P41. Parallélisation CUDA et OPENMP pour le traitement d'images satellitaires

Année 2016

Encadrants : R. FABLET – Département SC, PH. HORREIN – Département ELEC, G. MERCIER – Département ITI.

Partenaires : B. SAULQUIN, Société ACRI-ST

Mots clés : traitement d'image, télédétection, parallélisation, CUDA, OpenMP, données satellitaires, optimisation performance

Résumé :

L'objectif de ce projet est de proposer et de mettre en œuvre des stratégies d'accélération matérielle en utilisant le langage CUDA sur la base d'un code C parallélisé en OpenMP. Notre équipe a d'abord effectué une étude théorique sur ces deux techniques de traitement parallèle. Nous avons proposé par la suite une première version naïve du code initial parallélisé en CUDA pour enfin arriver à une version plus aboutie. Des mesures de performance ont été collectées à chaque étape.

1. Présentation et contexte du projet

Dans le cadre des développements réalisés par la société ACRI-ST pour le traitement des données satellitaires de couleur de l'eau issus du futur capteur ALCI, l'algorithme MEETC2 écrit en C et parallélisé en OpenMP nous a été fourni par la société ACRI-ST. Il a pour vocation de traiter des cartographies complètes de la surface de la Terre selon les latitudes avec une résolution de 300m et 21 bandes spectrales par pixel. Le traitement de ces grandes masses de données pose donc un problème de performances auquel nous devons proposer une solution.

2. Méthodologie développée pour aboutir

Le projet s'est déroulé en plusieurs étapes. La première regroupait la partie analyse du besoin et montée en compétences sur la technologie CUDA. Les membres de notre équipe ayant une maitrise plus ou moins avancée du C, nous avons dû rafraichir nos connaissances sur ce langage avant de monter en compétence par nous-mêmes en CUDA via des livres [1][2][3] et la plateforme Udacity. En parallèle nous avons pris possession du code fourni par ACRI-ST. Une deuxième étape a consisté à développer une première version 'naïve' en CUDA de ce code afin d'identifier les goulots d'étranglement au niveau de la performance via différents tests. Nous nous somme basés sur cette analyse pour établir une deuxième version optimisée de manière plus poussée. Ce code-ci a aussi fait l'objet de tests plus avancés afin d'atteindre nos conclusions finales.

3. Développement des différentes tâches et principaux résultats

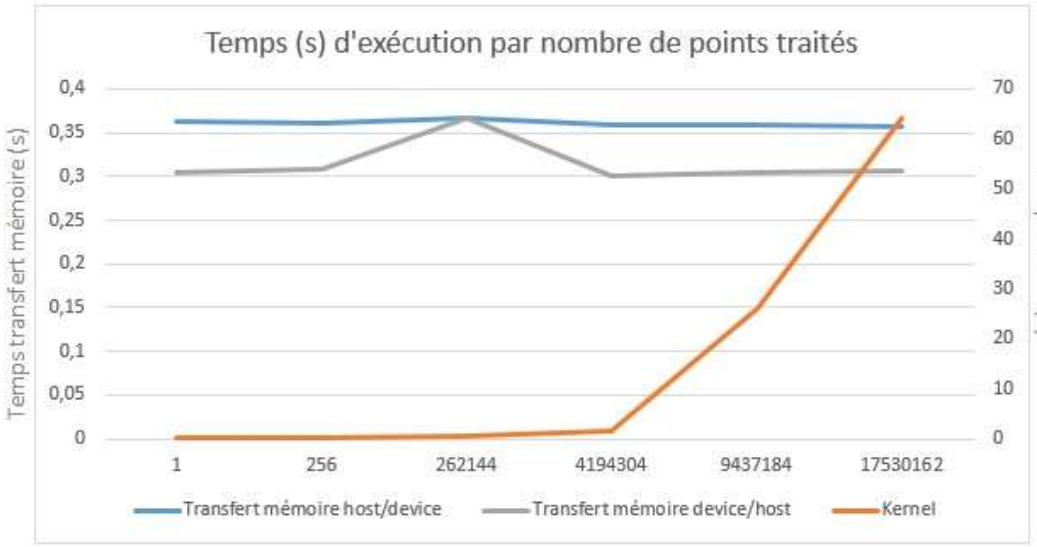
Notre projet n'a demandé aucunes dépenses d'ordre financier. Nous avons eu accès à une machine de Télécom Bretagne équipée de la carte graphique NVIDIA Tesla k80. Nous avons développé et testé avec un nombre limité de données sur nos machines personnelles. Des tests plus poussés sur le jeu

de données complet ont été réalisés sur la machine de l'école. Nous avons travaillé sur un code C parallélisé en OpenMP prenant en paramètre une image de 17530162 pixels.

31. Développement CUDA

Nous avons mis en place deux versions principales de code CUDA.
Tout d'abord une version naïve où nous avons tout simplement externalisé dans un Kernel la partie du code C à paralléliser (partie où un traitement identique était fait pour chaque pixel d'une image). Nous avons alors observé que les temps de transfert des données entre le CPU (~ host) et le GPU (~device) étaient trop important par rapport à ce qu'ils auraient dû être. Le temps total de traitement était aussi trop proche de celui du code OpenMP.
Nous avons donc développé une deuxième version diminuant les temps de transfert grâce à une réduction des variables échangées mais surtout diminuant le temps d'exécution du Kernel grâce à une optimisation de l'accès aux mémoires (partagées et globales) du GPU.

32. Tests de fonctionnalité et de performances



Les données modélisées ci-contre ont été extraites de la version 2 du code CUDA pour un nombre variable de points (pixels d'une image) et un nombre fixe de threads. On observe que les temps de transferts restent constants et sont très faibles en comparaison du temps d'exécution du Kernel. Ce dernier atteint un point de saturation autour de 4194304 points pour cette version.

Lorsque nous exécutons pour 17530162 points le code initial OpenMP et notre deuxième version en CUDA avec 1677216 threads, nous obtenons respectivement un temps total de 1337s et 96s. Cela prouve notre hypothèse initiale qui était de supposer que CUDA était plus performant qu'OpenMP dans ce contexte.

4. Conclusions et perspectives

Notre travail nous a permis de nous rendre compte de la performance de CUDA par rapport à OpenMP dans ce contexte bien précis. Cependant, il est possible d'aller encore plus loin au niveau de la parallélisation CUDA. Certaines pistes à explorer sont la synchronisation des threads, le partage de mémoire entre l'host et le device et globalement l'optimisation de l'utilisation de la mémoire. Une étude approfondie doit notamment être faite autour du nombre de threads pour supprimer le point de saturation. Une fois ces nouvelles modifications mises en place, il est possible d'avoir des temps encore inférieurs à ceux déjà observés.

Bibliographie

[1] Rirk David B., Hwu Wen-mei W.. *“Programming massively parallel processors”.* 2nd Edition. Décembre 2012
[2] Sanders J., Kandrot E. *“CUDA by Example: an introduction to general-purpose GPU programming”.* NVIDIA. 2010
[3] Cook S. *”CUDA Programming”*. Elsevier Libri. Décembre 2012