IN4342 Embedded System Lab Assignment 2:Canny Edge Detection

Group 11
Michiel van der Vlag 4369327 Andrea Pallini 4519086
Atul Pandaravila Biju 4511077 Christina Kylili 4520203
Christos Kyprianou 4516222 Dirk Vos 4391470

***Abstract*— One common scenario for embedded system development is having a team of engineers developing application on a desktop computer, and, once completed, give it to a team of system engineers to implement it on the final system. In this paper, we work as system engineers implementing a given application on the OMAP 3530 system. The application is an image processing application for edge detection that uses the canny edge detection-a well known edge detection algorithm, which was proven to provide good results when compared to other methods. The application is implemented in an efficient way on the target platform and executed according with the requirement for a set of data. Some minor modifications are done into the given application code in order to get them working on a heterogeneous platform, maintaining though the same functionality. The optimization of the code is divided in three main phases, at the beginning a deep profiling and understanding of the communication is performed.Then we implement the canny edge detection on GPP and DSP while in parallel exploring the NEON capabilities on GPP. Finally at the end we optimize the execution dividing the load in the best way between the two processors.**

I. INTRODUCTION

For this project, a canny edge detection algorithm is given and the goal is to optimize its performance using a multi-core heterogeneous system containing a general purpose processing unit (gpp) and a digital signal processing unit (dsp). The heterogeneous board given, is the Beagle board that contains an GPP processor with NEON support and a DSP. All the optimization were done having in reference a baseline solution in which the algorithm is simply running on the GPP processor using –O3 optimization level in the compiler.

Finding the optimal way to use the beagle board first requires an investigation on the algorithm and multiple test series' in order to decide the level of parallelism in which the two processors will collaborate. Also understanding the communication between the GPP and DSP is necessary. In order to make the above decisions, profiling tools were used and we obtained some insight regarding the load that every task gives to the processing unit. On considering that, we defined a strategy on how to use the DSP and GPP to optimize the execution time. Starting with only the GPP processor as a baseline, we test different scenarios where DSP processes part of the algorithm and thus reduce the load of the gpp as well as achieve lower overall execution times. Thus, during this assignment, multiple solutions were implemented and tested for three images with different sizes, texture and patterns in order to provide a general optimization of the algorithm in an embedded device.

The structure of the report follows the order: section II describes the image processing application *canny edge* while section III describes different features of the board based on processors collaboration. Profiling of the algorithm and the application's characteristics are also discussed in section IV. Section V explains team strategy along with the decisions made for optimizing the functions while in section VI, the final version is presented and discussed. Section **??** presents the team work and general conclusions are discussed in section **??**.

II. ALGORITHM: CANNY EDGE

As described during section I, edge detection is an image processing algorithm aiming to extract edges from a given image.Edge detection has a variety of applications in image processing and computer vision because it can give important information about the components that are included in an image and about the structural information of the objects included in the image. The algorithm use here is the canny algorithm. The algorithm performs in multiple stages starting with the use of a Gaussian smoothing (in the specific case), derivative computation, gradient direction, magnitude calculation, suppression and edge pixel calculation using hysteresis. As explained in section IV, Gaussian smoothing occupies the highest percentage of operating time when performing the algorithm. Since the specific part of the algorithm occupies 78% of the total execution time examining and understanding the specific part is considered the highest priority.

In image processing theory, image smoothing or image filtering in general is divided into two sub sections. First one is the creation of the filter and the second one is applying the filter to the image. Smoothing the image is a basic technique used in image processing that processes an input image in order to reduce the intense changes between adjacent pixels. For that a two-dimensional filter is used. Based on the operation a different filter is used such as median, average or Gaussian.

A Gaussian filter, as the name explains, is a filter created using a Gaussian distribution. That means the use of such filter adds more value to the nearest pixels to the filtered one and as the pixels get away from the filtering pixel their level of affection reduces. Just like a Gaussian distribution curve.The function that is used in the algorithm for creating the filter is:

```
void make_gaussian_kernel(float sigma, float
    **kernel, int *windowsize);
```

The use of this function returns a float array named *kernel* that includes the filter coefficient and the size of the array contained in the integer variable *windowsize*.

Creating the filter is the fist stage of the Gaussian smooth function. In the second stage the filter will pass through the image. As shown in figure 1, calculating the new value of a current filter demands reading all nearby pixels that are included in the filter. Specifically, for a $n \times n$ filter a $n \times n$ box is created around the processed pixel and each pixel is multiplied by its corresponding filter coefficient. After all pixels are multiplied, they are summed together and divided by the sum of the filter coefficients. The result is the new value of the processed pixel.

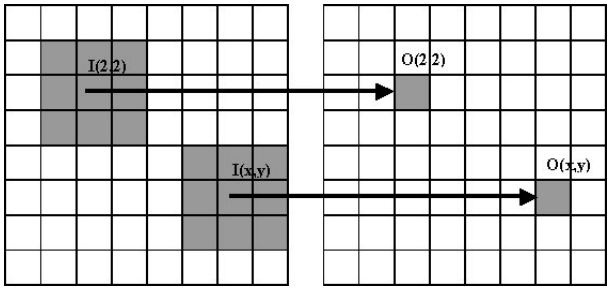


Fig. 1. 2D image filter (http://goo.gl/YV9SRQ)

At that point it shall be noted that kernel is created only depending by the *sigma* parameter which denotes the standard deviation of the Gaussian distribution. In the case of the given *canny edge* algorithm sigma is set to be 2.5 as a variable that user do not have the flexibility to change. This insight will be used further in optimizing the algorithm for the board.

The rest of canny edge receives the smoothed image and processes it in order to detect the edges of the image. In the specific algorithm, edges are extracted based on comparing gray levels (pixels are just black and white) between adjacent pixels to a threshold value.

III. The board: DSP, GPP and NEON

The board we have available is the Beagle board that consist mainly from an GPP processor and a DSP processor. The GPP and DSP can communicate in order to execute an application. Also the GPP processor can be supported with NEON.

1) DSP-GPP communication: In this assignment DSP and GPP a different communication method is used between DSP and GPP compared to the one used in the first assignment. The helper example pool_notify indicates the basic communication of the GPP and DSP, with the use of a shared memory region, the pool (fig. 2). Using the pool, the GPP places the data that need to be transferred in the pool. The DSP, after invalidation of its own cache, can write the data from the pool in its cache and can continue the execution part of the Gaussian function locally programmed on the DSP. When finished, the data will be put back in the pool so that the GPP will read the data from the pool. For the synchronization between the GPP and the DSP, semaphores and small notifications messages are used. Figure 3 shows the communication that is done in the pool_notify example, where the GPP place a simple vector to the pool. Is important to note, that first, GPP needs to send the addresses of the data buffer of the vector in a notification message, so the DSP knows from which address to read the data.

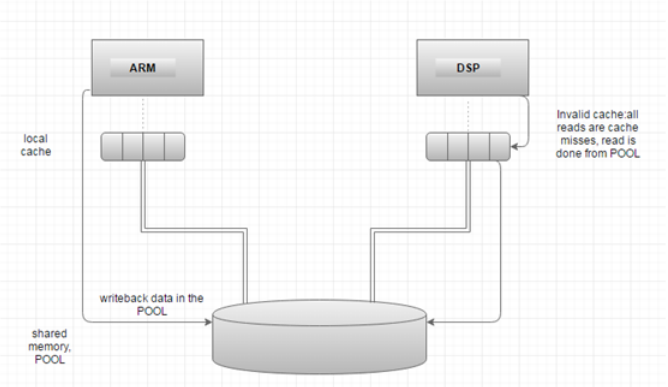


Fig. 2. Pool communication

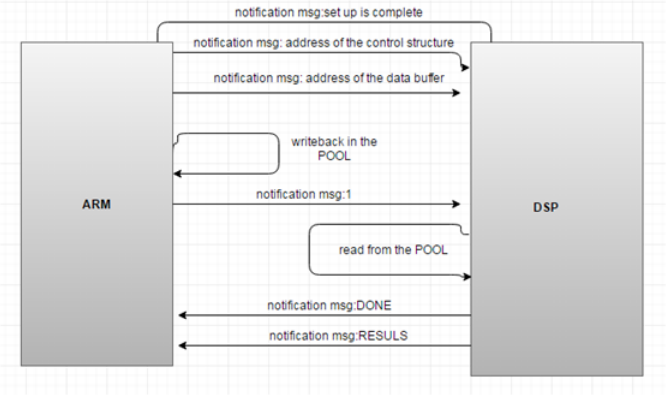


Fig. 3. Communication between GPP and DSP

We used this simple example and try to incorporate the whole algorithm in this, by initially testing the POOL communication using simple matrices. Is important to note how the write and read in the POOL is done. Figure 3 shows this. The POOL is just the main memory, and thus when GPP or DSP wants to write in the POOL, a simple write is not enough, since this will only write the data in the cache and not in the POOL. With the instruction "writeback" all the data are written in the POOL, not only in the cache. Similar, when GPP or DSP, needs to read the data from the POOL, a simple read will just fetch the data from the cache. Thus before reading, the cache is invalidated, causing cache misses, that will transfer the data from the POOL and not from the cache. With this way we ensure that always the data are the correct data retrieved from the POOL. This was the starting point to try the different optimization using DSP.

A. Neon Vectorized Instruction Set

The compiler is able to perform automatic vectorization on C or C++ source code. This gives access to high NEON performance without writing assembly code or using intrinsics. The code remains portable between different tools and target platforms. Because the C language does not specify parallelizing behavior, we gave additional hints to the compiler about where this is safe and optimal.

Vectorized calculations

Vectorization is a type of parallel computing where a single process performs multiple tasks at the same time. This is done by packing a set of elements together into an array and operating on them all at once. It's especially useful for multimedia purposes such as gaming, image processing, video encoding/decoding and much more. In order to vectorize our execution we used NEON's Intrinsics, which form an intermediate layer between assembly and C. They're a tad bit confusing, and make the C code more complex but thanks to the vectorization, it is possible to execute 4 operations (on integer number) per time in parallel. This implementation gives an incredible speedup to the execution of our program the figure below illustrates the logic implemented with NEON's Intrinsics.

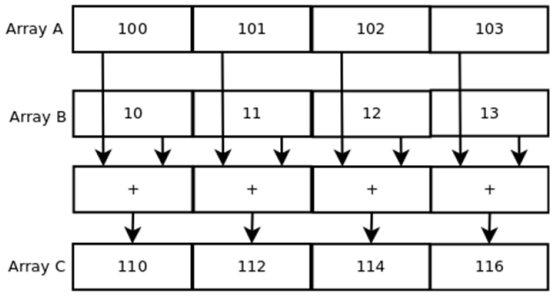


Fig. 4. The picture shows the algorithm used to improve the performance of our execution. Each element of each row of the first matrix is multiplied by the "corresponded" row of the second matrix. The sum of the result vector corresponds to the product matrix final row.

IV. Analysis of the algorithm using profiling

Profiling is a technique used to discover which parts of the given application are more important in the means of time and memory resource consumption. For that reason profiling provides crucial insight on which parts are critical bottlenecks to the performance of the application. Profiling is the first step to be done along with analysis of the algorithm in order to understand which parts to emphasize the optimization in order to achieve significant speed up.

Profiling is done on the baseline solution using MCPROF and GMPROOF tool. The results of the profiling presented in figure 5 for image *klomp.pgm*, show that the Gaussian function, the function that removes the noise from an image, is the most time consuming function, consuming more than 78% of the execution time. The size of the image doesn't influence the load division between the different functions as we got the same profile output for the other images as well. Furthermore, observing the pictures is interesting to see that while all functions operate only once, *follow_edges* function repeats it self for 1042 times. .

Based on the result provided by these tools, we focused our attention on optimization of the smoothed_gaussian function, because it is considered to provide the higher speedup if executed in an optimized way. In addition due to the long loop on *follow_edges* function is decided to concentrate neon resources on that in order to optimize is performance.

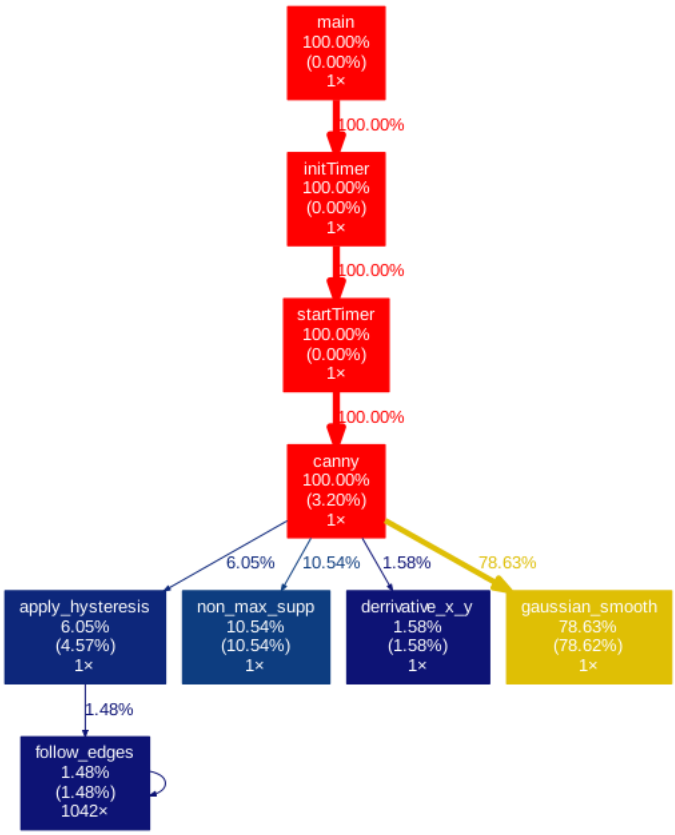


Fig. 5. Canny edge execution profiling

V. Strategy for Optimization and Implementation

Considering the results of the profiling, it is obvious that the bottleneck in the performance is the Gaussian function. Thus, in order to accelerate the whole algorithm we concentrated on optimizing the Gaussian smoothing part. In order to optimize the specific function, different techniques where employed such as sharing the processing load between GPP and DSP for the Gaussian function, using the maximum potential of neon processor and employing fix point libraries in order to reduce execution time of basic float operations on DSP processor.

A. Load balancing between DSP & GPP:

Having a heterogeneous processor, we considered sharing the processing load between the two processors in order to

achieve higher execution time. Even though in beagle board the main processor can be assumed to be the GPP, there are multiple communication techniques to send data to the DSP in order to collaborate together,

In the previous assignment message exchange was explored between the two processors but instead of that in the current assignment, we tried to exploit a memory component that the two processors share and exchange small message up to one unsigned integer 32 bit number as explained in section III.

Technically, communication between DSP and GPP can be done using short notification with the command

```
NOTIFY_notify
```

at GPP which sends a 32 bit unsigned int to the DSP. Similarly using the function

```
pool_notify_Notify (Uint32 eventNo, Pvoid
    arg, Pvoid info)
```

on GPP we are able to print notification messages received by the notifications sent by DSP.

Using the shared memory (pool) is slightly different since in that case the GPP first will allocate some space for a buffer, then save the buffer on the system, translate virtual memory to physical memory and notify DSP. A similar process is also undergoing to DSP when pool is used to send something.

Aiming to improve the execution time of the algorithm, one of the strategies followed is to test and observe on which processor the execution is faster and if combining the two processors would provide any benefits. At that point canny edge algorithm is set as a library and functions *Gaussian_smoothing* and *make_Gaussian_kernel* are set as independent functions instead called from function *canny*.

Having canny edge algorithm's different functions in a library enabled us to easily have access to all canny edge operation and especially to the Gaussian filtering which is the one decided to concentrate on. Using the *pool_notify.c* example as a starting point the changes in table I are done. Some similar changes are done in DSP *task.c* code file but in a smaller scale. The main differences are that in DSP kernel is not generated but added as a matrix. This decision was taken after considering the situation and concluded that for the user is not possible to alter kernel without changing the code thus making it a table would not make any significant difference.

With the changes on the code as shown in table I we are able to filter any part of image just by selecting the image's starting and ending row as an input parameter in Gaussian smooth filter. However, spitting the image in two parts created some problems during the filtering phase and later on edge detection. One of the problems is shown in figure 6. The reason we got that result is because when splitting the image in two, filtering it and then merging the two images, at the point where splitting happens the image filtering process misses important information. For

pool_notify.c	
Position	Change
main	Read image, allocate memory for buffer, create kernel and fill buffer with whole image.
create	Send notifications containing info regarding buffer memory address, number of rows and number of columns.
execute	Write into the pool and notify DSP to start filtering part of the image. GPP performs Gaussian smooth for its part of the image. Merging filter results and perform rest of canny edge.

TABLE I

BASIC CHANGES ON POOL_NOTIFY.C CODE FOR IMPLEMENTING DSP AND GPP SOLUTION.

that reason when combining again the two images have a slight difference in the split section thus the black line. To resolve the problem two solutions are used: first, when working on the DSP the whole image is sent so it cannot miss any information. DSP is used for filtering the lower part of the split image. In addition, when smoothing the image on GPP a certain offset of 20 rows is added instead of the initial image split row. Those offset rows added in the GPP filtering are part of the image that normally is processed on DSP. When merging the two images the first 10 offset rows are copied from GPP and the rest is copied from DSP. This way we achieve a smooth filter merging and remove the black line in the middle. The corrected result is presented in figure 7.

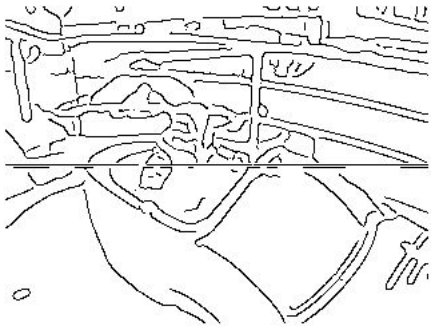


Fig. 6. Image detection error when splitting the image in two parts.

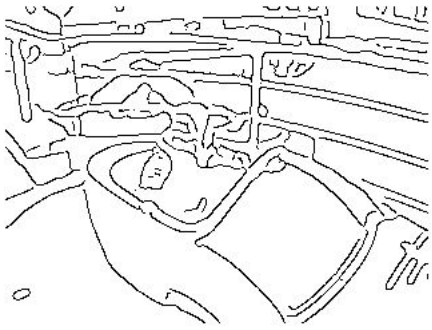


Fig. 7. Image detection error when splitting the image in two parts.

Proceeding with merging, several tests were made based on the portion of image each processor would filter. Specifically, the tests are done for the image being fully filtered on

GPP, 3/4, 1/2 of the image being filtered on GPP and finally the image being fully filtered on the DSP. When testing the algorithm three different images were used varying in size and characteristics. Those images are *klomp.pgm*, *tiger.pgm* and *square.pgm*. The algorithm execution times when splitting the image for filtering are presented in table II. In table III is shown the speedup compared with the baseline solution, it had been calculated dividing the baseline value with the execution time and we multiplied the result by 100.

Configuration	Square	Tiger	Klomp
Solely DSP (msec)	36	104	214
1/5 GPP 4/5 DSP (msec)	50	122	275
1/4 GPP 3/4 DSP (msec)	53	130	294
1/2 GPP 1/2 DSP (msec)	70	176	387
3/4 GPP 1/4 DSP (msec)	86	215	477
4/5 GPP 1/5 DSP (msec)	89	225	500
Solely GPP (the baseline) (msec)	91	236	538

TABLE II

EXECUTION TIMES OBTAINED WHEN SPLITTING THE IMAGE.

Configuration	Square	Tiger	Klomp
Solely DSP (%)	252	226	251
1/5 GPP 4/5 DSP (%)	182	193	195
1/4 GPP 3/4 DSP (%)	171	181	182
1/2 GPP 1/2 DSP (%)	130	134	139
3/4 GPP 1/4 DSP (%)	105	109	112
4/5 GPP 1/5 DSP (%)	102	104	107

TABLE III

PERCENTAGE OF SPEEDUP COMPARED WITH THE BASELINE VERSION OF THE RELATIVE IMAGE. THE FORMULA USED IS: (BASELINE/RESULT-OBTAINED)*100

Already the speedup is very positive and we achieved the base requirement for the assignment. However in the next sections another solution will be discussed, using vectorization on GPP side.

B. NEONIZING

The second optimization was to neonize all the functions executed on the GPP. For this we installed the NEON compiler of GPP. This compiler produced a report in which suggestions were made which function could be a target to neonize. We implemented all the suggestions. The Gaussian function was of course one of the functions which was targeted since this was the bottleneck when looking at the profile information. It was a great success implementing this. Overall we achieved a speedup of almost 33%. The baseline runs at 500 ms and the fastest neon implementation runs at 180 ms.

To convene the given image with a Gaussian function, a Gaussian blur is used. The image is converted into a matrix and computations are performed on it. The given program consists of a function which uses multiple Gaussian values to blur one pixel. These calculations depend heavily on the value of the pixel as well as its immediate neighbors. Usually C codes use loop iterations to perform calculations but we exploit the advantages of NEON, which is parallel computing. In such a scenario, we compute multiple values at the same time.

To store a blurred value in the Gaussian smooth function, a single-precision floating point variable is used but in NEON four single-precision floating points are used, which is 32-bits. NEON has 16 registers, so care had to be taken not to exceed that.

C. Qmath library

Texas Instruments offers a free library called QMath which allows the user to implement fixed point instructions very easily. You can change the position of the fixed point in a global define or use the specific functions for it which makes the design very flexible to use. This option is very interesting because the dsp porcessor (a C64+) has hardware for fixed point calculations and thus will get a good speedup using that hardware instead of regular floating point (which it has to do in software). Although we implemented the library for the dsp part and optimized the gaussian smooth function for fixed point, still we had to go with an integer version instead of the fixed point one because the image was returned distorted from the dsp, which had to do with the fixed point instructions. So with no further time left we decided to drop the qmath and further focus on a splitted workload version using both gpp and dsp. Texas instruments published an example of how normal math functions could be optimized by the right use of the qmath functionality as is seen in figure 8 below.

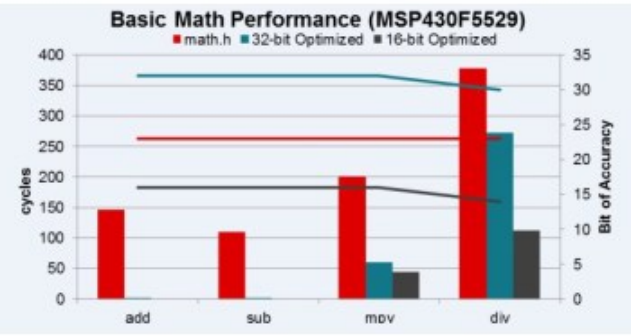


Fig. 8. qmath optimizations for math functions

The lines in figure 8 shows the accuracy of the fixed point for different instructions groups for different sources (math.h, qmath 32bit optimized, qmath 16bit optimized). What it also shows is the number of cycles needed to complete the instruction: In the gaussian smooth there are multiplications, divisions and additions which could all benefit a lot from the qmath functionality as the picture shows (the bars in the picture show the cycles needed).

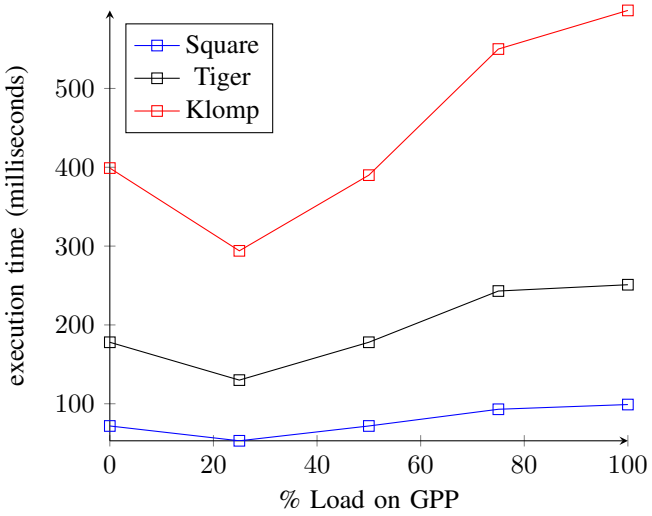
VI. FINAL VERSION AND RESULTS

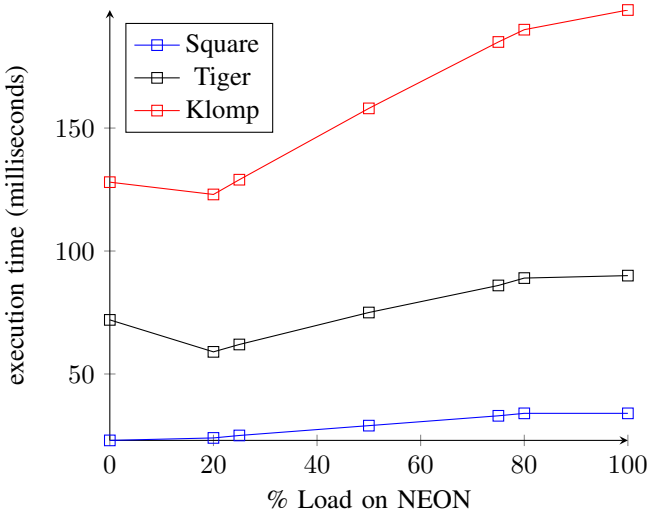
Concluding from the results, the best version is where the worklaod is divided 1/5 on the gpp and 4/5 on the DSP

(you can see the minima in the graph on this point). The final version includes the optimised gpp code using NEON intrinsics used on the gaussian smooth function and the magnitude_x_y function (these function are the most likely to give noticable speedup, as the MCProf tool showed us). Only the gaussian smooth function is moved to the dsp, because that function is the most likely to give speedup(indicated by MCProf) and was most suitable to allow concurrency (by dividing the image, the gpp could work on its own part and the dsp could work on its own part).

Configuration(with NEON)	Square	Tiger	Klomp
Solely DSP (msec)	23	72	128
1/5 GPP 4/5 DSP (msec)	24	59	123
1/4 GPP 3/4 DSP (msec)	25	62	129
1/2 GPP 1/2 DSP (msec)	29	75	158
3/4 GPP 1/4 DSP (msec)	33	86	185
4/5 GPP 1/5 DSP (msec)	34	89	190
Solely GPP (msec)	34	90	198

TABLE IV
EXECUTION TIMES USING NEON ON GPP.





VII. TEAMWORK AND CONTRIBUTIONS

The team is composed by 6 Embedded System Master students. Generally the overall teamwork for this project

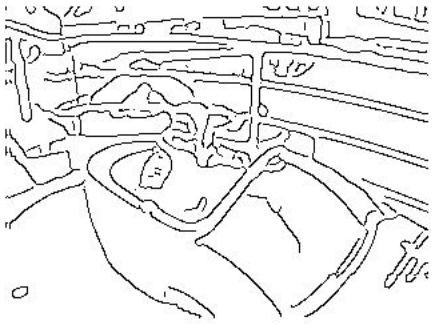


Fig. 9. Image detection results for klomp.pgm image.

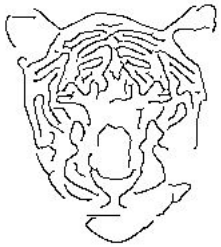


Fig. 10. Image detection results for tiger.pgm image.

was productive and satisfying. From the beginning we had a very good communication and collaboration. All of the members of the team were willing to work and contribute to the project. We were meeting in weekly basis and working on the project all together, each on different tasks, in groups of two. The main organization is analyzed in this section. Furthermore, when not in a meeting team members were able to connect to the beagle through a server that Michel installed in his house using a Raspberry pi board. This way team members were able to work any time anywere.

- Profiling: Michiel was the main member who run the profiling and explained us the results from them.
- Analysis: Christos at the beginning studied and understood deeply the canny edge detection code, and he also figured out how to and where optimize it. Andrea was more focus on the tutorial proposed (pool_notify example), he deeply understood how the communication works.
- NEON: Dirk and Atul were the members more involved in this solution. They also collaborated, when the solution was finished and complete, in merging their solution with DSP.
- DSP: The basic Gaussian function was added on DSP by Andrea and then further processed by Michiel, Christina and Christos. They also collaborated to merge this version with GPP and NEON.
- GPP: Christos, Michiel, and Christina were involved on this solution for most of the time, they worked on splitting the image between the two processors and they optimized the process. Further more Andrea contributed in order to merge the work completed on both side DSP and GPP.

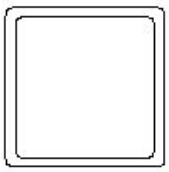


Fig. 11. Image detection results for square.pgm image.

- Report: Initially team progress was documented by Christina in a first draft version of the report. Later all the members of the team collaborate in the final version of the report adding their unique knowledge and contribution on the segment they worked.

VIII. CONCLUSIONS

This report describes the implementation and optimization of an the canny edge detection algorithm in an heterogoenous platform. In the optimization we tried to find the best solution in terms of execution time.

For that we try to execute the most time-consuming part of the algorithm in parallel using DSP and GPP. The two processors communicate and exchange information in order to achieve this. We tried changing the workload into the two processors until we conclude to the fastest implementation. After testing with various loads we concluded that the optimal division of the workload lays at 1/5 on the gpp and 4/5 on the dsp. The neon optimized the gaussian smooth function and the magnitude function with roughly 220% speedup.