IL2212 EMBEDDED SOFTWARE

Technical Report
Multicore Digital Signal Processing Application

JUN ZHANG JAAKKO LAIHO
junzha | jaakkol @kth.se

March 5, 2019

Contents

1 Introduction

1.1 Scope

The project is the laboratory assignment of the course IL2212 Embedded Software[1] at KTH, Sweden in January-February of 2019. The workload of the laboratory part of the course is reported to be 3 ECTS, which equates to 80 hours of work.[2]

1.2 Purpose

The purpose of this project is to enable our learning in terms of the established learning goals of the course. That goal is expressed followingly: ”Construction flow for software for embedded systems: System modeling, systems analysis and system synthesis.”[1]

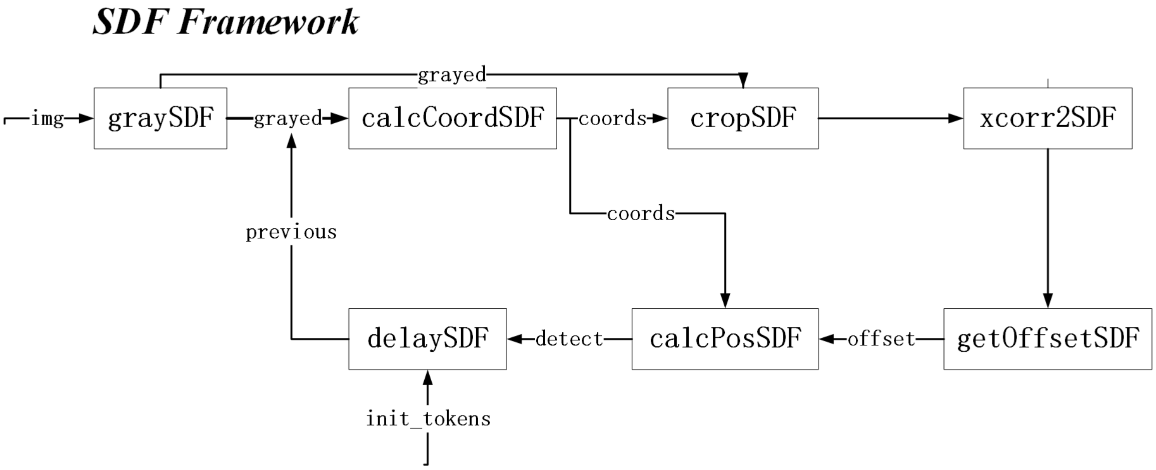
1.3 Application

This project is a practice in implementing a digital signal processing application on a multi-core hardware platform. More specifically, the task of the project is ”to implement an image tracking algorithm which tracks a (given) moving ’circle’ pattern in a series of image frames for the purpose of further processing.” [3] The application is implemented in three ways on the custom multi-processor hardware platform hosted on a DE2 FPGA: On a single core using the MicroC/OS-II, on a single core without OS, and on multiple cores without OS. The following assumptions are considered valid: [3]

- In any given input frame there can be at most one ’circle’ object.
- Between any two consecutive frames the ’circle’ object can move at most 15 pixels in any direction.
- There is an acceptable detection error of +/- 2 pixels.
- The first frame in a set of images is known to contain the object in the vicinity of a fixed position.

2 Application Model

Figure 1: SDF Framework.



2.1 Algorithm Descriptions

2.1.1 GraySDF

Grayscaling transforms the input RGB matrix to a grayscale matrix by performs so-called convolution, where ”the input samples are combined using a weighted sum with specific fixed weights associated with each offset input.” [4] It uses a Map[1]pattern to compute the output, where for each input sample and the next 2 input samples the output sample calculated as a weighted average. For a matrix input the processing time is $O(n^2)$, where n is the dimension of the image.

2.1.2 CropSDF

Cropping cuts the image to a smaller image of the given size. Its processing time is also $O(n^2)$, however in this case n denotes the dimension of the cropped image.

2.1.3 Xcorr2SDF

The function *xcorr2* reduces[2]the cropped and grayed image by adding together the greyscale values of a series of stencils[3], where the stencils are ”sub-images” based on the size of the pattern being searched. The summary value is then paired with the position of that value. The execution time depends on the dimensions of the cropped image and the size of the pattern to be detected, and its processing time is approximately $O(n^2)$.

[1] **Map**: ”In the map pattern, an elemental function is applied to all elements of a collection, producing a new collection with the same shape as the input. In serial execution, a map is often implemented as a loop (with each instance of the loop body consisting of one invocation of the elemental function), but because there are no dependencies between loop iterations all instances can execute at the same time, given enough processors”[4]

[2] **Reduce**: ”In the reduce pattern, a combiner function is used to combine all the elements of a collection pairwise and create a summary value. It is assumed that pairs of elements can be combined in different orders, so multiple implementations are possible.”[4]

[3] **Stencil**: ”A stencil is a map in which each output depends on a ’neighborhood’ of inputs specified using a set of fixed offsets relative to the output position.”[4]

2.1.4 GetOffsetSDF

The actor *getOffsetSDF* gets the position of the image with the greatest grayscale sum by reducing the paired datapoints with a comparer function, which returns the pair with the greater grayscale value. It has an execution time of $O(n)$.

2.1.5 CalcCoordSDF

CalcCoord calculates the coordinates of the next cropping point according to the previous known object position, unless it is the first time the function is being called, in which case it returns a default value based on the assumed known fixed point. Its execution time is $O(1)$.

2.1.6 CalcPosSDF

CalcPos calculates the position of the detected image by adding together the cropping point as well as the found offset and executes in $O(1)$.

2.2 Speculation

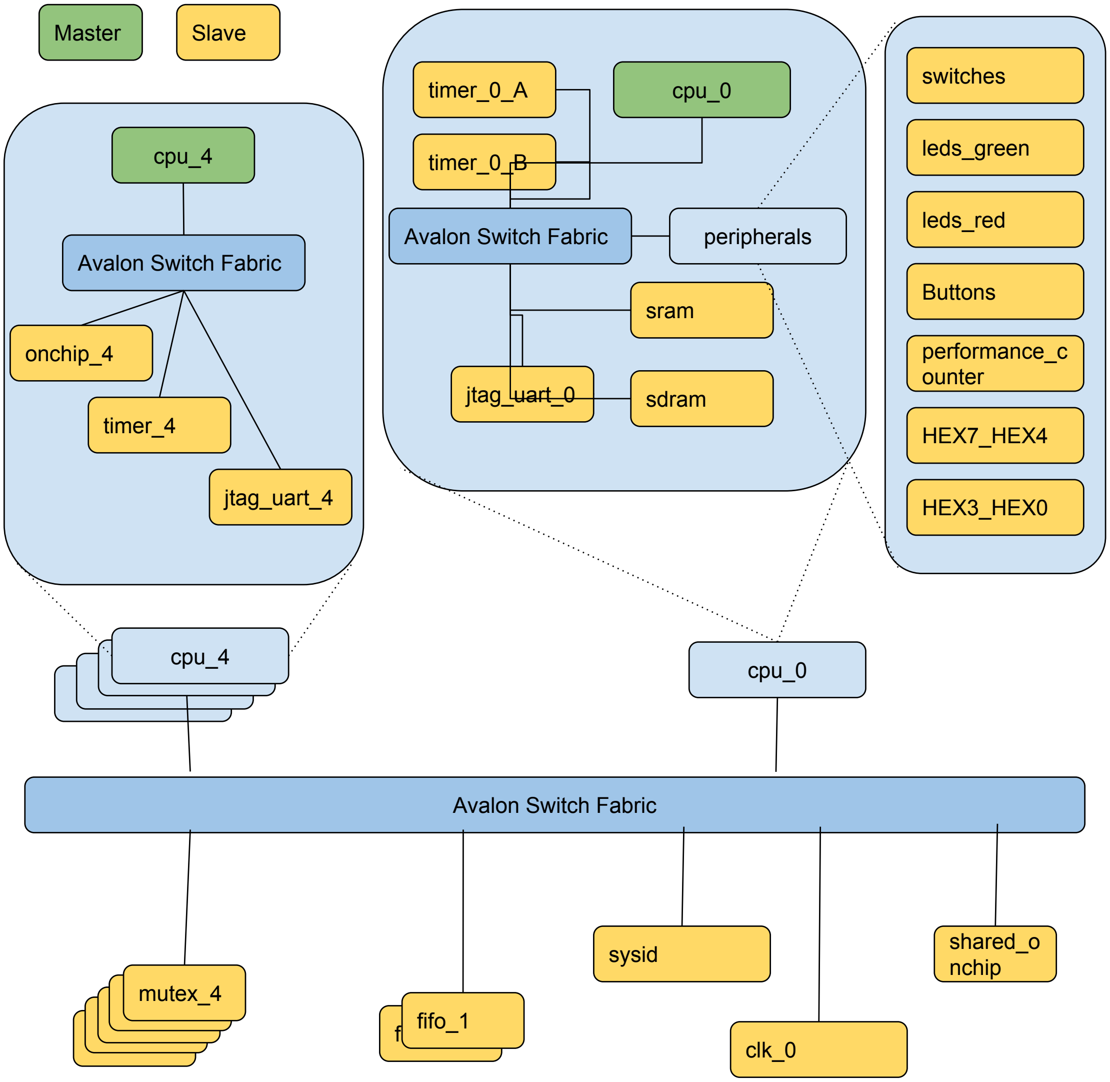
Mapping and reducing apply a function to all elements in a collection, where the ordering of the execution of the functions do not need to necessarily be sequential. In other words, there is reason to believe that significant performance increase can be achieved by parallelising the execution of grayscaling and xcorr2 onto multiple processors. While getting offset also makes use of the reduction pattern, its execution time is expected to be rather short, and thus parallelising its execution is not expected to yield great performance benefits.

3 Hardware Multi-processor Platform Specification

3.1 Platform Architecture graph

The hardware platform features multiple cores and peripherals a network-on-chip architecture that communicate via the Avalon Switch Fabric. There are 5 Nios II Processor cores in the system, and a wide selection of peripherals. Most peripherals are connected to the data master bus of processor core CPU-0. These peripherals include the following: switches, LEDs (both green and red), buttons, a hex display, and notably a performance counter. CPU-0 also has 2 timers, an sram and 8MB sdram and jtag uart component connected to its data bus, while the other 4 CPUs (cpu-1 to cpu-4) each have only an 8kB onchip memory, one timer, and a jtag uart component connected to their data master. Furthermore, the system also features 5 hardware mutexes, 2 on-chip fifo memories, 1 shared 8kB on-chip RAM memory, all of which connect to the data master bus of each CPU.

Figure 2: Network-on-Chip Architecture.[5]



3.2 Interconnection Network

Each component in the system communicates via encapsulated packet transactions, where the network-on-chip (NoC) interconnect transports the transactions between nodes in the system. The NoC treats the interconnect as a protocol stack where each layer implements the different functions of the interconnect. Each master or slave node is connected to the network through a network interface component. This interface receives the transaction or response and sends it to the network as a properly formatted packet. The packet is then delivered to the appropriate endpoint by the packet network for that packet, where it is passed to other network interfaces. The packet is then terminated by the network interfaces which also sends the command or response using the transaction layer protocol to the recipient.[6]

In other words, communication between the components happens through using the transaction interface of the Avalon Memory-Mapped interface in this case. Simply put, the network interfaces are the ones that communicate with each other, which in turn rely on the command and response networks provided by the network. This in turn shapes the question the software developer must answer into the following: ”How do I best map transactions to packets?” With the transaction layers being separated, the developer may optimize each layer independently without needing to consider the other layers.

4 Single Core+RTOS, Single Core+Bare-metal

4.1 Description

The application modeled as an SDF graph has a good mapping to the target architecture. this is because an actor in an SDF graph can be translated into a task (in case of a RTOS) or a function (in case of no RTOS). The signals in the SDF graph describe data dependencies of the actors - in an RTOS implementation using OS kernel objects such as message queues (mailboxes / semaphores in case each signal only has at maximum one token) will enable synchronisation of the tasks. Alternatively, in the bare metal implementation the beauty of a SDF graph shines through - a static schedule can be computed, in which case the main loop of the program will simply call the actors mapped as functions according to that schedule.

Table 1: Single Core Implementations.

	Single Core (RTOS)	Single Core (Bare Metal)
Throughput [s-1]	2.41	1.26
SRAM [bytes]	220000	145000

4.2 Schedule Graphs

Figure 3: Single-core RTOS schedule

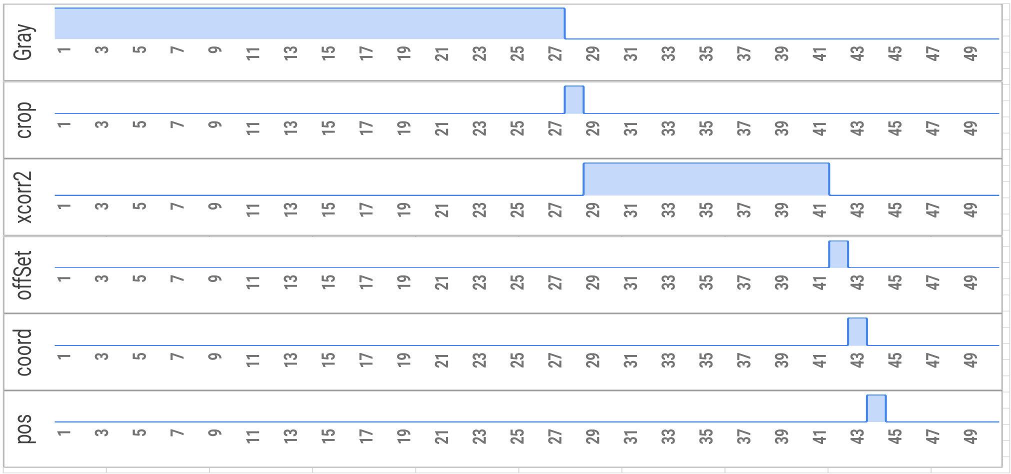
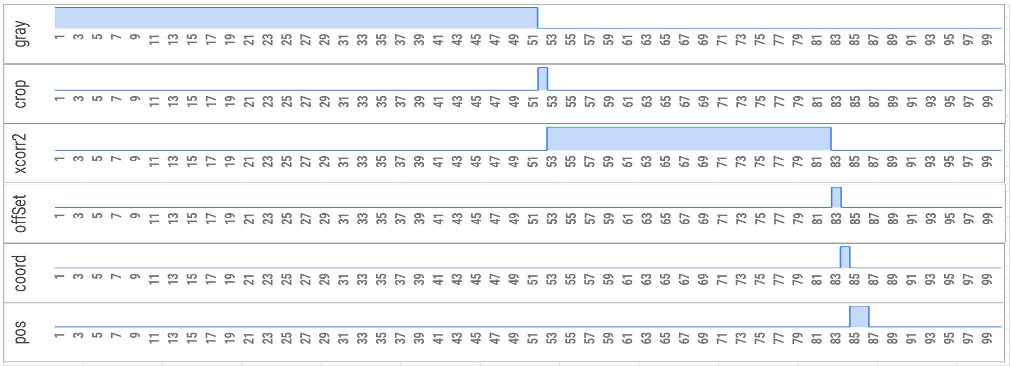


Figure 4: Single-core bare schedule



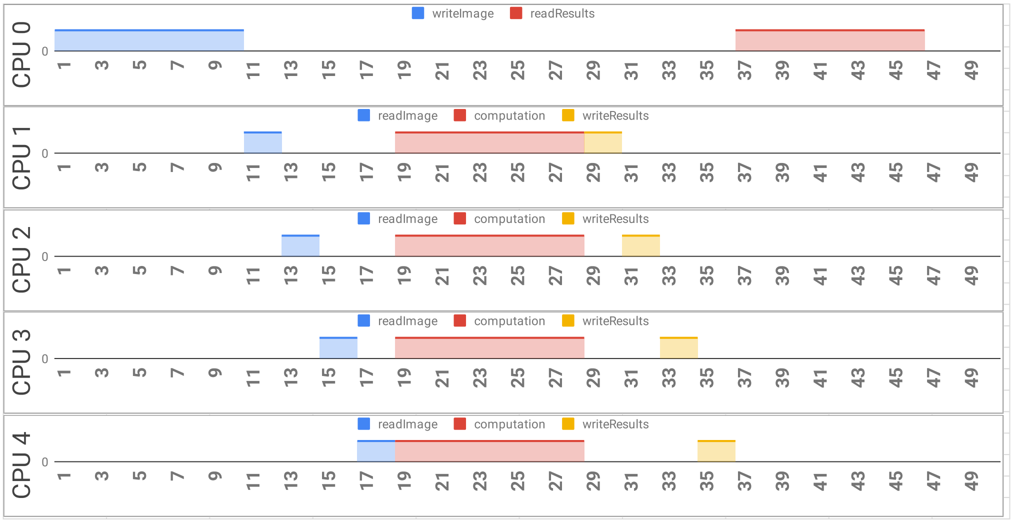
5 Initial Multi-core Implementation

There is a tradeoff for the system in adding more CPUs with the aim to increase computation power; increasing computation power by adding more CPUs also increases communication time. In other words there is a point at which the performance increase gained by the further distribution of computation is fully offset by the performance decrease caused by the additional communication cost. In order to determine how extensively computing can be offloaded from CPU 0 to CPUs 1 till 4 depends on our measurements of computation time as well as communication time.

We have discussed multiple ways to implement the application on a multicore architecture. One discussion was between delegating computation of both grayscaling and xcorr2'ing or only xcorr2'ing to CPUs 1 till 4. Another discussion was if it made sense to share these tasks between all 5 CPU's instead of 4 - after all, CPU0 is otherwise idle while CPUs 1 till 4 execute their tasks. Another discussion included using complex scheduling to attempt to produce optimal performance - however, we suspected it to become difficult to maintain.

5.1 Multicore schedule

Figure 5: Rough Multi-core Schedule



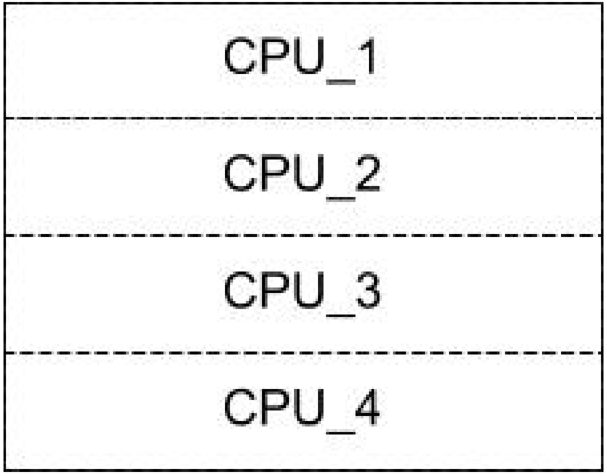
5.2 Implementation

Our design is simple - high performance with maintainability. Our multicore implementation of the application makes full use of the platform architecture. Where applicable, computing can be delegated to multiple CPU's simultaneously. As speculated earlier, the execution of functions *gray* and *xcorr2* can be distributed to multiple CPU's. Since they have long execution time, it makes sense for optimization purposes to distribute their computation. We have decided to split the image into 4 smaller sub-images by ”slicing” the image into 4 horizontal strips.

Furthermore, we have chosen to crop the image before grayscaling (as well as the writing of the cropped image to memory, naturally), since doing so reduces communication time and significantly increases computation speed of both grayscaling and xcorr2'ing, by reducing the dimensions of their input. While this does not follow the semantics of the initial model, the functionality of the application remains the same. In fact, such a model actually allows for tighter design constraints (less memory needed, higher throughput) which justifies the modification. CPUs 1 till 4 each read one segment of the cropped image, and individually perform grayscaling and xcorr2'ing on their own segment, and also compute the offset and position concurrently. Thereafter CPUs 1 till 4 write their found values to shared memory. CPU 0 then reads the 4 values and selects the actual correct result.

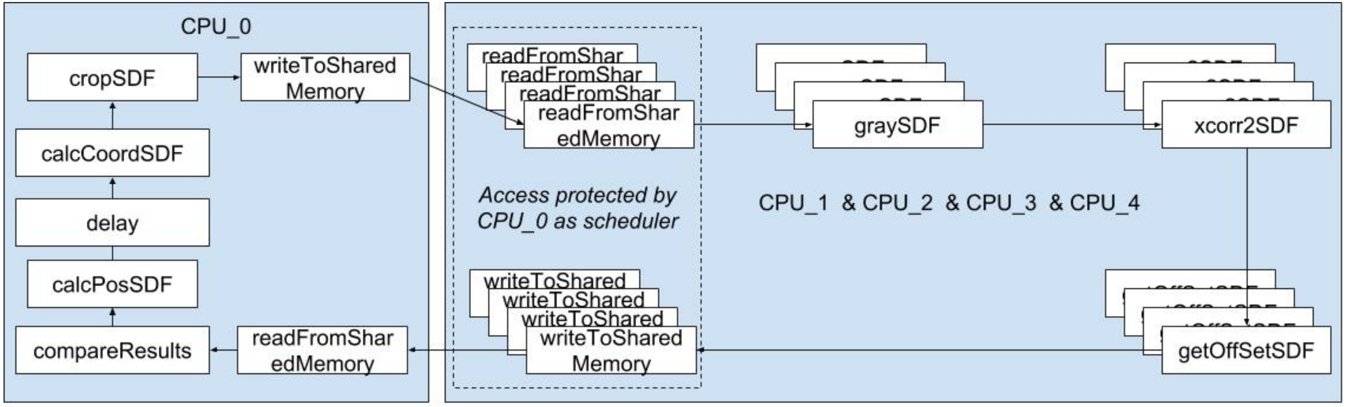
This distribution of computation, however, increases complexity by requiring synchronisation activities. Without having a RTOS at our disposal, we will rely on the hardware components provided by the platform: We will make use of the mutexes as though they were semaphores. In our implementation, CPU-0 is given

Figure 6: The cropped image to be processed is sliced into 4



the ability to 'suspend' the other CPUs by locking the mutex that the other CPUs try to lock before accessing shared memory. This way the behaviour of CPU-0 resembles that of a scheduler, which is able to ensure that only 1 CPU is accessing the shared memory at any given time. The SDF graph is updated to better reflect the process taking into account reading and writing to shared memory.

Figure 7: Multi-core SDF



6 Optimized Multi-core Implementation

6.1 Burst Reading

When CPU0 copies the image from SRAM to shared-onchip memory or other CPUs read the images from the shared-onchip memory, we implement burst reading to speed up the process, i.e using the int pointer instead of the unsigned char pointer to transmit the data. In such way, we can manage to transform the cropped image in around 600 cycles. The drawback of this approach is the alignment issue of the int datatype. When the computer read a value from an int pointer, it must start from a memory address which is a multiple of 4. If it is not, then it will start reading value from the last address (smaller) which is the multiple of 4. For example, if you try to use int pointer to read from address 15, then you actually read value from 12, which means you actually read 3 more extra value. Since we do the cropping of the original image first the read the image, we read some 3 more values when processing the third image. We get the detected coordinate which is one pixel shift right to the correct location. But it still fulfills the precision constraint. One way to improve this issue is adding one zero to the test images. This way can solve the problem of first image but not others.

6.2 Loop Unrolling

We do loop unrolling manually for grayscale function, croop function and xcroop function, the most computing-demanding functions. There is a trade off between footprint and speed here. If you unroll the whole functions, the program will take more space but consume less time. By taking this into consideration, we unroll the functions partially. But the measurements shows us the the speedup gain is good and the space cost is not that high.

6.3 Function Lining

In the optimized implementation, we take almost all the actors/functions into one big main function for every CPU. We don't think it is a good idea for actual development in the industry but it seems necessary in order to make it reach that throughput requirement.

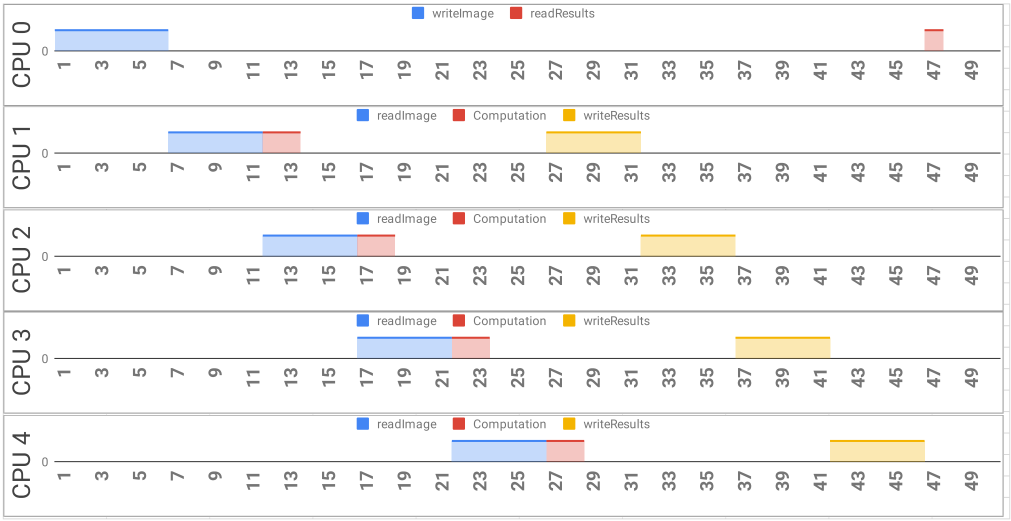
6.4 Others

Besides the above optimization mechanisms, we also use other methods. For example, we rewrite the grayscale algorithm and use shifting including instead of the multiplication. We adjust the delay time in the assemble code. We creat more intermediate variable to store some constants we use in the loops.

Table 2: Final Multi-Core Implementation.

Throughput [s-1]	SRAM [bytes]	OnChip CPU 1 [bytes]	OnChip CPU 2 [bytes]	OnChip CPU 3 [bytes]	OnChip CPU 4 [bytes]	OnChip Shared [bytes]	Total Memory [bytes]
201	12096	4344	4388	4380	4344	3888	33440

Figure 8: Optimised Multi-core Schedule in units of 100 microseconds



7 Further Improvements

Reaching the required 350 images per second requires our current performance of 201 images per second to be increased by about 75%, which means that the total throughput needs to be reduced from the current just under 5 milliseconds per image to about 2.86 milliseconds per image, resulting in a need to reduce image processing execution time by 2.14 milliseconds.

Our current implementation makes use of delays for two purposes: 1. To synchronize communication, and 2. to prevent multiple CPUs from accessing shared memory at the same time. Upon further inspection we discovered that performing purpose 2 is not at all necessary on the multi-core architecture because of how communication takes place: The Altera switch fabric performs weighted round-robin slave-side arbitration to ensure that the amount of access each core has to the peripherals is equally distributed (according to weights). In other words, it is not necessary to prevent multiple CPUs from accessing shared memory at the same time - this already happens on a lower level done by the arbiter.

This information results in the insight that the relatively time-consuming exclusive memory access mechanism is unnecessary, and can be removed. When cores 1 till 4 have finished computing their results, for each core to write the result to the shared memory takes in total 1.9 milliseconds. The result itself is only some bytes large, and takes very little time to write - it is the handshaking mechanism which fills the majority of the 1.9 milliseconds. In other words, repurposing the mutexes to be used so as to signal that a result is written (rather than passing control to the next core) could theoretically increase throughput by as much as 61%.

One possible optimization that can be performed is that grayscaling can be performed in the same loop as reading from memory by CPUs 1 till 4. A further optimization is to set some flags while grayscaling. For example, in this case, we know that the pattern has bigger gray value than the background does. Then we can choose 200 as a threshold (because we decrease the grayscale precision in order to increase the throughput). Whenever we get a gray value bigger than 200, we sets the flags to the column indexes of those pixels. Later we can use these flags to decrease the time complexity of xcorr function, i.e. we only perform the xcorr function when we know the column has at lease one pixel with a large gray value. In this way, we can avoid a number of unnecessary computation. Particularly, when processing the first image, CPU1 and CPU4 can almost avoid the xcorr function completely.

Another optimization that can be performed is that reading of the image by CPUs 1 till 4 can begin as soon as 1/4th of an image is written by CPU 0. This is because it takes more time for CPU0 to read from sram than to write to the shared onchip memory. So there will be some stalks when CPU0 transmits the image to the shared onchip memory, which can be utilized to schedule other CPUs to read the value. By doing this optimization, it is expected to increase the throughput by at most 9%.

8 Appendix

Table 3: Individual Contribution

Task	Jun	Jaakko
Haskell-to-SDF	X	(x)
Document process functions	X	(x)
Document Hardware architecture	(x)	X
Actor implementations as functions	X	
Singlecore synchronisation framework		X
Document single-core performance		X
Multicore design discussion	X	X
Multicore synchronisation framework	(x)	X
Function optimization	X	(x)

References

[1] I. Sander, "Embedded software." https://www.kth.se/student/kurser/kurs/IL2212?l=en. Visited on 02/22/2019.

[2] studentdata@kth.se, "Student at kth." https://www.kth.se/en/student/kurs/studieresultat-1.373671. Last changed: Jun 21, 2018.

[3] G. Ungureanu, "Laboratory." https://kth.instructure.com/courses/7693/pages/laboratory-information-about-the-project. Visited on 02/22/2019.

[4] A. D. R. Michael McCool and J. Reinders., *Structured parallel programming patterns for efficient computation*. Amsterdam ; Boston, Mass.: Morgan Kaufmann, 2012.

[5] J. Laiho, "Network-on-chip architecture graph." https://docs.google.com/drawings/d/1F3l7kJZKiIVBYh8ljDyU77SW0-kB9Sm1T0EeRqUxI50/edit?usp=sharing. Generated on 02/25/2019.

[6] K. Orthner, "Applying the benefits of network on a chip architecture to fpga system design." https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01149-noc-qsys.pdf. Visited on 02/22/2019.