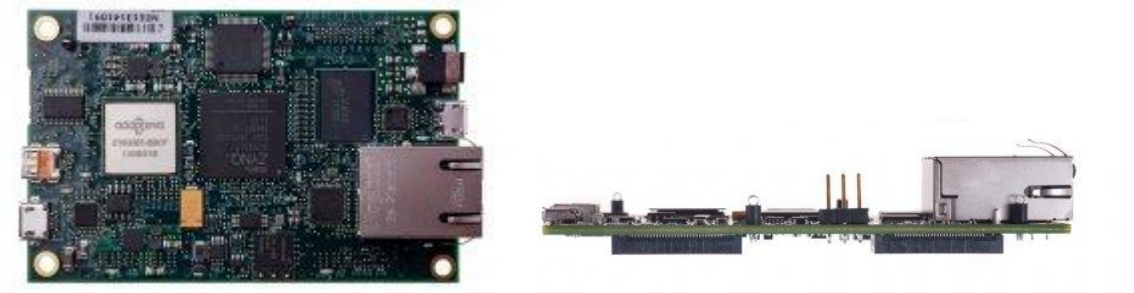
Belvire Antoine
Lavabre Mélanie
Recoules Frédéric
Salaün Kévin

Rapport de Projet Tutoré
Version 1.0

Carte Parallella



Tuteur M.Castan
Année 2014-2015

Sommaire

Introduction 1
I Prise en main de l'environnement Parallella 2
1) Prémices de notre réflexion 2
2) Programmes de tests 2
3) Identification des fonctions essentielles de la librairie 2
II Travail sur le projet java préexistant 3
1) Description du projet 3
2) Mise en place d'un JNI 4
3) Traduction en C du code de calcul 5
a - Architecture du programme 5
b - Description de la structure de calcul (calcul.h) 6
c - Synchronisation de l'affichage graphique 7
d - Portage minimale de la classe ListeCouleurs 7
e - Portage des méthodes de la classe CalculCascade2D 8
f - Échanges avec le code Java 8
Pistes d'amélioration 8
Conclusion 10
TABLE DES ANNEXES 11
Annexe 1 : Code du programme JNI qui change la valeur des variables depuis le C ou le Java. 11
Annexe 2 : Classes primaires du code JNI final 13
Annexe 3 : compilation de notre code 15
Étape 1 : installation de l'eSDK 15
Étape 2 : récupérer notre code sur GitHub 15
Étape 3 : compilation 15
Étape 4 : lancer le programme 15

Introduction

Notre mission initiale pour ce projet était de tirer partie des 16 cœurs de la carte en parallélisant un programme.

Après une phase d'appropriation des différents modules de la carte et des liens de communications qui les unissent, des tests de codes existants mis en ligne par la communauté Parallella, nous avons choisi d'adapter un programme écrit par M.Castan.

Ce programme, fait des calculs et affiche les points qui en résultent au fur et à mesure. Ce dernier a dû subir des adaptations pour pouvoir satisfaire nos contraintes à la fois de code (distinction de l'interface de lancement, de l'affichage et du calcul propre) et à la fois matérielles (mémoire disponible restreinte selon nos choix de stockage des données de calcul reçues et envoyées).

Dans un premier temps il sera question de nos premières réflexions, ensuite des programmes de test que nous avons réussi à lancer, des difficultés que nous avons connues avec la carte, des premiers appels à des fonctions propres à la carte Epiphany.

Par ailleurs, nous rentrerons dans les détails de nos choix de conception, du code écrit, des changements apportés au projet java qui nous a été donné, de la mise en place d'un JNI jusqu'à la transposition en C (avec une approche orientée objet) de certaines parties du programme.

I Prise en main de l'environnement Parallella

1) Prémices de notre réflexion

Nous avons parcouru les différents manuels de Parallella et Xilinx, et compris les liens entre eux. Nous nous sommes renseignés sur les différents types de mémoire existants (du plus local au plus global) ainsi que leurs capacités respectives et leurs modes et temps d'accès en vue de connaître les possibilités de calcul en termes de précision. Il nous a fallu du temps pour comprendre comment fonctionnaient les adresses des cœurs pour pouvoir les faire fonctionner simultanément en les ayant clairement identifiés.

2) Programmes de tests

Nous sommes parvenus à faire fonctionner des programmes de tests disponibles sur le dépôt github de la communauté Parallella comme le capteur de température, la multiplication de matrices ou encore le programme basé sur l'algorithme de Mandelbrot.

Cependant des modifications ont été nécessaires car toutes les fonctions n'étaient pas connues/maintenues dans les librairies utilisées lors de la compilation.

En outre, les difficultés liées à la carte ont été nombreuses. Il a fallu installer de nombreux kits de développement, les versions trouvées n'étaient pas toujours à jour et ne sont pas systématiquement accompagnées de documentation claire. En outre, la carte bien qu'enregistrée sur le réseau INSA avec une adresse unique n'est pas toujours joignable (à son adresse linaro@10.1.5.193) et l'on rencontre des complications à ce que la carte détecte en même temps des branchements de claviers, souris et écran.

3) Identification des fonctions essentielles de la librairie

Nous avons cherché à identifier la routine à mettre en place pour utiliser Epiphany, via la librairie e-hal.

La manipulation de la carte Epiphany suppose l'appel à la fonction e_init() car c'est elle qui permet l'initialisation des structures de données de HAL, ainsi que l'établissement d'une connexion à la plateforme Epiphany. NB : Les paramètres de la plateforme sont lus à partir d'un Hardware Descriptor File (chemin donné en argument de la fonction).

```c
#include <stdlib.h>
#include <stdio.h>
#include <e-hal.h>

// Pour les fonctions : e-hal-api.h
// Pour les types : e-hal-data.h

int main() {
	// e_platform_t : informations à propos de la plate-forme Epiphany (chip, external memory segments, géométrie)
	e_platform_t platform;
	// e_epiphany_t : structure représentant les données du groupe de coeurs
	e_epiphany_t dev;
	e_mem_t mbuf;

	unsigned row = 1;;
	unsigned col = 1;
	unsigned rows = 4;
	unsigned cols = 4;
	off_t base;
	size_t size;
	// e_init(char * hdf) : initialise la plate-forme, hdf est un fichier, si rien n'est spécifié (NULL) on cherche dans $EPIPHANY_HDF=/opt/adapteva/esdk/bsps/current/platform.hdf
	e_init(NULL);
	// e_get_platform_info(platform) : récupère des infos sur la plateforme
	// et les charge dans l'e_platform_t platform
	e_get_platform_info(&platform);
	// e_open(e_epiphany_t * dev, unsigned strtrow, unsigned strtcol, unsigned rows, unsigned cols) : ouvre le groupe de coeurs spécifié (stocké dans dev)
	e_open(&dev, 0, 0, platform.rows, platform.cols);

	e_alloc(&mbuf,0,0);
	e_free(&mbuf);
	// e_close(e_epiphany_t * dev) : relache le groupe de coeurs spécifié
	e_close(&dev); // contraire de e_open()

	// e_finalize() : termine la connexion avec le système Epiphany
	e_finalize(); // contraire de e_init()

	return 0;
}
```

Via e_open(), on définit un ensemble de travail constitué par un ou plusieurs cœurs identifiés par leurs coordonnées sur la carte de la plateforme Epiphany. Un groupe peut être constitué d'un seul cœur, comme de toute la carte. Les données de ce groupe de travail sont stockées dans une structure de type e_epiphany_t. C'est par la manipulation de ce même type que seront réalisés les accès en lecture/écriture.

La fonction qui fait écho à e_open() est e_close() et nous sert à fermer le groupe de cœurs préalablement ouvert. L'appel de cette fonction constitue évidemment un préalable à toute réallocation.

Par la suite e_alloc() nous sert à allouer un buffer dans la mémoire externe faisant suite à l'appel à e_init(), qui a permis de situer le début du segment mémoire alloué.

De la même façon, les accès au buffer en lecture/écriture se font par le biais de l'objet e_mem_buf. La désallocation est effectuée à l'aide de e_free().

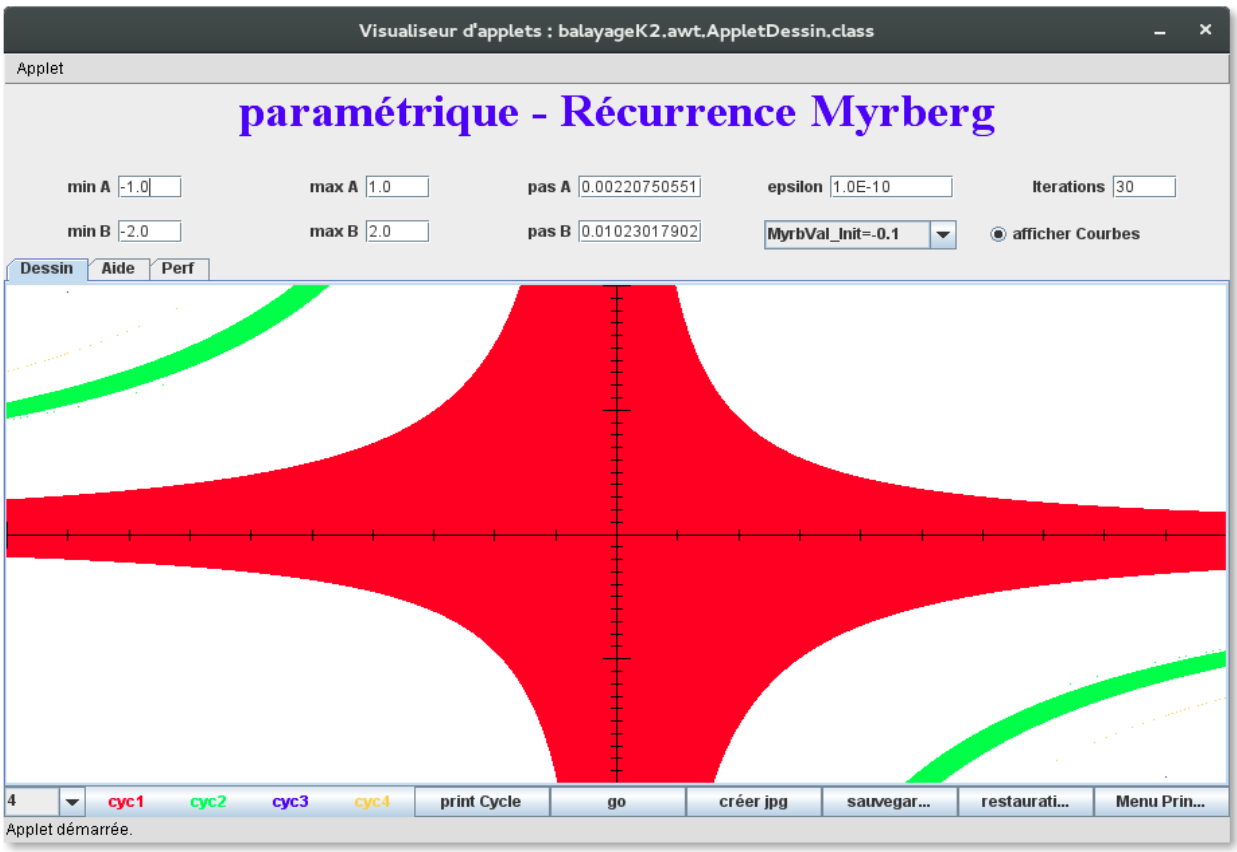
En fin d'utilisation, e_finalize() contribue à la libération des ressources allouées par e_init().

II Travail sur le projet java préexistant

1) Description du projet

Suite à l'apprentissage de l'api e-hal et des particularités techniques de l'architecture Epiphany, nous avons décidé d'appliquer ces nouvelles connaissances, en portant un programme conçu par notre tuteur afin qu'il tire profit des capacités de calcul parallèle de la carte Parallella. Dans l'idée c'est du C orienté objet qui a pour visée la facilitation des échanges et l'adaptation au plus près du code de base.

Ce programme s'appelle balayageK2 et est codé en Java. Il permet de visualiser, dans une interface graphique claire, différents types de balayage, exigeant des calculs conséquents en nombre.



Nous nous sommes focalisés sur un type de balayage (le travail étant à peu près le même pour chaque balayage) : le calcul “Plan de phase > CalculCascade2D > Cas quadratique”.

Grâce à notre tuteur, nous avons identifié la principale méthode de calcul qui devrait être portée de façon à être exécutée sur la carte Parallella et à mettre à l’épreuve ses performances supposées.

Dans un premier temps, nous avons identifié les principales difficultés que ce travail semblait poser.

Tout d’abord, balayageK2 étant écrit en Java et l’api e-hal ne fournissant que des fonctions C, nous devons trouver une solution pour appeler efficacement du code écrit en C depuis Java. Tout particulièrement, la communication des données semblait être un vrai défi pour nous.

La difficulté suivante est la réécriture du code Java de calcul en C, chose qui suppose la prise en compte des logiques de programmation des deux langages.

Enfin, la dernière difficulté identifiée est l’adaptation du code en C pour qu’il fonctionne sur l’architecture Epiphany. Comme montré dans la partie précédente, cela nécessite au moins deux au moins deux codes C distincts : un pour l’hôte, exécuté sur le processeur Zynq, et l’autre destiné à être exécuté par la carte Epiphany, qui réalise le calcul à proprement parler.

Après avoir réfléchi sur la meilleure façon d’aborder ce problème, nous avons décidé d’organiser le travail comme suit :

1. prise en main de la JNI, une bibliothèque faisant le pont entre Java et C
2. première réécriture du code de calcul de Java vers C
3. adaptation du code initial en Java afin d’intégrer notre nouveau code C grâce à JNI
4. test de l’ensemble sur une machine classique (nos ordinateurs personnels)
5. modification du code C afin de pouvoir être lancé sur la carte Epiphany
6. tests de l’ensemble sur la carte Parallella
7. création d’une librairie générique visant à faciliter le lancement de code C depuis une machine virtuelle Java
8. adaptation de notre code pour utiliser cette librairie
9. tests de l’ensemble sur la carte Parallella

Par manque de temps, les étapes 5, 6 et 7 n’ont pas été terminées au moment d’écrire ce rapport. Les étapes 8 et 9 n’ont pas été abordées.

2) Mise en place d’un JNI

Nous avons cherché à nous approprier la notion de Java Native Interface afin de pouvoir lancer, depuis un programme Java, des programmes C sur les différents coeurs de la carte pour tirer partie des 16 coeurs et paralléliser au maximum le calcul des points du programme de balayage Java.

Dans un premier temps nous avons suivi un tutoriel qui nous a permis d’afficher ‘HelloWorld’.

Ensuite nous avons rajouté des éléments à ce code, jusqu’à parvenir à changer les valeurs de deux variables et à les afficher, une première fois en faisant appeler des fonctions en C par le java et la deuxième fois dans le sens inverse.

Fichiers de code : Voir annexe 1

```
febroshka@febroshka-N55SF:~/Workspacetut/Part2/src$ gcc -I /home/febroshka/Téléc
hargements/jdk1.7.0_75/include/ -I /home/febroshka/Téléchargements/jdk1.7.0_75/i
nclude/linux/ -Wall -fPIC -c part2.c -o part2.o
febroshka@febroshka-N55SF:~/Workspacetut/Part2/src$ gcc -shared part2.o -o libpa
rt2.so
febroshka@febroshka-N55SF:~/Workspacetut/Part2/src$ java -Djava.library.path=. p
art2
La valeur de j :40
la valeur de i :33
febroshka@febroshka-N55SF:~/Workspacetut/Part2/src$ javac part2.java
febroshka@febroshka-N55SF:~/Workspacetut/Part2/src$ java -Djava.library.path=. p
art2
La valeur de j :2
la valeur de i :3
La valeur de j :40
la valeur de i :33
febroshka@febroshka-N55SF:~/Workspacetut/Part2/src$
```

Dans la version finale de notre JNI, nous avons créé deux classes java : la classe Ejava et la classe EChannel. La première contient des méhodes natives d'initialisation (init), de fermeture (close), de lancement (launch), d'enregistrement (register). La deuxième classe est dédiée à la connaissance de la signature des classes utilisées. Chose qui permet par la suite de transmettre automatiquement les arguments entre Java et Epiphany.

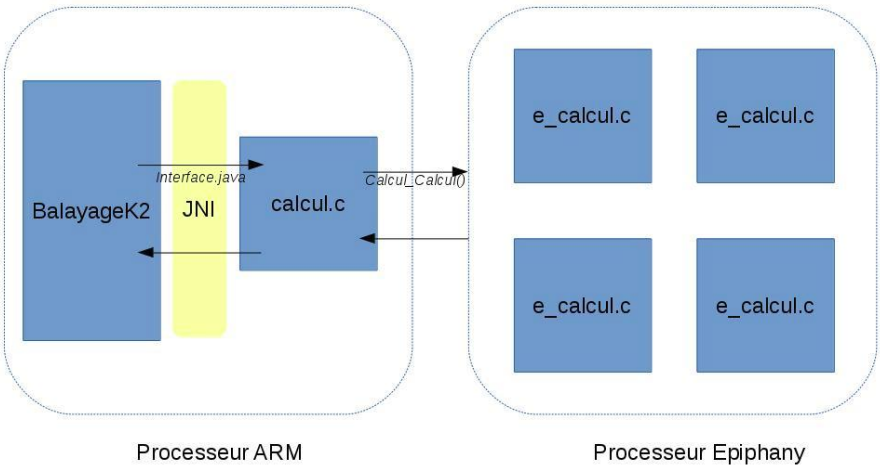
Fichiers de code : voir annexe 2

Toute la difficulté de l'implémentation est finalement d'être capables d'identifier le segment de mémoire via lequel transitent les données, pour pouvoir récupérer le résultat du traitement qui aura été fait sur ces calculs. On sait que le Zynq dispose d'un 1Go de mémoire embarquée. Notre programme principal aura la responsabilité de libérer les deux emplacements mémoire utilisés.

3) Traduction en C du code de calcul

a - Architecture du programme

L'architecture simplifiée de notre programme complet est peut être décrite par le schéma suivant.



Dans le le code Java, l'appel à balayageK2.calculCascade2D.calcul() est remplacé par un appel à une méthode d'une nouvelle classe Interface.java faisant appel à la JNI. Ainsi, on peut appeler depuis Java des fonctions natives dans calcul.c.

calcul.c sert à récupérer les données Java, initialise les coeurs Epiphany puis distribue le travail.

Trois points importants à noter :

- dans un premier temps, nous mettons tout l'algorithme de calcul dans calcul.c pour pouvoir tester si la transposition en C et dialogue avec le programme principal en Java fonctionnent, sur nos machines personnelles.
- dans un second temps, nous divisons la partie purement calculatoire de calcul.c dans e_calcul.c. C'est ce dernier code qui sera exécuté sur chaque coeur Epiphany. Ce travail n'est pas encore finalisé.
- enfin, dans un troisième temps, l'idée est de créer une librairie Java permettant d'interagir directement avec le processeur Epiphany et ainsi "nous passer" de calcul.c, moyennant le passage de paramètres adéquats et un e_calcul.c adapté. Ce travail n'est pas encore finalisé.

b - Description de la structure de calcul (calcul.h)

Nous avons décidé d'utiliser une approche orientée objet malgré le langage utilisé, ceci dans l'idée de faciliter les échanges avec le code en Java mais également afin ne pas trop transposer le code initial.

Ainsi, nous avons créé une structure reprenant les principaux attributs de la classe CalculCascade2D utiles pour notre calcul. Extrait de calcul.h :

```
typedef struct Calcul {

    // CalculBase
    double xPrec;
    double yPrec;
    ListeCouleurs * lcPrec;

    // Des vectors normalement
    double lstPtsX[80000];
    double lstPtsY[80000];
    double lstPtsC[80000];

    // Attributs
    signed char ordreCycle; // byte java
    int mMax; // 30 ?
    int nMax; // 30
    int m;
    double a;
    double b;
    double epsilonVal;
    int nombreLignes;
    int masqueIndiceLigne;// = nombreLignes-1

    int lstChoixPlanSelectedIndex;
    double ** lgN;//[nombreLignes][mMax];
    double * valInit;//[nMax];

    int indiceIterationCourante;
    int indiceIterationSuivante;
    int indiceIterationPrecedente;
    int noIterationCourante;

    int arretRunner;//booleen

    long long ctrCalculs;

    // Attributs "privés"
    int ctrV;
    int ctrH;
    int ctrD;
    int ctrG;

    // Méthodes
    int (*differentEpsilonPres)(struct Calcul * This, double x, double y);
    int (*egalEpsilonPres)(struct Calcul * This, double x, double y);
    void (*differerPoint2D)(struct Calcul * This, double x, double y, ListeCouleurs *lc);
    void (*calculM)(struct Calcul * This);
    void (*calcul)(struct Calcul * This);

} Calcul;
```

Pour lancer le calcul, les deux fonctions à utiliser sont :

- void Calcul_Creer (<arguments récupérés avec JNI>): en dehors de la structure, permettant justement de l'initialiser avec les données Java ;
- void (*calcul)(struct Calcul * This) qui permet de lancer le calcul à proprement parler

Le code initial du calcul en Java est relativement découplé du reste de balayageK2, ce qui est positif dans notre cas car cela limite le nombre de fonctions à porter en C.

Cependant, plusieurs difficultés majeures sont apparues.

c - Synchronisation de l'affichage graphique

Au moment de l'écriture de ce rapport, l'affichage graphique ne fonctionne pas.

Dans le code initial, le résultat des calculs sont pour la plupart directement envoyés au code gérant l'affichage. Il y a donc quelques appels gênants à des méthodes de la classe PanelDessin directement dans le code du calcul. Par exemple dans CalculCascade2D :

```
if (panelDessinDistant==null) {
    panelDessin.ajouterPoint(a, lgN[indiceItérationCourante][j], lc);
}
else {
    différerPoint2D(a, lgN[indiceItérationCourante][j], lc);//Color.red);
}
```

De même, dans CalculBase.différerPoint2D :

```
protected void différerPoint2D(double x, double y, Color c) {
    if (panelDessinBase.dansZoneAffichage(x, y)) {
        lstPtsX.add(x);
        lstPtsY.add(y);
        lstPtsC.add(c.getRGB());
    }
}
```

Après réflexion, nous avons décidé de modifier la logique du code afin de différer l'affichage de tous les points (pour un appel à Calcul()) à la fin de la boucle principale. En effet, on ne peut pas imaginer interrompre l'exécution du calcul à chaque nouveau résultat pour établir une communication lente avec le code Java (destiné à être sur un processeur différent !). Comme CalculBase.différerPoint2D a aussi un appel à panelDessin, nous le supprimons, quitte à calculer un peu plus de points que nécessaire.

Ainsi, nous éliminons les appels à PanelDessin. Mais cela crée une autre difficulté : la gestion en mémoire d'un grand nombre d'éléments. Or la mémoire interne d'un cœur du processeur Epiphany n'est que de 32 k. L'utilisation de la mémoire partagée pour chaque calcul semble nécessaire.

d - Portage minimale de la classe ListeCouleurs

Un autre problème a été l'utilisation dans le code du calcul à la classe ListeCouleurs.

En fait, le calcul sur chaque cœur doit renvoyer 3 listes d'éléments : lstPtsX, lstPtsY, lstPtsC. Les deux premières sont des listes de coordonnées (type double classique) mais la

dernière représente une liste de couleurs, qui est défini dans le code Java par une classe à part, ListeCouleurs.

Comme ces objets sont créés dans le cœur du calcul, nous avons décidé de porter une partie du code la classe ListeCouleurs en C. Ainsi, nous avons défini un type ListeCouleurs en C. Extrait de listeCouleurs.h :

```
typedef struct ListeCouleurs {

    // Attributs
    int nbrCouleurs;
    int valeur;
    int masqueCouleur;
    int nbrBitsParCouleur;

    // Méthodes (pas la peine de les mettre toutes) !
    void (*ajouterCouleur)(struct ListeCouleurs * This, int nbCouleurs);
    void (*Free)(struct ListeCouleurs * This);
    int (*equals)(struct ListeCouleurs * This, struct ListeCouleurs * lc);

} ListeCouleurs;
```

e - Portage des méthodes de la classe CalculCascade2D

Afin de limiter les appels à Java depuis le C à la simple communication des données initiales et à celle des résultats, nous avons porté quelques méthodes de la classe CalculCascade2D dans notre code en C sans trop grande difficulté.

f - Échanges avec le code Java

Cette partie a beaucoup changé au fil des versions. La version décrite dans ce rapport est susceptible de changer.

Le fait est que le code initial calcule énormément de points (plus de 80 000 par instance de calcul, donc pour une valeur de x !), pour ensuite réduire le nombre de points pour l'affichage.

Pour un affichage différé, les listes de coordonnées et de couleur sont stockées dans trois vecteurs côté C. Afin de réduire le nombre de calculs et d'échange, nous avons choisi de réduire le calcul au nombre de points de la fenêtre, et de renvoyer juste un tableau d'ordonnées et un tableau de couleurs correspondant.

Pistes d'amélioration

Sachant que javah génère le code JNI, une idée (qui sera peut être implémentée d'ici l'oral)de créer un javae qui analyse un fichier XML et qui crée automatiquement une classe pour chaque objet que l'on souhaite passer à la carte Epiphany ainsi que le '.h' à utiliser dans chacun des sous programmes Epiphany.

Dans le fichier XML on écrirait dans une partie déclarative les objets passés (objet et ses champs), on décrirait le nombre d'instances réellement transmises avec leur nom d'instanciation ainsi que les objets attendus en retour ce qui permet de générer un header avec le placement en mémoire des arguments prêts à être utilisés par le code C. Dans le header, on aurait la situation des variables et par passage du header on n'aurait pas à choisir les endroits de stockage mémoire. On pourrait y voir une analogie de cross compilateur.

Pour ce qui est des limitations de la carte, nous n'avons pas de réticences à utiliser les 1Go communs de la carte mais il semble paradoxal de devoir attendre après une même

ressource, la mémoire globale, alors qu'on pourrait directement les stocker sur les 32Ko. Or le problème s'il est trop gros, ne peut pas être stocké individuellement, pourquoi dès lors chercher à paralléliser le problème. En outre, un dilemme apparaît relativement au gain de temps dans le souci d'optimiser l'utilisation des coeurs et à la nécessité de rajouter du code pour le passage des arguments par la méthode décrite par la classe EChannel.

Par ailleurs, un souci annexe se pose, il n'y a pas de flags pour signaler les problèmes.

Conclusion

Nous sommes parvenus à établir une communication entre le programme de M.Castan jusqu'à la carte Epiphany via du code C. D'ici vendredi, le programme est susceptible d'évoluer grandement puisque nos principaux problèmes n'ont été résolus que récemment, ce mercredi 27 mai pour être exacts. Ce qui a été fait nous semble un début et reste limité à l'heure actuelle mais qui pourra être approfondi avec du temps supplémentaire.

A notre échelle, parmi les principales améliorations à apporter, on pourrait songer à un vraie bibliothèque permettant d'accéder de manière transparente à la puissance du processeur Epiphany depuis un code Java quelconque pour peu qu'une parallélisation soit profitable à ce code.

Nous avons apprécié de travailler sur un projet participatif, nous avons été séduits par le côté novateur et inédit du projet mais nous avons de fait pris conscience du travail titanesque que suppose de se renseigner et de se former en autonomie sur des notions nouvelles comme le JNI et les spécificités architecturales d'une carte comme Parallella.

TABLE DES ANNEXES

Annexe 1 : Code du programme JNI qui change la valeur des variables depuis le C ou le Java.

********** Fichier C **********

```c
#include "part2.h"


JNIEXPORT void JNICALL Java_part2_init
  (JNIEnv * env , jobject thisObj){

jclass thisClass = (*env)->GetObjectClass(env, thisObj);

jfieldID fidJ = (*env)->GetFieldID(env, thisClass, "j","I");  //I signature d'int

if (NULL == fidJ) return;

jfieldID fidI = (*env)->GetFieldID(env, thisClass, "i","I");

if (NULL == fidI) return;

(*env)->SetIntField(env, thisObj, fidJ, 40);
(*env)->SetIntField(env, thisObj, fidI, 33);

jmethodID fidF = (*env)->GetMethodID(env, thisClass, "printvar", "()V");

if (NULL == fidF) return;

(*env)->CallVoidMethod(env, thisObj, fidF);
```

********** Fichier de la librairie **********

```c
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

// Implementation of native method sayHello() of HelloJNI class
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
   printf("Hello World!\n");
   return;
}
```

********** Fichier Java **********

```
public class part2 {
    static {
        System.loadLibrary("part2");
    }
    int j;
    int i;
        public part2() {
        i = 3;
        j = 2;
    }
    public native void init();
        public void printvar(){
        System.out.println("La valeur de j :"+j);
       System.out.println("la valeur de i :"+i);
    }
        public static void main(String args[]){
        part2 O = new part2();
        O.printvar();
        O.init();
    }}
```

Annexe 2 : Classes primaires du code JNI final

Classe EJava.java

```
import java.lang.reflect.Field;


public class EChanel {

    private HashMap<Class<?>,String[]> fields;
    private HashMap<Class<?>,String[]> signatures;

    public EChanel() {
        fields = new HashMap<Class<?>, String[]>();
        signatures = new HashMap<Class<?>, String[]>();
    }

    public void register(Class<?> type) {
        if (this.fields.get(type) != null)
            return;
        Field[] F = type.getDeclaredFields();
        String[] fields = new String[F.length];
        String[] signatures = new String[F.length];
        for (int i = 0; i < F.length; i ++) {
            fields[i] = F[i].getName();
            if (this.fields.get(F[i].getType()) == null) {
                this.register(F[i].getType());
            }
            signatures[i] = F[i].getType().toString();
        }
        this.fields.put(type, fields);
        this.signatures.put(type, signatures);
    }

    public String[] getFields(Class<?> type) {
        return this.fields.get(type);
    }

    public String[] getSignatures(Class<?> type) {
        return this.signatures.get(type);
    }

}
```

Classe EChanel.java

```
import java.lang.reflect.Field;


public class EChanel {

    private HashMap<Class<?>,String[]> fields;
    private HashMap<Class<?>,String[]> signatures;

    public EChanel() {
        fields = new HashMap<Class<?>, String[]>();
        signatures = new HashMap<Class<?>, String[]>();
    }

    public void register(Class<?> type) {
        if (this.fields.get(type) != null)
            return;
        Field[] F = type.getDeclaredFields();
        String[] fields = new String[F.length];
        String[] signatures = new String[F.length];
        for (int i = 0; i < F.length; i ++) {
            fields[i] = F[i].getName();
            if (this.fields.get(F[i].getType()) == null) {
                this.register(F[i].getType());
            }
            signatures[i] = F[i].getType().toString();
        }
        this.fields.put(type, fields);
        this.signatures.put(type, signatures);
    }

    public String[] getFields(Class<?> type) {
        return this.fields.get(type);
    }

    public String[] getSignatures(Class<?> type) {
        return this.signatures.get(type);
    }

}
```

Annexe 3 : compilation de notre code

Le code n'est compilable complètement que sur la carte, à moins d'installer un cross-compilateur ARM et un cross-compilateur Epiphany (procédure non décrite ici).

Étape 1 : installation de l'eSDK

Récupérer le dernier kit de développement pour l'architecture Epiphany sur ftp://ftp.parallella.org/esdk/. À l'heure actuelle, il s'agit de la version 2015-1.

La procédure d'installation est détaillée dans un fichier INSTALL.

Étape 2 : récupérer notre code sur GitHub

Nous avons placé notre code sur GitHub :
https://github.com/Joe-Chip/Projet_tut_parallella.

Étape 3 : compilation

Il est nécessaire de la configurer afin d'utiliser le fichier ant build.xml. Celui-ci va appeler un autre fichier ant, buildJNI.xml qui construit la librairie partagée libcalcul.so (Java ⇔ C) ainsi que e_calcul.elf et e_calcul.srec (ARM ⇔ Epiphany) :

```
ant build.xml
```

Étape 4 : lancer le programme

Le chemin vers les bibliothèques C et Epiphany doivent être spécifié pour lancer correctement le main. Se placer dans le dossier racine du projet puis lancer avec la commande :

```
sudo LD_LIBRARY_PATH=/opt/adapteva/esdk/tools/host/lib  \
EPIPHANY_HDF=/opt/adapteva/esdk/bsps/current/platform.hdf \
java  -Djava.library.path=C balayageK2/Main
```

(Le sudo est nécessaire pour les appels à la carte Epiphany).

Note : à l'heure où cette annexe est écrite, l'affichage graphique ne fonctionne pas et seul données textuelles sont disponibles dans la console.