Il2212 Embedded Software Project

Tracking moving pattern in series of image frames

Group 15

Authors:

Mohaddeseh Basiri

Hamza Kadhum

Table of Contents

1. Objectives

The objectives of this lab project are to implement a concurrent data-flow application in different ways on a multiprocessor that is implemented on an FPGA. The knowledge gained from the lectures is applied during the project in order to derive an efficient implementation exploiting the parallelism provided by the architecture.

1.1. Goal of the project

The purpose of the project is to implement an image tracking algorithm which tracks a moving pattern, in this case a circle pattern, in a series of image frames for the purpose of further processing. The application is then implemented on a multi-processor hardware platform hosted on a DE2 FPGA, in three ways:

- On a single core, using the MicoC/OS-II
- On a single core, without OS
- On multiple cores, without OS, satisfying the design constraints

The design constraints are that the application should be able to process at least 350 images/second and the total code footprint should not pass 40Kbytes.

2. Application model

The application model has been given in Haskell language which describes functions and synchronous dataflow. The application model consists of 7 different functions. First function takes an image of 64 x 192 matrix and convert it to a gray image of 64 x 64 matrix. Proceeding to the next function cropping the gray image to 31x31 matrix in neighboring elements of the moving pattern, we are tracking, making it easy to detect. In the third function we calculate the neighboring elements of the moving pattern such that multiply it with the pattern we are looking for – after that the result will be a matrix of 27x27. So far, we have minimized the matrix to a size that allow us easily to locate the moving pattern. Moving on we gather the index for the maximum value, which is the center of the moving pattern, giving us a hint of neighboring elements. Before the feedback we calculate the position of the object with respect to the index of the maximum value. Through delay function we are able to introduce feedback in the system such that we can use previous pattern location as reference for current run. The Synchronous Data flow graph is shown below.

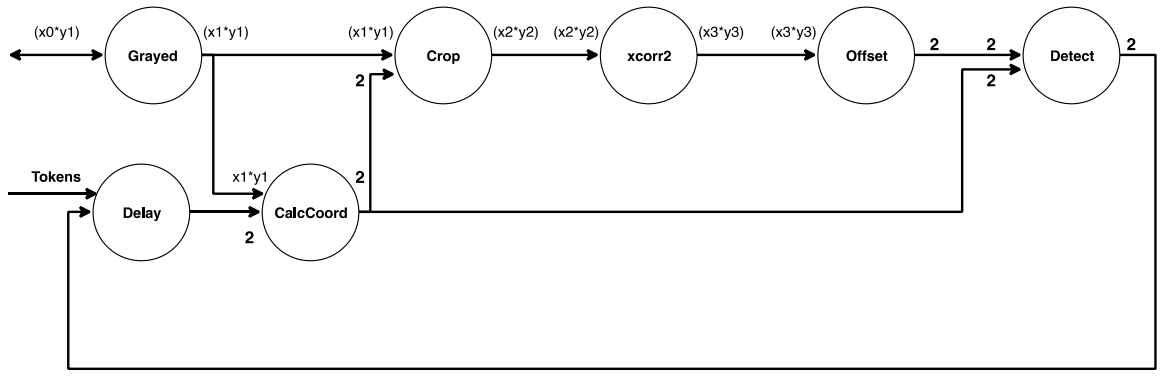

Figure 1. Synchronous Data flow graph of Tracker code.

Figure 1 represents the model in term of circles as actors and inputs and outputs as tokens. Tokens might represent a sequence of samples or signal. In our case it represents the matrix and its size as input and is passed on throughout the flow. The reason behind why we are using delay in the feedback is to avoid deadlock that can occur in the system. Observe that we initialize the delay with tokens.

3. Application Functions

This section will only consider the key functions of the model as follows:

3.1. Grayscale Function

The input of this function is an integer matrix and output matrix is floating point numbers with double precision. The function takes the input image in size of NxN and makes it to a gray image. In order to make it gray the values of the matrix are multiplied with RGB constants that results in gray. For every three elements in the rows of the matrix it multiplies them with RGB constants and adding them together. The sum of respective three elements is added in the corresponding column of a new matrix, called gray matrix. See following example.

RGB constants: R = 0.3125, G = 0.5625, B = 0.125

$$\text{Input (3x3) image} = \begin{matrix} 1 & 2 & 3 \\ 4 & 4 & 2 \\ 3 & 3 & 4 \end{matrix} \rightarrow \text{grayscale function} \rightarrow \text{Output gray image} = \begin{matrix} 1.812 \\ 3.75 \\ 3 \end{matrix} \quad (1)$$

3.2. Crop Function

Crop function is as named, a cropping function that reduces the size of the matrix on the input. The function inputs are integer and matrix double. The integer inputs are the index position of the maximum value in the matrix. The function uses the index position as starting position for cropping the matrix. The input matrix for this function is the matrix after the grayscale (see figure 1). The Matrix size seen at the output of the crop function is a 31x31 size Matrix. This function is important as it crops the size of the matrix to where we easily can locate the moving pattern. Additionally, in the feedback it will use previous position index as a reference to where it should start cropping.

3.3. xCorr2 Function

This function allows us to locate the pattern we are looking for. The function has two inputs, Matrix double and Matrix double and are defined in four different steps which are stencils, products and timesP.

Stencils is used to specify the position and neighboring elements of the moving pattern. It is defined as following

Stencils = StencilMat PatW PatH img (2)

The pattern we are looking for is stored in a pattern matrix that is predefined. The PatW respective PatH in (2) are the column respective row size of the pattern matrix. StencilMat function will return neighboring elements for each possible element with 5 vectors of size 5 elements.

TimesP mat = zipWithMat (*) pat mat (3)

This function applies multiplication pair-wise on each element on a matrix, multiplies the pattern matrix and input matrix (mat).

Products = MapMat timesP Stencils (4)

This function applies multiplication in pair-wise manner (timesP) on every possible value of the stencils.

The finish touch of the function is a function that applies addition on every possible value of products resulting in minimized size of the matrix. After this function the matrix size is reduced further which respect to the moving pattern we are looking for, the resulting matrix is 27x27 in size.

4. Interconnection Network

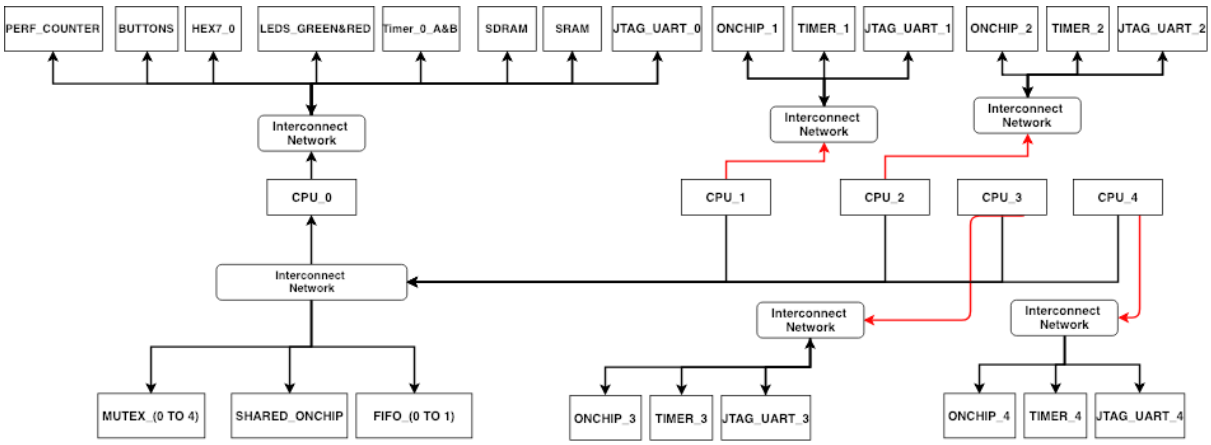

Figure 2. Hardware architecture connections

The Interconnection network is a network-on-chip topology [1]. It contains of command and response network. When a component wants to access other component, it sends a signal to command network in form of transaction layer protocol and deliver the response using the transaction layer protocol.

The benefits of NoC Architecture:

- Independent implementation and optimization layers
- Simplified customization per application
- Supports multiple topologies and options for different parts of the network
- Simplified feature development, interface interoperability, and scalability

5. Measurements

This section includes bare-metal, RTOS and multicore measurements that were done during the project.

5.1. Bare-metal implementation

For bare-metal implementation, we first measured the total execution time for the four input image sequences. We did the execution order as the synchronous data flow graph. Performance counter was used to measure the total execution time and the execution time for each function (actor as function). The results are shown in table 1 and 2, respectively:

Table 1. Measurement values

	Single Core (RTOS)	**Single Core (Bare Metal)**
Total execution time(s)	31.42712	26.296999
Throughput $[s^{-1}]$	1.2727	0.1521
SRAM [Bytes]	310k	186.6k

The code is not optimized yet resulting in sub-optimal throughput. The aim will be to optimize the code and look over how the architecture is connected such that we can improve the performance.

Table 2. Execution time for bare-metal functions.

Function	**Execution Time (sec)**
Get image function	0.17835
Grayscale function	5.48023
Crop function	0.01306
xCorr2 function	0.88919
Offset (posMax) Function	0.01120
Calcpos Function (Detect)	0.00001
CalcCoord Function	0.00001
Delay Function	0.00001

5.2. RTOS implementation

For measurement of execution time for RTOS tasks we used the same technique used to measure execution time for bare-metal. The total execution time for RTOS is revised in table 1. The following tables show execution time for each task and the context switch. The execution time is measured from the point the task executes to the point it's suspended. The context switch is measured from the point the task is about to finish to when RTOS start executing the next task as current task is suspended.

Table 3. Execution time for RTOS tasks.

Tasks	**Execution Time (sec)**
Get image function	0.21271
Grayscale Task	6.53494
Crop Task	0.01522
xCorr2 Task	1.06011
Offset Task	0.01338
CalcPos Task	0.00002
CalcCoord Task	0.00001
Delay Task	0.00001

Table 4. RTOS context switch time.

Context Switch Task	**Time (s)**
Grayscale to Crop Task	0.01553
Crop to xCorr2 Task	1.06026
xCorr2 to Offset Task	0.00904
Offset to CalcPos Task	0.00034
CalcPos Task to Calccord	0.00008

The synchronization and the execution flow of the RTOS tasks are done with help of semaphores. We have created semaphores such that almost every task is initialized as semaphore count of 0, the only semaphore with count of 1 is grayscale task semaphore, who will be the key of synchronization. The execution is explained as follows and is executed as shown in figure 3. Grayscale task executes first as it has the highest priority and the only task ready. When the task finish it signal the key to Crop task semaphore which it will acquire and executes Crop Task. When Crop Task has finished execution, it signals the key to xCorr2 Task which executes. Now that xCorr2 has finished executing It will signal the key to the next task, Offset Task. Then it's time for Offset Task to execute. The tasks are executed as shown in figure 3, which follows the explained procedure of semaphores. The highest respective lowest priority task is grayscale respective delay task.

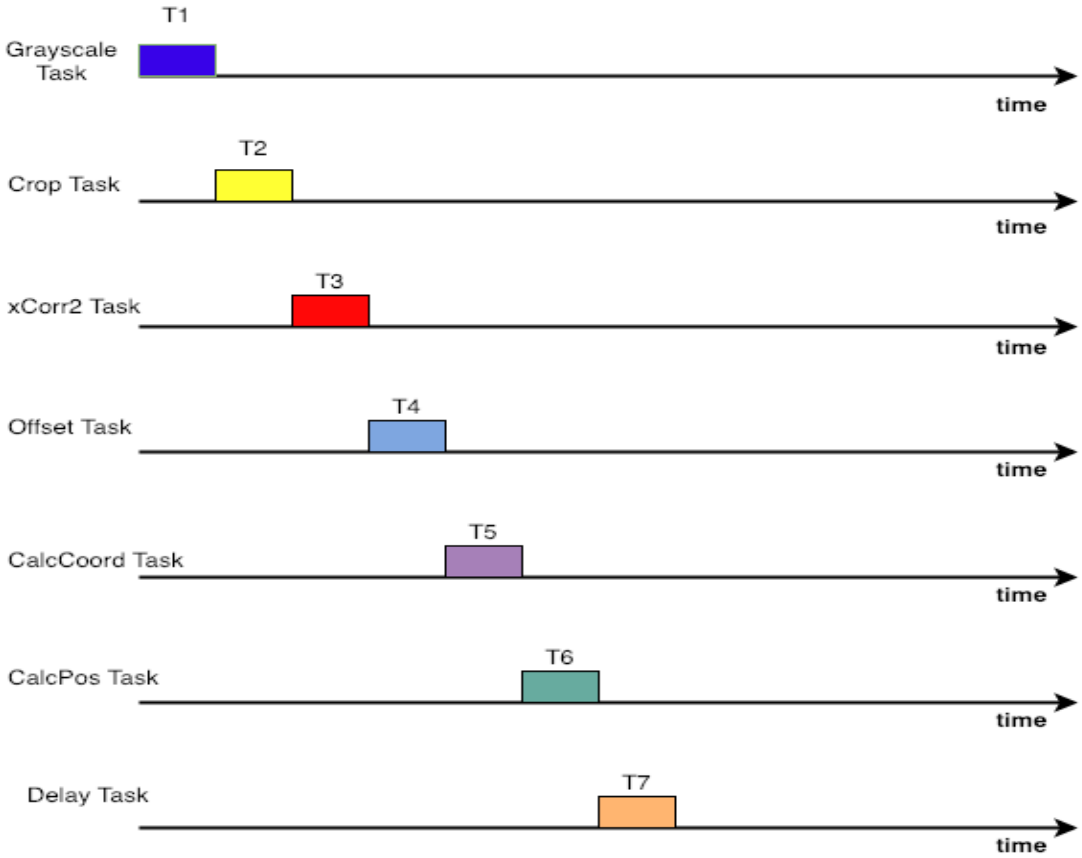

Figure 3. Schedule graph of tasks executions for one image sequence.

5.3. Multicore hardware architecture

The multicore implementation is an important aspect of this project, if not the most important. Multicore can be implemented differently, and this is up to the designer. The given hardware architecture (Interconnect Network) gives an idea of what is possible to implement and how we can implement it. Until this point the application model has been running on the SRAM and only Cpu0 accessing it – which describes the network. Our goal is to optimize the application to gain throughput of 350 images per second. The following subchapters will describe the components of the hardware architecture.

5.3.1. Central Processing Units

The architecture consists of 5 32-bit processors (CPUs) whereas only processor0 has access to the static random-access memory (SRAM) and the rest have their respective on-chip memory access, see figure 2. The processors can access the shared components fifo, shared on-chip memory and mutex through the interconnect network. The goal here is to make the processor communicate between each other and implement synchronization and resource sharing. By having this in mind we will be able to approach a reasonable design of the multicore application.

5.3.2. Shared on-chip memory

On-chip memories have faster access time than SRAM which means this type of memory will decrease the communication latency between processor and memory. The downside of On-chip memories is that they are usually small in size, which in our case is a size of 8192 kilobytes (8.2kB). The advantages of on-chip memory will increase the throughput of our application, but we are limited because the size of memory will eventually be an obstacle such that we need to consider code optimization. It is possible for the cpus to read from the memory while another cpu is writing. The shared on-chip memory is accessible by all the processors through the interconnect network (NoC).

5.3.3. Mutex

The mutex core ensure mutually exclusive ownership of a shared resource. Mutex are used for synchronization such that a shared resource is accessible once the mutex is locked and is then released when unlocked [3].

5.3.4. FIFO

FIFO (first in first out) is a method which can be used to process and reacquire data. FIFOs are slow compared to access time and writing time of on-chip memory. The idea here is to try to avoid using FIFO if it's not necessary [2].

5.4. Application on multicore

The application model has been run on RTOS and bare-metal without optimization and the throughput is not close to what the goal is. So far, we have gained all the information of the hardware architecture and having that in mind we can proceed on how we could design the multicore implementation.

The first process of implementation is to try dividing the application to run on all processors by making them communicate between each other, that is synchronization, and resource sharing – e.g. cpu0 has the image 64x192 on SRAM and will have to communicate with the other 4 cpus, which only have access to the shared on-chip, when there is data stored to acquire. Based on our previous bare-metal measurement of every function grayscale function takes the longest time (~5s). Splitting the grayscale function (Grayed actor in SDF) between all 5 CPUS will increase the performance drastically and this will be a good starting point for developing and optimize the multicore to reach the throughput.

The size of SRAM is 524kilobytes and on-chip shared memory has the size of 8192 bytes. The size of images are 12288 bytes which limits us in putting all the image to the shared on-chip memory which would've decreased the memory access time. The implementation is to divide the images matrix with size of 64x192 in such manner that 42x192 of that will be on the shared memory and 22x192 of that will be on SRAM. Figure 4 below shows the implementation of Grayed actor on multicore hardware.

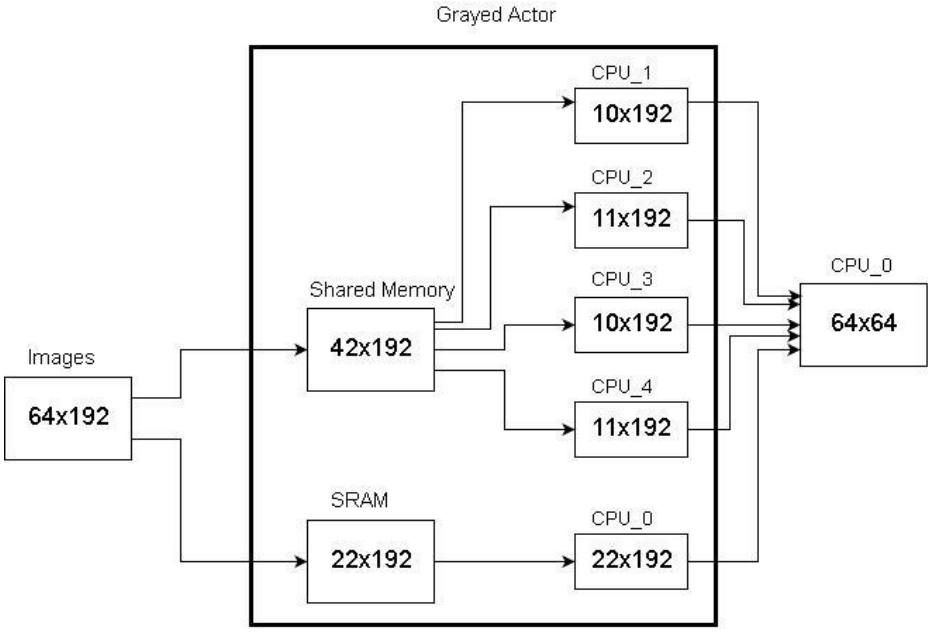

Figure 4: Grayed actor implementation on multicore.

Furthermore, we use mutex core with Avalon interface to coordinate accesses to the shared memory. The mutex core provides a protocol to ensure mutually exclusive ownership of the shared memory. We deploy all mutexes in such a way that allocate one mutex to each CPU and when each CPU want to access to the shared memory, we lock the related mutex and after finishing its work, unlock the mutex.

The synchronization between the CPUs was achieved through flags being sent through the FIFO and on-shared memory. Cpu0 will start first and will wait until the rest of the CPUs have started and are ready, the flags sent from the other CPUs are sent through FIFO and at the same time Cpu0 checks if there are 4 elements in the FIFO which means that the CPUs are ready. The synchronization is then done as following:

- Cpu0 puts the 42x192 image on the shared on-chip memory and sends a flag by setting a byte on the on-chip shared memory address with offset accessible by all the CPUs.
- The CPUs are waiting until Cpu0 is done with putting the image on the shared memory, during that time they are reading the memory address where the flag byte is located and are constantly looking for the flag which means the data is now stored on the memory.
- Once the CPUs receive the flag, they start acquiring data from different memory address offset assigned to each CPU such that they all can read from the memory at the same time without having to wait for each other.
- Once the CPUs have finished acquiring data from the shared on-chip memory they sent flags back to Cpu0 saying it has finished. If Cpu0 receives one flag from each Cpu it will respond with one flag, accessible by all Cpus, saying they can now put the data back on the shared memory. This is done to avoid overwriting the data in shared memory in case one cpu is slower than the other.
- Once the Cpus have finished computing they shove the data back to the shared memory.

This implementation did not achieve the goal of throughput, this is so far without optimization of the code. The following is the size of code on each CPU.

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]**
2	**84k**	**4565**	**4565**	**4754**	**4402**	**8k**	**110.2k**

Table 5. Size and throughput of multicore first implementation.

With this result we realized that we had to make the use of the cpus efficient and divide the tasks between the cores.

5.5. Optimization of Multicore

Multicore optimization is based on however an implementation can be designed in such way the end result is still accordingly the expected values. Optimization is a trade-off between accuracy and speed. For this project we have a detection error margin of +/- 2 pixels on the application.

The hardware architecture provides us with information of how the components are connected and knowing every component in the architecture allows us to make the right optimization decisions. The application model has to be optimization so that it's according to the SDF (actors) graph.

5.5.1. Optimization of shared on-chip memory

The first implementation suggested that Cpu0 is slowing down the performance of the application because it has the data stored on SRAM which has slower memory access compared to on-chip memory. The whole size of image cannot fit on the shared on-chip memory for the other cores to access the data. The cropped image after crop actor is 31x31, see SDF graph figure 1. To make is possible we should crop the image, place it on shared memory and divide the data between the CPUs to compute the grayscale actor, reaching parallelism.

The network (data bus) between components is a 32-bit width, which implies that reading and writing 32-bit will always be more efficient than other data type. This information implies that writing the data from SRAM to shared memory is faster if done as 32-bit writing or reading.

5.5.2- Optimization of functions (actors)

The first implementation included only grayscale split between the CPUs but it's also possible to split xCorr actor between the CPUs giving us further optimization of the application. Resulting in xcorr and grayscale function being performed parallel on all 4 CPUs which leave us to CPU0 which will only do the simplest, fastest and almost none time consuming functions – CalcCoord and ObjectPos. The importance of function split on the CPUs is determined by their possibility of parallelism and execution time.

5.5.3. Optimization of C code

Optimization of the C code improves the performance exponentially. C code optimization included arithmetic optimization such that we are using shift operations instead of integer multiplication and division – this optimization increased the performance substantially. We continued to optimize the code such that avoiding function calls which introduced stack memory manipulation resulting in decrease in performance and unrolling for loops which gave additional performance gain. We followed this protocol to optimize C code [4].

5.5.3. Optimized application model

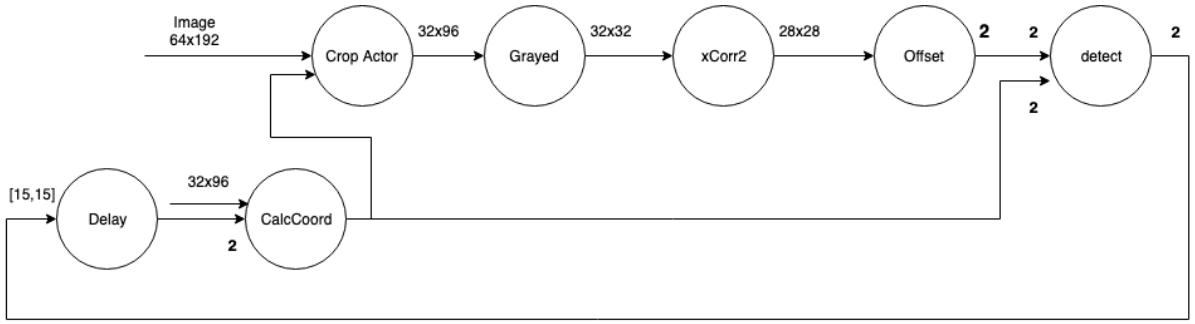

Figure 5. SDF graph after optimization.

The SDF graph in figure 5 describes the application model after the optimization. The image 64x192 is cropped and written to shared memory. The image size 32x96 is because we are writing as 32-bit and alignment is needed – the size of cropped image 32x96 (3072 bytes) is now placed on the shared memory. The 32x96 image is then split between the processors, where each CPU takes 8x96 (768 bytes) data. The shared memory block is divided and defined for each CPU, every CPU has its specific memory block, e.g. cpu1 has memory block address from base to 0x100 and cpu2 has memory block address from 0x100 to 0x200. Gray, xCorr2 and Offset function is split between the processor so that they are working in parallel. Cpu0 tasks are now only cropping and writing the image to shared memory, cropping includes calccoord, delay and detect function which has neglected able execution time. Once the processor has finished computation, they will put back the data to the shared memory where Cpu0 will access it and show the result.

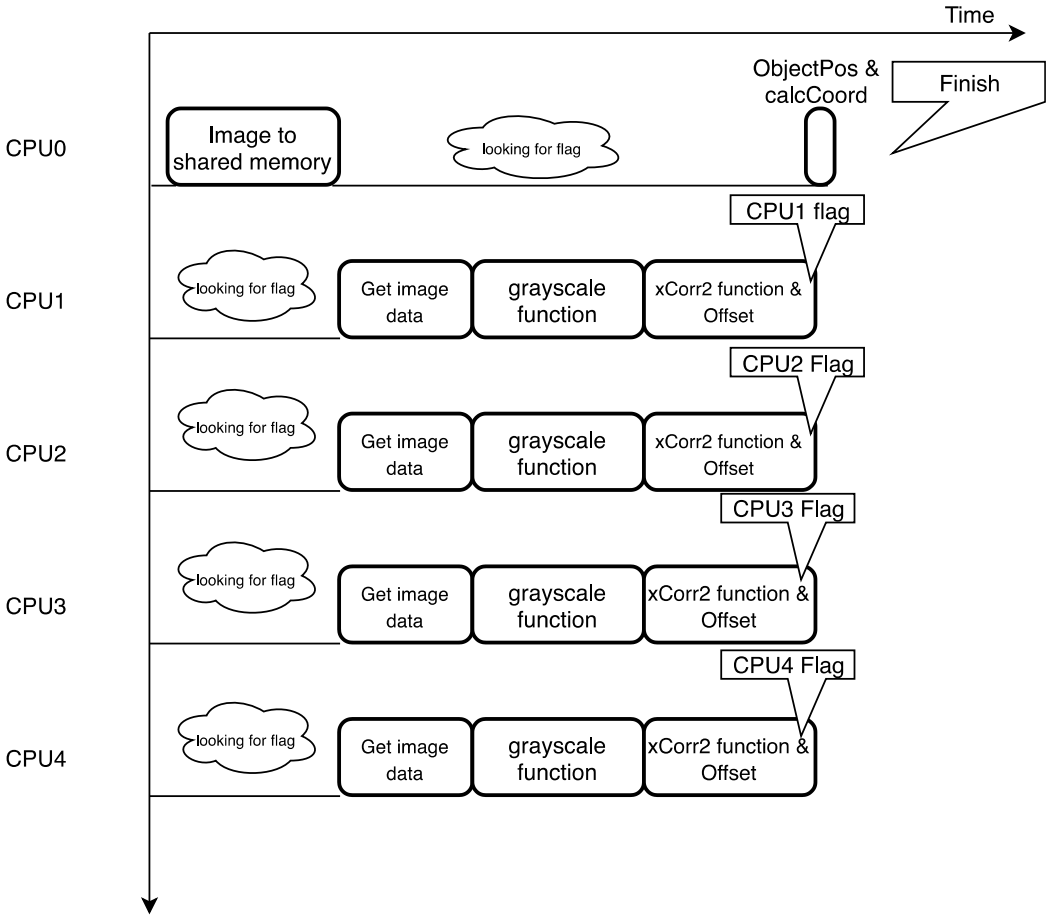

Figure 6. Rough schedule of application.

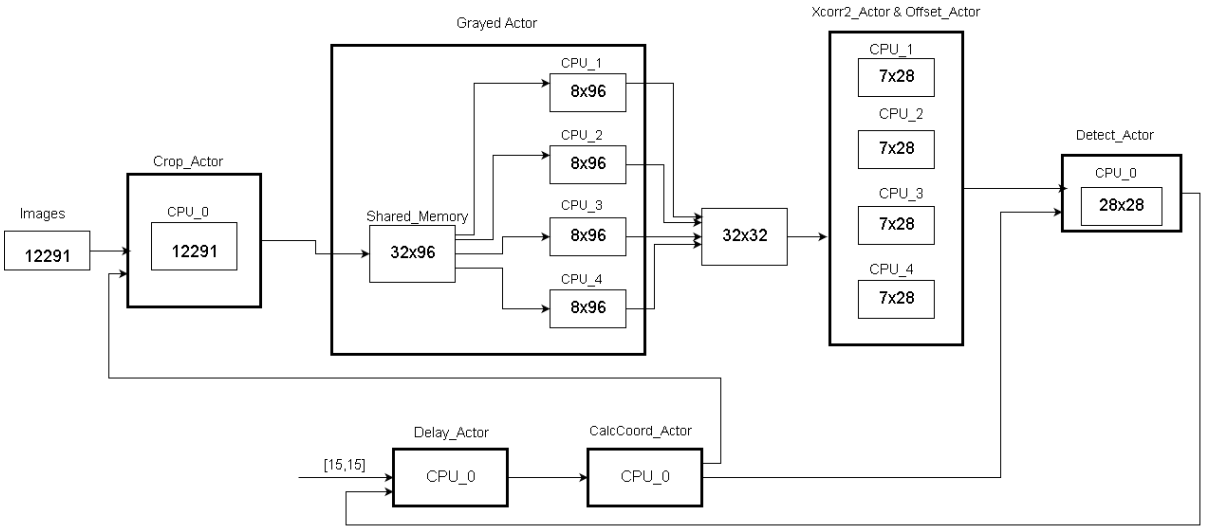

Figure 7. Parallelism of functions between CPUs.

Figure 7 describes the behavior of the optimized application model – how the functions and data are split between CPUs to achieve parallelism. As mentioned, we are now cropping first and then putting the image of size 32x96 on the shared memory and from there every CPU have access to the image. We split the 32x96 image in four resulting in 8x96 image on each CPU which in their turn will do the grayed function. After the grayed function we will have image size of 32x32 split between four CPUs, that is 8x32 image on every CPU except CPU0. Because each CPU has its own data to work with, they can proceed to calculate the xcorr2 and offset actor for given data. When the computation of every function is done, they put back the results of every function – the important result is obtained from xcorr2 and offset because we gain the location of the pattern.

5.5.4. Synchronization

The synchronization of the multicore is achieved by sending flags through the shared memory. Every CPU has it respective flag located in their respective memory block offset.

The application starts with CPU0 waiting for the other CPUs to be ready to run. CPU1 to CPU4 will indicate they are ready through FIFO communication, CPU0 will keep checking the FIFO status looking for four elements, once it detects four elements it will stop looking and start the application because now all the CPUs are ready to run. CPU0 starts cropping and writing the image to the shared memory meanwhile the other CPUs are looking for a flag that indicate if the image is available on the shared memory. Once the CPUs detect the flag, they will start acquiring image data from their respective memory block and start computation of functions. Meanwhile CPU0 is looking for a flag from each CPU – Receiving all flags mean that the CPUs are done with the computation and that CPU0 can now gather the values stored in specific memory block. The values given by CPUs and collected by CPU0 is the position of maximum value, that is index number and maximum value from each CPU – those values are sent through a flag stored on shared memory. The values are then compared between each other to find the position of the maximum value which is used to find the pattern and where to crop for the next image.

5.6. Optimized Multicore measurements

This section will show the execution time for different number of images. Starting with execution time of one image, one run.

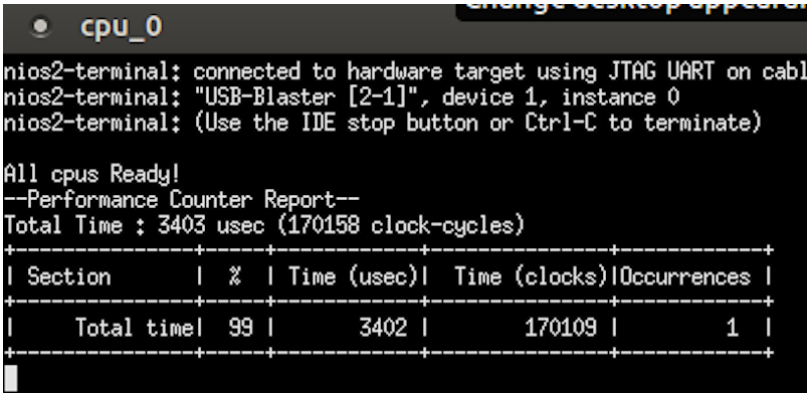

Figure 8. Execution time of one image.

Throughput of one image is 293, to make sure the throughput is correct we run the application for four images and observe the throughput.

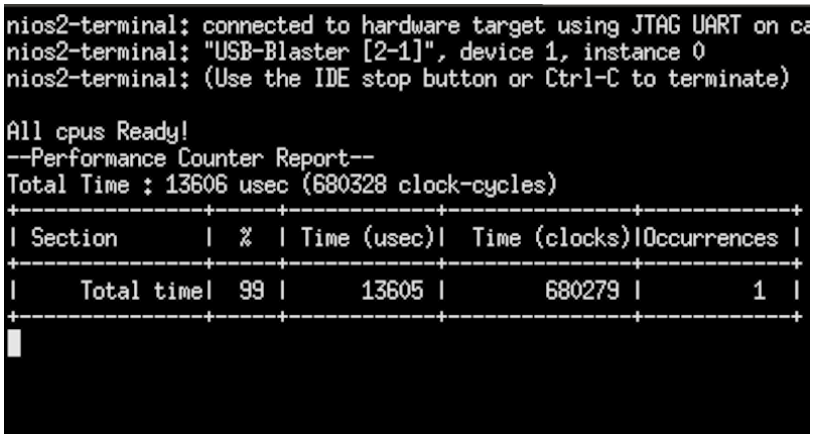

Figure 9. Execution time of four images.

Total execution time for four images, see figure 8, is a four multiple of one image execution time – 13605/4 = 3401 usec.

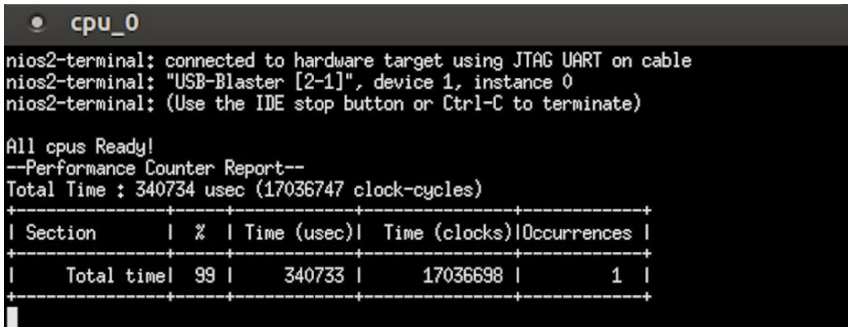

Figure 10. Execution time of hundred images.

Total execution time for hundred images, see figure 9, is about 100 multiple of one image execution time – 340733/100 = 3407 usec. This proves that the throughput is the same overall.

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]**
293	**6704**	**7604**	**7640**	**7604**	**7604**	**3092**	**40248**

Table 6. Throughput and size of multicore after optimization.

6. Discussion

6.1. Optimized application model

When we started to optimize the application model we were considered about the final result, which are the location values of detecting the pattern, the position of maximum value and detect indexes (detect actor). The values obtained from Haskell are equal to those seen at the output of bare metal implementation. First step was to redraw the SDF graph to see the optimized application model such that we were sure that the implementation was similar. We realized that Calccoord actor didn't need any matrix input from gray actor but the size and delay which we were able to make as constants. The graph before was crop of 64x64 to 31x31, but we wanted to crop first the 64x192 image. The solution was simple because we only needed to change the cropping of column, such that the starting point is correct for matrix before gray actor. We changed the cropping constant for column such that whatever position we achieve from detect and delay actor is multiplied by 3, because the image is not yet grayed. We run the new cropping code on bare-metal to see if we achieve the same results as Haskell and it worked.

The bottleneck of the application is mainly the time it takes to write the image from SRAM to shared memory. In the beginning we were reading and writing as unsigned char, both from SRAM and to shared memory. We realized that this was very inefficient because we are not actually using the whole width of the data bus which slowed down the whole application. We implemented writing and reading of 32-bit, such that we are reading 4 bytes at a time from SRAM and writing 4 bytes at a time to shared memory. This change increased the performance of the application.

6.2. Challenges

Optimizing the multicore to achieve the throughput requires a good understanding of the hardware architecture. From our first implementation we have learned that FIFO is slow and should be avoided If possible, that's why we have moved on to only use shared memory for resource sharing and synchronization, which we eventually came across other factors.

The challenges that we faced during this projection were throughput not reacting to optimization. The issue was that we initialized variables and matrixes as unsigned char, which is 1 byte compared to the processor being 4 byte, this itself slowed the whole application a lot and as soon as we changed everything to integer the throughput jumped to almost 200 from 8. We spent a lot of time on this issue which also limited the time left on the project for optimization.

6.3. Synchronization

This part was the main purpose of implementing multicore, efficient synchronization increased the performance. Understanding how the architecture works we could easily implement handshake synchronization such that the CPUs know when and what to do. We first tried to synchronize the multicore with FIFO but realized it was slow compared to mutex and shared memory communication. Moving on to the mutex we implemented the synchronization with mutex, but we were unsure if it really worked because our throughput was very low and didn't react to the optimization, as mentioned before we believe the error was because we were not reading or writing in integer. Moving on we synchronized the multicore with shared memory but that was at the same time we fixed the error of throughput.

The synchronization did work, for now, and after discussing with TAs we realized our method of synchronization was unsafe and we should have done it with mutex instead. This is true even though or implementation worked it's unsafe in real-time systems. Mutex provide exclusive ownership of a shared resource in such case we are sure no other CPU is polling the shared resource.

6.4. Further optimization

The throughput of 293 did not reach the goal of 350. Because we wasted a lot of time on the error of throughput not increasing when optimizing we had limited time before the deadline. We managed to jump from throughput of 8 to 293 in 1 day, that's optimization according to [4] and synchronization.

Further optimization would be to unroll for-loops, this is because we were able to increase the throughput by 30% by only unrolling for-loops of one function. We have looked over the code and came across that we could unroll Xcorr2 actor loop to increase the performance. Other option is to optimize the synchronization such that we use mutex instead of flags. This implementation includes that one on-chip CPU reads the image as soon as it's seen on the shared memory, that is when writing from SRAM to shared memory we will release the mutex for that on-chip CPU. The procedure continues, for every part of image available on shared memory we unlock (release) mutex for that given on-chip CPU that will access it. This will increase the performance of the application and will guarantee a safer synchronization.

7. Conclusion

The aim of this project was to understand how multicore is implemented and how hardware architecture could limit the throughput. This project gave us good understanding of multicore and how to optimize an application to reach great throughput. We learned the different types of optimization that could increase performance in an application and how different implementations of multicore could either increase or decrease the performance. We also realized how complex it could be if there were more cores that has to communicate with each other – that resulting in much better performance. Overall the project gave us a good picture of multicore.

References

[1] Kent, Orthner. *Applying the Benefits of Network on a Chip Architecture to FPGA System Design.* Intel. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01149-noc-qsys.pdf (Acquired 2018-02-03).

[2] Altera. 2009. *On-chip FIFO Memory Core.* Intel. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/qts_qii55002.pdf (Acquired 2018-02-20).

[3] Altera. 2009. *Mutex Core*. Intel. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu_nii51020.pdf (Acquired 2018-02-20).

[4] *Tips for Optimizing C/C++ code.* https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf (acquired 2018-02-25).