ISIMA
Institut Supérieur d'Informatique de
Modélisation et de leurs Applications

Campus des Cézeaux
BP 10 125
63 173 AUBIERE CEDEX

Rapport de projet
3ème année ingénieur
Génération de code OpenCl

Maxime Escourbiac - Jean-Christophe Septier
Responsable ISIMA : Jonathan PASSERAT-PALMBACH & Romain REUILLON

9 mai 2012

Remerciements :

Nous tenons à remercier **Jonathan PASSERAT-PALMBACH** , qui nous a épaulés pour la réalisation de ce projet. Il a été très présent pour nous apporter son aide et ses conseils.

Nous remercions **Romain REUILLON** pour nous avoir aidés à débuter le projet lors de réunions.

Nous remercions également **Olivier CHAFIK**, qui nous a apporté son aide lors du développement de notre solution, afin d'intégrer nos modifications à son projet.

Table des figures

1.1 Différences entre un CPU et un GPU 2
1.2 Architecture simplifiée d'un Streaming MultiProcessor d'une carte NVIDIA Fermi C2050, selon [PASSERAT 2011] 3
1.3 Exemple d'un Warp pour le Warp-Level Parallelisation 5

2.1 Architecture d'OpenCL. 6
2.2 Exemple de code source OpenCL. 7
2.3 Exemple de kernel OpenCL. 8

4.1 Diagramme UML simplifié de ScalaCLCollections. 16
4.2 Diagramme UML simplifié de ScalaCLPlugin. 17
4.3 Sortie en mode VERBOSE 22

Résumé :

Dans le domaine de la simulation, il est nécessaire de réaliser un certain nombre de réplications d'une même expérience, afin d'obtenir des résultats statistiquement corrects. Pour réduire le temps d'exécution de ces réplications, la parallélisation est utilisée. Un des moyens utilisés pour paralléliser est l'emploi de GPU (Graphics Processing Units). Cette solution permet à faible coût d'exécuter les mêmes opérations grâce aux threads : les nombreuses unités de calcul parallèle disponibles sur un GPU.

Les chercheurs du LIMOS ont proposé une publication proposant une nouvelle approche de parallélisation, permettant de résoudre une limitation matérielle des GPU au niveau de l'accès à la mémoire vive et des exécutions de codes indépendants. Cette solution se nomme Warp-Level-Parallelism.

Notre projet a consisté à implémenter ce paradigme dans un langage de haut niveau, nommé ScalaCL. ScalaCL est un projet Open Source créé par Olivier Chafik, permettant de générer du code OpenCL à partir d'un code Scala.

La première étape de notre projet a été d'analyser la publication afin de comprendre l'intérêt de ce nouveau paradigme pour la réplication de simulations.

La seconde étape a été ensuite d'insérer ce paradigme dans le projet ScalaCL. Nous avons d'abord réaliser une longue analyse sur le fonctionnement de cet outil. L'implémentation a été divisée en deux partie : l'implémentation de nouvelles collections ScalaCL, puis l'implémentation d'un plugin du compilateur ScalaCL pour convertir nos fonctions ScalCL en kernel OpenCL.

Keywords :

OpenCL, Scala, Programmation parallèle, GP-GPU, ScalaCL, Warp-Level-Parallelism

Abstract :

Recent simulations need multiple replications to observe a real tendency and conclude on the results. The main issue of replicating experiences is the time needed to running them all. To solve this problem, a good practice is to parallelize the simulations. This approach requires parallel computers such as a multiprocessing machine (with many cores). In this project, we will focus on GP-GPUs (General-Purpose Graphics Processing Units). These devices provide great computational power at a low cost but are limited to perform the same operation on several items of data through threads : the independent computational units of GP-GPU device. This method is called Single Instruction, Multiple Threads (SIMT).

LIMOS researchers proposed an approach to improve the general performance of SIMT. The concept is to use small groups of threads, called warps to execute independent replications. They named their solution Warp Level Parallelism (WLP).

Our project consists in implementing a Warp-Level-Parallelism approach in a high level language Scala using ScalaCL project. ScalaCL is an open source project developed by Olivier Chaffik to adapt Scala to a GP-GPU context using OpenCL API.

The first step of our project was to analyse indetail LIMOS's publication to understand the mechanism of Warp Level Parallelism and to adapt the NVIDIA CUDA implementation of WLP, provided in the publication to benchmark the approach, to an OpenCL compliant native code.

The second step was to insert WLP to ScalaCL project. Before the implementation we had to analyse the models adopted by Olivier Chafik to convert Scala code into a C/C++ code. The project is divided in two parts, the first is ScalaCL Collections which contains OpenCL packed collections offering the same behaviour such as the standard Scala Collections, and the second is ScalaCL Compiler Plugin which converts Scala functions into OpenCL kernels at compile time.

As a result, we propose a ScalaCL WLP implementation and have contacted ScalaCL's original developer (Olivier Chaffik) for our feature to be inserted in the ScalaCL project.

Keywords :

OpenCL, Scala, Parallel Computing, GP-GPU, ScalaCL, Warp-Level-Parallelism

Table des matières

1 Utilisation des GPU dans le contexte de la simulation **2**
- 1.1 Les cartes GPU . . . 2
- 1.2 Méthode de parallélisation . . . 3
  - 1.2.1 Le paradigme SIMT et le Thread Level Parallelism . . . 4
  - 1.2.2 Introduction au paradigme Warp-Level . . . 4
  - 1.2.3 Le paradigme Warp-Level . . . 4

2 OpenCL **6**
- 2.1 Définition . . . 6
- 2.2 L'architecture . . . 6

3 Les spécificités du Scala **9**
- 3.1 Introduction . . . 9
- 3.2 Object . . . 9
  - 3.2.1 Singleton . . . 9
  - 3.2.2 Factory . . . 10
  - 3.2.3 Objet compagnon . . . 10
- 3.3 Trait . . . 10
- 3.4 La programmation fonctionnelle . . . 11
  - 3.4.1 Les fonctions littérales . . . 11
  - 3.4.2 Le type fonction . . . 11
  - 3.4.3 Les closures . . . 12
- 3.5 L'écriture de plugins de compilation . . . 13
  - 3.5.1 le compilateur Scala . . . 13
  - 3.5.2 Ajouter un plugin scala . . . 13

4 ScalaCL **15**
- 4.1 Introduction . . . 15
- 4.2 Description de l'architecture . . . 15
  - 4.2.1 ScalaCLCollections . . . 15
  - 4.2.2 ScalaCLPlugin . . . 17
- 4.3 Insertion du WLP dans ScalaCL . . . 18
  - 4.3.1 Introduction . . . 18
  - 4.3.2 Modifications sur ScalaCLCollection . . . 18
  - 4.3.3 Modifications sur ScalaCLPlugin . . . 20
- 4.4 Premiers résultats . . . 21

Introduction

Dans le cadre de notre troisième année de l'ISIMA, nous avons réalisé un projet sous le tutorat de Jonathan PASSERAT-PALMBACH concernant la parallélisation dans un contexte de réplication de simulations. En effet, il est nécessaire, lorsque l'on réalise une simulation, de réaliser un certain nombre de réplications pour pouvoir conclure sur les résultats obtenus, et obtenir un intervalle de confiance. On utilise donc depuis quelques années la parallélisation sur GPU. C'est une solution bon marché pour réaliser des parallélisations sur une machine de bureau.

Toutefois, le paradigme actuel de parallélisation pose problème, à cause de la limitation matérielle d'accès à la mémoire vive des GPU. Les chercheurs du LIMOS ont réalisé une étude décrivant un nouveau paradigme de parallélisation, le Warp-Level-Parallelism. Ce paradigme permet de pallier la limitation matérielle des GPU lorsqu'on les utilise dans le cadre de la réplication de simulations.

Il nous a été demandé d'implémenter ce paradigme. Toutefois, l'utilisation de celui-ci devait être simple d'utilisation. Nous avons donc choisi de réaliser notre projet sur une technologie récente et en cours de développement : ScalaCL. Cet outil, créé par Olivier Chafik, est une extension open-source du langage Scala, permettant de transformer du code Scala en code OpenCL, afin de pouvoir être lancé sur GPU.

Pour ce projet, nous avons donc étudié les avantages du Warp-Level-Parallelism dans le contexte de la réplication de simulations sur GPU, mais aussi le fonctionnement du projet ScalaCL.

Pour vous présenter ce rapport, nous commencerons par expliquer les différentes notions que nous avons eu besoin d'apprendre pour la réalisation de ce projet (notamment les différents paradigmes de parallélisation, et les différents langages utilisés dans ce projet), puis nous vous présenterons le projet ScalaCL. Enfin nous expliquerons l'implémentation du nouveau paradigme de parallélisation.

Chapitre 1

Utilisation des GPU dans le contexte de la simulation

1.1 Les cartes GPU

Un GPU (Graphics Processing Unit), ou processeur graphique est un circuit intégré qui permet d'assurer les fonctions d'affichage d'une machine. Il possède une structure hautement parallèle qui le rend efficace pour toutes les tâches de calcul graphique. Il est généralement relié à la carte mère via un port PCI express.

Initialement utilisé pour une large palette de tâches graphiques comme le rendu 2D/3D ou le décodage de flux vidéo, le GPU est utilisé depuis peu (environ 5 ans, avec le développement de NVIDIA CUDA) dans le domaine du calcul haute performance afin de paralléliser les calculs et donc de réduire le temps d'exécution des applications. On parle dès lors de GPGPU pour *General Purpose computing on Graphics Processing Units*, ou plus communément de *GPU computing*.

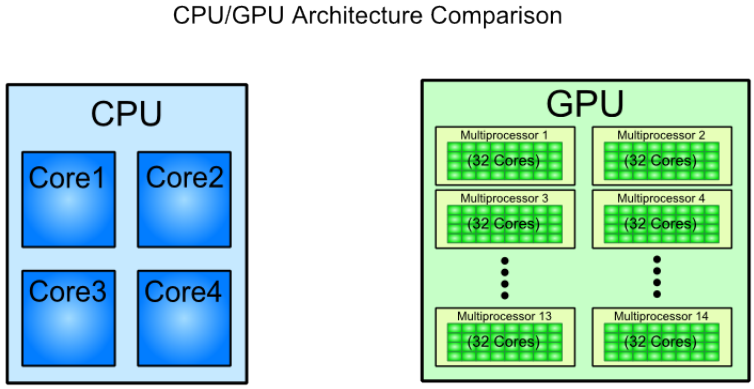

FIGURE 1.1 – Différences entre un CPU et un GPU

Contrairement au CPU qui possèdent un faible nombre de Cores, l'architecture des cartes GPU compte un ensemble de multiprocesseurs (ou SM pour Streaming Multiprocessor). Chacun de ces SM possèdent des unités de calculs, nommée Streaming Processor (SP). Un warp scheduler va permettre à un groupement de 32 SP nommés warps d'accéder à la mémoire.

Un thread GPU n'a pas tout à fait le même sens qu'un thread CPU. À l'inverse des threads CPU, les threads GPU sont extrêmement légers ce qui signifie qu'un changement de contexte est une opération peu coûteuse. Les nouvelles cartes NVDIA possèdent maintenant deux warp scheduler, contrairement aux anciens modèles qui n'en possédaient qu'un. Cela permet donc d'accélérer l'accès à la mémoire.

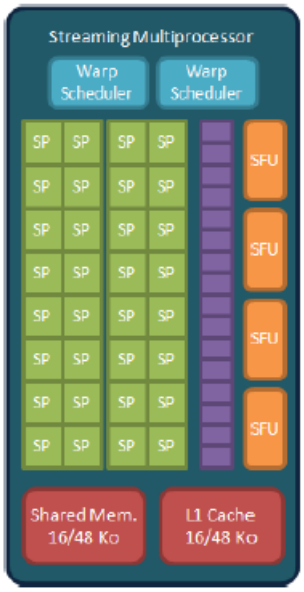

FIGURE 1.2 – Architecture simplifiée d'un Streaming MultiProcessor d'une carte NVIDIA Fermi C2050, selon [PASSERAT 2011]

1.2 Méthode de parallélisation

Afin de réaliser plusieurs réplications d'une simulation, permettant ainsi de réaliser un intervalle de confiance pour celle-ci, la parallélisation est utilisée depuis les années 90 sous le nom de MRIP (Multiple Replications in Parallel) [PAWLIKOWSKI 1994]. Chaque réplication est réalisée en parallèle.

Depuis peu, on utilise des cartes graphiques afin de profiter de leur grand nombre de threads disponibles. Mais une des limites des GPU est la médiocrité de l'accès mémoire. En attendant une amélioration de la part des constructeurs, notamment par l'amélioration de la mémoire du GPU (GDDR), il est possible d'améliorer les performances grâce à une autre vision de la parallélisation du programme.

1.2.1 Le paradigme SIMT et le Thread Level Parallelism

La méthode actuelle de parallélisation utilisée par OpenCl ou CUDA est le paradigme SIMT (Single Instruction Multiple Threads). C'est une amélioration du paradigme SIMD (Single Instruction, Multiple Data). Avec SMID, la même instruction est exécuté en parallèle avec des données différentes en entrées.

SIMT permet donc de réaliser plus facilement des applications GPU. Le développeur crée donc une seule fonction, appelée Kernel, et manipule les threads plutôt que les vecteurs utilisés en SIMD.

Ces kernels se subdivisent en blocs composés de warps . Un warp est un groupe de threads pour le scheduling, mais ce n'est pas une partie de circuit exécutant du code à part entière. Les kernels vont ensuite être exécutés sur un Streaming MultiProcessor. Chaque SM dispose d'une mémoire partagée ou Shared Memory.

1.2.2 Introduction au paradigme Warp-Level

Il existe toutefois deux problèmes pour l'application de ce paradigme à la réplication de simulations. En utilisant une réplication par thread, peu de threads sont utilisés. Hors, les performances des GPU sont optimales lorsque les cartes sont pleinement utilisées. Mais le principal problème est que, lorsque l'on réalise une simulation stochastique, les tirages de nombres aléatoires vont se faire au même point. Il va donc y avoir un problème pour les Threads appartenant à un même Warp. L'exécution se fera chacun son tour pour avoir accès à la mémoire, et l'exécution sera donc sérialisée.

C'est pour cette raison que l'objectif de notre projet est donc d'implémenter un nouveau paradigme palliant ces problèmes : Warp-Level Parallelism.

1.2.3 Le paradigme Warp-Level

Pour accélérer l'exécution de plusieurs réplications d'une simulation stochastique, un autre paradigme a été créé : le Warp-Level Parallelism (WLP). En effet, pour prendre en compte les contraintes décrites précédemment, on décide de n'exécuter qu'un Thread par Warp. Ainsi, le Warp-Scheduler n'aura à réaliser que l'accès à la mémoire pour une seule réplication. Il n'y a donc plus les problèmes d'accès au même point de plusieurs Thread lors du tirage des nombres aléatoires. Par ailleurs, aucune branche divergente n'apparaît, ce qui évite les portions de code exécutées en série.

Enfin, baser le calcul au niveau des warps permet de créer des threads virtuels, puisque 32 threads seront créés par warp. Ainsi, on augmente l'occupation de la carte, améliorant de ce fait l'ordonnancement. On autorise donc un seul Thread par warp à s'exécuter sur les 32 possibles.

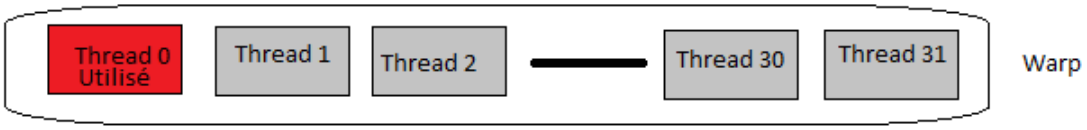

FIGURE 1.3 – Exemple d'un Warp pour le Warp-Level Parallelisation

Sur la Figure 1.3, on voit que seul le Thread 0 est activé dans son Warp. Les autres ne fonctionnent pas.

Après différents tests réalisés, on remarque pour une simulation simple avec un nombre de réplications situé entre 0 et 600 que les performances avec le paradigme Warp-Level Parallelism sont bien meilleures (ratio max x6) que pour le Thread Level Parallelism. Toutefois, arrivé à un très grand nombre de réplications (> 700), les performances convergent du fait du nombre limité de Warps pouvant être gérés par le GPU.

La publication [PASSERAT 2011] ne fait état que d'une implémentation CUDA de ce paradigme. De plus, les auteurs soulignent l'intérêt d'une automatisation de la méthode, évitant à l'utilisateur d'insérer des lignes désactivant certains threads dans son code.

Chapitre 2

OpenCL

2.1 Définition

OpenCL (Open Computing Langage) est un projet ambitieux, initialement lancé par Apple. Les spécifications ont été proposées et acceptées par le consortium Khronos, qui par ailleurs s'occupe aussi d'OpenGL.La première version officielle (1.0) est sortie en juin 2008. Actuellement, la version 1.2 de l'API est proposé aux fabriquant. Le but de ce projet est de permettre aux développeurs de tirer parti des énormes capacités de calcul des processeurs graphiques (GPUs) ou des processeurs classiques (CPUs) d'aujourd'hui. Chaque fabricant propose son implémentation de l'API pour le matériel qu'ils fournissent. Cette situation est ambiguë, car il existe une hétérogénéité dans leurs performances. Le cas de NVIDIA est symptomatique. En effet, NVIDIA est propriétaire de la plateforme CUDA qui est un concurrent direct d'OpenCL. Ce qui a été un frein pour le déploiement de l'API car NVIDIA possède 32% de part de marché, et est leader dans le domaine des cartes graphiques haut de gamme.

2.2 L'architecture

Voici un schéma representant l'architecture d'OpenCL.

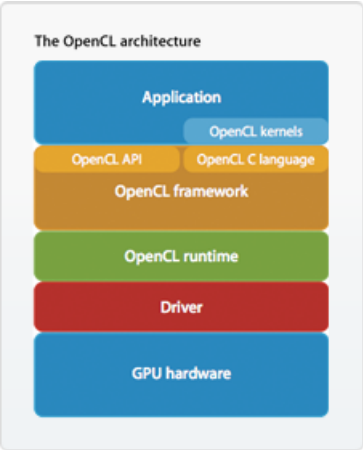

FIGURE 2.1 – Architecture d'OpenCL.

L'utilisation d'OpenCL peut se décomposer en deux partie. La première est la partie

qui va manager le code à exécuter et la deuxième la partie du code qui va être directement exécuté sur la plateforme utilisée. Pour manager son code source OpenCL, l'API propose plusieurs étapes.

- Création d'un contexte selon le device que l'on veut utiliser (CPU ou GPU).
- Allocation des buffers mémoire pour les entrées-sorties.
- Récupération du code source à exécuter.
- Création des kernels à exécuter.
- Paramètrage du nombre d'e réplications à effectuer
- Planification de l'exécution sur le device.

Il est intéressant de noter que le code source des kernels peut être récupéré de plusieurs façons. Le kernel qui est dans l'exemple ci-dessous provient d'un fichier source que le programme va compiler directement à l'exécution. On peut aussi utiliser directement le binaire de la fonction ce qui permet d'éviter une phase de compilation qui peut être longue. Voici un exemple de code de création et de lancement de kernel.

```
// create a compute context with GPU device
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// list the available devices
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &nb_devices);
clGetContextInfo(context, CL_CONTEXT_DEVICES, nb_devices, devices, NULL);

// create a command queue on the first GPU
queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(float)*2*num_entries, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*num_entries, NULL, NULL);

// create the compute program
program = clCreateProgramWithSource(context, 1, &fft1D_1024_kernel_src, NULL, NULL);

// build the compute program executable
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the compute kernel
kernel = clCreateKernel(program, "fft1D_1024", NULL);

// set the args values
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, sizeof(float)*(local_work_size[0]+1)*16, NULL);

// create N-D range object with work-item dimensions and execute kernel
global_work_size[0] = num_entries;
local_work_size[0] = 64;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size, local_work_size, 0, NULL, NULL);
```

FIGURE 2.2 – Exemple de code source OpenCL.

Le code source de calcul possède quelques particularités par rapport à un code classique en C. Un kernel est en effet toujours une fonction dont le prototype commence par le mot-clé **__kernel**. Pour la suite, on peut utiliser des fonctions permettant de récupérer des informations sur l'exécution du code à proprement parlé comme par exemple le numéro de work-item. Voici un exemple de code source pouvant être directement exécuté sur un device compatible OpenCL.

```
// This kernel computes FFT of length 1024. The 1024 length FFT is decomposed into
// calls to a radix 16 function, another radix 16 function and then a radix 4 function

__kernel void fft1D_1024 (__global float2 *in, __global float2 *out,
                          __local float *sMemx, __local float *sMemy) {
  int tid = get_local_id(0);
  int blockIdx = get_group_id(0) * 1024 + tid;
  float2 data[16];
  // starting index of data to/from global memory
  in = in + blockIdx;  out = out + blockIdx;
  globalLoads(data, in, 64); // coalesced global reads
  fftRadix16Pass(data);      // in-place radix-16 pass
  twiddleFactorMul(data, tid, 1024, 0);
  // local shuffle using local memory
  localShuffle(data, sMemx, sMemy, tid, (((tid & 15) * 65) + (tid >> 4)));
  fftRadix16Pass(data);               // in-place radix-16 pass
  twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
  localShuffle(data, sMemx, sMemy, tid, (((tid >> 4) * 64) + (tid & 15)));
  // four radix-4 function calls
  fftRadix4Pass(data); fftRadix4Pass(data + 4);
  fftRadix4Pass(data + 8); fftRadix4Pass(data + 12);
  // coalesced global writes
  globalStores(data, out, 64);
}
```

FIGURE 2.3 – Exemple de kernel OpenCL.

La décomposition de la partie calcul et de la partie scheduler est une fonctionnalité très interrésante. Car cela oblige aux développeurs à rendre leurs programmes plus modulaire. De plus, il est très facile pour une entité de créer un "launcher" universel qui permettra d'économiser du temps de dévellopement de la partie non-métier du programme.

Pour conclure sur cette partie, on peut dire que OpenCL est une API qui a de l'avenir. En effet malgré certaines réticence de NVIDIA, OpenCL a été adopté par les autres constructeurs comme AMD ou Intel qui en font leur fer de lance. Intel propose désormais une implémentation matérielle d'OpenCL dans leurs derniers processeur basé sur l'architecture Ivy Bridge.

Chapitre 3

Les spécificités du Scala

3.1 Introduction

Scala est un langage de programmation multi-paradigme conçu à l'École polytechnique fédérale de Lausanne (EPFL) [ODERSKY] pour exprimer les modèles de programmation courants dans une forme concise et élégante. Son nom vient de l'anglais Scalable language qui signifie à peu près « langage adaptable » ou « langage qui peut être mis à l'échelle ». Il peut en effet être vu comme un métalangage. [1]
Ce chapitre montre les fonctionnalités avancées de ce langage qui ont été utilisées lors de ce projet.

3.2 Object

Object est une particularité Scala. On peut s'en servir principalement de 3 façons, en singleton, en factory et pour finir en objet compagnons.

3.2.1 Singleton

Object est une particularité Scala. Cela permet d'implémenter très facilement le design patern **singleton**. Pour l'utiliser, il suffit de remplacer le mot clé class par **object** lors de la déclaration. **object** n'est pas un élément qui peut être instancié. Donc il est impossible d'avoir plusieurs réplications de cet objet. On retrouve le comportement du design pattern Singleton.

```
object MaitreDuJeu () {
    def kick(...) = {...}
    def slay(...) = {...}
}
```

A l'appel, on obtiendra.

```
MaitreDuJeu.kick(...)
MaitreDuJeu.slay(...)
```

1. Langage qui permet la définition formelle d'un langage de programmation

3.2.2 Factory

Une autre utilisation du type object est l'implémentation du design pattern factory. Il suffit de créer une méthode qui retourne l'objet créé. En voici un exemple ci-dessous.

```
object JoueurFactory () {
   def creerJoueur(nom : String) : Joueur = {
      return new Joueur(nom);
   }
}
```

3.2.3 Objet compagnon

La troisième et dernière utilisation présentée est la possibilité de l'associer à une classe de même nom et qui permettra à cette classe de posséder des attributs et des membres statiques. Voici une comparaison entre un code Scala et son équivalent Java.

```
class maClass {
    private int monAttr1;
    public static int monAttr2;

    public static void maMethode() {
        ...
    }
}
```

```
class maClass() {
    private var monAttr1 : int ;

    object maClass {
        private var monAttr2 : int ;
        def maMethode() = {
            ...
    }
}
```

3.3 Trait

Un trait est une classe abstraite, mais avec quelques particularités :

- Un trait est déclaré avec le mot clé "trait" alors que les classes abstraites le sont avec "abstract class".
- On ne peut pas déclarer de champs dans le constructeur d'un trait, c'est à dire écrire val ou var avant le nom d'un argument du constructeur principal(cette restriction sera supprimée dans les prochaines versions de Scala).
- On peut hériter de plusieurs traits.

Exemple :

```
trait monTrait {
        def maMethode() = {...}
}
```

Ce type de données est préféré aux classes abstraites car l'héritage, en Scala, possède un facteur bloquant. En effet, il est impossible pour une classe d'hériter de plusieurs classes abstraites.

3.4 La programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui limite les effets de bords en interdisant les opérations d'affectation. Les programmes fonctionnels purs sont un ensemble de fonctions imbriquées les unes dans les autres. La contrainte principale de la programmation fonctionnelle est qu'une fonction ne peut avoir qu'une seule sortie.

Il existe des langages, comme le Scala, qui peuvent utiliser à la fois le paradigme fonctionnel et le paradigme impératif. On appelle ces langages des langages impurs. Ils ont le gros avantage de pouvoir s'adapter au type de problèmes à résoudre au dépend de la baisse de robustesse engendrée par l'interdiction des opérations d'affectation.

3.4.1 Les fonctions littérales

Concrètement, une fonction littérale est une fonction sans nom. Elle est souvent utilisée comme argument de fonction. Par exemple, la méthode foreach de la classe Array[T] prend une fonction en paramètre et va l'appliquer à tous les éléments contenus dans le tableau. Une fonction littérale se présente sous la forme suivante :

```
(x:Int, y:Int) => x + y
```

où les arguments sont situés entre parenthèses, et le corps de la fonction est situé après la flèche.

3.4.2 Le type fonction

L'une des faiblesses de Java est l'absence de pointeur de fonction. En effet, on ne peut pas passer une fonction en paramètre dans Java. Scala propose de corriger cela à l'aide du type fonction. Le Scala étant un langage tout objet, une fonction est un objet comme un autre. Elle peut être affectée à une variable **val** ou **var**. Une variable fonction se déclare comme ceci :

```
val sum : (int, int) : int =  (a : int, b : int) => a + b
val res = sum(4,5)
// res = 9
```

On remarque que pour la déclaration de sum, on définit le type **int, int) : int**. Le prototype de la fonction donne le type de la variable. On peut utiliser le type fonction en paramètre de méthode.

Un exemple classique est l'implémentation d'une fonction qui applique un traitement spécifique à un objet passé en paramètre.

```
def operation( x : int, y : int, op : (int,int) => int ) : int = {
    op(x,y)
}
```

La méthode opération peut être appelée avec **sum** comme paramètre, car il y a concordance entre leur prototype. On peut l'utiliser avec le code suivant.

```
val result = operation(2,3,sum)
// val = 5
```

On retrouve ici le comportement des pointeurs de fonction en C, fonctionnalité que l'on peut simuler en Java, à l'aide d'une interface avec la méthode de calcul a implémenter. Une autre application est de faire des prédicats, utilisables sur des collections, à l'instar de la bibliothèque standard C++ (STL) sur la méthode **sort**.

3.4.3 Les closures

Par définition, une closure est une fonction qui possède un état interne. C'est à dire que ce type de code s'exécute selon un contexte prédéfini avant. Cependant, la closure est toujours à l'écoute des variables qu'on lui a intégrées. L'exemple suivant va montrer plus en détail son fonctionnement classique.

```
def compteur( debut : int) = (x : int) => x + debut
val cpt1 = compteur(1);
// cpt1 est un pointeur de fonction
val res = cpt1(3);
// res = 4
```

L'exemple suivant montre quant à lui les effets externes après la déclaration de la closure.

```
var debut = 0;
def compteur = (x : int) => x + debut
val cpt1 = compteur();
// cpt1 est un pointeur de fonction
val res = cpt1(3);
// res = 3
debut = 10; // Modification de la variable initiale
val resModif = cpt1(5)
```

```
// resModif = 15
```

On remarque que la modification de la variable **debut** affecte le comportement de la closure. En effet, toutes les variables externes utilisées dans les closures sont des références. Donc les valeurs prises en comptes sont les valeurs actuelles lors de l'appel de la fonction.

3.5 L'écriture de plugins de compilation

3.5.1 le compilateur Scala

Par définition, le compilateur scala se veut simple et flexible. La flexibilité provient du fait que l'on peut très facilement modifier le comportement du compilateur à l'aide de plugins externes. Attention, cette fonctionnalité doit être utilisé avec précaution. La norme Scala conseille de vérifier si des bibliothèque peuvent permettre d'implémenter le problème en questions. Il est recommandé d'utiliser des plugins dans les cas suivants :

- Contrainte supplémentaire à la compilation (Ajout de robustesse au code).
- Optimisations à la compilation.
- Modification de la syntaxe Scala.

L'avantage de l'utilisation de plugin est la non-modification du compilateur de base. En effet, les plugins disponibles sont distribuables de la même façon qu'une library. Un plugin peut être classé en deux catégories. La première est l'ajout d'une phase de compilation comme un pre-check ou transformation d'une partie de code spécifique. La deuxième est la modification du comportement des annotations.

3.5.2 Ajouter un plugin scala

Le plugin en exemple ci-dessous, propose de détecter les divisions par zéro évidente. Le code est tiré du cite officiel (www.scala-lang.org).

```
class DivByZero(val global: Global) extends Plugin {
  import global._
  val name = "divbyzero"
  val description = "checks for division by zero"
  val components = List[PluginComponent](Component)

  private object Component extends PluginComponent {
    val global: DivByZero.this.global.type = DivByZero.this.global
    val runsAfter = "refchecks"
    val phaseName = DivByZero.this.name
    def newPhase(_prev: Phase) = new DivByZeroPhase(_prev)

    class DivByZeroPhase(prev: Phase) extends StdPhase(prev) {
      override def name = DivByZero.this.name
      def apply(unit: CompilationUnit) {
        for ( tree @ Apply(Select(rcvr, nme.DIV), List())) <- unit.body;
             if rcvr.tpe <:< definitions.IntClass.tpe) {
          unit.error(tree.pos, "definitely division by zero")
        }
      }
    }
  }
```

}

Le code du plugin est composé d'une seule classe dérivante de Plugin. Elle possède un seul paramètre pour son constructeur qui est de type **global**. Le plugin doit définir au minimum un object dérivant PluginComponent.

Le nom des object doivent être inclus dans la liste **components**. De plus, chaque composant doit définir une méthode **newPhase** qui crée la phase du compilateur. Et pour finir, chaque phase doit redéfinir la méthode **apply** pour implémenter la partie concrète du compilateur.

Comme on peut l'apercevoir au travers de l'exemple précédent, la réalisation d'un plugin Scala est une tâche extrêmement légère par rapport à la modification d'un compilateur libre d'un langage comme le C. Cette fonctionnalité a un fort potentiel pour la création d'environnement de développement robuste et fiable. On pourrait penser que dans l'avenir, ce genre de mécanisme permettra au Scala d'être facilement maintenable par des communautés de développeurs.

Chapitre 4

ScalaCL

4.1 Introduction

ScalaCL est un langage dédié [1] développé par Olivier Chafik. ScalaCL est scindé en deux parties. La première est ScalaCLCollections qui réécris les collections standards de Scala, en l'adaptant au contexte OpenCL.

La deuxième et dernière partie est un plugin de compilation Scala permettant d'utiliser les éléments implémentés dans ScalaCLCollections. L'objectif final est d'obtenir un programme exécutable sur un GP-GPU en utilisant l'API OpenCL. Depuis peu, ScalaCL a été intégré au projet nativelibs4java [2].

4.2 Description de l'architecture

4.2.1 ScalaCLCollections

D'un point de vue modélisation, ScalaCLCollections est composé de deux parties. La première est la réécriture de collections Scala et la seconde est une modélisation de méthodes appelées avec OpenCL. Actuellement, trois collections ont été implémentées.

- CLArray[T].
- CLFilteredArray[T].
- CLFunction[A,B].

CLArray et CLFiltered reprennent le comportement des classes issues des collections standards de Scala. Cependant, des fonctionnalités ont été ajoutées. Elles possèdent une structure asynchrone. En effet, il est impossible de lire une donnée en cours de lecture et vice-versa. Ce comportement est quasi-indispensable dans des environnements hautement parallèles.

CLFunction se situe entre le modèle et les collections car le plugin ScalaCL convertit les fonctions écrites en Scala en instance de CLFunction. Cependant, il est possible de créer manuellement cette instance afin de travailler au plus près des collections. Cette méthode est tout de même déconseillé car le plugin permet d'optimiser la fonction modéliser.

Voici un exemple de code utilisant directement CLFunction :

1. Langage de programmation dont les spécifications sont dédiées à un domaine d'application précis
2. Repository de library d'inter-opérabilité entre Java/Scala et C, C++, ObjectiveC

```
def testMapArrayCapture {
    val rg = (0 until n).map(_ * 10).toCLArray

    val f: CLFunction[Int, Int] =
      (
        (x: Int) => rg(x) - x,
        Array("_1[_] - _"),
        impl.CapturedIOs(
          Array(IntCLDataIO.asInstanceOf[CLDataIO[Any]]),
          Array(),
          Array()
        )
      )


    val fCapt = f.withCapture(
      Array(rg.asInstanceOf[CLArray[Any]]),
      Array(),
      Array()
    )
    same(a.map(fCapt), cla.map(fCapt))
  }
```

La modélisation comporte plusieurs caractères intéressants. En effet, il s'agit d'un modèle très complet qui gère toutes les fonctionnalités et les particularités des fonctions. Voici un diagramme UML simplifié pour voir comment sont articulés ces éléments.

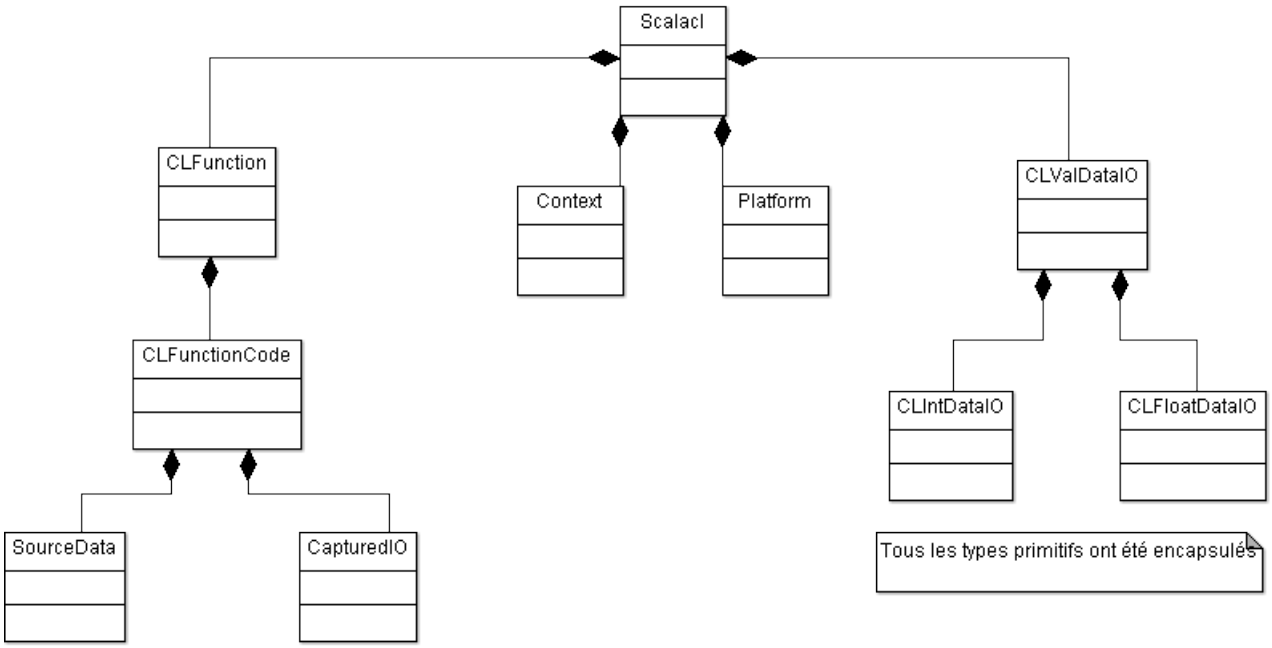

FIGURE 4.1 – Diagramme UML simplifié de ScalaCLCollections.

On remarque que ce diagramme peut être décomposé en trois parties distinctes. La première regroupe **CLFunction** et ses classes filles. Elles servent à modéliser les fonctions en terme d'exécution. **SourceData** comporte la partie métier du code tandis que **CapturedIO** se concentre sur la gestion de la mémoire pour les paramètres d'entrée et la/les valeur(s) de sortie.

La deuxième, CLValDataIO et ses classes filles permettent l'encapsulation des types primitifs issus d'OpenCL. Les types implémentés sont :

- Booléen.
- Byte.
- Char.
- Double.
- Float.
- Int.
- Long.
- Short.
- Tuple.

La troisième et dernière partie assemble **Scalacl**, **Context**, **Platform**. Ces trois classes contiennent des informations sur les conditions d'exécution de la fonction. **Platform** s'occupe de la partie hardware de la machine où est exécuté le code. Tandis que **Context** se concentre sur le device utilisé (CPU ou GPU).

4.2.2 ScalaCLPlugin

La fonction principale du plugin est d'analyser le code Scala utilisant ScalaCLCollection pour le tranformer en CLFunction puis en code C/C++ OpenCL. L'architecture de la partie plugin est vraiment très intéressante. En effet, Olivier Chaffik l'a développé de manière à être très modulaire et robuste. Elle est constituée d'un noyau qui implémente une interface pour chaque fonctionnalité. Voici un diagramme UML qui résume l'architecture.

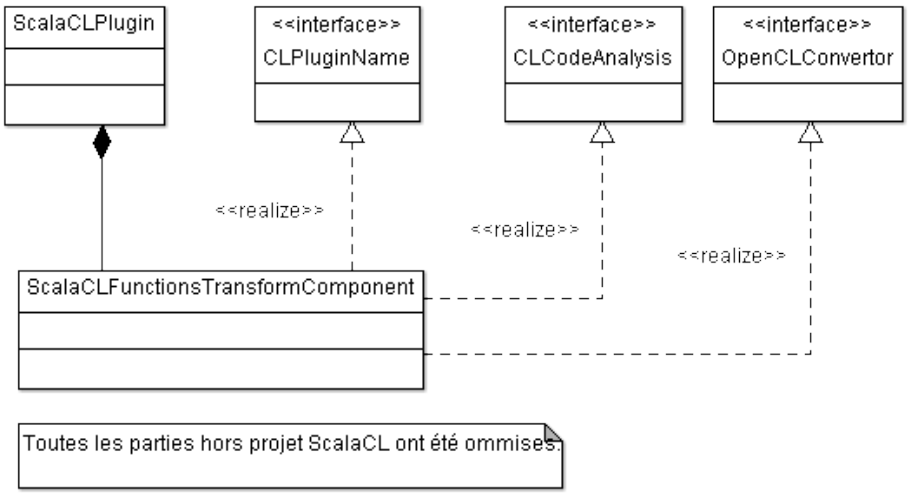

FIGURE 4.2 – Diagramme UML simplifié de ScalaCLPlugin.

La fonction noyau décrite dans le paragraphe précédent est **ScalaCLFunctionTransformComponant**. Elle contient tout la partie fixe du code du plugin. Son objectif principal est d'analyser le code et de le transformer en modèle CLFunction. Cette classe implémente plusieurs interfaces.

CLPluginName définit les éléments comme le package de collections qui seront utilisés pour la construction de la CLFunction. **CLCodeAnalysis** donne, quant à lui, des compléments sur la lecture et la transformation du code Scala lu.

OpenCLConverter est le trait qui transforme le modèle CLFunction en langage C/C++. Cette partie du code est très critique. La moindre modification peut entrainer des erreurs difficiles à résoudre. C'est pour cela qu'il est déconseillé de modifier cette implémentation de ce trait.

4.3 Insertion du WLP dans ScalaCL

4.3.1 Introduction

L'objectif final de ce projet était d'insérer le paradigme Warp-Level dans le projet ScalaCL. Après plusieurs échanges avec Olivier Chafik, nous avons conclu sur la démarche à réaliser pour implémenter cette nouvelle fonctionnalité. Les modifications ont été faite sur les deux composantes du projet, la partie Collection et la partie Plugin.

4.3.2 Modifications sur ScalaCLCollection

Pour commencer, il a fallu modifier le modèle de CLFunctionCode. En effet, l'implémentation du paradigme Warp-Level nécessitant d'effectuer un test au début du code du kernel. C'est pour cela que deux paramètres ont été ajouté à la méthode **buildSourceData** de CLFunctionCode. il s'agit de **bodyPrefix** et de **bodySuffix**. Ces deux paramètres sont des éléments qui seront insérés systématiquement en début et en fin du kernel. Voici une partie du code utilisant cette fonctionnalité.

```
outputFunction(
      "__kernel void array_array",
      Seq(sizeParam, kernelParams),
      Seq(
        bodyPrefix,
        indexHeader,
        sizeHeader,
        kernDeclsArray,
        assigntArray,
        bodySuffix
      ),
      kernelsSource
    )
```

Par ailleurs, Olivier Chafik, nous a conseillé de créer un nouveau package pour mieux exploiter la modification précédente. Le nouveau package obtenue est scalaclWLP. Il reprend la quasi-totalité du code du package scalacl. En effet, dans notre cas, il était pas nécessaire d'apporter des modifications aux collections. La modification se fera uniquement dans la déclaration de la méthode **buildSourceData**.

Le test à ajouter pour le paradigme Warp-Level est constant. La solution retenue a été de modifier la valeur par défaut des paramètres **bodyPrefix** et **bodySuffix**. Voici la déclaration originale de la méthode buildSourceData.

```
def buildSourceData[A, B](
    outerDeclarations: Array[String],
    declarations: Array[String],
    expressions: Array[String],
    includedSources: Array[String],
    extraArgsIOs: CapturedIOs = CapturedIOs(),
    bodyPrefix: Array[String] = Array(),
    bodySuffix: Array[String] = Array()
  )
```

Voici la version adapté pour le paradigme Warp-Level.

```
def buildSourceData[A, B](
    outerDeclarations: Array[String],
    declarations: Array[String],
    expressions: Array[String],
    includedSources: Array[String],
    extraArgsIOs: CapturedIOs = CapturedIOs(),
    bodyPrefix: Array[String] = Array("if(get_global_id(0)==0){"),
    bodySuffix: Array[String] = Array("}")
  )
```

Cette solution n'est sans doute pas la plus propre en terme de génie logiciel. Elle a cependant l'avantage de ne pas trop modifier l'architecture globale de ScalaCLCollection. L'autre solution aurait été de garder un seul package et d'implémenter une buildWLP-SourceData qui aurait été appelé selon le paradigme que l'on veut utiliser.

4.3.3 Modifications sur ScalaCLPlugin

Comme l'avait conseillé Olivier Chafik nous avons créer un nouveau plugin pour ScalaCL. Tout comme avec la partie Collection du projet, il y a une grande majorité du code en commun.

Nous avons utilisé la même architecture que le plugin existant. C'est à dire un noyau ScalaFunctionsTransformComponant implémentant plusieurs interfaces. La majeure partie du travail a consister a connecter les modifications qui ont été faite sur les Collections. Pour cela, l'interface CLPlugin a été retouché pour obtenir **CLWLPPluginName**.

Voici la nouvelle implémentation du trait

```
val ScalaCLPackage          = M("scalaclWLP")
val ScalaCLPackageClass     = ScalaCLPackage.tpe.typeSymbol
val CLDataIOClass = C("scalaclWLP.impl.CLDataIO")
val CLArrayClass = C("scalaclWLP.CLArray")
val CLFunctionClass = C("scalaclWLP.impl.CLFunction")
val CLRangeClass = C("scalaclWLP.CLRange")
val CLCollectionClass = C("scalaclWLP.CLCollection")
val CLFilteredArrayClass = C("scalaclWLP.CLFilteredArray")
val scalacl_ = N("scalacl")
val getCachedFunctionName = N("getWLPCachedFunction")
val Function2CLFunctionName = N("Function2CLFunction")
val withCaptureName = N("withCapture")
```

Le paradigme Warp-Level ne nécessitant pas une analyse ou une écriture particulière du code. Il n'a pas été de corriger fondamentalement **CLCodeAnalysis** et **OpenCLConvertor** mis à part les liaisons avec CLWLPPluginName.

Le noyau, quant à lui, a subi plusieurs transformations. La première est le choix des interfaces à implémenter. Au lieu de CLCodeAnalysis et CLPluginNames, on obtient CLWLPCodeAnalysis et CLWLPPluginNames.

C'est grâce à CLWLPPluginName que le plugin appellera désormais getWLPCachedFunction au lieu de getCachedFunction qui est situé dans ScalaCLCollections.

4.4 Premiers résultats

L'objectif de ce projet sera atteins si les kernels générés par ScalaCL comporteront le test initial en début de code. Pour vérifier ceci, nous avons utilisé la variable d'environnement **SCALACL_VERBOSE** . Elle permet d'afficher une grande partie des traces d'executions de ScalaCL. Voici un code de test de notre implémentation.

```
  def main(args: Array[String]) {

  implicit val context = Context.best(GPU)
  val filt = (
    (x: Int) => (x % 2) == 0,
    Array("(_%2)==0")
  ): CLFunction[Int, Boolean]

  for (dim <- 1 until 1000) {
    val openclArray = (0 until dim).toCLArray.filter(filt).toArray.toSeq
    val openclRangePre: CLIndexedSeq[Int] = (0 until dim).cl.filter(filt)
    val openclRange = openclRangePre.toArray.toSeq
    val scala = (0 until dim).toArray.filter(filt).toArray.toSeq
    }
}
```

On obtient donc en sortie :

```
[ScalaCL] Creating kernel with source <<<
__kernel void array_array(int __cl_size, __global const int* __cl_in0, __global int* __cl_out1) {
    if ( get_global_id(0) == 0) /*Test ISIMA*/ {
    int __cl_i = get_global_id(0);
    if (__cl_i >= __cl_size) return;
    __cl_out1[__cl_i] = (__cl_in0[__cl_i] % 2) == 0;
    }
}

 __kernel void filteredArray_filteredArray(int __cl_size, __global const int* __cl_presence,
                                           __global const int* __cl_in0, __global int* __cl_out1) {
     if ( get_global_id(0) == 0) /*Test ISIMA*/ {
     int __cl_i = get_global_id(0);
     if (__cl_i >= __cl_size) return;
     if (!__cl_presence[__cl_i]) return;
     __cl_out1[__cl_i] = (__cl_in0[__cl_i] % 2) == 0;
     }
 }

__kernel void range_array(int __cl_size, __global const int* __cl_in0, __global int* __cl_out1) {
    if ( get_global_id(0) == 0) /*Test ISIMA*/ {
    int __cl_i = get_global_id(0);
    if (__cl_i >= __cl_size) return;
    __cl_out1[__cl_i] = ((__cl_in0[0] + (__cl_i) * __cl_in0[2]) % 2) == 0;
    }
}

>>>
```

FIGURE 4.3 – Sortie en mode VERBOSE

On remarque que pour chaque kernel généré, on obtient bien le test inséré dans la méthode buildSourceData de CLFunction. Donc pour un code donnée, on peut obtenir sa version dans ce nouveau paradigme.

Conclusion

Lors de ce projet, nous avons réalisé une étude approfondie de l'ensemble des notions nous permettant de mener ce projet à terme. Nous avons d'abord analyser la publication sur laquelle le projet s'est basé. Nous avons donc étudié les différents paradigmes de programmation sur GP-GPU, afin de comprendre les subtilités et les avantages de l'utilisation du Warp-Level-Paradigm dans le cadre de la réplication d'expérience.

Nous avons ensuite étudié Scala, afin de comprendre les fonctionnalités et les particularités de ce langage. Ce langage novateur apporte une nouvelle dimension dans le monde de la programmation. En effet, il était assez difficile pour nous d'imaginer un langage qui puisse être autant adaptable aux besoins de l'utilisateur. Il ne fait aucun doute que le Scala sera bientôt un langage phare.

La plus grande partie du projet a ensuite été l'étude de l'architecture ScalaCL, afin de savoir comment on pourrait intégrer le paradigme Warp-Level sans modifier le comportement du code existant. Cette partie a été notre plus grande difficulté. En effet, ScalaCL est un projet récent et ambitieux , et peu d'informations sont disponibles. Les nombreuses mises à jour, modifiant les parties. dans lesquelles nous avions insérer notre fonctionnalité nous a également posé problème.

Nous avons finalement réussi à implémenter le Warp-Level-Parallelism avec ScalaCL. Il est désormais possible de proposer nos modifications, afin qu'elles puissent être intégrer finalement dans la branche principale projet.

Bibliographie

[PASSERAT 2011] J. Passerat-Palmbach, J. Caux, P. Siregar, and D. Hill, "Warp-level parallelism : Enabling multiple replications in parallel on gpu," in Proceedings of the European Simulation and Modeling Conference 2011, 2011, pp. 76–83, iSBN : 978-90-77381-66-3.

[PAWLIKOWSKI 1994] Pawlikowski, K., Yau, V., and McNickle, D. (1994). Distributed stochastic discrete-event simulation in parallel time streams. In Proceedings of the 26th conference on Winter simulation, pages 723–730. Society for Computer Simulation International.

[ODERSKY] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," Technical Report IC/2004/64, EPFL Lausanne, Switzerland, Tech. Rep., 2004.