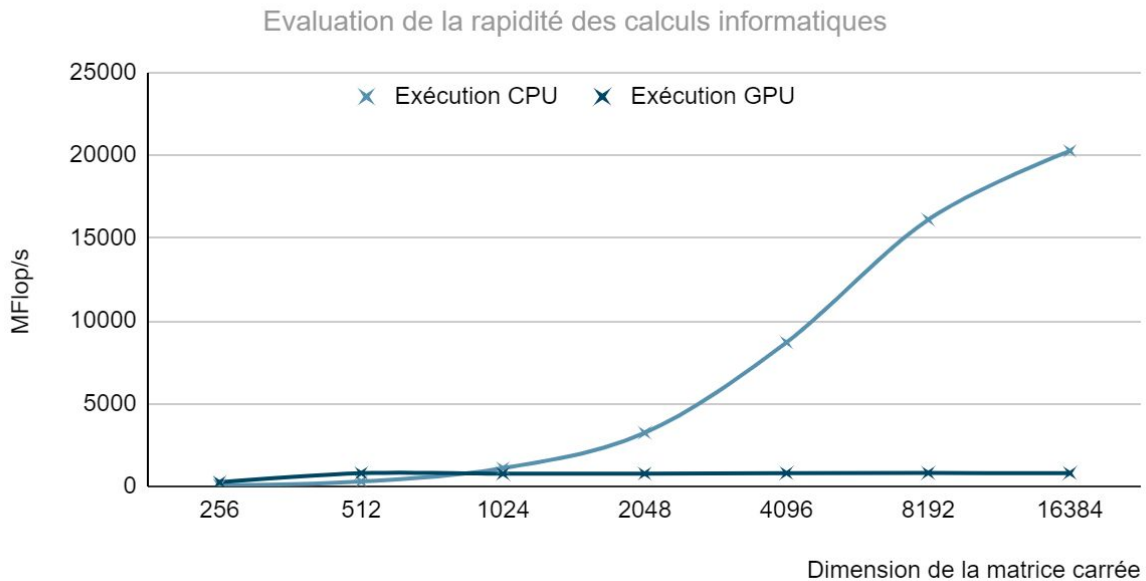
On remarque que pour le CPU les performances sont légèrement meilleures pour de petites matrices. La différence est certainement dû à un peu grand nombre de communications entre le matériel pour l'exécution sur GPU.

Cependant pour des matrices de grandes tailles, le GPU est nettement plus performant.

Comparons maintenant le nombre Mflop/s :

| Matériel / Taille matrice | 256 | 512 | 1024 | 2048 | 4096 | 8192 | 16384 |
|---------------------------|---------|---------|-----------|-----------|------------|-------------|-------------|
| CPU (Mflop/s) | 244.8 | 793.3 | 766.0 | 772.4 | 803.9 | 806.1 | 802.9 |
| GPU (Mflop/s) | 65.3 | 294.1 | 1098.8 | 3232.2 | 8696.6 | 16111.7 | 20269.5 |
| Mflop | 132 864 | 527 872 | 2 104 320 | 8 402 944 | 33 583 104 | 134 275 072 | 537 051 136 |

Comparaison des performances de calcul sur la méthode de la puissance



Ainsi, on remarque encore une fois que les performances sur GPU sont bien meilleures pour des matrices de grande taille.

Ainsi sur CPU lorsque l'on double la taille de la matrice ($\sim \times 4$ pour le nombre de Mflops), on quadruple le temps d'exécution. Le nombre de Mflops semble converger vers 800 sur CPU.

Alors que pour le GPU le temps d'exécution est triplé à partir d'une taille de 4096. Puis multiplié par 3.6 pour une taille de matrice de 16 384 et 32 768. Pour une taille de matrice de 32 768, le temps est d'environ 14 secondes. Le nombre de Mflops semble converger vers environ 20 000.

Ainsi en termes de performances, le GPU est meilleur pour le calcul matriciel.

IV - Optimisation des performances GPU

Optimisations effectuées:

Nous avons dans un premier temps optimisé l'*ILP* (*Instruction Level Parallelism*). Par se faire, plutôt que d'attribuer 1 seule ligne par thread, nous avons tenté d'en attribuer plusieurs par thread, tout en diminuant le nombre de thread.

Ainsi, attribuer 16 lignes par threads semblent donner les meilleures performances, tout en adaptant le nombre de blocs en fonction de la taille de la matrice. Le nombre de thread est fixé à 8 par bloc.

Nous avons ensuite utilisé les opérations atomiques, pour paralléliser les différentes sommes séquentielles et permettre l'accès exclusif de chaque thread à une ressource.

Dans le code v2, on choisit le nombre de lignes calculées par un thread, avec la constante *NB_ELEM*. L'algorithme adapte ensuite le nombre de thread par bloc.

Certains kernels ont changé, notamment dans la multiplication matrice-vecteur. On profite de la boucle pour calculer en même temps la somme des éléments du vecteur propre, avec *atomicAdd*. Le kernel normalisant le vecteur profite lui aussi de la boucle pour calculer la somme des erreurs quadratiques, avec *atomicAdd*. Ainsi le calcul des carré et les soustractions X-Y, sont répartis sur plusieurs thread, sur plusieurs blocs. Et la somme est parallélisée.

Enfin une dernière version implémente les routines cuBLAS.

Les tests sont effectués sur une matrice de taille 8 192, 16 384 et 32 768 afin d'observer au mieux les différences de temps d'exécution.

| Matériel / Taille matrice | 8192 | 16384 | 32768 |
|--------------------------------------|------|-------|-------|
| GPU - v1 (temps d'exécution en s) | 1.2 | 3.8 | 13.9 |
| GPU - v2 (temps d'exécution en s) | 1.2 | 3.3 | 10.9 |
| GPU - v3 (temps d'exécution en s) | 0.5 | 0.9 | 2.9 |

On remarque qu'avec les optimisations que nous avons apportées nous arrivons à légèrement réduire le temps d'exécution pour de grande matrice. Notamment pour une taille de 32768. Cependant, remarque que l'implémentation avec les routine cuBLAS, surpasse largement les 2 implémentations précédentes. L'implémentation 3 est d'environ 3.5 fois plus rapide pour les grandes matrices (16384 et 32768).

| Matériel / Taille matrice | 8192 | 16384 | 32768 |
|---------------------------|---------|---------|----------|
| GPU - v1 (Mflop/s) | 15889.3 | 20269.5 | 23770.5 |
| GPU - v2 (Mflop/s) | 16654.3 | 18375.9 | 24001.3 |
| GPU - v3 (Mflop/s) | 41058.5 | 83997.9 | 112973.9 |

Encore une fois on remarque que l'implémentation cuBLAS est largement meilleure que les 2 autres implémentations.

Information sur l'implémentation cuBLAS:

Nous avons utilisé les routines:

- cublasSgemv: pour la multiplication matrice-vecteur
- cublasSnrm2: pour le calcul de la norme euclidienne et de l'erreur
- cublasSscal: pour multiplier le vecteur résultat par la norme
- cublasScopy: pour copier le vecteur résultat dans le vecteur X, pour une nouvelle itération.
Et le résultat de $(X-Y)$ dans un vecteur dédié.
- cublasSaxpy: pour soustraire le vecteur X et le vecteur Y.

V - Conclusion

Dans ce TP nous avons eu l'occasion d'implémenter diverses versions de la méthode de la puissance itérée. Une première version sans optimisation montre qu'il est préférable d'effectuer des calculs matriciels sur GPU plutôt que sur CPU. Le GPU surpassant largement le CPU. Attention toutefois, les comparaisons dépendent aussi de la puissance du matériel utilisé.

Nous avons ensuite tenté d'optimiser la première version, notamment en déterminant le nombre optimisé de thread par bloc, et de calculs par thread.

Enfin, les meilleurs résultats sont issus de l'implémentation avec cuBLAS. Ce qui est normal puisque la librairie contrairement à nos précédentes implémentations, est optimisée pour être exécutée sur GPU et pour le calcul matriciel.

Ainsi, ce TP a permis une première approche de l'utilisation de CUDA, certaines optimisations auraient pu être envisagées pour la version 2. Notamment l'utilisation de grille et de bloc de thread 2D, ou bien l'utilisation et l'implémentation du chargement de la mémoire partagée, afin de réduire les temps d'accès mémoire du GPU.

Le GPU est donc, comparé au CPU, le matériel le plus adéquat, pour effectuer la méthode de la puissance.