TER - Rendu 1

Emile Cadorel, Guillaume Gas, Jimmy Furet, Valentin Bouziat

11 mars 2016

Chapter 1

Introduction

1.1 La Programmation GPGPU

La programmation GPGPU[1] (General Purpose on Graphic Processing Units), est un principe informatique visant à utiliser le processeur graphique comme processeur de calcul générique. L'objectif étant de bénéficier de l'architecture massivement parallèle d'un processeur graphique et ainsi résoudre des problèmes pour lesquels nous avons des algorithmes parallèles efficaces.

En effet le processeur graphique possède une capacité de calcul beaucoup plus grande que celle d'un processeur classique. Pour peu qu'il existe un algorithme parallèle pour un problème donné. La programmation GPGPU n'est applicable qu'avec utilisation des pilotes et des bibliothèques associées. Ces pilotes et bibliothèques généralement proposés par les constructeurs de processeur (ex: Intel, Nvidia, ...) permettent la programmation sur la carte graphique pour des informaticiens chevronnés.

Il existe deux piliers dans la programmation GPGPU, OpenCL : bibliothèques ouvertes par Khronos Group, et Cuda : développée par Nvidia et nécessitant un GPU Nvidia. Les bibliothèques proposent la création de noyaux (kernel), fonction exécuté sur le GPU, la gestion de mémoire sur le GPU et le passage de données entre la RAM du GPU et la RAM du CPU.

La programmation GPGPU étant d'assez bas-niveaux, il existe des bibliothèques permettant d'abstraire certains principes, comme la gestion mémoire.

1.2 SPOC

SPOC[2], Stream Processing with OCaml est une bibliothèque qui propose une abstraction pour la programmation GPGPU. Son principal but est de fournir une solution portable, multi-GPGPU, et hétérogène tout en gardant de hautes

performances.

Elle permet ainsi l'utilisation des systèmes Cuda/OpenCL avec OCaml, l'unification de ces deux API se faisant via une liaison dynamique, ainsi que d'abstraire également les transferts de données entre le CPU et la carte graphique.

Enfin, elle permet aussi d'utiliser le typage statique afin de vérifier les noyaux de calcul, écrits en Sarek.

Il existe deux solutions afin d'exprimer les noyaux :

- Sarek : un langage proche de OCaml. Sarek est compatible avec les bibliothèques existantes, ses performances sont facilement prévisibles et optimisables.
- L'interopérabilité avec les noyaux Cuda/OpenCL : permet de réaliser des optimisations supplémentaires, compatible avec les bibliothèques actuelles, mais moins robuste que Sarek.

1.3 Les Squelettes de programmation

Les squelettes de programmation[3] sont une forme de programmation, où il existe des éléments génériques pré-définis - tel que map, reduce, farm, pipe etc.... Ces squelettes peuvent être personnalisés grâce à du code utilisateur.

Les squelettes[4] permettent une abstraction de haut niveau et aident les utilisateurs à implémenter des algorithmes complexes comme les algorithmes parallèles. Les algorithmes suivant sont généralement implémentés en tant que squelette dans les bibliothèques de programmation GPGPU.

- map[5] - applique une fonction donnée à toutes les entrées d'un tableau.
- reduce - réduction d'un tableau dans une variable, en fonction d'un operateur donné.
- sort - tri un ensemble d'élément à l'aide d'une relation d'ordre donné.
- stream-filtering - applique la même opération que le reduce mais en écartant quelque élément selon un critère donné.
- ...

Chapter 2

Présentation du sujet

2.1 Introduction au domaine

La programmation GPGPU permet de profiter de l'architecture massivement parallèle du GPU. Le GPU possède une architecture SIMD (Single Instruction Multiple Data), proposant de nombreuse unités de calcul.

Le GPU est decoupée en plusieurs TCP (Texture Processing Cluster), eux-même découpés en plusieurs SM. Les SM (Streaming Multi-processor) possèdent une mémoire locale, ils possèdent plusieurs SP (Streaming Processor) permettant des calculs sur nombres flottants, ainsi que des unités SFU (Special Function) permettant le calcul de fonctions spécifiques - cos, sin... . Une mémoire globale est la disposition de chacun des clusters mais est beaucoup plus lente que leur mémoire locale.

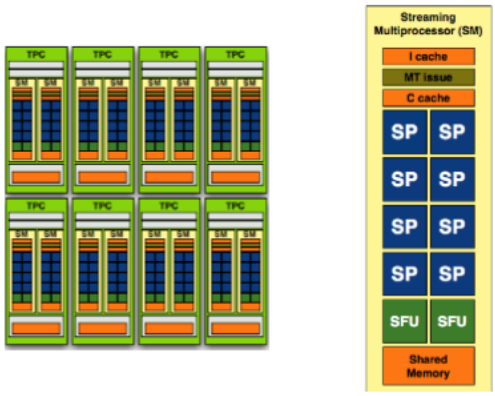

Figure 2.1: Représentation d'une architecture GPU - ref : Introduction à la programmation GPGPU avec Cuda, Mathias Bourgoin Sylvain Jubertie.

L'architecture d'un GPU étant SIMD (Single Intruction Multiple Data), il n'est pas conseillé de créer des programmes complexes proposant beaucoup de conditions, ce qui ferait perdre l'intérêt parallèle de la carte, en effet certains coeurs seraient en attente la plupart du temps.

2.2 Analyse de l'existant

Actuellement, la version de SPOC disponible permet déjà d'effectuer des calculs sur la carte graphique, que ce soit en utilisant Cuda ou OpenCL. Elle nous donne également le choix entre l'utilisation de noyaux Sarek ou bien en Cuda/OpenCL. De plus, SPOC met à notre disposition trois bibliothèques, Sarek en faisant partie, les autres permettant d'utiliser des kernels ecrits en cuda, ou opencl.

Sarek est un langage interne à Spoc permettant de définir des kernel dans un langage proche de ocaml. Ce langage est compilé par Kirc, qui en génère un code exécutable soit par cuda, soit par opencl en fonction de l'architecture de l'ordinateur hôte préalablement detectée.

Spoc propose aussi une bibliothèque Compose, celle-ci permet déjà d'utiliser 3 squelette - Map, Reduce et Pipe. Ces squelettes prennent en paramètre une fonction kernel codé en Cuda ou en Opencl. Le nombres de thread et de block de calcul à créer sur le GPU sont définis de matière automatique.

Chapter 3

Objectifs et organisation

Notre travail tout au long du projet consistera en :

- Modifier le langage Sarek.

Pour créer des squelettes pour Spoc, il faudra modifier le compilateur de Sarek, Kirc. Ajouter un élément de syntaxe au langage Sarek, cet élément sera ensuite remplacé par les fonctions créées par les utilisateurs de Spoc.

```
let foo = fun a -> a + 1;

let map = kern a b n ->
  let open Std in
  let open Skel in
  let idx = global_thread_in in
  if idx < n then
    b.[<idx>] <- { a.[<idx>] }
;;

Kirc.gen map foo;;
```

Le code ci dessus sera alors transformé de la sorte :

```
let map = kern a b n ->
  let open Std in
  let idx = global_thread_in in
  if idx < n then
    b.[<idx>] <- a.[<idx>] + 1;;
```

Nous pourrons aussi, définir de nouveaux squelettes pour la bibliothèque d'extension Compose, mais l'utilisateur devra alors connaître les kernels Cuda ou OpenCL pour les utiliser.

- Créer des squelettes, pour Sarek.

- Effectuer des test de performances sur des calculs parallèlisés en les comparant à une exécution séquentielle, mais aussi comparer nos squelettes à d'autres squelettes présent dans d'autres bibliothèques GPGPU.
- Réaliser une documentation.

Bibliography

[1] *Introduction à la programmation GPGPU avec Cuda, Mathias Bourgoin Sylvain Jubertie.*

[2] *Page d'accueil du projet SPOC.* http://mathiasbourgoin.github.io/SPOC/.

[3] *Skeleton Programming for Heterogeneous GPU-based Systems.* http://homepages.inf.ed.ac.uk/msteuwer/papers/hips2011.pdf.

[4] *SkePU.* http://www.ida.liu.se/labs/pelab/skepu/index.html.

[5] *The map operation.* https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_unitsMap.