Renderscript

Boris Mansencal

23/03/2018

Renderscript

Boris Mansencal

23/03/2018

Renderscript

Renderscript est un système qui permet d'executer des **calculs** de façon **performante** (parallélisée) assez **simplement**.
Il est particulièrement adapté aux CPUs et GPUs multicoeurs, et notamment pour le traitement d'images.

https://developer.android.com/guide/topics/renderscript/compute.html

Renderscript

Renderscript est un système qui permet d'executer des **calculs** de façon **performante** (parallélisée) assez **simplement**.
Il est particulièrement adapté aux CPUs et GPUs multicoeurs, et notamment pour le traitement d'images.

https://developer.android.com/guide/topics/renderscript/compute.html

On peut se passer de RenderScript pour écrire du code très performant. On peut utiliser le NDK (Native Development Kit) qui permet d'utiliser du code C/C++. Mais c'est beaucoup plus compliqué : il faut gérer soit même le fait qu'on peut avoir différentes configurations hardware (CPU, GPU, nb coeurs, ARM v5/v7, x86, ...) ; gérer aussi les synchronisations, ...
RenderScript a été développé pour simplifier tout cela.
Les différences matérielles sont complètement cachées. Le développeur n'écrit qu'une version du code.
Le code RenderScript sera en fait compilé, à la volée, à l'exécution de l'application, sur le smartphone.

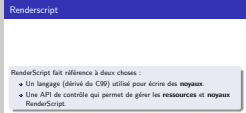
Dans les premières versions d'Android ($<$ 4.1 / API level 16), il y avait une partie Graphique (en plus de cette partie Calcul). Mais elle a été abandonnée.

Nous vous encourageons fortement à lire la documentation Android sur Renderscript. Cette présentation n'en est qu'une version édulcorée.

Renderscript

RenderScript fait référence à deux choses :

- Un langage (dérivé du C99) utilisé pour écrire des **noyaux**.
- Une API de contrôle qui permet de gérer les **ressources** et **noyaux** RenderScript.



Pour un code utilisant du Renderscript, il va donc y avoir deux parties : une partie en langage proche proche du C99, une partie en langage haut niveau (Java pour nous).

Cette API de contrôle est disponible en Java, C++ et dans le script lui-même

Renderscript : configuration

Pour utiliser RenderScript, il faut le préciser dans le *build.gradle* du répertoire *app*.
Dans la section *defaultConfig*, rajouter les lignes :

```
renderscriptTargetApi 18
renderscriptSupportModeEnabled true
```

Renderscript nécessite Android 2.3 (API level 9).

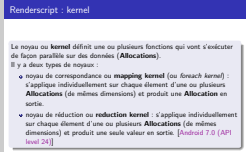
Renderscript : configuration
Pour utiliser RenderScript, il faut le préciser dans le *build.gradle* du répertoire *app*.
Dans la section *defaultConfig*, rajouter les lignes :
renderscriptTargetApi 18
renderscriptSupportModeEnabled true
Renderscript nécessite Android 2.3 (API level 9).

renderscriptTargetApi : la valeur est entre 11 et le dernier API level disponible (27 pour Oreo/8.x.0, 28 pour Android P/9.0, 29 pour Android10/10). Ici, on dit qu'il faut au minimum Android 2.3 (API level 9), mais cela dépend en fait de ce qui est importé. On en parle plus loin.

Renderscript : kernel

Le noyau ou **kernel** définit une ou plusieurs fonctions qui vont s'exécuter de façon parallèle sur des données (**Allocations**).
Il y a deux types de noyaux :

- noyau de correspondance ou **mapping kernel** (ou *foreach kernel*) : s'applique individuellement sur chaque élement d'une ou plusieurs **Allocations** (de mêmes dimensions) et produit une **Allocation** en sortie.
- noyau de réduction ou **reduction kernel** : s'applique individuellement sur chaque élement d'une ou plusieurs **Allocations** (de mêmes dimensions) et produit une seule valeur en sortie. [Android 7.0 (API level 24)]



- **Allocation** : c'est en gros un buffer qui stocke un type donné.
- **Mapping kernel** : c'est typiquement ce qu'on va utiliser en traitement d'image. Les données d'entrée sont nos pixels, et on va appliquer un noyau sur chaque pixel, individuellement et indépendamment des autres pixels, pour produire de nouveaux pixels.
- **Reduction kernel** : on peut par exemple s'en servir pour calculer la somme des éléments de l'Allocation d'entrée.

Pour ces TDs, nous allons nous intéresser aux **mapping kernels**.

Renderscript : kernel

Concrètement, un noyau est écrit dans un fichier *.rs*.
Ces fichiers sont généralement dans un répertoire *rs*.

- Sous Android studio, click droit sur *app* puis *New* → *Folder* → *RenderScript Folder*

Renderscript : kernel
Concrètement, un noyau est écrit dans un fichier .rs.
Ces fichiers sont généralement dans un répertoire rs.
Sous Android studio, click droit sur \app puis New → Folder → RenderScript Folder

Le répertoire apparait comme *renderscript* sous Android Studio mais s'appelle bien *rs* sur le disque. Il va être ici : app/src/main/rs

Renderscript : kernel

Un exemple de script : **invert.rs**

```
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

uchar4 RS_KERNEL invert(uchar4 in, uint32_t x, uint32_t y) {

  uchar4 out = in;
  out.r = 255 - in.r;
  out.g = 255 - in.g;
  out.b = 255 - in.b;

  return out;
}
```


Renderscript : kernel
Un exemple de script : invert.rs
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)
uchar4 RS_KERNEL invert(uchar4 in, uint32_t x, uint32_t y) {
uchar4 out = in;
out.r = 255 - in.r;
out.g = 255 - in.g;
out.b = 255 - in.b;
return out;
}

Langage proche du C99. Ce noyau va être compilé par le compilateur RenderScript, sur le smartphone cible. [C'est aussi ce qui se passe pour les *shaders* graphiques (en OpenGL par exemple)].

- version(1) : c'est actuellement la seule version valable.
- com.android.rssample doit être remplacé par le package name de votre projet.
- La fonction *kernel* appelable depuis Java est décorée par RS_KERNEL
- Cette fonction ne concerne qu'un seul élément de la ou les Allocations d'entrée et va s'appliquer en parallèle pour chaque élément pour remplir l'Allocation de sortie. Dans notre cas, si notre Allocation contient les pixels d'une image, la fonction va s'appliquer en parallèle sur cheque pixel. Elle prend les valeurs pour un pixel en entrée, et produit les valeurs pour un pixel en sortie. On ne se préoccupe pas du parcours des buffers/Allocations, ni de la parallélisation. C'est fait pour nous.
- on voit apparaitre le type *uchar4* : struct en C99, qui contient 4 uchar, nommés r, g, b, et a. Jusqu'à maintenant, en Java, nos couleurs était stockées comme des ints et on accédait aux uchars.
- on a accès à la position du pixel (x, y) mais on ne s'en sert pas ici. On aurait pu l'omettre.
- Ce script est très simple. On peut aussi avoir des variables globales ou des fonctions auxiliaires (appelées *invokable functions*, et non décorées par RS_KERNEL).

Renderscript : kernel

Un exemple de script : **invert.rs**

```
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

uchar4 RS_KERNEL invert(uchar4 in, uint32_t x, uint32_t y) {

  uchar4 out = in;
  out.r = 255 - in.r;
  out.g = 255 - in.g;
  out.b = 255 - in.b;

  return out;
}
```

Est-ce que vous avez une idée de comment faire *toGray()* ?

Renderscript : kernel
Un exemple de script : invert.rs
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)
uchar4 RS_KERNEL invert(uchar4 in, uint32_t x, uint32_t y) {
uchar4 out = in;
out.r = 255 - in.r;
out.g = 255 - in.g;
out.b = 255 - in.b;
return out;
}
Est-ce que vous avez une idée de comment faire toGray()?

- Il faut passer par des “int” ou des “floats”.

Renderscript : kernel

Un exemple de script : **gray.rs**

```
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

uchar4 RS_KERNEL toGray(uchar4 in) {

  float4 pixelf = rsUnpackColor8888(in);

  float gray = (0.30*pixelf.r
                + 0.59*pixelf.g
                + 0.11*pixelf.b);

  return rsPackColorTo8888(gray, gray, gray, pixelf.a);
}
```


Renderscript : kernel
Un exemple de script : gray.rs
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)
uchar4 RS_KERNEL toGray(uchar4 in) {
float4 pixelf = rsUnpackColor8888(in);
float gray = (0.30*pixelf.r
+ 0.59*pixelf.g
+ 0.11*pixelf.b);
return rsPackColorTo8888(gray, gray, gray, pixelf.a);
}

- On voit apparaitre le type *float4* : struct en C99, qui contient 4 floats, nommés r, g, b, et a.
- *rsUnpackColor8888* transforme les canaux uchar (dans [0 ; 255]) en floats dans [0 ;1]
- *rsPackColorTo8888* transforme les canaux floats dans [0 ; 1] en uchar (dans [0 ; 255]).
- On aurait aussi pu utiliser *float4 convert_float4(uchar4)* et *uchar4 convert_uchar4(float4)*.

Renderscript : kernel

Un exemple de script (amélioré) : **gray.rs**

```
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

static const float4 weight = {0.299f, 0.587f, 0.114f, 0.0f};

uchar4 RS_KERNEL toGray(uchar4 in) {

  const float4 pixelf = rsUnpackColor8888(in);

  const float gray = dot(pixelf, weight);

  return rsPackColorTo8888(gray, gray, gray, pixelf.a);
}
```


Renderscript : kernel
Un exemple de script (amélioré) : gray.rs
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)
static const float4 weight = {0.299f, 0.587f, 0.114f, 0.0f};
uchar4 RS_KERNEL toGray(uchar4 in) {
const float4 pixelf = rsUnpackColor8888(in);
const float gray = dot(pixelf, weight);
return rsPackColorTo8888(gray, gray, gray, pixelf.a);
}

- Ce code est potentiellement plus rapide. Il utilise une fonction *dot()*, qui est vectorisée : les 4 multiplications peuvent se faire en parallèle. (Dans les faits, il ne semble pas y avoir de différence sur une fonction aussi simple)
- On peut voir la déclaration d'une variable globale : *weight*. Elle est "static const" : *static* indique qu'elle est propre au script et ne sera pas visible depuis le Java, *const* qu'elle ne sera pas modifiée. On voit au passage qu'on peut déclarer des variable "const", comme en C/C++.

Renderscript : kernel

Un exemple de script : **gray.rs**

```
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

uchar4 RS_KERNEL toGray(uchar4 in) {

  const uchar gray = (30*in.r
                   + 59*in.g
                   + 11*in.b)/100;

  return (uchar4){gray, gray, gray, in.a};
}
```


Renderscript : kernel
Un exemple de script : gray.rs
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)
uchar4 RS_KERNEL toGray(uchar4 in) {
const uchar gray = (30*in.r
+ 59*in.g
+ 11*in.b)/100;
return (uchar4){gray, gray, gray, in.a};
}

Encore une autre version : ici, on fait tous les calculs avec des entiers.

Renderscript : kernel

Est-ce qu'on peut **calculer un histogramme** en RenderScript ?

Renderscript : kernel
Est-ce qu'on peut **calculer un histogramme** en RenderScript ?

Qu'est-ce qui pose problème ? → le calcul est *parallélisé* et donc (potentiellement) des "threads" différents essayent de modifier la même case de l'histogramme en même temps !

Quelle est la solution ? → mettre une section critique ! Solution plus efficace : faire des opérations atomiques : incrément atomique de la case de l'histogramme.

Renderscript : kernel

Est-ce qu'on peut **calculer un minimum ou maximum** en RenderScript ?

Renderscript : kernel
Est-ce qu'on peut **calculer un minimum ou maximum** en RenderScript ?

Pas avec un **mapping kernel** (ou *foreach kernel*)) : car celui-ci produit une valeur de sortie par élément en entrée et nous on veut une seule sortie.
Mais ça serait possible avec un **reduction kernel** [si on est en Android 7.0 (API level 24)].

Renderscript : côté Java

Accessible de deux façons :

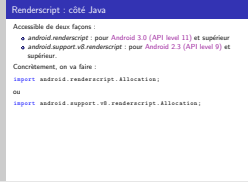
- *android.renderscript* : pour Android 3.0 (API level 11) et supérieur
- *android.support.v8.renderscript* : pour Android 2.3 (API level 9) et supérieur.

Concrètement, on va faire :

```
import android.renderscript.Allocation;
```

ou

```
import android.support.v8.renderscript.Allocation;
```

Si on utilise la “Support Library API” (c-a-d qu’on a “renderscriptSupport-ModeEnabled true” dans le build.gradle de l’app) alors il faut avoir :
import android.support.v8.renderscript.Allocation ;
sinon (avec “renderscriptSupportModeEnabled false” dans le build.gradle de l’app) il faut avoir :
import android.renderscript.Allocation ;

Renderscript : côté Java

Différentes étapes à suivre

1. **Créer un contexte RenderScript**
2. **Créer des Allocations** pour passer les données vers/depuis le script
3. **Créer le script**
4. Copier les données dans les *Allocations* si nécessaire
5. Initialiser les variables globales potentielles des scripts si nécessaire
6. **Lancer le noyau**
7. **Récupérer les données des Allocations**
8. Détruire le context, les *Allocation(s)* et le script [optionnel]

Renderscript : côté Java
Différentes étapes à suivre
Créer un contexte RenderScript
Créer des Allocations pour passer les données vers/depuis le script
Créer le script
Copier les données dans les Allocations si nécessaire
Initialiser les variables globales potentielles des scripts si nécessaire
Lancer le noyau
Récupérer les données des Allocations
Détruire le context, les Allocation(s) et le script [optionnel]

Pourquoi passe-t-on les données par ces Allocations ?
En fait, les deux codes ne s'adressent pas au même niveau : Java est au niveau "Android VM" et Renderscript au niveau "natif".
Il faut penser ces opérations comme des opérations pouvant s'effectuer sur un périphérique différent : la carte graphique/GPU par exemple, qui ne partage pas (forcément) la même mémoire que le CPU. On envoie donc les données sur la carte, on lance le calcul sur la carte, puis on rapatrie le résultat côté CPU. C'est exactement le comportement des langages de calcul sur GPU comme CUDA et OpenCL, ou des biblothèques 3D (OpenGL, Vulkan, DirectX).
Donc ces fonctions de création de contexte et d'allocations peuvent être coûteuses.

Renderscript : côté Java

```
private void toGrayRS(Bitmap bmp) {
  //1) Creer un contexte RenderScript
  RenderScript rs = RenderScript.create(this);

  //2) Creer des Allocations pour passer les donnees
  Allocation input = Allocation.createFromBitmap(rs, bmp);
  Allocation output= Allocation.createTyped(rs, input.
      ↪ getType());

  //3) Creer le script
  ScriptC_gray grayScript = new ScriptC_gray(rs);

  //4) Copier les donnees dans les Allocations
  // ...
  //5) Initialiser les variables globales potentielles
  // ...

  //6) Lancer le noyau
  grayScript.forEach_toGray(input, output);

  //7) Recuperer les donnees des Allocation(s)
  output.copyTo(bmp);

  //8) Detruire le context, les Allocation(s) et le script
  input.destroy(); output.destroy();
  grayScript.destroy(); rs.destroy();
}
```


1. Le *this* est un Context (android.content.Context). On considère ici qu'on est dans une Activity, qui hérite de Context.
2. On crée l'Allocation d'entrée en copiant les données d'une bitmap. On crée l'Allocation de sortie à partir d'un type donnée, ici le même que l'Allocation d'entrée, donc que de la bitmap d'entrée.
3. Une classe ScriptC_<filename> (côté Java) est automatiquement créée pour le script <filename>.rs. Donc ici une classe ScriptC_gray pour le script gray.rs.
4. Ici, on a copié les données dans *input* à la création, donc on n'a rien à faire.
5. Ici, on n'a pas de variable globale (non static const) dans le script précédent donc rien à initialiser.
6. Notre script (côté Java) a une méthode forEach_<mappingKernelName> créée automatiquement (où *mappingKernelName* est le nom de la méthode décorée dans notre script). Pour un noyau de réduction, la méthode s'appelerait reduction_<reductionKernelName>. Si on avait des fonctions auxiliaires (*invokable unctions*), on pourrait les appeler avec invoke_<invokableFunctionName>.
7. On recopie les données une fois calculées.
8. Si on ne le fait pas, ça devrait être fait par le Garbage Collector.

En réalité certaines de ces opérations sont asynchrones (lancement des noyaux notamment). Mais on ne le détaillera pas ici (voir la doc).

REMARQUE : Ce n'est sans doute pas le plus efficace de créer le contexte RenderScript et les Allocations à chaque appel de la méthode...

Renderscript : variables globales *non-static*

Si le script *filename.rs* a une variable globale *non-static* nommée <**var**>, la classe ScriptC_<filename> aura une méthode **get**_<**var**>.
Si cette variable n'est pas *const*, la classe aura aussi une méthode **set**_<**var**>.

Renderscript : variables globales *non-static*
Si le script *filename.rs* a une variable globale *non-static* nommée <**var**>, la classe ScriptC_<filename> aura une méthode **get**_<**var**>.
Si cette variable n'est pas *const*, la classe aura aussi une méthode **set**_<**var**>.

Une variable *static* dans le script ne sera pas visible depuis le Java.
Une variable *non-static* mais *const* sera visible et accessible en lecture, via **get**_<**var**>.
Une variable *non-static* et *non const* sera visible et accessible en lecture et écriture, via **get**_<**var**> et **set**_<**var**>.
Attention : seule la valeur d'initialisation de la variable globale (à sa création), est visible côté Java. Toute modification de la variable globale dans le script ne sera pas visible côté Java. On ne peut pas se servir de cette variable pour passer des résultats du renderscript vers le Java.

Renderscript : variables globales *non-static*

Variable globale côté Renderscript : **brightness.rs**

```
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)

float brightnessScale = 0.5;

uchar4 RS_KERNEL changeBrightness(uchar4 in) {

  //...
}
```


Renderscript : variables globales *non-static*
Variable globale côté Renderscript : **brightness.rs**
#pragma version(1)
#pragma rs java_package_name(com.android.rssample)
float brightnessScale = 0.5;
uchar4 RS_KERNEL changeBrightness(uchar4 in) {
//...
}

Ici, on fait un script pour changer la luminosité.
On a une variable global non-static et non-const : *brightnessScale*, initialisée à 0.5.
Côté Java, on va donc avoir accès aux méthodes get_<brightnessScale> et set_<brightnessScale>().
Si on fait un get_<brightnessScale>(), on obtiendra la valeur initiale, 0.5.

Renderscript : variables globales *non-static*

Variable globale côté Java :

```
private void changeBrightnessRS(Bitmap bmp) {
  //1) Creer un contexte RenderScript
  RenderScript rs = RenderScript.create(this);

  //2) Creer des Allocations pour passer les donnees
  Allocation input = Allocation.createFromBitmap(rs, bmp);
  Allocation output = Allocation.createTyped(rs, input.
      ↪ getType());

  //3) Creer le script
  ScriptC_brightness bScript = new ScriptC_brightness(rs);

  //4) Copier les donnees dans les Allocations
  // ...
  //5) Initialiser les variables globales potentielles
  bScript.set_brightnessScale(0.8);

  //6) Lancer le noyau
  bScript.forEach_changeBrightness(input, output);

  //...
```


- 1) et 2) ne changent pas.
- 3) le nom du script change, car notre fichier rs s'appelle **brightness.rs**.
- 5) On initialise la valeur en appelant set_<brightnessScale>().
- 6) la fonction à appeler a aussi changé : forEach_changeBrightness() car notre fonction (décorée) dans le fichier rs s'appelle changeBrightness().
- Le reste est identique au code précédent.

Renderscript : précision des calculs flottants

Il est possible de jouer sur la précision des calculs flottants, en définissant au choix dans le script :

- **#pragma rs_fp_full**
- **#pragma rs_fp_relaxed**
- **#pragma rs_fp_imprecise**

Cela peut permettre de gagner en performance.
→ à tester !

Renderscript : précision des calculs flottants
Il est possible de jouer sur la précision des calculs flottants, en définissant au choix dans le script :
- #pragma rs_fp_full
- #pragma rs_fp_relaxed
- #pragma rs_fp_imprecise
Cela peut permettre de gagner en performance.
→ à tester !

Les macros sont listées de la plus précise à la moins précise.
rs_fp_full est le défaut si rien n'est spécifié.
Il est généralement assez sûr d'utiliser **rs_fp_relaxed**. Ca correspondrait à peu près à compiler du C/C++ avec -ffast-math

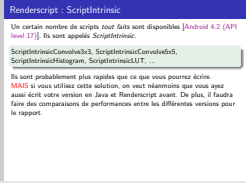
Si vous jouez sur ce paramètre, vos mesures de temps doivent apparaître dans le rapport !

Renderscript : ScriptIntrinsic

Un certain nombre de scripts *tout faits* sont disponibles [Android 4.2 (API level 17)]. Ils sont appelés *ScriptIntrinsic*.

ScriptIntrinsicConvolve3x3, ScriptIntrinsicConvolve5x5, ScriptIntrinsicHistogram, ScriptIntrinsicLUT, ...

Ils sont probablement plus rapides que ce que vous pourrez écrire.
MAIS si vous utilisez cette solution, on veut néanmoins que vous ayez aussi écrit votre version en Java et Renderscript avant. De plus, il faudra faire des comparaisons de performances entre les différentes versions pour le rapport.



RenderScript : références

- RenderScript Overview : https://developer.android.com/guide/topics/renderscript/compute.html
- Exemple RenderScript et AsyncTask : https://github.com/googlesamples/android-BasicRenderScript

RenderScript : références
- RenderScript Overview : https://developer.android.com/guide/topics/renderscript/compute.html
- Exemple RenderScript et AsyncTask : https://github.com/googlesamples/android-BasicRenderScript

On ne présente pas les AsyncTask, mais on vous encourage à regarder la documentation Android à ce sujet. C'est une solution pour avoir une application qui reste fluide...