

# Laboratoire 1

## Exploration de différentes architectures et configuration de l'Odroid-C2

ELE4205 - Département de génie électrique  
Polytechnique Montréal

17 août 2018

### Table des matières

1	Introduction	2
2	Configuration de votre compte	2
3	Compilation et exécution du logiciel de test en natif	2
3.1	Manipulations . . . . .	2
4	Compilation de l'application sous Yocto	7
4.1	Manipulations . . . . .	7
5	Exécution de l'application de test dans une architecture ARM sous QEMU	10
5.1	Manipulations . . . . .	10
6	Exécution de l'application de test dans une architecture x86 sous QEMU	12
6.1	Manipulations . . . . .	12
7	Installation de l'application de test sur l'Odroid-C2	14
7.1	Démarrage automatique du réseau par USB . . . . .	17
8	Évaluation	18

# 1 Introduction

Dans ce premier laboratoire, vous allez configurer votre environnement de travail. Puis, vous allez compiler et exécuter une application pour différentes plateformes cibles en compilation native et en compilation croisée. Vous allez utiliser différents outils : **GIT**, **GCC**, **QEMU**, **Bitbake**, etc. Ces outils vous seront présentés plus en détails dans le cours et les laboratoires suivants.

L'objectif de ce laboratoire est de vous familiariser avec l'environnement de compilation croisée du projet **Yocto** ainsi que de vous donner un aperçu sommaire des outils que vous allez utiliser durant la session. Vous utiliserez également l'outil **QEMU** qui permet d'exécuter une image d'un système d'exploitation en virtualisation ou en émulation sous différentes architectures telles que **ARM**, **x86**, **PowerPC**, ..

## 2 Configuration de votre compte

Le **shell** par défaut au laboratoire est le **tcsh** pour lequel l'invite de commande est donnée par

```
[user@poste:/repertoire\_courant ]\%
```

Vous devez configurer l'invite du **bash** (shell requis par **Yocto**) avec la commande

```
PS1="[u@h:w]\$ "
```

que vous pouvez ajouter dans votre fichier **.bashrc** avec

```
% gedit .bashrc
```

## 3 Compilation et exécution du logiciel de test en natif

Le logiciel de test qui sera utilisé est un calculateur de fractales de l'ensemble **Mandelbrot** *Multi-Thread*. Celui-ci permet de générer une vidéo d'un zoom progressif à l'aide de la librairie **OpenCV**.

### 3.1 Manipulations

1. Obtenez les sources de l'application avec ces commandes dans une invite de commandes<sup>1</sup> (ici vous devez remplacer **4205\_nn** par votre compte usager) :

---

1. À partir de maintenant nous allons tronquer l'invite du **shell** à **%** pour raccourcir les affichages des commandes. Le *shell* **tcsh** a une invite **%** alors que le *shell* **bash** que nous utiliserons plus loin a une invite que se termine par **\$**

```
% cd /export/tmp/
% mkdir 4205_nn
% chmod 700 4205_nn
% cd 4205_nn
% git clone https://<votre_utilisateur_bitbucket>@bitbucket.org/rgourdeau/ele4205-labo1.git
% cd ele4205-labo1
```

La commande `chmod 700 4205_nn` s'assure que seul votre compte a accès à ce répertoire (protection contre le plagiat). Vous devriez avoir ces fichiers dans le répertoire :

```
% ls
CMakeLists.txt  conf  include  labo1.bb  README.md  src
```

2. Compilez le programme avec CMake en version *Release* (optimisée et sans débogage) dans un répertoire que vous nommerez **build** :

```
% mkdir build
% cd build
% cmake -DCMAKE_BUILD_TYPE=Release ../
```

Vous devriez voir ce message :

```
Release Build
-- The CXX compiler identification is GNU 4.8.5
-- The C compiler identification is GNU 4.8.5
-- Check for working CXX compiler: /bin/c++
-- Check for working CXX compiler: /bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working C compiler: /bin/cc
-- Check for working C compiler: /bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
```

```
% make
Scanning dependencies of target Fractale
[ 33%] Building CXX object CMakeFiles/Fractale.dir/src/Fractale.cpp.o
[ 66%] Building CXX object CMakeFiles/Fractale.dir/src/ImagePNG.cpp.o
[100%] Building CXX object CMakeFiles/Fractale.dir/src/main.cpp.o
Linking CXX executable bin/release/Fractale
[100%] Built target Fractale
```

3. Exécutez maintenant le programme pour effectuer une vidéo de 6 secondes à un taux de 30 images par seconde. On doit donc écrire 180 frames.

```
% ./bin/release/Fractale -f 180
```

Vous devriez voir un fichier vidéo nommé `test.avi` apparaître dans le répertoire actuel. Vous pouvez le visionner pour voir le résultat.

4. Maintenant, on s'intéresse à profiler l'application. Faites la commande :

```
% sudo /users/Cours/ele4205/commun/scripts/perf-fractale-release
```

ce script contient les commandes suivantes

```
perf record ./bin/release/Fractale -f 180
perf report
```

pour lesquelles nous vous avons autorisé une exécution en *super user* (**root**).

Samples: 70K of event 'cycles', Event count (approx.): 60335291571

Overhead	Command	Shared Object	Symbol
30.72%	Fractale	Fractale	[.] Fractale::calculer
24.42%	Fractale	libm-2.12.so	[.] __ieee754_log
9.05%	Fractale	libgcc_s-4.4.7-20120601.so.1	[.] __muldc3
6.15%	Fractale	Fractale	[.] ImagePNG::EcritureImage
3.93%	Fractale	libm-2.12.so	[.] __ieee754_log10
3.11%	Fractale	libm-2.12.so	[.] __ieee754_hypot
2.55%	Fractale	libm-2.12.so	[.] __ieee754_sqrt
1.79%	Fractale	Fractale	[.] __muldc3@plt
1.40%	Fractale	libm-2.12.so	[.] __log10
1.06%	Fractale	libm-2.12.so	[.] __floor
0.82%	Fractale	libopencv_highgui.so.2.4.11	[.] png_write_find_filter
0.73%	Fractale	libz.so.1.2.3	[.] 0x0000000000004e13
0.68%	Fractale	libz.so.1.2.3	[.] 0x0000000000004de3
0.67%	Fractale	libm-2.12.so	[.] __hypot
0.55%	Fractale	libm-2.12.so	[.] __finite
0.50%	Fractale	libz.so.1.2.3	[.] adler32
0.42%	Fractale	libz.so.1.2.3	[.] 0x0000000000005505
0.33%	Fractale	libm-2.12.so	[.] isnan
0.32%	Fractale	libopencv_highgui.so.2.4.11	[.] png_do_bgr
0.30%	Fractale	libz.so.1.2.3	[.] 0x0000000000004e19
0.27%	Fractale	Fractale	[.] main

Press '?' for help on key bindings

Vous verrez une interface graphique apparaître à la console. On observe que la fonction de calcul de **Fractale** est celle qui consomme le plus de temps d'exécution et en deuxième une opération de la librairie **glibc** et en troisième une fonction de la librairie du compilateur **gcc** `--muldc3`. Si on fait une petite

recherche sur Google, on tombe sur cet article : [http://locklessinc.com/articles/complex\\_multiplication/](http://locklessinc.com/articles/complex_multiplication/).

En effet, l'implémentation d'une multiplication de nombre complexes vérifie si on est en présence de variables flottantes valides ou si on est en présence de symboles spéciaux tels que NaN ou Inf. Dans notre cas, nous pourrions négliger de tels cas puisqu'ils ne risquent pas d'arriver dans le cas de calculs de fractales.

Pour continuer, on remarque les appels aux bibliothèques de **OpenCV** qui sont négligeables ainsi que les appels *Kernel* principalement dû à la création des *threads*.

- Également, cet outil permet de voir les instructions en assembleur qui prennent le plus de temps. Mettez votre curseur pour sélectionner `Fractale::Calculer`, appuyez sur la touche `a` ou `enter` et choisir `Annotate Fractale::Calculer`. Observez l'opération `add $0x1,%r13d` qui prend 17% du temps d'exécution.

```
Fractale::calculer /export/tmp/4205_1/e1e4205-label/build/bin/release/Fractale
0.00      addsd 0x8(%rsp),%xmm0
movsd    %xmm1,0x38(%rsp)
0.25      addsd 0x10(%rsp),%xmm1
movsd    0x394b(%rip),%xmm4      # 408a00 <typeinfo for Fractale
movapd   %xmm0,%xmm2
movapd   %xmm1,%xmm3
0.06      mulsd  %xmm0,%xmm2
11.76     mulsd  %xmm1,%xmm3
11.02     addsd  %xmm3,%xmm2
17.03     ucomis %xmm2,%xmm4
5.85      jb     17f
17.24     12f: add  $0x1,%r13d
0.14      cmp    %r13d,%ebp
          jg     e8
          138: add  $0x1,%r14d
          add  $0x18,%r15
0.04      cmp    0x18(%rsp),%r14d
          jl     60
          14b: addl  $0x1,0x1c(%rsp)
          addq  $0x8,0x20(%rsp)
          mov   0x1c(%rsp),%edx
          cmp   %edx,0x2c(%rsp)
press 'h' for help on key bindings
```

- Quittez l'interface de perf en appuyant sur `q` et mesurez maintenant le temps d'exécution de l'application :

```
% time ./bin/release/Fractale -f 180
20.200u 0.074s 0:10.76 188.3%    0+0k 0+63568io 0pf+0w
```

La troisième colonne est celle qui nous intéresse, soit 10.76 secondes.

L'application de test utilise par défaut quatre threads. En théorie, le nombre de threads optimal est déterminé par le nombre de coeurs du processeur.

- Observez les informations du CPU avec cette commande :

```
% lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
```

```

Byte Order:           Little Endian
CPU(s):               8
On-line CPU(s) list:  0-7
Thread(s) per core:   2
Core(s) per socket:   4
Socket(s):            1
NUMA node(s):         1
Vendor ID:            GenuineIntel
CPU family:           6
Model:                26
Stepping:             5
CPU MHz:              1596.000
BogoMIPS:             5600.68
Virtualization:       VT-x
L1d cache:            32K
L1i cache:            32K
L2 cache:             256K
L3 cache:             8192K
NUMA node0 CPU(s):   0-7

```

On observe qu'il y a 8 CPUs, ainsi que 4 coeurs par socket et 2 threads par coeur. **Le nombre optimal de threads théorique pour cet ordinateur est donc de 8 threads!**

8. Mesurez le temps d'exécution de l'application mais cette fois-ci avec 8 threads :

```

% time ./bin/release/Fractale -f 180 -t 8
21.966u 0.140s 0:08.52 259.3%    0+0k 0+63568io 0pf+0w

```

En pratique, **il se peut que le nombre optimal théorique de threads ne soit pas celui en pratique.** Essayez avec 64 threads par exemple :

```

% time ./bin/release/Fractale -f 180 -t 64
24.516u 0.381s 0:06.65 374.2%    0+0k 0+63568io 0pf+0w

```

Vous verrez également que le temps d'exécution sera souvent différent à la fois précédente. **Le gain de performance en utilisant des threads est donc relatif à la complexité du programme.** **Il reste néanmoins que l'utilisation des threads est d'une grande importance pour les systèmes modernes à plusieurs coeurs.**

Maintenant, compilez l'application mais cette fois-ci en mode Debug :

```

% cd /export/tmp/4205_nn/ele4205-labo1/
% mkdir build_debug
% cd build_debug
% cmake -DCMAKE_BUILD_TYPE=Debug ..
% make
Scanning dependencies of target Fractale
[ 33%] Building CXX object CMakeFiles/Fractale.dir/src/Fractale.cpp.o

```

```
[ 66%] Building CXX object CMakeFiles/Fractale.dir/src/ImagePNG.cpp.o
[100%] Building CXX object CMakeFiles/Fractale.dir/src/main.cpp.o
Linking CXX executable bin/debug/Fractale
[100%] Built target Fractale
```

Exécutez maintenant le build debug de Fractale :

```
% ./bin/debug/Fractale
```

Vous verrez que l'application est beaucoup plus lente que la version Release !

Vous pouvez interrompre le programme avec `ctrl+C`.

Faites son profilage avec la commande suivante

```
% sudo /users/Cours/ele4205/commun/scripts/perf-fractale-debug
```

ce script contient les commandes suivantes

```
perf record ./bin/debug/Fractale -f 10
perf report
```

On conclut que la majorité du temps a été consacrée dans la librairie standard `libc`. En effet, la compilation `Debug` introduit des symboles qui permettent de tracer l'exécution et n'utilise pas les options d'optimisation de vitesse d'exécution comme la version `Release`. Pour cette raison, on a avantage à développer en `Debug` et livrer le logiciel final en version `Release`.

## 4 Compilation de l'application sous Yocto

On s'intéresse maintenant à compiler l'application de test dans un système d'exploitation Linux compilé sous Yocto.

### 4.1 Manipulations

1. Clonez le répertoire `GIT` du projet `Yocto` dans votre répertoire `/export/tmp/4205_nn/`, si ce n'est pas déjà fait :

```
% cd /export/tmp/4205_nn/
% git clone -b krogoth git://git.yoctoproject.org/poky.git
Cloning into 'poky'...
remote: Counting objects: 408471, done.
remote: Compressing objects: 100% (96863/96863), done.
remote: Total 408471 (delta 304893), reused 408281 (delta 304703)
Receiving objects: 100% (408471/408471), 148.66 MiB | 3.97 MiB/s, done.
Resolving deltas: 100% (304893/304893), done.
```

2. On doit maintenant inclure le `repo` d'`open-embedded` pour avoir le support requis pour notre système dont une recette pour `OpenCV`. Faites ces commandes :

```
% cd poky
% git clone -b krogoth git://git.openembedded.org/meta-openembedded
```

3. Nous allons maintenant cloner la couche du `Odroid-C2` qui sera utilisé comme plateforme pour le cours.

```
% git clone -b master https://github.com/akuster/meta-odroid.git
% cd meta-odroid
% git checkout 89685506742fa9d9c1860f3eebae5850e6235bdf
% cd ..
```

4. Clonez maintenant avec votre compte `Bitbucket` le répertoire `GIT` du `labo1` :

```
% git clone https://<votre utilisateur>@bitbucket.org/rgourdeau/ele4205-labo1.git
```

5. Entrez en `bash` pour configurer l'environnement de `bitbake` :

```
% bash
$ source oe-init-build-env build-arm

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
  core-image-minimal
  core-image-sato
  meta-toolchain
  adt-installer
  meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
$
```

6. Vous devez maintenant ajouter les « couches » requises pour générer le système. La dernière commande permet de vérifier que toutes les couches ont été ajoutées.

```
$ bitbake-layers add-layer /export/tmp/4205_nn/poky/meta-odroid/
$ bitbake-layers add-layer /export/tmp/4205_nn/poky/meta-openembedded/meta-oe/
$ bitbake-layers add-layer /export/tmp/4205_nn/poky/ele4205-labo1/
$ more conf/bblayers.conf
```

7. Finalement, modifiez le fichier de configuration du build pour mettre comme cible une architecture `ARM` émulée par `QEMU` et des flags supplémentaires pour



des fonctionnalités nécessaires pour inclure OpenCV ainsi que l'application de test dans l'image. Ajoutez (ou modifiez si elles existent déjà dans le fichier, à vous de vérifier) ces lignes dans `conf/local.conf` :

```
LICENSE_FLAGS_WHITELIST = "commercial"
LICENSE_FLAGS_WHITELIST += "commercial_libav"
LICENSE_FLAGS_WHITELIST += "commercial_x264"

PREFERRED_PROVIDER_jpeg = "libjpeg-turbo"

MACHINE = "qemuarm"
IMAGE_INSTALL_append = " _fractale"
EXTRA_IMAGE_FEATURES = "debug-tweaks _\
ssh-server-openssh _\
eclipse-debug _\
tools-profile"

TOOLCHAIN_HOST_TASK_append = " _nativesdk-cmake"
```

Les premières lignes permettent de s'assurer de l'inclusion de bibliothèques commerciales requises par OpenCV pour la compression vidéo. On a ensuite la sélection de la bibliothèque jpeg (il y a plus d'un choix), puis l'architecture cible, l'ajout du programme fractale à l'image et les outils pour l'accès distant et le débogage.

Dans les laboratoires, nous avons fait un *prefetch* les téléchargements dans un répertoire commun. Ajoutez le chemin du répertoire des téléchargements :

```
DL_DIR = "/export/tmp/4205_nn/yocto_downloads"
```

et créer le répertoire et décompresser les fichiers que nous avons pré-téléchargés (afin d'éviter un engorgement réseau) avec la commande :

```
$ tar -C /export/tmp/4205_nn -xf /export/tmp/4205_1/yocto_downloads.tar.gz
```

Soyez patient, il faut décompresser une archive  $\approx 11\text{Go}$ !

8. Vous êtes maintenant prêts à compiler l'image Yocto. Faites ces commandes<sup>2</sup> (la première permet de créer les fichiers de la distribution avec les bonnes permissions) :

```
$ umask a+rx u+rw
$ nice bitbake core-image-minimal
```

La compilation devrait prendre environ une heure. Lorsque la compilation sera terminée, passez à la prochaine section.

---

2. `nice` permet de donner la plus basse priorité à une tâche, ainsi vous pouvez continuer à travailler sur poste de travail sans ralentissement apparent.

En attendant la fin de la compilation, un tutoriel vous sera présenté au laboratoire.

## 5 Exécution de l'application de test dans une architecture ARM sous QEMU

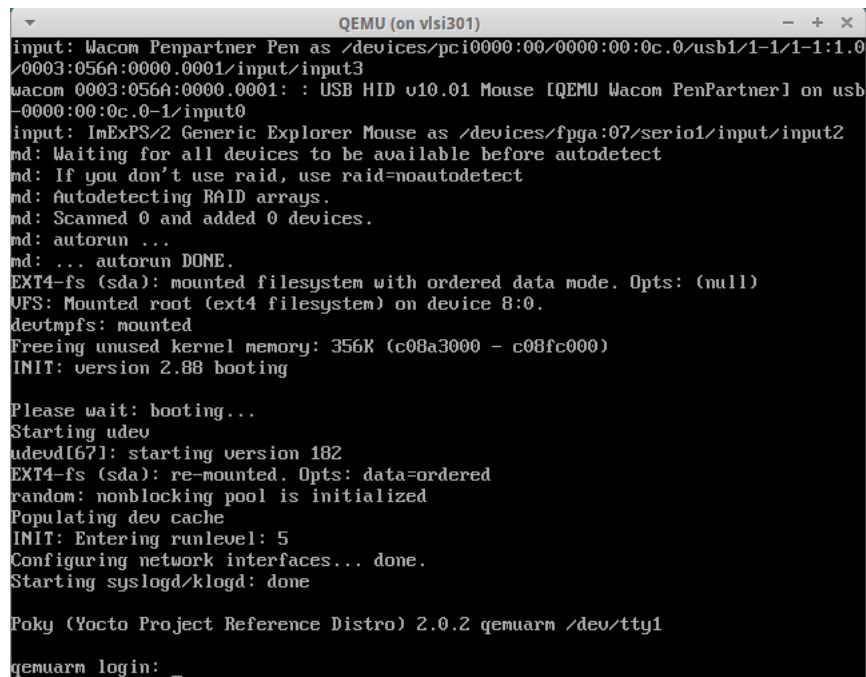
Vous êtes maintenant prêts à simuler le système d'exploitation Linux Yocto bati sur votre machine.

### 5.1 Manipulations

1. Pour exécuter QEMU, il faut s'assurer d'utiliser l'option `slirp` qui ne requiert pas de `sudo` :

```
$ runqemu qemuarm slirp
```

2. Vous verrez une console affichée qui vous demandera le login. Entrez-y `root`.



```
QEMU (on vlsi301)
input: Wacom Penpartner Pen as /devices/pci0000:00/0000:00:0c.0/usb1/1-1/1-1:1.0
/0003:056A:0000.0001/input/input3
wacom 0003:056A:0000.0001: : USB HID v10.01 Mouse [QEMU Wacom PenPartner] on usb
-0000:00:0c.0-1/input0
input: ImExPS/2 Generic Explorer Mouse as /devices/fpga:07/serio1/input/input2
md: Waiting for all devices to be available before autodetect
md: If you don't use raid, use raid=noautodetect
md: Autodetecting RAID arrays.
md: Scanned 0 and added 0 devices.
md: autorun ...
md: ... autorun DONE.
EXT4-fs (sda): mounted filesystem with ordered data mode. Opts: (null)
UFS: Mounted root (ext4 filesystem) on device 8:0.
devtmpfs: mounted
Freeing unused kernel memory: 356K (c08a3000 - c08fc000)
INIT: version 2.88 booting

Please wait: booting...
Starting udev
udevd[671]: starting version 182
EXT4-fs (sda): re-mounted. Opts: data=ordered
random: nonblocking pool is initialized
Populating dev cache
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 2.0.2 qemuarm /dev/tty1
qemuarm login: _
```

3. On peut maintenant exécuter le programme de test. Si tout est bien installé, le programme devrait se situer dans `/usr/bin`. Exécutez le programme pour qu'il écrive seulement une seule image :

```

root@qemuarm:~# which fractale
/usr/bin/fractale
root@qemuarm:~# fractale -f 1
Image #0 terminée!
root@qemuarm:~#

```

4. Le programme est définitivement plus long à exécuter. Regardez ce qu'il se passe avec perf :

```

root@qemuarm:~# perf record fractale -f 1
root@qemuarm:~# perf report

```

QEMU (on vlsi301)

Samples: 2K of event 'cpu-clock', Event count (approx.): 624250000

Overhead	Command	Shared Object	Symbol
10.13%	fractale	libgcc_s.so.1	[.] __aeabi_dsub
9.41%	fractale	libgcc_s.so.1	[.] __aeabi_dmul
9.29%	fractale	fractale	[.] Fractale::calculer
7.93%	fractale	libgcc_s.so.1	[.] __muldc3
6.81%	fractale	libgcc_s.so.1	[.] __aeabi_cdrcmple
5.33%	fractale	libm-2.22.so	[.] __log_finite
3.44%	fractale	libgcc_s.so.1	[.] 0x0000d254
3.32%	fractale	libgcc_s.so.1	[.] __aeabi_dcmple
3.24%	fractale	libgcc_s.so.1	[.] __aeabi_dcmpeq
2.48%	fractale	fractale	[.] 0x0000d2d8
2.44%	fractale	libgcc_s.so.1	[.] __aeabi_fmuloplt
2.28%	fractale	libm-2.22.so	[.] 0x0000d182c
2.04%	fractale	libm-2.22.so	[.] __sqrt_finite
1.76%	fractale	libgcc_s.so.1	[.] 0x0000d2e4
1.72%	fractale	libm-2.22.so	[.] 0x0000d1bd8
1.68%	fractale	libgcc_s.so.1	[.] __ledf2
1.60%	fractale	fractale	[.] ImagePNG::EcritureImage
1.44%	fractale	libgcc_s.so.1	[.] __aeabi_ddiv
1.24%	fractale	fractale	[.] 0x0000d2e3c
1.24%	fractale	libm-2.22.so	[.] 0x0000d1854
1.08%	fractale	libgcc_s.so.1	[.] 0x0000d2f0
1.04%	fractale	libgcc_s.so.1	[.] __Unwind_GetDataRelBase@plt
1.00%	fractale	libm-2.22.so	[.] 0x0000d1c00
0.96%	fractale	libm-2.22.so	[.] __log10_finite
0.96%	fractale	libm-2.22.so	[.] 0x0000d2104
0.88%	fractale	fractale	[.] 0x0000d2c8c
0.80%	fractale	libm-2.22.so	[.] 0x0000d1828

Press '?' for help on key bindings

The screenshot shows a QEMU window titled 'QEMU (on vlsi301)' with a blue header bar containing 'aeabi\_dsub /lib/libgcc\_s.so.1'. The main area displays the disassembly of the '.text' section. On the left, a vertical column lists execution times in seconds. The instructions are color-coded: red for the first two, green for the next three, and black for the last five. The instructions include 'eor', 'push', 'lsl', 'teq', 'tegeq', 'orrsne', 'munsne', 'beq', 'lsr', 'rsbs', 'rsblt', 'ble', and 'add'. The last instruction is 'eor r1, r3, r1'.

Time (s)	Address	Label	Instruction
6.32	0000d8c8	<__aeabi_dsub>	eor r3, r3, #-2147483648 ; 0x80000000
26.09	0000d8cc	<__adddf3>	push {r4, r5, lr}
			lsl r4, r1, #1
			lsl r5, r3, #1
			teq r4, r5
			tegeq r0, r2
			orrsne ip, r4, r0
			orrsne ip, r5, r2
			munsne ip, r4, asr #21
			munsne ip, r5, asr #21
13.83			beq dadc <__adddf3+0x210>
			lsr r4, r4, #21
			rsbs r5, r4, r5, lsr #21
			rsblt r5, r5, #0
1.98			ble d920 <__adddf3+0x54>
			add r4, r4, r5
			eor r2, r0, r2
			eor r3, r1, r3
			eor r0, r2, r0
			eor r1, r3, r1

Press 'h' for help on key bindings

On observe que les opérations élémentaires sont celles qui occupent le plus de temps d'exécution. Effectivement, QEMU ne virtualise pas mais émule l'architecture ARM. Il doit donc s'occuper d'interpréter les instructions d'assembleur ARM pour les transférer à votre machine qui effectue les calculs sous une architecture x86-64 !

Par exemple, lorsqu'on regarde le disassembly de \_\_aeabi\_dsub, on remarque que la section de l'instruction push {r4, r5, lr} est celle qui occupe le plus de temps.

Terminer votre session QEMU avec la commande :

```
root@qemuarm:~# poweroff
```

## 6 Exécution de l'application de test dans une architecture x86 sous QEMU

Cette section vous fera exécuter l'application de test dans une architecture x86 émulée par QEMU.

### 6.1 Manipulations

Vous avez deux options pour cette partie :

1. Utiliser une image est déjà compilée pour vous (recommandé vu le temps requis et la durée du laboratoire) et vous pouvez la récupérer dans le répertoire `/export/tmp`

Vous pouvez extraire et lister les fichiers requis avec ces commandes

```
$ tar -C /export/tmp/4205_nn/poky/build-arm/ \
    -xvf /export/tmp/4205_1/qemux86.tar.gz
$ ls tmp/deploy/images/qemux86
```

Puis, exécuter l'image avec cette commande :

```
$ runqemu qemux86 slirp
```

ou celle-ci en cas d'erreur de fichier non-trouvé :

```
$ runqemu tmp/deploy/images/qemux86/bzImage-qemux86.bin \
    tmp/deploy/images/qemux86/core-image-minimal-qemux86.ext4 slirp
```

2. Batir votre propre image. Démarrer un nouveau terminal.

```
% cd /export/tmp/4205_nn/poky/
% bash
$ source oe-init-build-env build-x86
$ cp ../build-arm/conf/*.conf conf
```

Remplacer `MACHINE ?= "qemuarm"` par `MACHINE ?= "qemux86"` dans le fichier `conf/local.conf`. Car seulement la machine change dans notre compilation à partir des mêmes meta.

```
$ umask a+rx u+rw
$ nice bitbake core-image-minimal
$ runqemu qemux86 slirp
```

Vous aurez alors une machine QEMU qui s'exécute. Exécutez l'application de test et son profilage :

```
root@qemux86:~# perf record fractale -f 1
```

L'application prend relativement moins de temps que sur la machine émulée ARM.

Regardons le profilage de l'application :

```
root@qemux86:~# perf report
```

The left screenshot shows a QEMU disassembly window with the following columns: Overhead, Command, Shared Object, and Symbol. The instructions are sorted by overhead, with the top entries being:

Overhead	Command	Shared Object	Symbol
33.77%	fractale	libgcc.s.so.1	[.] __muldc3
20.14%	fractale	fractale	[.] Fractale::calculator
11.56%	fractale	fractale	[.] _ZNSt6Bios_base4initC1E0p
7.66%	fractale	fractale	[.] ImagePNG::EcritureImage
2.28%	fractale	libm-2.22.so	[.] _log10
2.18%	fractale	[kernel.kallsyms]	[k] _raw_spin_unlock_irq
1.94%	fractale	[kernel.kallsyms]	[k] _do_softirq
1.68%	fractale	libgcc.s.so.1	[.] 0x000023c0
1.41%	fractale	libm-2.22.so	[.] _log10_finite
1.37%	fractale	libm-2.22.so	[.] __hypot
0.73%	fractale	fractale	[.] 0x00002500
0.74%	fractale	fractale	[.] _ZN20i1UideoWriterD1Ev0p
0.72%	fractale	[kernel.kallsyms]	[k] _do_page_fault
0.62%	fractale	libm-2.22.so	[.] __hypot_finite
0.60%	fractale	libm-2.22.so	[.] _fmod
0.56%	fractale	libm-2.22.so	[.] _fmod_finite
0.56%	fractale	fractale	[.] _ZN5olsEi0plt
0.44%	fractale	libm-2.22.so	[.] __x86_get_pc_thunk.bx
0.33%	fractale	libm-2.22.so	[.] __cabs
0.31%	fractale	libm-2.22.so	[.] __isinf_as
0.22%	fractale	libm-2.22.so	[.] __x86_get_pc_thunk.dx
0.27%	fractale	libm-2.22.so	[.] 0x00038c4b
0.25%	fractale	libm-2.22.so	[.] _finite
0.22%	fractale	fractale	[.] 0x00002720
0.21%	fractale	[kernel.kallsyms]	[k] get_page_from_freelist
0.19%	fractale	fractale	[.] main
0.18%	fractale	[kernel.kallsyms]	[k] _raw_spin_unlock_irqrestore

The right screenshot shows a detailed view of the instruction `fstp %st(5)` at address `3ab`. The instruction is highlighted in orange, and the overhead percentage `85.06` is shown in orange. The instruction is part of a block of code that includes `fstp %st(0)`, `fstp %st(0)`, `fstp %st(0)`, `fstp %st(1)`, `fstp %st(4)`, `fstp %st(0)`, `fstp %st(0)`, `fstp %st(1)`, `fstp %st(4)`, and `fstp %st(4)`.

Cette fois-ci, ce sont les multiplications de nombres complexes qui prennent le plus de temps d'exécution. Si on observe son *disassembly*, l'instruction `fstp %st(5)` prend jusqu'à 85%! En faisant une petite recherche, cette instruction indique de copier le premier élément de la pile dans le cinquième emplacement de la pile du coprocesseur et retourne (*pop*) l'élément qui a été copié.

Terminer votre session QEMU avec la commande :

```
root@qemux86:~# poweroff
```

## 7 Installation de l'application de test sur l'Odroid-C2

Vous devez créer votre image (temps de compilation  $\approx$  une heure, à démarrer avant la fin du laboratoire). Démarrer un nouveau terminal.

```
% cd /export/tmp/4205_nn/poky/
% bash
$ source oe-init-build-env build-oc2
$ cp ../build-arm/conf/*.conf conf
```

Il faut remplacer `MACHINE ?= "qemuarm"` par `MACHINE ?= "odroid-c2"` dans le fichier `conf/local.conf`. Car seulement la machine change dans notre compilation à partir des mêmes meta.

Lancer Bitbake

```
$ umask a+rx u+rxw
$ nice bitbake perf
```

Vous allez avoir éventuellement une erreur de compilation sur `perf`. Il faudra modifier un *header* comme suit

```
$ gedit tmp/work-shared/odroid-c2/kernel-source/arch/arm64/include/uapi/asm/sigcontext.h
```

La ligne 61 doit être modifiée pour obtenir

```
#define ESR_MAGIC          0x45535201

struct esr_context {
    struct _aarch64_ctx head;
    __u64 esr;
};

#endif /* _UAPI__ASM_SIGCONTEXT_H */
```

et reprendre la compilation.

```
$ nice bitbake core-image-base
```

Les fichiers pour votre image seront dans le répertoire /export/tmp/4205\_nn/poky/build-oc2/tmp/deploy/images/odroid-c2/ dont le fichier core-image-base-odroid-c2.sdcard qui pourra être écrit directement sur la carte microSD avec l'adaptateur en utilisant la commande<sup>3</sup>

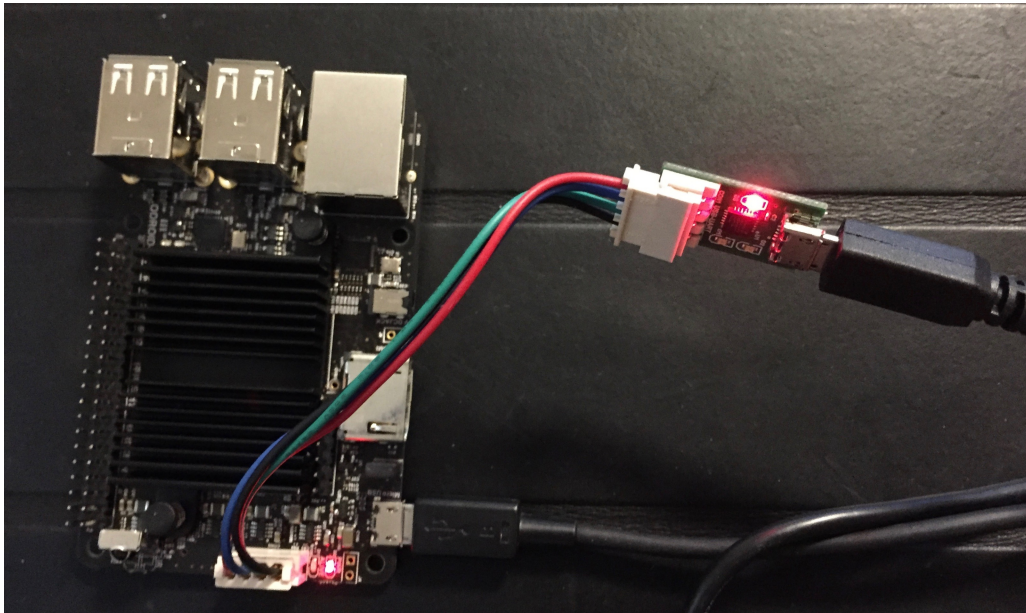
```
$ cd tmp/deploy/images/odroid-c2/
$ dd if=core-image-base-odroid-c2.sdcard of=/dev/sdc
$ sync && sync
```

La commande **sync** est très importante car elle permet de vider les *file buffers* ce qui termine l'écriture « physique » sur la carte **microSD**.

Insérer la carte **microSD** dans l'emplacement prévu sous le **Odroid-C2**. Le branchement des différents câbles vous sera montré au laboratoire et est illustré à la figure ci-dessous.

---

3. **Attention** : ici, on fait l'hypothèse que la carte microSD est le *device* /dev/sdc (3<sup>e</sup> disque physique). Pour savoir, si c'est /dev/sdb, /dev/sdc ou autre, utiliser la commande **df -h**.



- Brancher le câble UART-USB, par [console série](#), en premier et démarrer le programme `minicom`. Les options par défaut dans le laboratoire permettent la communication.
- Puis, brancher le câble micro-USB qui, entre autre, fournit l'alimentation.

Après le démarrage de l'Odroid-C2 et une liste de messages du système, une invite de commande vous permettra un *login*. L'utilisateur `root` n'a pas de mot de passe. Vous pouvez tester l'exécution du programme `fractale` et utiliser les programmes `perf` et `time` (de la même façon que lors de l'emulation avec `QEMU`) pour analyser la performance du programme.

Nous allons maintenant [activer le réseau du côté de l'Odroid-C2](#) (chargement du module `usb-gadget-ethernet` puis activation d'`usb0` dont la configuration est donnée dans le fichier `/etc/network/interfaces`).

```
# modprobe g_ether
# ifup usb0
```

Du côté de l'ordinateur de développement, [il faut activer manuellement le réseau sur le `USB gadget ethernet`](#) avec la commande :

```
$ sudo /users/Cours/ele4205/commun/scripts/ifconfig-enp0s29f7-up
```

équivalente à `ifconfig enp0s29f7uX 192.168.7.1 up` (ici le X est un numéro identifiant un « port usb »), mais pour laquelle un `sudo` est autorisé au laboratoire.

Vous pouvez maintenant communiquer avec le Odroid-C2 en SSH :



```
% ssh root@192.168.7.2
```

## 7.1 Démarrage automatique du réseau par USB

Nous allons maintenant reconfigurer notre image pour un démarrage automatique du réseau. Il faut d'abord charger automatiquement le module `g_ether`. L'ajout de la ligne suivante dans le fichier `conf/local.conf` :

```
KERNEL_MODULE_AUTOLOAD += "g_ether"
```

va créer pour nous le fichier

```
/etc/modules-load.d/g_ether.conf
```

pour le chargement automatique du module. Il nous faut rebâtir l'image avec cette options (c'est très court comme compilation) :

```
% cd /export/tmp/4205_nn/poky/  
% bash  
$ source oe-init-build-env build-oc2  
$ nice bitbake core-image-base  
$ cd tmp/deploy/images/odroid-c2/  
$ dd if=core-image-base-odroid-c2.sdcard of=/dev/sdc && sync && sync
```

Il faut maintenant démarrer notre Odroid-C2 avec `minicom` éditer le fichier

```
/etc/network/interfaces
```

pour ajouter

```
# ifup automatique  
auto usb0
```

juste avant

```
iface usb0 inet static  
address 192.168.7.2  
netmask 255.255.255.0  
network 192.168.7.0  
gateway 192.168.7.1
```

Vous pouvez utiliser la commande `vi /etc/network/interfaces` pour l'édition (et oui, un peu de `vi` : voir l'[aide mémoire](#)).

Une fois que cela est fait, un redémarrage avec

```
root@odroid-c2:~# reboot
```

suivi d'un

```
$ sudo /users/Cours/ele4205/commun/scripts/ifconfig-enp0s29f7-up
```

sur le poste de travail va vous permettre une connexion SSH (attention, effacer l'ancienne clé du 192.168.7.2 dans `.ssh/known_hosts` de votre poste de travail).

Votre Odroid-C2 est maintenant configuré. Si tout fonctionne, vous n'avez plus besoin de `minicom` pour communiquer à moins d'un problème de `boot`. Seul le démarrage avec le câble d'alimentation suivi de

```
$ sudo /users/Cours/ele4205/commun/scripts/ifconfig-enp0s29f7-up
```

et un `ssh` seront requis.

## 8 Évaluation

Il n'y a pas de rapport à écrire pour ce laboratoire. La présence est obligatoire et ce laboratoire compte pour 3% de la note finale. Vous devez compléter toutes les étapes pour réussir ce laboratoire.