Politecnico di Milano - Department of Electronics, Information and Bioengineering

POLITECNICO
MILANO 1863

OpenCL on ARM CPU

[Coding Project]

Student	Andrea Gussoni
ID	853423
Course	Advanced Operating Systems
Academic Year	2016-2017
Advisor	Giuseppe Massari
Professor	William Fornaciari

May 20, 2018

Contents

1 Introduction

The main purpose of this document is to sum up the work done during the development of the project for the *Advanced Operating Systems* course.

1.1 Problem statement

Quoting the project assignment of the project, the goal is to *Compile a OpenCL runtime (pocl) on a ARM board. Run some benchmarks. Provide a comparison in terms of execution time, power/energy consumption.*
Let's characterize more in detail the hardware and software used for the project.

1.2 Hardware

1.2.1 ARM Board

The main ARM board used for the project is an **ODROID-XU3** produced by **Hardkernel co., Ltd.** [3] provided by the **HEAP lab**.
The specifics are:

- A CPU belonging to the **Arm big.LITTLE** series, in particular a Samsung Exynos5422 Cortex™-A15 2Ghz and Cortex™-A7 Octa core CPU.
- A **Mali-T628 MP6** GPU that is certified for OpenGL ES 3.1/2.0/1.1 and OpenCL 1.2 Full profile.
- This configuration of the board is equipped with **2Gbyte LPDDR3 RAM** PoP stacked.

You can visit the product page [7] for further details on the hardware.
I also used an **ODROID-XU4** of my own to advance in the completion of the project during the summer. I opted for this model since the previous model wasn't available from the producer, and the SOC platform (CPU and GPU) is identical with respect to the **ODROID-XU3**, except for small differences with ports and integrated peripherals, but I don't expect that this to influence the results of the benchmarks, also because the final results proposed here have been **always** produced with the board present in the **laboratory**.
There is a small chance that problems may arise from the fact that I manly tested the auto-deployment scripts on my personal board during the summer when the University was closed. Keep that in mind if there are some problems with the deploy scripts, since it may simply be a difference on packages names or a broken dependency.

1.2.2 Power Measurement

For the energy consumption measurements I used the **Hardkernel Smart Power** [11] provided me by the laboratory. I also had available an **Hardkernel Smart Power 2** [12], but unfortunately it wasn't compatible with the measurement software(detailed explanation on this in the software paragraph 2.3).

1.2.3 x86 Platform

The comparison of performances with the **x86** platform have been made on a Thinkpad X1 Carbon 3rd gen. that mounts and **Intel i5 5200U CPU** and 8 GB of ram.

1.3 Software

In this section we will describe the software component used for the development of the project.

1.3.1 OS

For what concerns the OS used during the development, I used the **Ubuntu 16.04.2 Kernel 4.9** image downloaded from the Hardkernel site. I used the suggested utility called **Etcher** [1] to flash the image to the eMMC of the ODROID-XU4. I assume that also the flash of the ODROID-XU3 has been done in a similar way.

1.3.2 OpenCL Runtime

For the benchmarks we actually used two OpenCL runtimes. The one used for the integrated Mali GPU is provided in the distribution repositories directly by the Hardkernel developers, and can be installed via the **mali-fbdev** package.
Instead for the CPU platform we manually fetched and compiled the runtime provided by the **Portable Computing Language (pocl)** [8] work group, version 0.14.

1.3.3 Benchmark Suite

The benchmark suite used is the **Rodinia Benchmark Suite**, version 3.1. This suite includes a lot of benchmarks specifically designed for systems that provide accelerators, and thus belong to the **heterogeneous computer systems** category. In fact the benchmarks provides parallelization features for three of the main parallel computing paradigms, that are **OpenMP, CUDA, and OpenCL**. We will of course use only the OpenCL benchmarks. The project has been started and it is mantained by the Computer Science Department of **University of Virginia** [14].

1.3.4 Result Analysis

For what concerns the gathering and the analysis of the results obtained from the run of the benchmarks, I mainly take advantage of **Bash** and **Python(v2)** scripts to collect the results, and also of **Gnuplot** [2] to create graphs representing the results.

2 Summary Of The Work

2.1 Becoming familiar with the OpenCL framework

Before starting the project I never worked with OpenCL, so I decided to research information through the documentation available online to have a grasp of a how an OpenCL application works. I used the documentation provided by the **Khronos Gropu** [4] as the main source of information about the C++ OpenCL Wrapper API.
In the meantime I tried to compile and play with **pocl** on my laptop, just to understand how to start from an OpenCL application and run it on the hardware. My main reference has been the pocl website, in particular the documentation [8] on the pocl project website.
I had some previous experience with the **LLVM** framework [6], that pocl uses for compiling the runtime, so this part was not too difficult to manage, also because the required version of LLVM (3.8) is the default version shipped by the Ubuntu distribution, but anyway I had it already compiled on my machine.
Once I had the runtime compiled and ready for my laptop, I moved to becoming familiar with the designated benchmark suite.

The first impact with the benchmark suite has been a little problematic since, for reasons that will be more clear when reading the section dedicated to the modifications made at the **Rodinia Benchmark Suite** 2.5, the suite is tailored for running on GPU, and since the pocl runtime on my laptop only exposed a CPU device, I wasn't able to run even a single benchmark out of the box, and not having yet developed the skills necessary to debug and work with the C++ OpenCL Wrapper API, I was having some difficulties.
For this reason I decided to begin with something simpler, and I searched for other Benchmark Suites online. I searched for a little bit and found the **ViennaCL** [13] suite.
This time things went better, and after some experiments and tentatives I managed to run some benchmarks of the suite on my laptop, and reading the code I began to understand how the initialization and run of an OpenCL platform worked.
During the documentation phase I become aware of the existence of the **Beignet** project, an Open Source OpenCL implementation to support the integrated GPUs on Intel chipsets, so I had the opportunity to experiment a little also with a GPU device even before working on the board.
At this point I felt that I had the prerequisites to start working with the **ODROID**, so I began the work on the board.

2.2 Build of the runtime

The first challenge to tackle was the retrieval and compilation of the OpenCL runtimes.
The runtime for the **Mali GPU** is already provided in the Hardkernel repository, so a simple `sudo apt -get install mali-fbdev` does the trick.
For what concerns the pocl runtime instead we need to start from scratch.
The first thing to do is to retrieve the last version of the OpenCL runtime (currently the last available versio is the 0.14) from the website. The next thing to do is to decompress the archive with a simple `tar xvfz pocl-0.14.tar.gz`.
Pocl take advantage of **LLVM** to build itself, so we need to install a few dependencies from the package manager before being able to compile it. We can find at the dedicated page on the official wiki a list of all the packages needed for the build. Basically we need LLVM and a bunch of development package of it, CMake to build the Makefiles, the standard utilities for compiling (gcc, lex, bison), and some

packages to have an Installable client driver (**ICD**), in order to be able to load the appropriate OpenCL at runtime.
What we need to do on our system is basically:

```
sudo apt-get update && sudo apt-get upgrade -y
sudo apt-get install -y vim build-essential flex bison libtool
   libncurses5* git-core htop cmake libhwloc-dev libclang-3.8-
   dev clang-3.8 and llvm-3.8-dev zlib1g ocl-icd-libopencl1
   clinfo libglew-dev time gnuplot clinfo ocl-icd-dev ocl-icd-
   opencl-dev qt4-qmake libqt4-dev libusb-1.0-0-dev
```

At this point we can proceed and build pocl. To do that we enter the directory with the sources and create a folder called *build* in which we will have all the compiled stuff. At this point we take advantage of **CMake** for actually preparing our folder for the build. Usually a `cmake ../` should suffice, but on the ODROID we have a little problem.

Since our CPU is composed of four cortex a7 and four cortex a15 cores, CMake can't by itself understand what is the target CPU to use for the build. Luckily the two types of cores shares the **same ISA**, so we can explicitly tell CMake to use the cortex a15 as a target type of cpu. All we have to do is to launch `cmake -DLLC\_HOST\_CPU=cortex-a15 ../`.
At this point we are ready for the build, just type `make -j8` and we are done. We can also run some tests with `ctest -j8`, just to be sure that everything went smooth, and finally install the runtime in the system with `sudo make install`. At this point if everything went fine we will have a `pocl.icd` file in `/etc/OpenCL/vendors/`, and running `clinfo` we should be able to see our brand new OpenCL runtime.

Additionally in order to be able to use the runtime for the **Mali GPU** we need to place a file named `mali.icd` containing:

```
/usr/lib/arm-linux-gnueabihf/mali-egl/libOpenCL.so
```

at the path `/etc/OpenCL/vendors/`.
This should conclude the part regarding the OpenCL runtime deploy, and at this point we should be able to see both the CPU pocl platform with an eight cores device and the Mali GPU platform with two devices of four and two cores respectively invoking `clinfo`.

2.3 Build of the power measurement utility

At this point we should get and compile the utility for measuring the power consumption of the board. The utility used is a modified version of the official utility provided by Hardkernel, that simply stores the consumption detected in a csv file, that we can later use for results analysis and plotting. For building the utility we start from this repository [10].

The use of the utility has been kindly granted to me by *Michele Zanella*, who is the main maintainer of the utility. He also helped me understanding how to make the utility work on the board, debugging a problem with the setup of the USB interface and kindly agreed to publish on his repository a dedicated branch were all the unnecessary Qt dependencies have been removed.
As first step we can retrieve the repository with the following bash command:

```
git clone https://bitbucket.org/zanella_michele/
    odroid_smartpower_bridge
```

At this point we should switch to the **no_qt** branch with a simple `git checkout no_qt`. In this branch all the non essential dependencies to Qt libraries have been removed, in order to avoid cluttering the board with the full KDE framework just for storing an integer representing the consumption. Of course if we want to have available the original GUI interface we need to compile the version present on the **master** branch.

Unfortunately the HIDAPI library provided with the sources of the utility has been already compiled for x86 and stored in the repository, causing an error when trying to link the utility.
To avoid this we need to recompile the library, by entering the HIDAPI folder and giving the following commands:

```
qmake
make clean
make
```

At this point enter the smartpower folder and compile the utility with:

```
qmake
make clean
make
```

now we should have in the `linux` folder a binary named `SmartPower`, this self-contained binary is the utility that we need. Please take care to install also the dependencies necessary for building this utility, in particular `qt4-qmake libqt4-dev libusb-1.0-0-dev`.

In addition, in order to be able to communicate through USB to the device even if we are not root, we need to add a file name `99-hiid.rules` in the path `/etc/udev/rules.d/` containing the following:

```
#HIDAPI/libusb
SUBSYSTEM=="usb", ATTRS{idVendor}=="04d8", ATTRS{idProduct}=="
    003f", MODE="0666"
```

Reached this point we should be able to take power measurements. To test it simply launch the **SmartPower** binary with as argument the file in which you want to store the results, let it run for a while and then stop it with a **SIGUSR1** signal. In the file you should find the power consumption (double check it with the display of the power measurement device). Also take into account that there is a known bug in the software, meaning that sometimes the utility is not able to retrieve the consumption and the process become a zombie process in the system. Take into consideration this if you have trouble in taking measurements, and before starting a new measurement please be sure that no other SmartPower process is running.

I also had the new version of the SmartPower device, but unfortunately they changed the interface and now it is no more possible to read the measurements via USB with the utility.

2.4 Build of the benchmarks

For what concerns the benchmarks, we start from the vanilla **Rodinia 3.1** benchmark suite, taken directly from the site of Virginia University [14] (you need to register on the site, and then you'll receive via mail a link to the real download page). Unfortunately the benchmarks are **not ready for**

running out of the box.
Some of them presents some bugs, and you need to apply a lot of fixes and modifications to successfully run them on the ODROID. Since the modifications are really big (I estimate that making the benchmarks usable has in fact taken most of the time of the development of the project), I opted for creating a repository that I initialized with the sources of the benchmarks and on which I worked.
You can find **the repository** here [9]. There are multiple branches on the repository since I worked in parallel on CPU and GPU benchmarks to make them work, and later I tried to merge all the results in a single branch to use for the benchmarks.

In addition to bugs and other problems the main difficulty was that the creator of the benchmarks **hard-coded** in the source the OpenCL platform, device and type of device to use. This meant that if you wanted to run benchmarks on different OpenCL devices you had to manually modify the source, recompile the benchmark and run it. At the beginning of the development I also followed this approach and specialized a different branch for running the benchmarks on CPU or GPU.
But this approach bugged me, since the main advantage and the ultimate goal of having an OpenCL application should be to be able to run it on different devices and accelerators with the minimum effort possible. So in the end I modified heavily the benchmarks in order to take as parameter the platform, the device and the type of device to use. I then added different **run scripts** that contain the right parameters for each available device.
In this way we **compile** the benchmarks **once**, and then at runtime we select the platform and device to use. The selection simply implies to use the `run-cpu` or `run-gpu` script. In this way we have the more *transparent* interface as possible.

2.5 Work on the Benchmark Suite

In this section I'll try to explain what are the main problems that I found in trying running the Rodinia Suite, and how I overcame them.
As said previously I decided to create a new repository containing the benchmark sources in order to keep track of the work and have a better organization over all the code base.
The first two steps where to initialize the repository with the original sources of the suite and then to remove all the **CUDA** and **OpenMP** related folders and references. I opted for this strategy and not for completely avoiding inserting them in the repository to facilitate keeping track of all the changes made at the code base, in the eventuality that in the future, when a new official release of Rodinia will be released, we want to re-apply all the changes.

The next problem to solve was the fact that all the benchmarks (with the exception of a couple) had hard-coded in the source code the OpenCL platform, device, and type of device to use, meaning that they always expected to find a GPU available on the platform and device with index zero.
The first idea that came to my mind was to create two branches on the repository, one to use with CPU and one to use with GPU. I then proceeded to work in parallel on the two branches modifying the source code of the benchmark to use the right device. This approach worked and in the end I was able to run the benchmarks on the two different types of device.

But this solution didn't really satisfied me, since was in some way **not coherent** with the OpenCL ultimate goals. Writing an application in OpenCL should give you the possibility to have a portable application that is able to run on different devices with the minimum effort possible. With the branches approach in order to switch from an executable for CPU to one for GPU we needed to switch between the branches and recompile the executable. In addition I find this kind of approach really not elegant

since the setup and initialization of the OpenCL devices is all done at runtime, so there is no particular reason for having those parameters hard-coded in the source code. We can in principle pass all those information at runtime when executing the benchmark. So I tried to make another step and, taking inspiration from the couple of benchmarks that already followed this kind of approach, I implemented a platform, device, and device type selection through passing different parameters to the command line.
As a general guideline the convention is to specify *-p* and an index to specify the platform to use, *-d* and an index to specify the device, and *-g* and a boolean with the meaning of using or not a GPU. For example, if we want to execute a benchmark on platform 0, device 1 and on GPU we need to pass something like this

```
-p 0 -d 1 -g 1
```

Instead if we want to execute on platform 1, device 0 and on CPU we pass something like this

```
-p 1 -d 0 -g 0
```

All this made possible the creation of different run scripts for the different types of execution. Look in the benchmarks folder for the various run-something scripts and see how we invoke the benchmark with different parameters in case we want to execute something on the Mali GPU or on the CPU.
In some situations was not possible to do this (parameters already taken or parameter parsing made in a way not compatible with this restructuring), and I'll specify this cases in each subsection explaining in detail the modifications made at the single benchmark. Also consider executing the benchmark binary without parameters (or with `-help`) to get an usage summary with all the necessary flags.
I'll now add a subsection for each benchmark trying to detail the modifications introduced with a brief explanation of them.

2.5.1 Backprop

The benchmark didn't use correctly the `clGetPlatformIDs` primitive, not retrieving at all the platforms present on the system. Modified this and added parameter parsing for the OpenCL initialization.

2.5.2 Bfs

The benchmark sources imported a **timer** utility for debug purposes that consisted of ad-hoc X86 assembly instructions to get the time in different execution points. This obviously prevented the compilation on an ARM device. Removed this dependency since we time the execution in a different manner, so we do not use this mechanism. The parameter follows the general guidelines.

2.5.3 Cfd

This benchmark didn't compile for problems with the import of the *rand()* function, so we fixed this. In addition the platform and device selection was not parametrized, so changed this. In this case we use the standard convention on the parameters as explained before.

2.5.4 Dwt2d

Implemented the device selection and fixed a bug with a `char` variable not compatible with our architecture. Since the -d flag was already taken in this benchmark to specify the dimension we use -i for the device id specification.

2.5.5 Gaussian

This benchmark already presented a prototype of platform and device selection. Added the possibility to select also the device type and changed some minor details in the use of the OpenCL primitives.

2.5.6 Heartwall

At first implemented the device selection as in the other case, and reduced the work group size in order to be compatible with the board. Unfortunately in the end the execution on CPU always returned the `CL_OUT_OF_HOST_MEMORY` error, and even with the minimum work group size the execution on CPU was not possible. I decided to disable and remove this benchmark since having only the data relative to the execution on GPU made no sense for the final comparative.

2.5.7 Hotspot

In this case there was an additional problem with work group size that was not compatible with CPU device. Reduced this work group size and implemented the device selection as described before.

2.5.8 Hybridsort

In this benchmark implemented the device selection adding a parameter parsing routine that works in parallel with the already present argument parsing routine, since integrating the two was too problematic.

2.5.9 Kmeans

In this case the only problem was with the platform retrieval, as in backdrop. Changed this and implemented device selection as described before.

2.5.10 LavaMD

In this benchmarks there were multiple problems. The first thing was the work group size too big to be handled on our device, so I reduced this.
The other more subtle problem was with the size of the parameter passed to the OpenCL kernel. Since the C++ `long` type has different sizes on 32-bit and 64-bit architectures (respectively 32-bit and 64-bit), while the `long` type in OpenCL code is always 64-bit wide, during the execution of the benchmark we received strange errors indicating some problems with the maximum size of the argument.
At first I thought that simply the benchmark was not adequate to be run on this platform, but after receiving similar strange errors with other benchmarks I decided to investigate more. After firing up gdb and some tentatives to understand what caused the `SEGFAULT` I decided to go for a step by step execution in parallel on two 32-bit and 64-bit devices. I finally found that the problem was with the `clSetKernelArg()` function. In fact I noticed that the the parameter passed to the kernel were different in size, and the kernel always expected arguments multiple of 64-bit.
Once understood this I modified the C++ variables corresponding to the arguments from type `long` to type `long long`, fixing this bug.
I find that this type of bug is really subtle, since for someone not knowing in detail the internals of OpenCL is really difficult to spot and solve a situation like this. In some way this should be prevented with some coding convention, for example always using the `long long` type for 64-bit wide variables. When writing an application that should be portable relying on behavior of the compiler for a specific

architecture should not be acceptable.
Also in this benchmark we implemented the device selection as described before.

2.5.11 Leukocyte

The first problem with this benchmark was an error with a Makefile target that prevented the compilation at all. Make was erroneously trying to compile also the header files resulting in an error when linking the final executable. Once fixed this the other problems encountered where with the dimension of the work group size that needed to be reduced. In addition also the initialization of the OpenCL context was done in a customary way not really functional, so I rewrote it in a more standard way.

2.5.12 Lud

In this benchmark the main change was the introduction of device selection as described before. Also fixed the use of the `clGetPlatformIDs` primitive to get all the platforms available on the board.

2.5.13 Nn

Also in this case there was already present a prototype of platform and device selection. Changed some details on the initialization of the OpenCl context to take into account the addition of device type specification.

2.5.14 Nw

Also in this benchmark the main change was the implementation of the device selection, and in doing this we changed also the parameter parsing for the already required parameters.

2.5.15 Particlefilter

In this benchmark I mainly implemented the device selection. Take care that in this case the argument order is important for compatibility reason with the argument parsing already there.

2.5.16 Pathfinder

Implemented device selection following the guidelines defined before. In this case the task had been a little difficult since we have a lot of function activations between the parameters parsing and the actual OpenCL context initialization, so we have a lot of parameters passing between the modules. The alternative was to use a global object to store the parameters, but I don't like this approach since in case of problems we can't simply debug looking at the function parameters to spot problems, but we need to trace the state of a global object, thing that I find not elegant and prone to synchronization errors.

2.5.17 Srad

In this benchmark we needed to reduce the work group size to be compatible with the ODROID, and implement the device selection as showed before. Also in this case the problem regarding the size of the arguments for the kernel manifested, so changed the size to match the one that the OpenCL kernel is expecting as done in LavaMD.

2.5.18 Streamcluster

Also in this benchmark we had the same problem already showed for LavaMD and Srad with the size of the kernel arguments. Fixed this and implemented the device selection, also fixing another bug with the initialization of the OpenCL Context with the `clCreateContextFromType()` primitive.

2.5.19 Consideration valid for all the benchmarks

Please keep into account that the code base of the benchmark has probably been modified by a lot of different developers, with different styles and approach to the OpenCL framework.
One problem that you can spot as soon as you look at a single commit is that there is no convention on the use of spaces or tabs(who would have guessed it?), so code is often misaligned and present trailing white-spaces and is really awful to look at with the editor set in the wrong way.
To avoid cluttering the commits with a lot of blank space removals and substitutions of tabs with whitespace I preferred to disable on my editor all the mechanisms that corrected this thing and leave the source code with misaligned lined but at least highlighting only the changes really made to the source.
I then tried to solve as much as possible all this things in a later commit that has the only purpose to obtain a source code less horrible to look at.
I apologize for this inconvenient and I ask you to not look at this problems within the commits, since I preferred to keep them as little as possible to have a better chance to spot the real modifications made and to avoid to get lost in a commit with thousands of line added and removed to fix a tab.

2.6 Running the benchmarks

Arrived at this point we should have a working version of the benchmarks. We can then proceed to run them on our board. We can take advantage of the scripts present in the folder of each benchmark to run it on the different devices available.
As the names of the run scripts say:

- `run-cpu` runs the benchmark on the 8 cores of the CPU belonging the pocl OpenCl platform
- `run-gpu-primary` runs the benchmark on the GPU device composed of 4 cores belonging to the Mali OpenCL platform
- `run-gpu-secondary` runs the benchmark on the GPU device composed of 2 cores belonging to the Mali OpenCL platform

We can also use the targets present in the Makefile inside the benchmark directory to conveniently run the sequence of all the benchmarks. We have:

- `OPENCL_BENCHMARK_CPU` to run all the benchmarks on the CPU
- `OPENCL_BENCHMARK_GPU_PRIMARY` to run the benchmarks on the GPU device 1 (4 cores)
- `OPENCL_BENCHMARK_GPU_SECONDARY` to run the benchmarks on the GPU device 2 (2 cores)
- `OPENCL_BENCHMARK_ALL` to run the benchmarks on alle the three previous devices
- `OPENCL_BENCHMARK_GPU` to run the benchmarks on the GPU device 1 (kept for compatibility reasons)

These targets automatically invoke the `time-and-save` bash script that is responsible of timing the benchmarks and collecting the power measurements using the `SmartPower` utility. Then it saves the results in differents files all named `total.dat` each one in the right sub-folder in the `results` folder in the benchmark folder. The files are basically *csv* files with three columns, and each record is composed of, in order:

- the name of the benchmark
- the run time of the run expressed in seconds
- the consumption expressed in Watt/hour

3 Analysis of the results

For the analysis of the results I opted for using **Python** and in particular the **numpy** and **matplotlib** libraries to analyze and plot the results obtained before.
At first I decided to use gnuplot to plot the results, but since the charts are quite populated I found matplotlib being more practical for this purpose.
I divided this step in three scripts that also correspond to different conceptual phase, and for this reason I preferred to keep them separate. We have three phases:

- Preprocessing: elimination of some faulty power measurements.
- Average Computation: we compute the *average* and the *stderr* between all the runs of each benchmark.
- Plot: the real creation of the final charts.

Let's see in more detail each phase.

3.1 Preprocessing

Unfortunately there is a bug in the measurement utility that in some occasions create a stall of the the process taking the measurement, leading to blank power measurement value in the *.csv* file were the utility stores the results or worse to a duplication of the previous value in the total.dat file.
To check this as first thing we need to look at the complete file and find duplicated values on consecutive lines. There might be a small chance that the values are correct, for example this sometimes happens with the *backprop* and the *bfs* benchmarks that have a really similar run time, and this actually means that also the power measurement will be very similar. We exclude this situation from our preprocessing by looking at the time values, and if their difference is less than 2.5 seconds we do not purge the power consumption value from the results. In all the other cases if we have a duplicate this is symptom of a problem with the measurement, so we purge the second identical power measurement record.

3.2 Average Computation

At this point we can proceed in computing the **average** and the **standard errror** of the measurements belonging to the same benchmark.
It is in fact suggested to repeat the run of the benchmarks an adequate number of times to, at the end, have a value that not affected by temporary high load of the system caused by reasons external to the benchmark, and also for getting an approximate idea of the accuracy of the results. In the following of this report the results have been obtained by averaging **10 runs** of each benchmark for each platform supported by the machine.
I took advantage of the *average* and *stderr* functions provided by *numpy* for conducting this analysis. The results are then stored in two files named *average.csv* and *stderr.csv* in which we have an entry per benchmark containing respectively the average and the standard error of time and power consumption.

3.3 Plot

The creation of the final charts is conducted with the help of the *matplotlib*, starting from the data already produced by the *analyze* script in the `average.csv` and `stderr.cvs` files.
Since the benchmarks have really different duration and energy consumption, having in a single plot

the data of all the benchmarks does not make much sense, since comparing a bar with an height of 180 seconds with another of eight of 2 seconds can't give much information.
So as first thing I decided to split the benchmarks in **two categories**, one with the becnhmarks that have a duration on CPU less than 30 seconds, and the other the remaining benchmarks.
I then organized the benchmarks in a plot chart having on the **x-axis** the various benchmarks, with **three bars** for each platform on which the has been run (CPU, GPU 4 cores, GPU 2 cores).
In this way we can easily spot the differences between the various platforms. Obviously to keep the charts readable there are two charts, one for the execution time and one with the power consumption.

3.4 Charts

In this section there are the charts representing the data produced, remember that all the benchmarks have been executed **10 times** per platform, per device, so these are actually the averages with also the representation on each bar of the standard error.

Following the charts relative to the benchmarks execution on the ODROIDXU3:

Figure 1: Execution time ODROIDXU3, *short* category

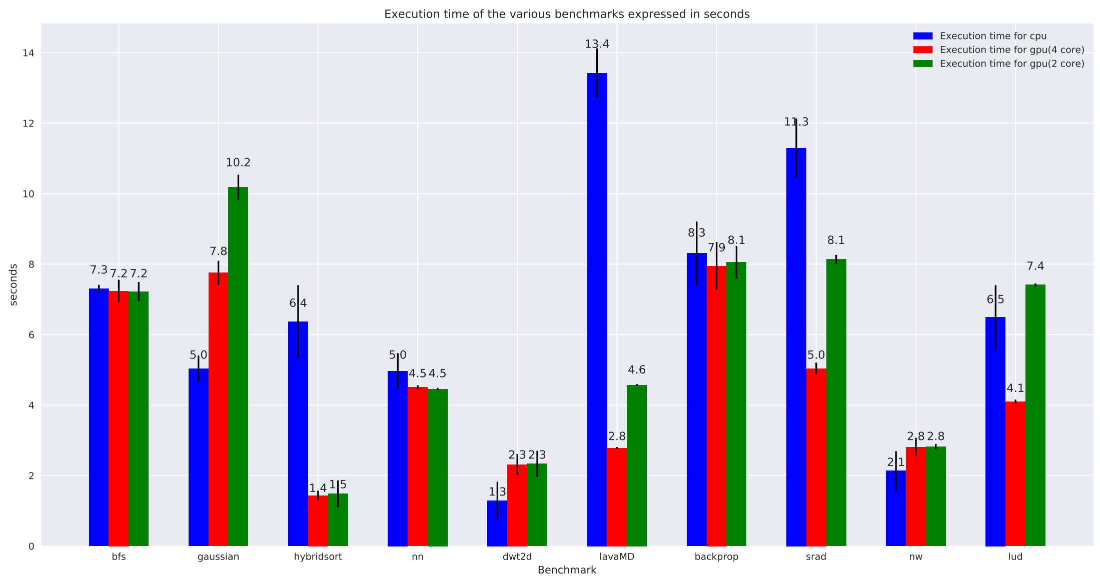


Figure 2: Execution time ODROIDXU3, *long* category

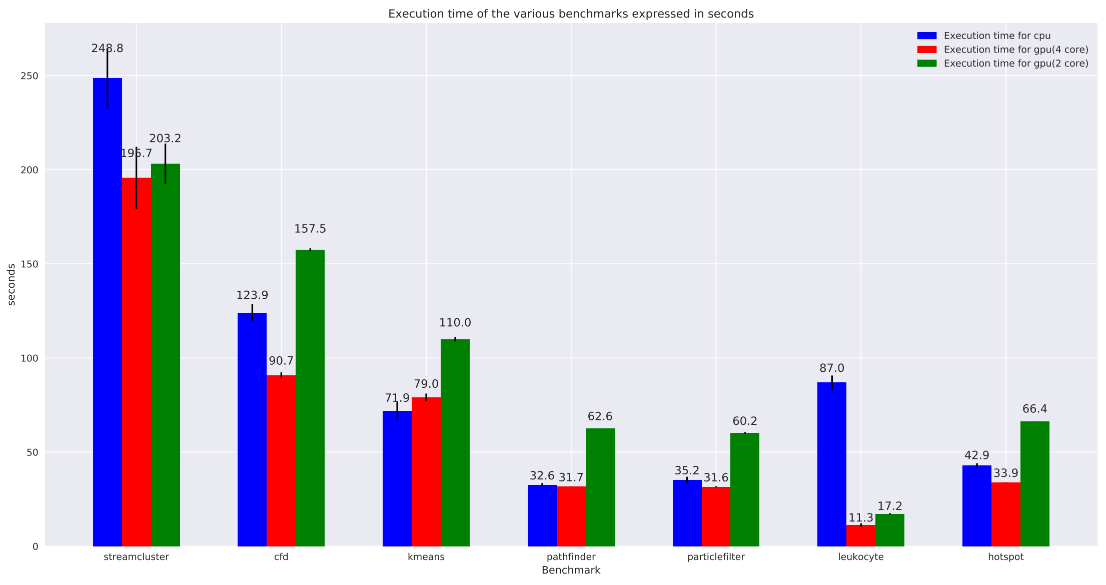


Figure 3: Power consumption ODROIDXU3, *short* category

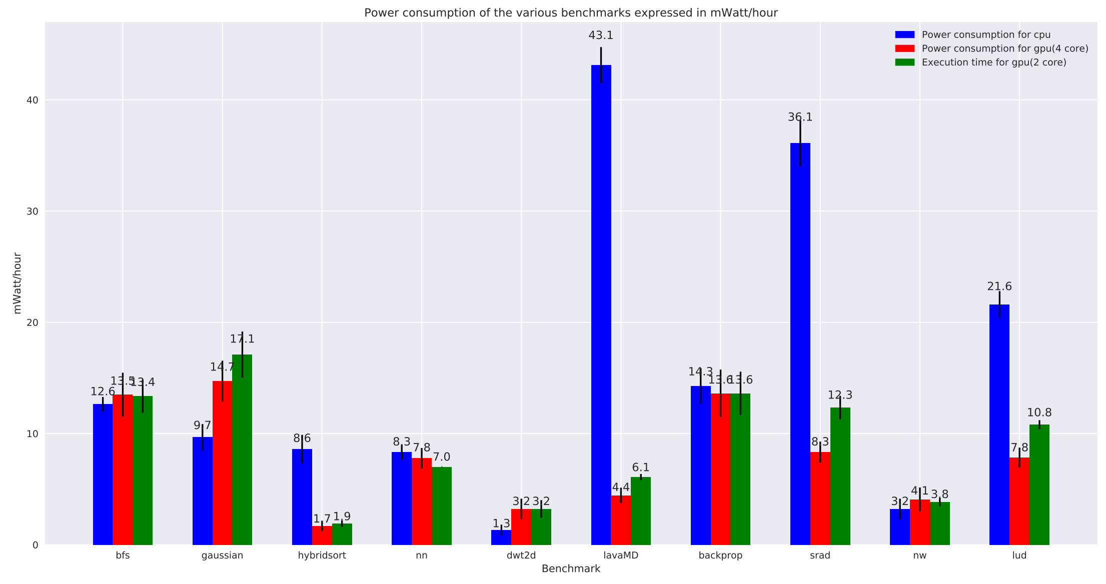
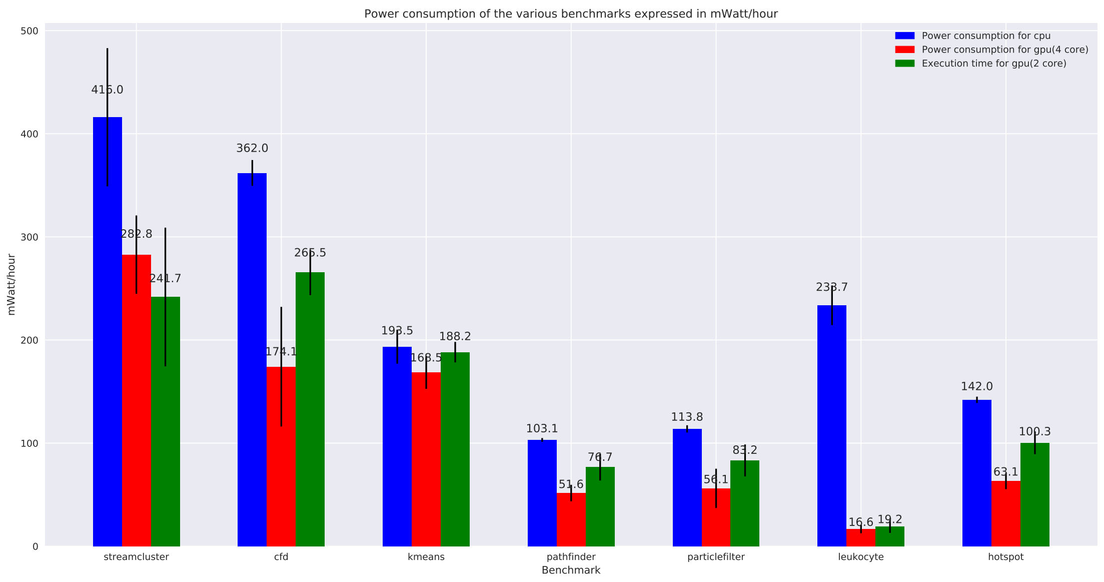


Figure 4: Power consumption ODROIDXU3, *long* category



Following the charts relative to the benchmarks execution on the ODROIDXU4:

Figure 5: Execution time ODROIDXU4, *short* category

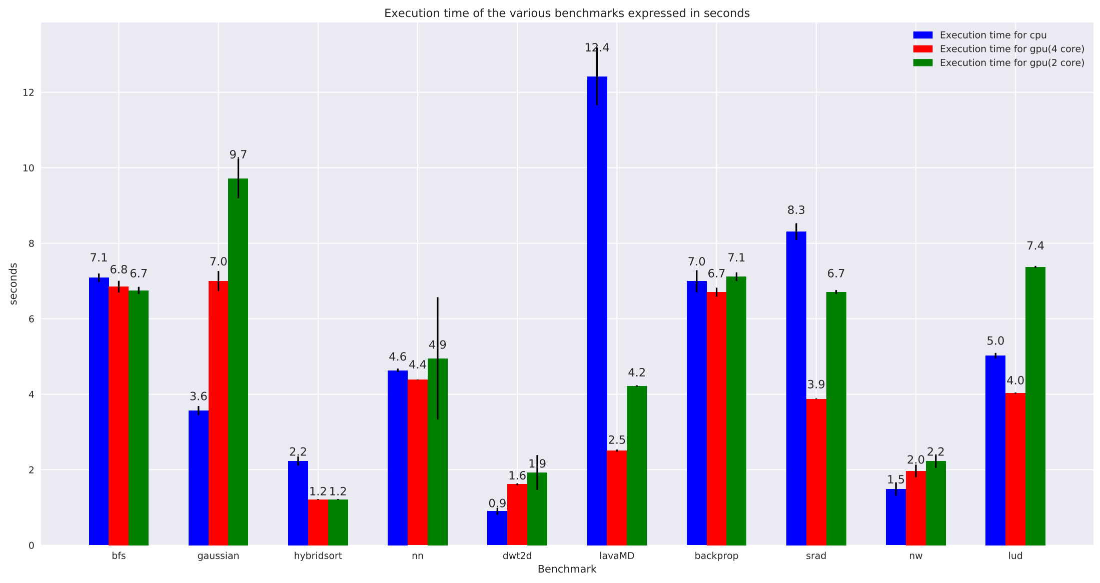


Figure 6: Execution time ODROIDXU4, *long* category

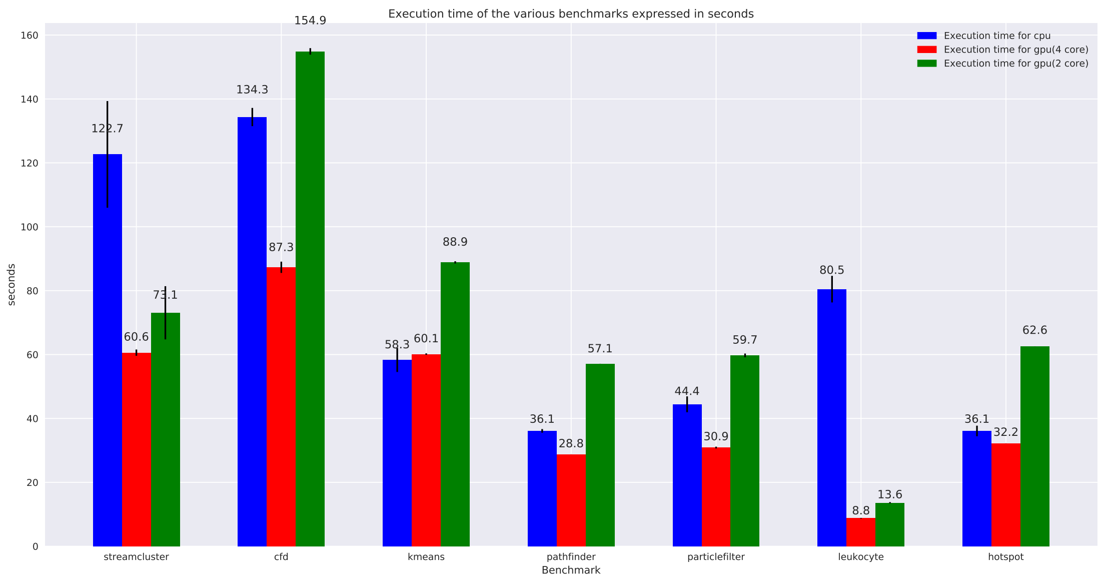


Figure 7: Power consumption ODROIDXU4, *short* category

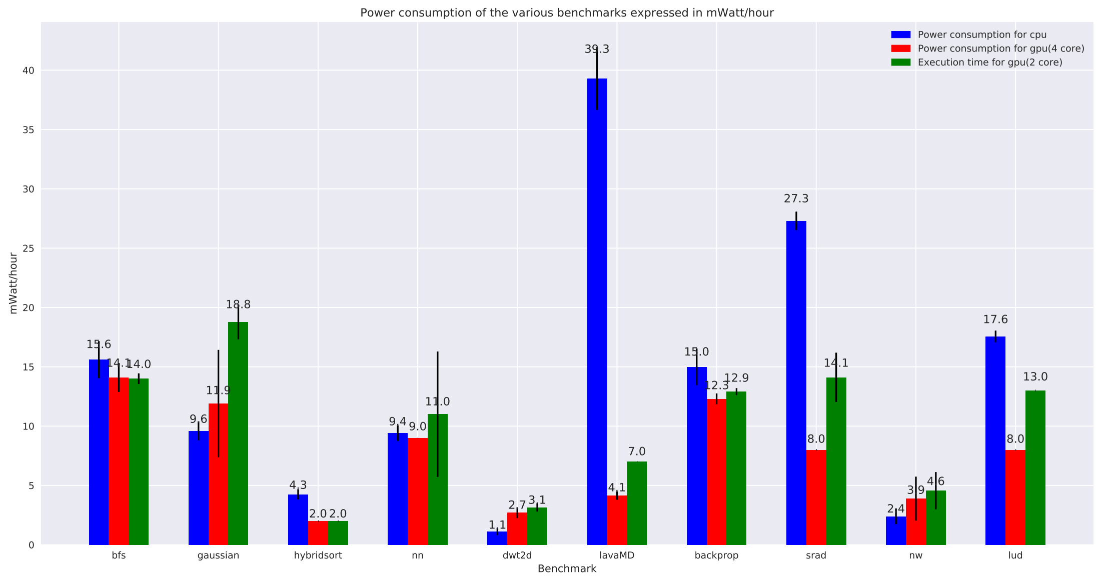
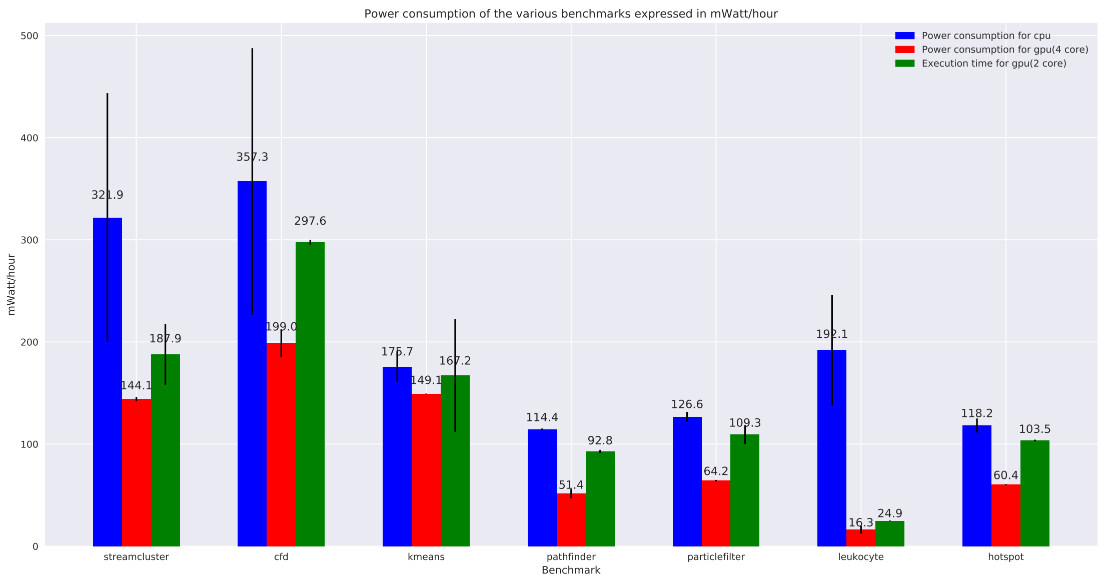


Figure 8: Power consumption ODROIDXU4, *long* category



I also made some measurement of only the execution time of the benchmarks on my laptop, whose characteristics are reported in the Hardware section 1.2 (I couldn't take power consumption measurements and even if I could have taking the power consumption of a such complex machine wouldn't have made much sense). Here are the charts relative to the benchmarks execution on my laptop (X86), only CPU, only execution time:

Figure 9: Execution time on Thinkpad X1, *short* category

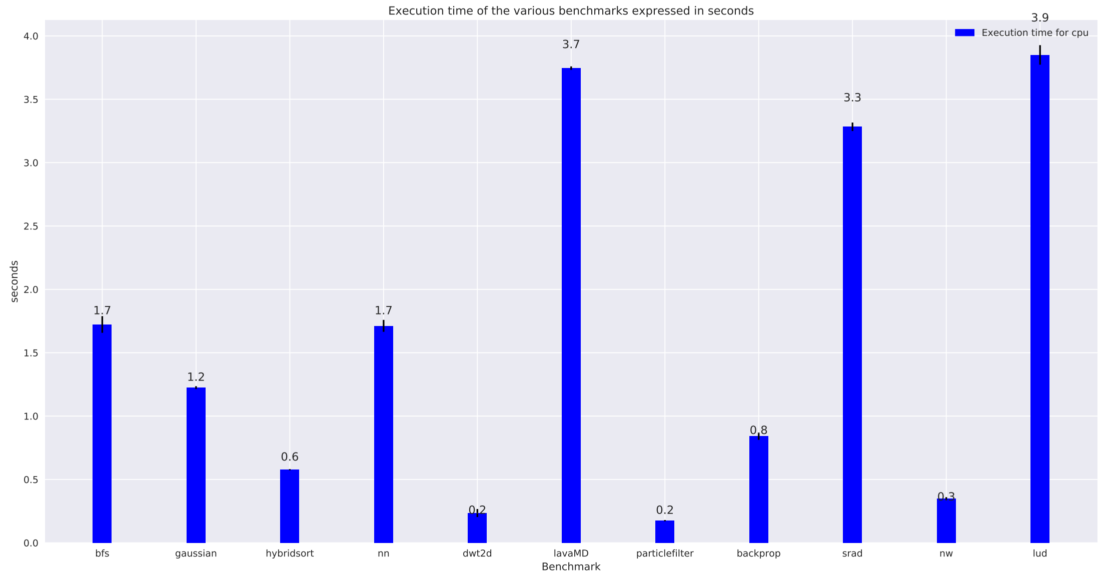
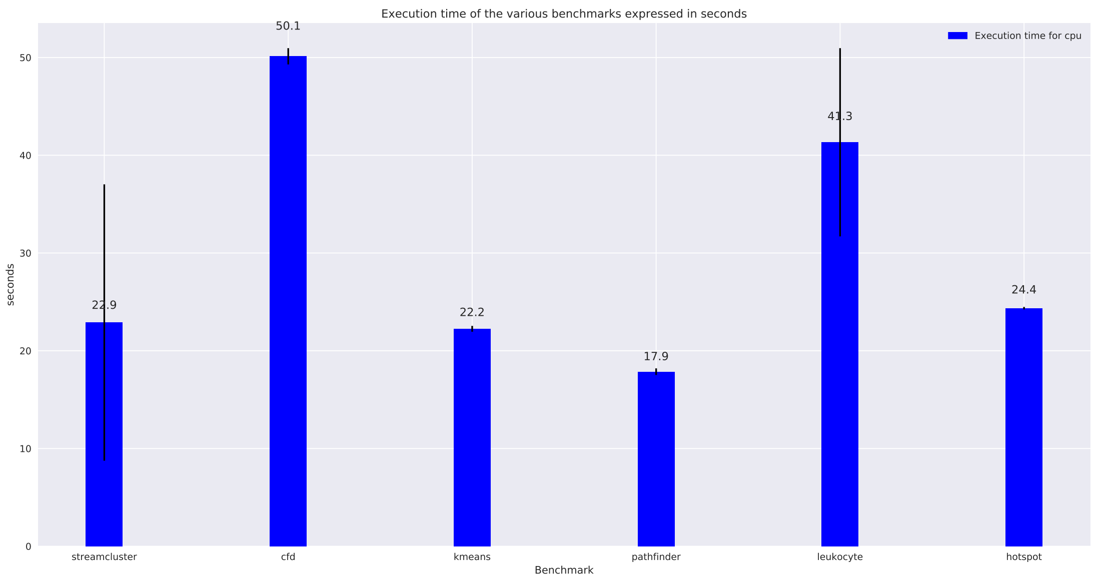


Figure 10: Execution time on Thinkpad X1, *long* category



3.5 Comments on the results

Let's analyze first the results obtained on the ODROIDXU3 . We can clearly see that on average running a benchmark on the GPU (especially on the one with 4 cores) is really beneficial for the execution time 2. This is particularly evident for the benchmarks which belong to the *long* category, where we have more time to see the effects of the increased computational power. On the *short* category instead in some cases we have an inversion of the roles, but this can be explained with the additional overhead that a GPU computation brings. In fact we need to copy the buffers on which the OpenCL kernels will work between the central memory and the memory of the GPU, and on benchmarks that are really short this additional overhead compensate the reduced execution time.

What is really surprising is the power consumption of the GPU 4(4 cores). We can clearly see that for basically all the benchmarks (except for the *gaussian* and *dwt2d* benchmarks, but we can explain this with the fact that for example gaussian is a benchmark which does not give opportunity to much parallelization, in fact it implements the gaussian elimination algorithm on a matrix, that is per se a sequential task, while dwt2d is a task such short that probably presents the problems of the copy of the buffers between the CPU memory and the GPU one) the power consumption is drastically reduced in the case of a GPU computation. Often, also the computation with the GPU composed of only 2 cores, even presenting an execution time higher than the one on CPU can achieve a better power efficiency. We can have an hint of this noticing that when running the benchmarks on the GPU the fan of the board spin significantly more rarely than when running the benchmarks on the CPU. This means that the SOC is dissipating less heat, a clear clue that the board is draining less power.

Also the results for the ODROIDXU4 presents more or less the same trends, and this is expected since the hardware configuration of the two boards is basically identical. What instead we can notice is that especially for the benchmarks belonging to the *long* category we have a reduction in execution time

and power consumption when moving from the ODROIDXU3 to the ODROIDXU4. At first this result may seem strange, since as we've said the SOC on the two boards should be identical, but I think that we can explain this noticing that while the XU3 was running the OS on a SD card, the XU4 was instead running the OS on a eMMC, which have a speed considerably superior to the one of even a good SD card. And this, together with the fact that often the benchmarks work on huge input files that we have to load from the storage, explains why the execution time reduced so much.
Let's have a detailed look at the results in 6: we can see for streamcluster we have an execution time on GPU of 60 seconds, when instead on the CPU we have more than the double, and the same trend is verified for the power consumption (more than double). For leukocyte instead we have an execution time 10 times higher for the CPU with respect to the GPU, and this is probably due to the intrinsic parallel nature of the benchmarks.
The main exception to this trend is the gaussian benchmark. In this case the GPU computation is always disadvantaged, probably due to the structure of the benchmark that is not so much parallel.
Another thing that is worth highlighting is that especially on *long* benchmarks the standard error computed on the power consumption is quite high. This means that during the measurement external factors may have influenced the measurements, even if obviously during the tests we had no other task running on the board. Probably the scheduling mechanism of the Linux Kernel caused this high variance, so maybe in the future will be interesting to try to deepen in this aspect and try to isolate the possible causes.

As a final note we can see that the execution times on the X86 platform are significantly slower 10, but this is something that in a way we can forecast. An X86 machine (even if with only 4 cores) has a lot of additional support hardware in the CPU, and a **CISC** architecture and achieve performances considerably superior with respect to a **RISC** one.
But if we look at the results of some benchmarks (e.g cfd or leukocyte), if we assume to be able to optimally parallelize the execution on all the devices available on the ODROID (not considering contention on the central memory), we can imagine that the final execution time could be comparable or at least of the same order of the one of the X86 architecture, leave alone the power consumption, that will be probably one order of magnitude inferior.
Obviously we can't be sure of this, and more research and experiment should be conducted in this direction.
But also without speculation, we can notice that for example the leukocyte benchmark is always really fast on the ARM GPU, even faster that on the X86 machine. This is because the benchmark is intrinsically designed to be run on a GPU platform (it consists of image frames elaboration) and thus we can conclude that our little ODROID can beat a **beast** such as Intel i5 CPU, under certain assumptions.

3.6 Detailed results

In this section we provide the detailed results obtained with the ODROIDXU3.

Table 1: Results on CPU

Name	Ex time avg.	Power avg.	Ex time stderr	Power stderr
bfs	7.305	12.6	0.109	0.6
gaussian	5.033	9.7	0.371	1.2
hybridsort	6.367	8.6	1.037	1.3
nn	4.969	8.3	0.502	0.7
dwt2d	1.293	1.3	0.532	0.5
lavaMD	13.425	43.1	0.696	1.6
streamcluster	248.779	416.0	15.785	67.0
cfd	123.94	362.0	4.612	12.5
kmeans	71.942	193.5	4.715	16.5
pathfinder	32.635	103.1	0.84	1.8
particlefilter	35.226	113.8	1.691	3.5
backprop	8.307	14.3	0.9	1.6
srad	11.294	36.1	0.839	2.1
leukocyte	86.971	233.7	3.705	19.2
nw	2.135	3.2	0.557	0.9
lud	6.498	21.6	0.905	1.2
hotspot	42.938	142.0	1.152	3.1

Table 2: Results on GPU (4 cores)

Name	Ex time avg.	Power avg.	Ex time stderr	Power stderr
bfs	7.237	13.5	0.321	2.0
gaussian	7.751	14.7	0.344	1.8
hybridsort	1.434	1.7	0.131	0.5
nn	4.509	7.8	0.055	0.9
dwt2d	2.315	3.2	0.3	0.9
lavaMD	2.781	4.4	0.029	0.7
streamcluster	195.689	282.8	16.543	38.0
cfd	90.693	174.1	1.69	58.0
kmeans	79.049	168.5	2.063	15.9
pathfinder	31.747	51.6	0.064	8.0
particlefilter	31.599	56.1	0.272	19.1
backprop	7.947	13.6	0.678	2.1
srad	5.038	8.3	0.169	0.9
leukocyte	11.284	16.6	0.863	4.0
nw	2.808	4.1	0.262	1.1
lud	4.099	7.8	0.058	0.9
hotspot	33.946	63.1	0.08	7.8

Table 3: Results on GPU (2 cores)

Name	Ex time avg.	Power avg.	Ex time stderr	Power stderr
bfs	7.217	13.4	0.274	1.5
gaussian	10.181	17.1	0.357	2.1
hybridsort	1.482	1.9	0.38	0.3
nn	4.457	7.0	0.032	0.0
dwt2d	2.335	3.2	0.371	0.8
lavaMD	4.57	6.1	0.025	0.3
streamcluster	203.219	241.7	10.673	67.2
cfd	157.451	265.5	0.81	22.0
kmeans	109.956	188.2	1.189	9.9
pathfinder	62.599	76.7	0.049	13.0
particlefilter	60.217	83.2	0.396	15.5
backprop	8.051	13.6	0.462	1.9
srad	8.133	12.3	0.13	1.1
leukocyte	17.193	19.2	0.4	6.3
nw	2.823	3.8	0.075	0.4
lud	7.41	10.8	0.045	0.4
hotspot	66.436	100.3	0.051	11.0

4 Conclusions and Future Works

4.1 OpenCL and Heterogeneous Computing

We have seen that even a quite affordable and small ARM board has real potential in running computational intensive applications. With a fraction of the cost of an X86 platform we can achieve really surprising results, also remembering that the power consumption is really limited.
This, together with applications written for **heterogeneous computing platforms**, can really open the road to embedded devices running heavy tasks with a fraction of the costs of a standard X86 platform.
The development of the Linux Kernel for ARM devices has really advanced a lot in the last years, and now we have a lot of alternatives for running a Linux-based distribution on an ARM device. So having available our favorite platform is no more a problem.
What I really think is important is to have a common programming language or framework that enables exploiting all the computational power provided by a board like an ODROID with the minimum effort possible in porting the applications.

I never worked with OpenCL before, but the approach that underlies the project is really promising, since once we have a well written OpenCL kernel we can basically run it on different types of devices without additional effort. If we imagine this applied on a dedicated board with a lot of GPU devices and a *small* CPU that only serves for running the OS and dispatching the tasks, we can easily obtain power efficient devices able to run heavy tasks. In addition to this we could have also other types of accelerators (such FPGAs, co-processors, cryptographic accelerators) that could benefit from this type of architecture.
There are also other examples of parallel and heterogeneous computing oriented platforms and programming paradigms (such as CUDA or OpenMP), but OpenCL really suits well the environment of embedded and low power platforms in my opinion.
The main challenge will be to exploit as much as possible all the computational power provided by these kind of boards, since we have seen that using a GPU can reduce by an half or more the execution of graphical oriented applications, and not exploiting all the hardware available on our device is really a waste.
Another main goal should be to understand when a task is more efficient on a GPU/CPU/Accelerator, and dispatch it accordingly to this criteria. For example in the case of our benchmarks we would like to have a policy that dispatch the streamcluster task on the GPU, while the gaussian one on the CPU, where it is more efficient.

4.2 Possible future extensions

A natural continuation for this project will be to better investigate how much the performances of the benchmarks are affected by the read/write speed on the main storage. Some benchmarks work on really huge input files, so it is probable that this fact affects in a huge way the final data that we get from the benchmarks.
We could try to run the benchmarks on inputs of different sizes, or track how much of the time is used for reading the files and how much for doing the actual computation. We could also try to preallocate in the main memory the data for example using a **tmpfs** file system and try to re-execute the benchmarks and see the differences.

Another possible extension could be to try to investigate how the benchmarks execution time and power consumption are affected by scheduling policy of the Linux kernel. We could try to change the scheduling class [5] when executing the benchmarks and try to see how the results are affected by this changes.

Another interesting thing to do would be to try to compare the power consumption of an X86 machine with the one of the ODROID, but of course we would need an appropriate criteria to fairly compare the two measurements.
Another interesting thing could be to try to execute the benchmarks on only the big or LITTLE core provided by the CPU. In this manner we can see how much the performances vary when using the low power cores or when using the high power ones, or even better understand if the LITTLE cores are in some way a bottleneck for the application.

List of Figures

List of Tables

References

[1] *Etcher Website*. URL: `https://etcher.io/`.

[2] *Gnuplot Website*. URL: `http://www.gnuplot.info/`.

[3] *HardKernel Website*. URL: `http://www.hardkernel.com/`.

[4] *Khronos website*. URL: `https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/`.

[5] *Linux Scheduling*. URL: `https://www.usenix.org/legacy/publications/library/proceedings/usenix01/freenix01/full_papers/alicherry/alicherry_html/node5.html`.

[6] *LLVM Website*. URL: `https://llvm.org/`.

[7] *ODROIDXU3 specifications webpage*. URL: `http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127`.

[8] *Pocl Documentation*. URL: `http://portablecl.org/docs/html/`.

[9] *Rodinia Custom Repository*. URL: `http://gogs.heisenberg.ovh/andreagus/rodinia-benchmark.git`.

[10] *SmartPower Custom Repository*. URL: `https://bitbucket.org/zanella_michele/odroid_smartpower_bridge`.

[11] *SmartPower1 product page*. URL: `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137361754360`.

[12] *SmartPower2 product page*. URL: `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G148048570542`.

[13] *ViennaCl Website*. URL: `http://viennacl.sourceforge.net/`.

[14] *Virginia University Website*. URL: http://lava.cs.virginia.edu/Rodinia/download.htm.