Accélération sur GPU d'une simulation radar avec OpenACC

Maxime Martelli, Cyrille Enderli, Nicolas Gac, Antoine Vermesse, Alain Mérigot

Accélération sur GPU d'une simulation radar avec OpenACC

Maxime MARTELLI[1,2,3], Cyrille ENDERLI[3], Nicolas GAC[2], Antoine VERMESSE[3], Alain MÉRIGOT[1]

[1]Laboratoire des Signaux et Systèmes, CentraleSupélec, CNRS, Université Paris Sud, Université Paris-Saclay, FRANCE

[2]Laboratoire des Systèmes et Applications des Technologies de l'Information et de l'Énergie, ENS Cachan, CNRS, Université Paris Sud, Université Paris-Saclay, FRANCE

[3]Thales DMS France, Elancourt, FRANCE

maxime.martelli@l2s.centralesupelec.fr

Résumé – Cet article propose une méthodologie pour accélérer un environnement d'une simulation RADAR (**RA**dio **D**etecting **A**nd **R**anging), en passant d'une implémentation sur processeur (CPU) à une implémentation sur puce graphique (GPU). Nous utilisons les outils de programmation GPU les plus utilisés, comme CUDA [1], et plus précisément OpenACC [5], un langage de programmation par directives. **La contribution majoritaire est d'évaluer la vitesse de mise en oeuvre d'une solution d'accélération sur un algorithme**, tout en fournissant des étapes clefs d'accélération sur GPU, en analysant non seulement les performances brutes, mais aussi la facilité et la rapidité de programmation. Concernant notre cas d'étude, l'accélération maximale obtenue sur GPU est de 8.2 avec CUDA et de 4.56 avec OpenACC par rapport à l'implémentation de référence sur CPU.

Abstract – This article gives a methodological approach to accelerating an environment of a RADAR (**RA**dio **D**etecting **A**nd **R**anging) simulation, from a CPU to a GPU implementation. We focus our attention on the most common tools for GPU programming like CUDA [1], but more specifically on OpenACC [5], a directive based parallel programming language.

Our contribution is providing key steps for accelerating a software simulation of a radar algorithm on a GPU, with a particular focus on performance but also on the ease of programming. Maximum achieved execution time speedup on GPU architecture for our typical use case of radar processing is of 8.2 for CUDA and of 4.56 for OpenACC compared to the reference implementation on CPU.

1 Introduction

A mesure que la connaissance du monde qui nous entoure gagne en ampleur, sa modélisation devient plus précise et complexe. Ces modèles nécessitent une puissance de calcul toujours plus importante, et, à défaut d'avoir les ressources de calcul suffisantes, ils sont souvent simplifiés, donc moins précis. Pour permettre une modélisation toujours plus poussée, il est alors crucial d'avoir une architecture matérielle des plus performantes.

La démocratisation du calcul hétérogène vient de la nécessité de trouver une nouvelle dynamique de croissance dans les architectures matérielles. En effet, la fin de la Loi de Moore a été annoncée pour 2021 [8], et il n'est donc plus possible d'augmenter la puissance brute de manière exponentielle comme cela était le cas jusqu'à présent. Un relais de croissance est d'adopter une démarche d'adéquation algorithme architecture pour avoir un système plus efficace, contenant plusieurs architectures hétérogènes adaptées aux algorithmes à accélérer.

Le domaine du traitement radar nécessite de très nombreux calculs. Leur efficacité et leur précision dépend des tests effectués en amont, et une étape essentielle est de simuler aussi précisément que possible les cibles que les capteurs d'un radar pourraient rencontrer dans leur environnement final. Aussi, la simulation radar, qu'il s'agisse de l'environnement comme des traitements, est primordiale pour valider les algorithmes avant de définir le système embarqué réel.

Une simulation radar fiable est ainsi cruciale pour réduire les coûts et améliorer la robustesse du système. Le cas d'étude présenté dans cet article fait partie d'un outil industriel de simulation et de validation de traitements radars. Ce simulateur inclut de nombreux environnements, comme des nuages, des autoroutes, différents sols, ou encore des cibles mouvantes. Il intervient pour tester en amont à la fois les traitements, mais aussi la réponse à certains environnements du radar, tout en validant dans une moindre mesure certain capteurs analogiques.

A mesure que les radars gagnent en complexité, il y a un besoin réel d'accélération en simulation. Comme évoqué précédemment, l'une des solutions est d'utiliser d'autres architectures que les CPUs. La plupart des plateformes de calcul moderne sont hétérogènes, incluant souvent des CPUs, des GPUs, des FPGAs, ou encore des CPUs manycore. Pour programmer ces systèmes, il existe de nombreux outils et langages comme OpenMP [7], OpenCL [6], OpenACC ou CUDA, qui permettent de gérer le parallélisme ainsi que les transferts mémoires efficacement. Certains de ces outils ont pour objectif de permettre une portabilité rapide entre différentes architectures, tout en restant efficaces au niveau des performances, comme

OpenACC, un langage de programmation par directives pour les CPUs et les GPUs.

La contribution principale de nos travaux est de donner une méthodologie d'accélération d'algorithmes pour OpenACC sur GPU, appliqué à notre outil de simulation radar. L'implémentation sur GPU d'algorithmes radar a déjà été évalué par le passé pour les logiciels radars [10] et l'exécution de traitements radars non simulés [9], et les résultats montrent une accélération efficace par rapport à une implémentation sur CPU.

La suite de cet article est structuré comme suit : la Section 2 est consacrée à l'introduction de quelques concepts radars, et à la présentation de l'outil de simulation ainsi qu'à l'algorithme accéléré. Nous décrivons notre méthodologie d'accélération pour OpenACC à la Section 3, tout en discutant à la Section 4 des résultats obtenus tant en terme de performance brute qu'en facilité de programmation.

2 Présentation du problème

2.1 Radar : concepts préliminaires

Au début du 20eme siècle, le développement de la radio et des communications sans fil a ouvert la voie au radar. Utilisé dans un vaste panel d'applications, de la météorologie aux systèmes de défense, c'est aujourd'hui une technologie cruciale.

Un Radar Doppler utilise l'effet Doppler-Fizeau [2]. Le signal émis par l'antenne a une fréquence précise, et l'écho reçu de la réflexion sur la cible a une autre fréquence. En corrélant ces deux fréquences, on peut en déduire la vitesse radiale de la cible. Il est alors possible de construire une carte **D**istance **A**mbiguë **V**itesse **A**mbiguë (**DAVA**) comme illustré en Fig. 1. Le principe est d'afficher de manière concise les cibles détectées tout en les caractérisant en distance et en vitesse.

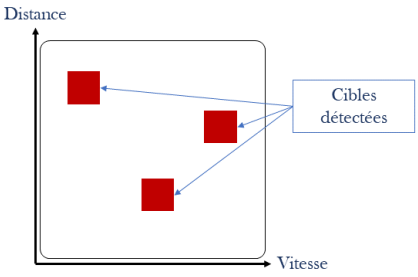

Figure 1: Carte d'ambiguïté en Distance et en Vitesse (DAVA)

Chaque carré de la carte représente une cible détectée par les premiers traitements du radar. Une carte DAVA réelle contient des échos additionnels en provenance de l'environnement radar.

2.2 Simulateur d'Echo Numérique

Le **S**imulateur d'**E**cho **N**umérique (**SEN**) est un outil industriel de simulation et de validation au service du développement d'algorithmes innovants en traitement du signal pour les radars aéroportés. Avec le SEN, les radars sont modélisés, pour simuler de façon représentative les échos reçus dans un environnement qui peut inclure des nuages, des forêts denses, des déserts, des océans, des cibles et des brouilleurs. Les échos produits par le SEN se situent au niveau de la sortie de la démodulation complexe du radar. Il permet alors une validation des traitements radars sans avoir recours a des essais en vols coûteux.

Parmi ses nombreuses librairies d'environnements, nos travaux se sont focalisés sur un brouilleur radar, le **B**rouilleur **C**orrélé **L**ocalisé en **D**istance et en **F**réquence (**BCLDF**).

2.3 BCLDF

L'objectif du BCLDF est de brouiller le radar afin de masquer une cible, en saturant une zone localisée sur la carte DAVA du radar, comme illustré en Fig. 2.

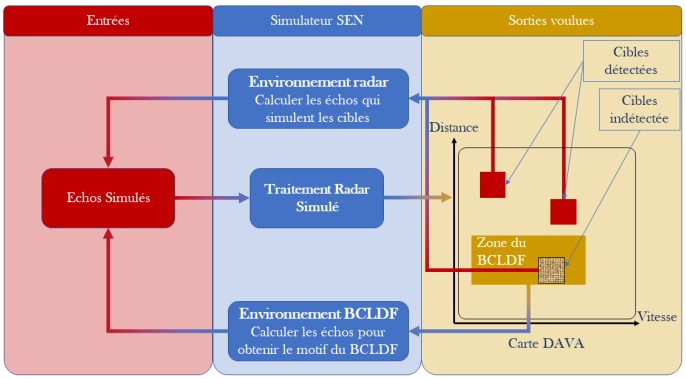

Figure 2: BCLDF et Simulateur SEN

En brouillant une zone rectangulaire sur la carte DAVA, le BCLDF empêche le radar de détecter une possible cible à l'intérieur du motif. Pour y parvenir, on doit calculer les échos qu'un radar est supposé recevoir pour que son traitement génère une telle carte DAVA. L'algorithme était déjà implémenté sur CPU dans le SEN, et notre objectif était de l'accélérer sur GPU, en évaluant la programmation avec OpenACC.

2.4 Plateforme CPU-GPU

Notre plateforme de co-processing est composée d'un CPU Intel Xeon E5-2667 [3] et d'une carte graphique GTX 1080Ti [4]. La version des outils CUDA est la 8.0, celle d'OpenACC la 15.10. Les outils de profiling utilisés sont le *Nvcc Visual Profiler* et celui inclut dans le compilateur PGI pour OpenACC. Les spécifications sont précisées dans le Tableau 1.

3 Méthodologie

Cette section présente certaines de nos implémentations sur GPU avec OpenACC. La méthodologie est divisée en trois

Table 1: Spécifications matérielles

Spécifications	CPU	GPU
Type	Intel Xeon E5-2667	GTX 1080Ti
Fréquence	2.9 GHz	1.48 GHz
Cœurs physiques	6	3584
Mémoire Globale	96 GB	11 GB
Bande passante	51.2 GB/s	484.4 GB/s
Performance FP32	500 GFLOPS	11.34 TFLOPS

catégories principales : l'analyse de l'algorithme, les considérations mémoires, et l'expression du parallélisme. Chaque optimisation est expliquée dans la catégorie lui correspondant. Le facteur d'accélération obtenu par rapport à l'implémentation sur CPU est détaillé à la Figure 3.

Pour une comparaison pertinente, nous avons également implémenté une version optimisée en CUDA de notre algorithme. **Quelque soit la version, le *facteur d'accélération* inclut les transferts mémoires CPU/GPU.**

3.1 Analyse d'algorithme

Analyse du code : la première étape, qui conditionne le reste de notre méthodologie, est d'analyser les propriétés intrinsèques de notre algorithme. Il s'agit ici de vérifier que l'algorithme peut être implémenté sur l'architecture cible, et sinon, comment l'adapter. Par exemple, les fonctions récursives sont très peu efficaces sur des architectures parallèles, et doivent être évitées autant que faire se peut. Dans le cas ou un traitement récursif est essentiel, il peut alors être plus efficace de sauvegarder le contexte sur GPU, de faire le calcul sur CPU, et de reprendre le déroulement de l'algorithme sur GPU. Le temps perdu lors de ces copies additionnelles peut alors être négligeable en comparaison d'une exécution trop lente sur GPU.

Conversion d'algorithme : la seconde étape consiste à adapter certains traitements inadéquats sur l'architecture cible, tout en s'assurant que la précision de cette nouvelle version de l'algorithme répond bien au cahier des charges.

L'environnement BCLDF s'appuie sur des transformées de Fourier rapides (FFT), et la version CPU utilise une librairie récursive, kissFFT. Il y a alors deux options possibles : soit effectuer tous les traitements sur GPU sauf la FFT qui se fera sur CPU comme expliqué précédemment, soit nous devons trouver une librairie aussi précise qui soit optimisée pour les GPUs. Dans notre cas, nous avons choisi la seconde option, en utilisant la librairie cuFFT de Nvidia. En effet, la première solution était trop longue à l'exécution par rapport à l'utilisation de cuFFT. En modifiant l'algorithme initial, nous nous sommes assurés que les signaux de sortie étaient bien en accord avec le cahier des charges : la différence mesurée en absolu était inférieure à 0.01% (***Optimization 1***).

Partition du code CPU/GPU : L'objectif ici est de partitionner le code en accord avec les points évoqués précédemment. L'inconvénient principal d'une architecture hétérogène est l'ajout des temps de transferts mémoires entre les architectures. Mais la latence qui en résulte peut être cachée en exécutant de manière concurrente les calculs et les transferts. Il s'agit ici de tirer parti des outils de profiling pour analyser les motifs d'accès aux données, afin d'identifier au mieux quelles portions sont facilement parallélisables. Bien que ces outils s'avèrent particulièrement efficaces pour des codes simples, leur efficacité se doit d'être couplée par une analyse précise du programmeur pour repérer des motifs d'exécution plus intriqués.

3.2 Considérations mémoires

Localité des données : à l'aide des directives comme *enter data* et *exit data*, il est possible d'optimiser les transferts mémoires en délimitant des zones mémoires, et les clauses *copyin* et *copyout* permettent de choisir quelles données seront à transférer sur le GPU. OpenACC permettant avec un même objet de représenter une zone mémoire sur chaque architecture, on peut choisir avec la directive *update* de mettre à jour les données sur le CPU avec les données sur le GPU (***Optimization 3***).

Zone mémoire unique : pour un parallélisme optimal, il est possible, quant il s'agit de manipuler une variable en lecture seule, de le préciser à l'aide du mot clef *const*, alors que le mot clef *restrict* permet lui de préciser au compilateur que le pointeur correspondant est le seul à modifier la zone mémoire pointée. Le compilateur va alors implémenter des optimisations d'accès poussées (***Optimization 4***).

3.3 Expression du parallélisme

Vectorisation : OpenACC définit trois niveaux de parallélisation, la plus fine étant la *vectorisation*. Il fonctionne comme le parallélisme SIMD, et les opérations sont effectuées pour tous les index d'un vecteur simultanément. Par exemple, utiliser le type *int8* (vecteur de 8 entiers) pour une addition de vecteur va réduire son exécution par un facteur de 8. C'est au programmeur de bien choisir la granularité du parallélisme.

Déroulage de boucle : la parallélisation de boucle est au centre de la programmation parallèle. La première étape est de s'assurer qu'il n'y a pas de dépendance entre chaque itération de la boucle, auquel cas le compilateur ne pourra pas pleinement extraire son parallélisme. Il est possible, en utilisant les clauses *reduction* et *atomic*, de résoudre certaines dépendances de boucles (***Optimization 5***).

4 Résultats et discussions

Temps d'exécution : la Figure 3 montre le facteur d'accélération à l'exécution des différentes versions GPU comparées à l'exécution de référence sur CPU. Toutes les versions GPU incluent les temps de transferts mémoires entre le CPU et le GPU. La quatrième optimisation *OpenACC Opt4* démontre l'importance de la gestion de la mémoire pour obtenir des

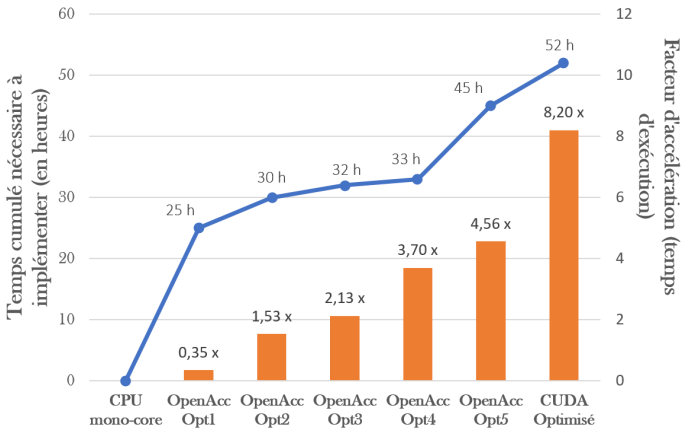

Figure 3: Facteur d'accélération du temps d'exécution & Temps d'implémentation : CPU - OpenACC & CUDA GPU

performances significatives, avec une accélération de 3.7 comparé au temps d'exécution sur CPU. On constate également que la version la plus optimisée avec OpenACC (*OpenACC Opt5*) est plus lente qu'une version CUDA optimisée. Cela est dû au fait qu'OpenACC a toujours certaines lacunes au niveau de la gestion de la mémoire.

Temps de programmation : en considérant la productivité de programmation, la figure 3 montre le temps cumulé nécessaire à l'implémentation de chaque version.

Le long temps d'implémentation de la première optimisation (*OpenACC Opt1*) est indépendante quelque soit l'outil de programmation utilisé, mais dépend par contre de l'architecture cible. Cela vient du fait que cette étape s'appuie sur l'analyse du code existant, et à son adaptation à la nouvelle architecture. Les optimisations suivantes (*OpenACC Opt2* à *OpenACC Opt4*) sont obtenues rapidement, et la dernière version quant à elle est plus ardue, regroupant tous les concepts d'optimisations évoqués précédemment. Parce que la programmation GPU est similaire quelque soit l'outil utilisé, la transition d'un code OpenACC à un code CUDA est assez rapide, et la version CUDA optimisée a été écrite à partir de la version la plus optimisée d'OpenACC. Si un programmeur était amené à n'utiliser que CUDA, sa méthodologie d'optimisation (et le temps nécessaire) aurait suivi un motif similaire à celui obtenu avec OpenACC.

CUDA ou OpenACC : pourquoi pas les deux ? : des résultats mis en valeur précédemment, on peut se demander qui d'OpenACC ou de CUDA serait le meilleur outil. Dans notre cas d'étude, parceque le parallélisme demandait une réécriture significative du code, les deux versions sont assez différentes en comparaison du code CPU de référence. Par contre, les versions en OpenACC sont plus facile à lire que celles en CUDA. En superposant les temps d'implémentation et d'exécution des versions, on observe que le meilleur compromis pour OpenACC est de s'arrêter à la version 4. Après, les optimisations demandent trop d'investissement pour des résultats trop faibles, et il est alors préférable de se tourner vers CUDA pour les optimisations poussées.

5 Conclusion

Dans cet article, nous avons présenté comment certaines optimisations ont impacté les performances d'un algorithme de traitement radar, d'une version CPU à des versions GPU. Le facteur d'accélération maximal est de 4.56 avec OpenACC et de 8.2 avec CUDA. OpenACC est aisé à prendre en main, et est très lisible par rapport à un code en C classique. Sa promesse de peu interférer avec le code CPU n'a été que partiellement tenue ici. En effet, notre algorithme initial était inadapté au parallélisme de données et a du être réécrit en profondeur. Toutefois, les modifications du code en OpenACC restent significativement plus faibles par rapport à CUDA, et, dans certains cas, a l'avantage de toujours pouvoir être exécuté sur CPU.

Parce qu'OpenACC et CUDA partagent la même vue conceptuelle de l'architecture GPU, il est aisé de passer d'un outil à l'autre. Cela peut être rassurant, le développeur ne s'enfermant pas dans une seule voie en choisissant l'un ou l'autre. Récemment, Nvidia s'est porté acquéreur d'un des compilateurs d'OpenACC, *PGI Accelerator*^{TM}, et a communiqué sur les progrès à venir de ce dernier, notamment en ce qui concerne les lenteurs des transferts mémoires. Leur but est à terme de converger au niveau des performances vers CUDA. En résumé, OpenACC est une bonne solution de prototypage rapide d'un algorithme sur GPU, mais, pour tirer parti au mieux de l'architecture, il convient de se tourner vers CUDA.

References

[1] CUDA URL https://developer.nvidia.com/cuda-zone

[2] Doppler-Fizeau effect URL https://en.wikipedia.org/wiki/Doppler_effect

[3] Intel Xeon E5-2667 specifications URL http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2667.html

[4] Nvidia GeForce 1080Ti specification URL https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877

[5] OpenACC standard URL https://www.openacc.org/

[6] OpenCL standard URL https://www.khronos.org/opencl/

[7] OpenMP standard URL https://www.openmp.org/

[8] The International Technology Roadmap For Semiconductors 2.0. Semiconductor Industry Association (2015)

[9] Degurse, J.F., et al.: Architecture GPU pour radar de surveillance spatial et pour radar aéroporté. GRETSI (2013)

[10] Zhang, Q., Deng, Y.: Toward of a GPU accelerated software navigation radar. IEEE 4th International Conference on Software Engineering and Service Science (2013)