KTH ROYAL INSTITUTE OF TECHNOLOGY

Application for image tracking to be run in three implementations: on single core with MicroC/OS-II, on single-core bare-metal, and on multi-core bare-metal

Group 19

Cioffi Giulia <cioffi@kth.se>

Ieva Antonia <aieva@kth.se>

Contents

1 Introduction

The goal of this project is to implement an application which tracks a given pattern, a circle, in a series of image frames and removes it from them. The application runs on the multi-core DE2 Altera FPGA board and should satisfy throughput and memory usage requirements. The final implementation has been gradually reached: first the application has been implemented on a single core using the MicroC/OS-II Real-Time Operating System (RTOS), then without the RTOS (bare-metal), afterwards all the five cores have been used and then this final version has been optimised.
The application has been realised starting from the analysis of an already working one, implemented as an executable process network written in the ForSyDe-Haskell language. After studying the application, a Synchronous Data Flow (SDF) graph has been drawn from it. Before writing the code for the application, the architecture of the DE2 board has been studied in order to understand how its components are interconnected and how they communicate. Then the C code for the application has been written, using the algorithm described by the SDF graph and eventually optimising it.

2 Application specifications

The application receives in input a series of image frames of arbitrary size in the PPM-format (P3); for each frame a pair of coordinates, corresponding to the X and Y position of the image pattern, is computed and then used to remove that pattern from the image.
The image frames in the input sequence are assumed to have a fixed size 64x64 and are stored in the SRAM. The P3 graphic format contains information stored as bytes, so each image is basically described as an array of bytes: the first and second bytes respectively contain the width and height of the array of pixels of the image; then the third byte contains the information about the maximum colour intensity (the higher this number, the higher the number of shades and mixed colours that can be obtained). An additional dummy byte has been added to start reading the useful pixels at byte number four, in order to easy the reading since the memory is aligned. Each pixel is described using three consecutive bytes, respectively representing the red, green and blue colour (rgb format). Therefore, to represent an image of 64x64 pixels, we need (64x64x3) + 4 bytes.
The image pattern that the application must recognise is a "circle" inscribed in a squared frame of 5x5 pixels. Since we are working with arrays, the circle is actually shaped as a rhombus.
The application works correctly if the sequence of input images respects some specifications:

1. the first image frame must contain the circle in the vicinity of a fixed position, whose coordinates (x,y) are (15,15);
2. in any given frame there can be at most one circle object;
3. between any two consecutive frames the circle object can move at most 15 pixels in any direction.

The application can run in two different modes, according to the user selection:

- *debug mode*: performs the image tracking algorithm on a set of input images, printing out the grayed input and the grayed-and-patched output in ASCII-art format and also the coordinates of the identified object;

- *performance mode*: the performance counter is enabled; the algorithm is performed looping over a set of images and stopping after a certain number of iterations, then printing the performance data.

The application has been tested with a sequence of 4 images, each one with a dimension of 64x64 pixels. Those images are saved in the SRAM memory starting from address 0, occupying a total space of 4x[(64x64x3)+4] = 49168 bytes.

3 Application model

The application has been modelled taking as reference a "golden-model" application written in the ForSyDe-Haskell language. Studying it, we draw the SDF graph of the application, which is shown in Figure 1.

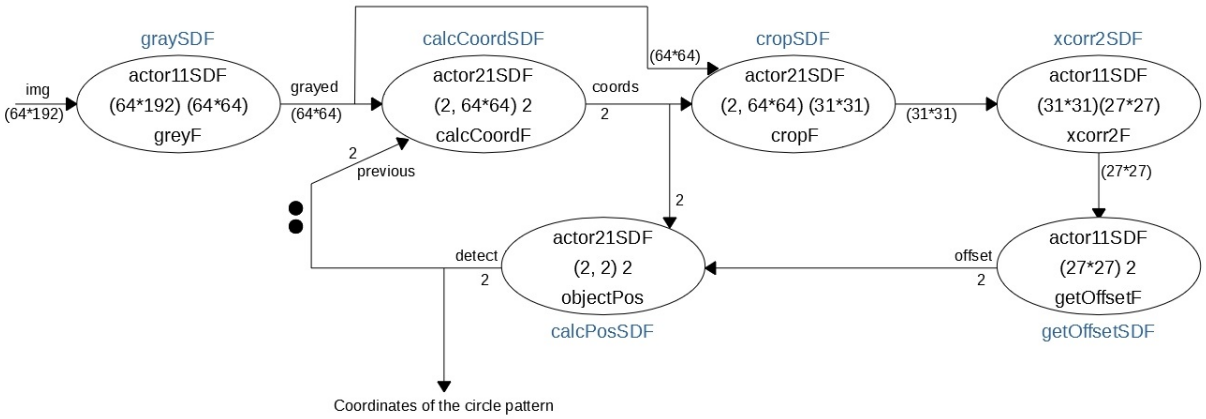

Figure 1: SDF graph of the application.

The application is divided in 6 nodes which implement particular tasks of the algorithm.

- *graySDF* takes in input the image in the rgb format and "grayes" it reducing the three bytes of the red, green and blue colours into a single byte. This operation is done by computing the expression r*0.3125 + g*0.5625 + b*0.125. Therefore the output array has a total number of bytes equal to one third of the input one.

- *calcCoordSDF* takes in input the grayed image and the coordinates of where the pattern was found in the previous image (if it is the first image of the sequence, the coordinates are equal to the first fixed position (15;15) as said in the specification, available as inital tokens). From these inputs, it performs the *calcCoord* function which computes the coordinates of the pixel from which the image cut should start. The coordinates are computed in such a way that the cut frame would still include the circle in case of any movement of 15 pixels in each directions.

- *cropSDF* takes in input the grayed image and the coordinates of the up left most pixel starting from which the crop must be performed and then crops the image into a frame of 31x31 pixels.

- *xcorr2SDF* takes in input the 31x31 matrix and, per each element of the matrix, a little matrix of 5x5 neighbouring pixels is multiplied by a fixed matrix (5x5) filled with 0 values and 1s only in the positions that allow to shape the circle pattern. Then all the elements

of the product matrix obtained are summed and a single value is computed and saved, sequentially, in a matrix 27x27.

- *getOffsetSDF* takes in input the 27x27 matrix containing all the sums of products and finds the coordinates of the maximum value (which corresponds to initial pixel of the 5x5 pattern which containes the circle).

- *calcPosSDF* takes in input the coordinates of the pixel starting from the 31x31 array was cut and the coordinates of the pixel (related to the 31x31 array) where the circle pattern was detected and add them up (coordX + offsetX, coordY + offsetY). The output is the absolute position of the top left most pixel of the circle pattern in the grayed image. Those coordinates are passed back to the second node.

4 Hardware architecture

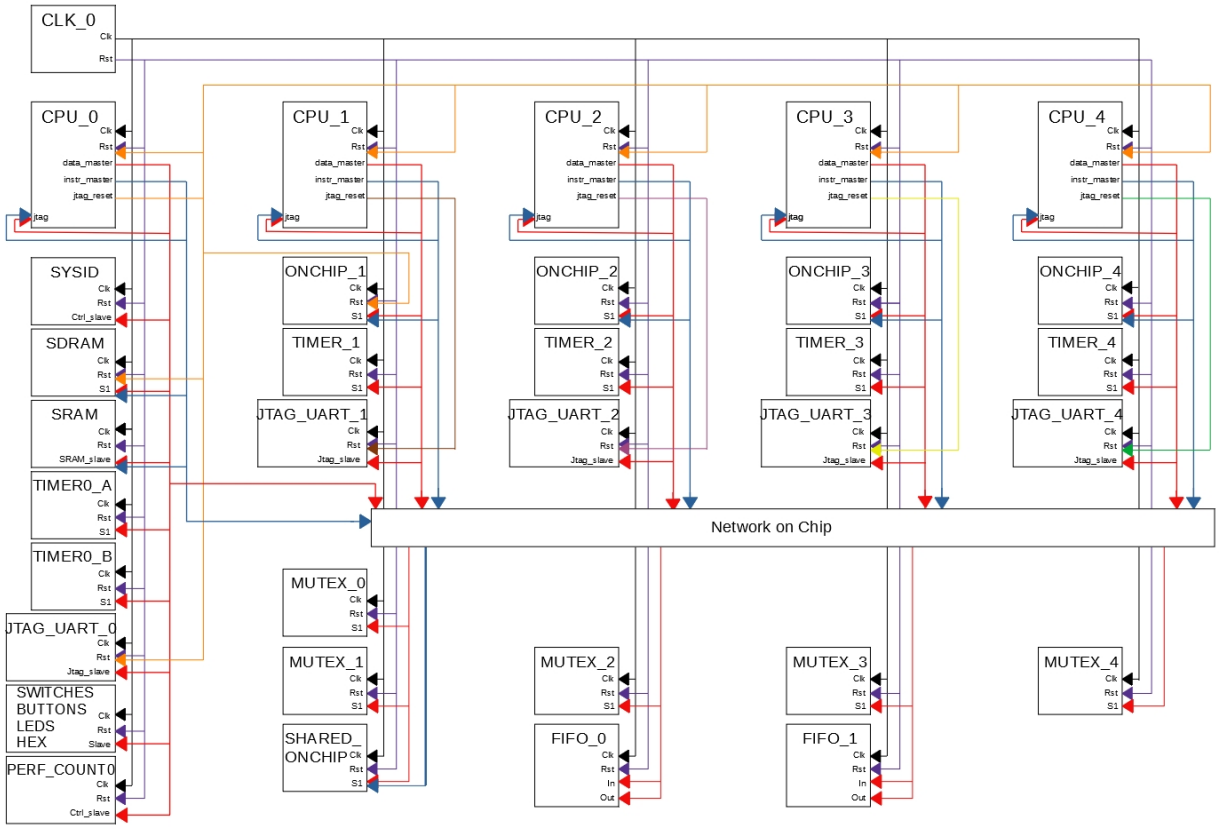

Figure 2: Hardware architecture of the Altera DE2 board.

Figure 2 shows the hardware architecture of the DE2 Altera board with the following blocks:

- **5 CPUs**, each one connected to different blocks. Cpu_0 controls the System ID Peripheral, SRAM, SDRAM, two timers, the jtag_uart block, the available peripherals (switches, buttons, leds, and hexadecimal displays), and the performance counter. The other four cores control their own onchip memory, timer and jtag_uart block. All the five cores can control the shared_onchip memory, the five mutex blocks (which manage the hardware resource sharing), and the two FIFOs.

The ALU supports the additions, subtractions, multiplications and divisions; however multiplications and divisions are performed, respectively, as repeated additions and subtractions. The floating point unit is not present.

- **Memories**: the Nios II architecture provides memory-mapped I/O access with memory alignment. Little-endian byte ordering is used. Addresses are on 32 bits, allowing access up to a 4-gigabyte address space.
  - SDRAM controller: its size is 8192;
  - SRAM controller: it can be accessed only by cpu_0 and its size is 8388608 bytes;
  - 4 on-chip memories (RAM or ROM): each one can be accessed by its own cpu and their sizes are 12288;
  - Shared Onchip memory: can be accessed by all 5 cores and its size is 12288 bytes;
- **5 mutex blocks**: the mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. All the mutex blocks can be accessed by all the cores.
- **2 FIFOs**: they can be accessed by all the cores and used for message passing among them.
- **Network on Chip**: it breaks the problem of communication between entities into smaller problems, performing the arbitration using the weighted-Round Robind policy.
- **Clock source**;
- **6 timers**;
- **Performance counter unit**;
- **System ID Peripheral**;
- **Peripherals**: 5 JTAG UART connections, switches, leds, buttons, hexadecimal displays;

5 Single core implementation with RTOS

In this first implementation, we wrote a C code application that uses the MicroC/OS-II on the DE2 board and which runs using a single processor (cpu_0) of the multi-core architecture.
The application is structured in such a way that each node of the SDF graph is implemented as a task and all the tasks are synchronised using OS kernel objects: semaphores. The main function of the application only creates a *StartTask* which, in turns, creates all the other six tasks, containing the functionality of the related node of the SDF graph, and also six kernel objects (semaphores all initialised to 0) used for synchronization.
The execution of the application is synchronised in this way: the first task, which corresponds to the *graySDF* node, starts executing immediately and, after have finished its execution, posts the first semaphore *Task1Sem* so that the second task can start executing. The the first task stops waiting on the last semaphore *Task7Sem*. In the same way, each task waits for a semaphore to be posted by the precedent task, then executes, and then posts its own semaphore to let the next task starts executing. Finally the task corresponding to the node *calcPosSDF* posts the last semaphore *Task7Sem* and the first task starts executing again working on another image.
Timers have not been used in this implementation, since only one processor could be used. Synchronising the tasks using semaphores would allow to avoid to waste time: as soon as one

task completes its execution, the following one starts immediately.
The results obtained with this implementation are the following:
Throughput = 13.47 img/s
SRAM usage = 178024 bytes (without images).

5.1 Feedback received

The suggestions received for this first implementation was about the coding style:

- usage of an equal data type whenever possible;
- usage of significative names for variable;
- description of the function performed by each task.

6 Single core bare-metal implementation

This second implementation could not exploit the RTOS to implement the task synchronisation. The only way to let the tasks execute in a correct order was to call (or resume) them in an ordered way from the main() function, since the program runs sequentially. Therefore, the *StartTask* is not needed anymore, since it's only function was to create the six tasks which performed the tracking algorithm. Now the algorithm is performed by six functions (not tasks anymore) which are called in a sequential way from the main() function.
The results obtained with this implementation are the following:
Throughput = 1.55.
SRAM usage = 74288 bytes (without images).

6.1 Feedback received

Since the C code written for this implementation was very similar to the one used in the previous one, the suggestions were exactly the same.

7 Multi-core bare-metal implementation

The algorithm described in Section 6 has been divided to be performed by the 5 available cores. The first implementation was a non-optimised one, able to reach a still very high throughput. Then some code optimization techniques have been applied to reach the final optimised implementation.

7.1 Non-optimised implementation

The algorithm described in the SDF graph shown in Section 3 has been modified in order to obtain a better implementation of the application. The functions have been divided in the following way among the 5 cores:

- CPU_0 reads the coordinates called as *previous* and computes the coordinates $cropX$ and $cropY$ from which it performs the graying and the cropping. The cropped grayed image 36x36 is saved in the Shared Onchip Memory.

- When all the image has been written in the Shared Onchip Memory, CPU_1, CPU_2, CPU_3 and CPU_4 read from the Shared Onchip Memory a different portion of image. Each core tries to find the circle pattern in its read portion of image, performing the operations described in nodes *xcorr2SDF* and *getOffsetSDF*. Then each core writes in a different location of the Shared Onchip Memory the maximum value obtained adding the product of the 5x5 matrices with the searched pattern and the coordinates in which this maximum was found.

- When all the other cores have finished their computations and written inside the Shared Onchip Memory, CPU_0 reads the four maximum values found and finds the absolute maximum among them: its related coordinates are used to perform the operation of the SDF node *calcPosSDF*.

The scheduling of the now performed algorithm is shown in Figure 3.

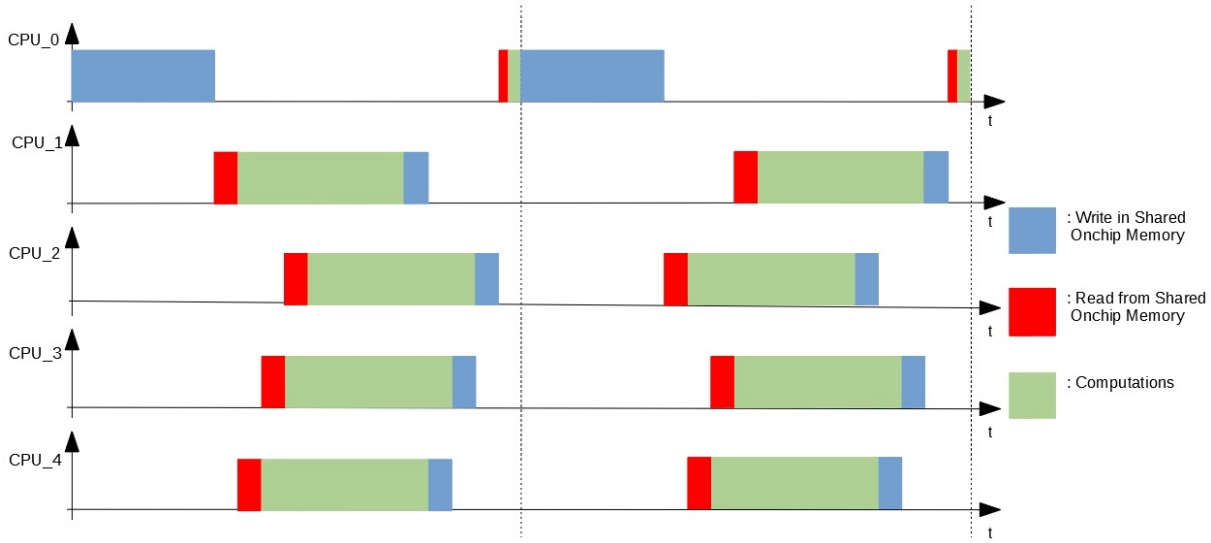

Figure 3: Schedule of the non-optimised multi-core implementation.

Synchronisation among the cores is performed using two FIFOs and mutex0 in the following way:

- CPU_0 waits that the other four cores write on FIFO_0 that they are ready to start. Then it starts writing on the Shared Onchip Memory.

- Operations made on Shared Onchip Matrix (read or write) are always preceded and followed by the lock and unlock of mutex0, so that arbitration does not need to be performed.

- When CPU_0 finishes to write on the Shared Onchip Memory the cropped grayed image, it writes four messages on FIFO_1 which are read by all the other cores.

- Each core tries to access the Shared Onchip Memory as soon as it reads the message from CPU_0 in FIFO_1. Since the operations on the Shared Onchip Memory are protected by mutex0, the first cpu which locks the mutex access the Shared Onchip Memory. Then all the others access it in the same order in which they tried to lock the mutex.

- The order in which the cores read the messages on FIFO_1 and then access the Shared Onchip Memory is random.

- When each of the four cores finishes its operation and writes its results in the Shared Onchip Memory, it writes a message in FIFO_0.
- When CPU_0 reads all the four messages from FIFO_0, it access again the Shared Onchip Memory and makes the final computations.

The results obtained with this implementation are the following:
Throughput = 5.79.
Total memory usage = 26288 bytes (without images).

7.1.1 Feedback received

The suggestions that we received were related to the mathematical computations and the synchronisation:

- do not use multiplications and divisions;
- do not waste available bus space reading or writing only unsigned chars, since the bus width is 32 bits;
- synchronise CPU_1, CPU_2, CPU_3 and CPU_4 using mutexes and allow to start them as soon as possible.
- do not perform the graying function in CPU_0, but on the other cores. The less the operations performed on CPU_0, the fastest the execution time.

7.2 Optimised implementation

The SDF graph of this final implementation is shown below in Figure 4.

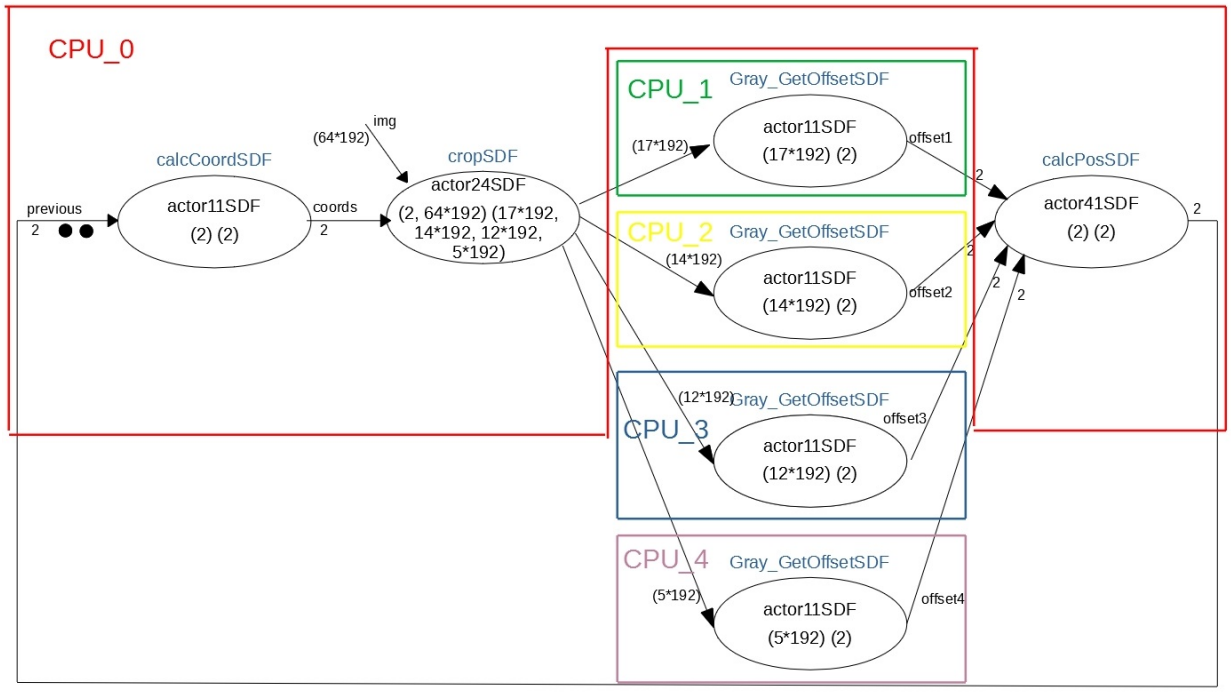

Figure 4: SDF of the final multi-core implementation.

The decided scheduling and mapping are shown in Figure 5 below.

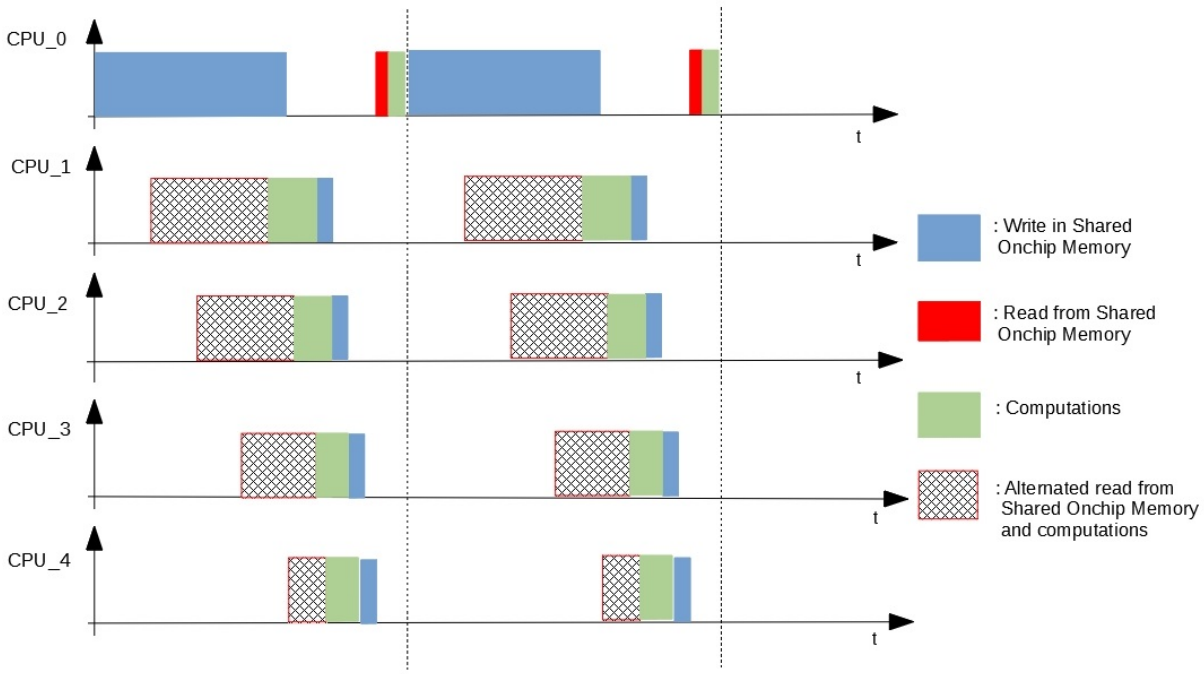

Figure 5: Schedule of the optimised multi-core implementation.

The optimised implementation has the following changes with respect to the previous one:

- mutex1, mutex2, mutex3 and mutex4 (instead of messages on FIFO_1) are used to let the other four cores start executing. Mutexes allows to give a precise order of execution of the other four cpus.
- The main idea was to give to CPU_ few operations as much as possible since it works with an external memory (SRAM), then its operations are slower with respect other CPUs. Then, CPU_0 just copies in the Shared Onchip Memory a portion of rgb image. In order to take advantage from the bus size of CPUs, that is 32bits, CPU_0 reads integer numbers from SRAM and copies it directly in the Shared Onchip Memory. Moreover, it copies all the rows the belongs to the image that we want to crop, so that the rgb image results cropped only in an horizontal way. This is done since the memories allow only an aligned access.
- CPU_1, CPU_2, CPU_3 and CPU_4 start performing their tasks whenever the portion of image on which they should work has been written in the Shared Onchip Memory by CPU_0. Each CPU reads only its own portion of the rgb cropped image, performs the graying and tries to find the pattern; if this operation is sucessfull, the offset coordinates are written in the Shared Onchip Memory. The sizes of the portions of images assigned to the cpus are the following:
  - 17 rows in CPU_1;
  - 14 rows in CPU_2;
  - 12 rows in CPU_3;
  - 5 rows in CPU_4.

The sizes are decreasing values: by assigning few values to the last cpu, CPU_0 should wait only for a little time before reading the offset and computing the absolute coordinates. Less time is wasted if more data are assigned to cores which can start working earlier. Arbitration is allowed in this case, since the CPU_0 is able to write in the Shared Onchip Memory while the others CPUs are performing the graying.

- The pattern search is simplified, assuming that we are looking for a white circle: after the "greying" we expect to find pixels of a value similar to 255 where the circle pattern is present.
- Multiplications and divisions are never performed; shifts are performed instead.

The results obtained with this implementation are the following:
Throughput = 204 img/s.
Total memory usage = 31891 bytes (without images).

8 Results

Table 1 below contains a summary of the measured performances of the single core implementation. Then Table 2 and 3 show the measured performances of the multicore implementations (respectively non-optimised and optimised).

	Single Core (RTOS)	**Single Core (bare metal)**
Throughput[s^{-1}]	1.35	1.55
SRAM [bytes]	178024	74288

Table 1: Performances of the two implementation on single core

Throughput [s^{-1}]	**SRAM [bytes]**	**Onchip CPU1 [bytes]**	**Onchip CPU2 [bytes]**	**Onchip CPU3 [bytes]**	**Onchip CPU4 [bytes]**	**Onchip shared [bytes]**	**Total memory [bytes]**
5.79	14248	2680	2680	2680	2680	1320	26288

Table 2: Performances of the non-optimised implementation on multi-core

Throughput [s^{-1}]	**SRAM [bytes]**	**Onchip CPU1 [bytes]**	**Onchip CPU2 [bytes]**	**Onchip CPU3 [bytes]**	**Onchip CPU4 [bytes]**	**Onchip shared [bytes]**	**Total memory [bytes]**
204	14220	2696	2700	2700	2656	6919	31891

Table 3: Performances of the optimised implementation on multi-core

9 Further improvements

From Table 1 it is possible to notice that in two the single core implementations the bare-metal implementation reached a slightly higher throughput since no time is wasted in performing system calls of the OS. Regarding the multi-core implementations, a large increase in the throughput has been reached performing the optimisation: from 5.79 img/s to 204 img/s. The aimed throughput was 350 img/s, less than twice of the one we obtained.
Further improvements can be performed on our application in order to reach the aimed throughput:

- First of all, CPU_0 can copy in the Shared Onchip Memory only the fully cropped image, playing with the integer pointers in order to reduce the access to memories. With our implementation we are copying 64x36x3 rgb pixels (the entire stripe of image in which the cropped is included), performing (64x36x3)/4 integer readings and the same number of integer writings. Introducing this improvement only (36x36x3)/4 integer readings and writings should be necessary.. As a matter of fact, these operations are performed with the CPU_0, that has all its own variables and the images in the SRAM memory and each operation on this memory is very slow.
  To coumpute how much the further improvement would have been, we need to perform some analysis on the step performed. In the first moment, we copied the cropped rgb image in the shared onchip memory reading in unsigned char: 36x3x36 = 3888 readings were performed. The throughput that we had at that time was about 120img/s. Then we read in integer the entire stripe of the cropped image: (64x3x36)/4 = 1728, performing 2160 readings less. Just doing this, we reached a throughput around 190 img/s. If we would have copied only the (36x3x36)/4 = 972 integers of the cropped image, taking into account the previous improvement, we would have reached a throughout of around 230img/s.

- Another improvement can be to avoid the usage of FIFO. The access to the FIFO of course slows down the system: the greater the number of resources used, the greater the time required by the system. A better and hopefully faster solution can be to use shared variables in the Shared Onchip Memory to synchronize the CPUs, since the access to it is faster.
  In order to understand how much the access to a FIFO slows down the system, it can at least be compared to the access to a Mutex (which is also a far resource). In a previous implementation, we used mutexes to determine which cores should access the Shared Onchip Memory: the throughput at that time was around 60img/s. Then we tried to eliminate the usage of mutexes, letting the arbitration to manage their accesses. This simple change brought us to a throughput of around 120img/s. Taking into account that CPU_1, CPU_2, CPU_3, CPU_4 made two accesses each per image and that the FIFO is accessed 5 times per image, the throughput would have increased by around 38, leading to around 270img/s. Furthermore, we need to consider that the access to a FIFO (which contains a message) is for sure more time consuming than the access to a Mutex which carries a 1 bit value. For this reason we have reason to think that the throughput increase would have been even higher, probably leading to a value around or higher than 300img/s.

- Last important improvement can be the usage of variables that are integers. In our implementation we worked mostly with unsigned char variables to save memory space. Since all processors have 32-bit bus, it is more natural and faster for a processor to manage data that are integers: C compiler may be able to optimize the code much better for 32-bit variables than 8-bit ones. However this feature has not been practically measured in any way; it would have been probably helpful to read the obtained assembly code.

About the memory constraint, it was fully achieved in both multi-core since we worked directly with the cropped images, then no improvements are necessary in this field.

A Appendix

A.1 Allocation of responsibilities

The two group members worked together in almost all the steps of the application developement, in all the four implementations. However, each participant played a more important role in different phases of the project.
Starting from the model of the application written in the ForSyDe-Haskell language, Giulia obtained the corresponding SDF graph. Moreover, she wrote the algorithm used to split the image through 5 cpus.
Antonia worked more on the synchronization between cpus, then on low-level of the scheduling problem.
Anyway, we worked together for all the other phases of the project,