Implementations of Image Processing on DE2 Board

Session 3 (Feb.26)

Jiaqi LI
Embedded Systems
KTH Royal Institue of Technology
Stockholm, Sweden
Email: jiaqili@kth.se

Guanghao GUO
Embedded Systems
KTH Royal Institue of Technology
Stockholm, Sweden
Email: gguo@kth.se

***Abstract*—The paper illustrates the implementation of a concurrent data-flow image processing application on multiprocessor and single processor (with and without RTOS) respectively. The main focus is how to take advantage of shared resources and schedule task in real time operating system. By measuring the execution time and memory footprint, we can compare differences among bare-metal implementation, RTOS implementation and multi-core implementation.**

1. Introduction

In this laboratory, the requirement is to process image by three different implementations. These implementations are single core bare-metal, single core with RTOS and multi-core. During this lab, we can put our knowledge of embedded software into practice and learn how to optimise codes to get maximum throughputs and minimum memory footprints.

2. Hardware Architecture

2.1. Interconnection Between Components

Platform architecture is shown as Figure 1.

The interconnection network is an Avalon Memory Mapped Interface (Avalon-MM), which is an address-based read/write interface for master and slave components, connected by a system interconnect fabric. In this case the processors are the only masters and the other components are acting as slaves. A various amount of components can be described using the Avalon-MM making it very useful. [1]

Through analyzing the given QSYS file and documents, we observed following facts:

1.The data master is connected to both memory and peripheral components while the instruction master is only connected to memory components.

2.The given architecture has five CPUs.

3.The CPUs are connected to the shared on-chip memory and only CPU 0 has access to other peripherals such as buttons, switches.

4.There are five mutexes and two fifo buffers which are shared between all CPUs.

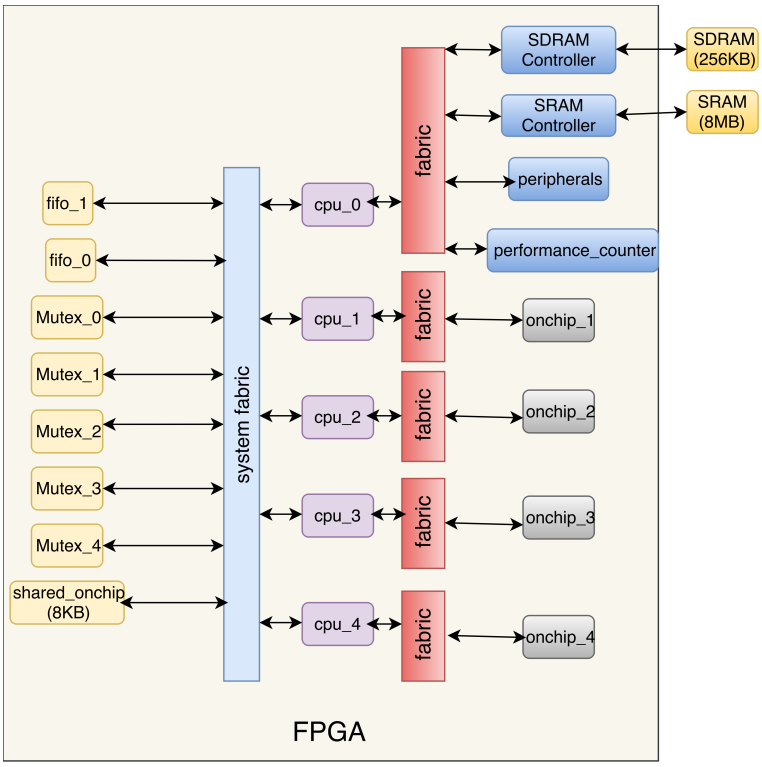

Figure 1. Platform Architecture

2.2. Shared Resourses

The shared memory can be accessed by all the CPUs and thus mutex logic is used to secure the communication between shared memory and the CPUs. For example, the data in the shared memory is accessed by any one CPU,

it will lock the mutex so that simultaneously another CPU will not be able to access it. Other CPUs have to wait till the mutex is unlocked.

Mutex:The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource. The mutex core provides a hardware-based operation allowing software in a multiprocessor environment to determine which processor owns the mutex. There are 5 mutex available.

On-Chip fifo Memory:a contiguous memory space with dedicated segments of memory allocated for 16 channels. Data is delivered to the output interface in the same order it was received on the input interface for a given channel.

2.3. SRAM and peripherals

SRAM/SDRAM: Only CPU 0 has access to the SDRAM of 8MB and the SRAM of 512 KB.

Peripherals: Every CPU has internal peripherals and only CPU 0 is connected to external peripherals such as buttons, LEDs, switches and seven segment display.

3. Functional Specification of Module

3.1. Synchronous Dataflow

There are 5 SDF actors for processing an image: rgbToGray, resizeImg, brightness correct, sobel and toASCII. RgbToGray is an actor with 1 input signal and 1 output signal. When rgbToGray fires, it consumes $x_0 * y_1$ tokens and produces $x_1 * y_1$ tokens. resizeImg is a SDF with 1 input signal and 1 output signal. When resizeImg fires, it consumes $x_1 * y_1$ tokens and produces $x_2 * y_2$ tokens. Brightness correct is a SDF with 1 input signal and 1 output signal. When brightness correct fires, it consumes $x_2 * y_2$ tokens and produces $x_2 * y_2$ tokens. Sobel is a SDF with 1 input signal and 1 output signal. When sobel fires, it consumes $x_2 * y_2$ tokens and produces $x_3 * y_3$ tokens. ToASCII correct is a SDF with 1 input signal and 1 out put signal. When brightness correct fires, it consumes $x_3 * y_3$ tokens and produces $x_3 * y_3$ tokens. Then the final tokens will become the out put of the system. Hence, the system consumes $x_0 * y_1$ tokens and produces $x_3 * y_3$ tokens. When The SDF for image processing is shown as Figure 2.

3.2. Procedures for Image Processing

Convert RGB to Gray

A grayscale image is one in which the value of each pixel is a single sample representing only an amount of light, that is, it carries only intensity information. Images of this sort are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest. And in order to get grayscale of every pixels in a typical RGB pictures, the Formula (1) is used.

$$Y = 0.3125 \times R + 0.5625 \times G + 0.125 \times B \qquad (1)$$

Resize an Image

The next step is to resize an image. The strategy for resizing is to merge the four pixels into one pixel. As result, the total size will decrease to one fourth of the original picture. Formula for resizing is shown as Fomula 2.

Brightness Correction

In order to correct the brightness level in the picture, the maximum and minimum value of the brightness need to be found out before adjustment of brightness level of the image can be done. Brightness level need to be adjusted by using four threshold levels according to the maximum and minimum value of the pictures which has been detected in the former procedure.

Sobel Edge Detection

The Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. Formula of Sobel operator is shown as below.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$G_x = image * K_x \qquad G_y = image * K_y$$

$$G = \sqrt{G_x^2 + G_y^2}$$

Output ASCII images The last step is to convert the grayscale into ASCII characters and output these characters on terminal to show the result of image processing.

3.3. Importance of SDF

The SDF model consists of functions describe above, the overall function of this SDF is to take a series of RGB images, resize them and extract the edges, transfer every pixel into equivalent ASCII code which represent the brightness level. Within this program, a control progress is implemented to detect the contrast of brightness, if the contrast is not enough for edges detection, the image will be corrected to fulfill the requirement.

4. Parallel Patterns and Parallelization

Pipelining can be applied in this application that apply a sequence of operations to each element in a data set.Because the image processing is typical streaming application, and caculations on different pictures are independent.Therefore this application has a potential to realize pipeline parallelization. Pipeline parallelization can execute a sequence of calculations on a sequence of data elements(images). Therefore, we can assign each task to a PE and design a mechanism to forward data from one task to another.Speedup limited by number of stages and the slowest stage. If cost of sending individual elements ,communication cost,is high, aggregation is important. And on the contrary if a task consume too much time, we can spill this task into two different tasks.

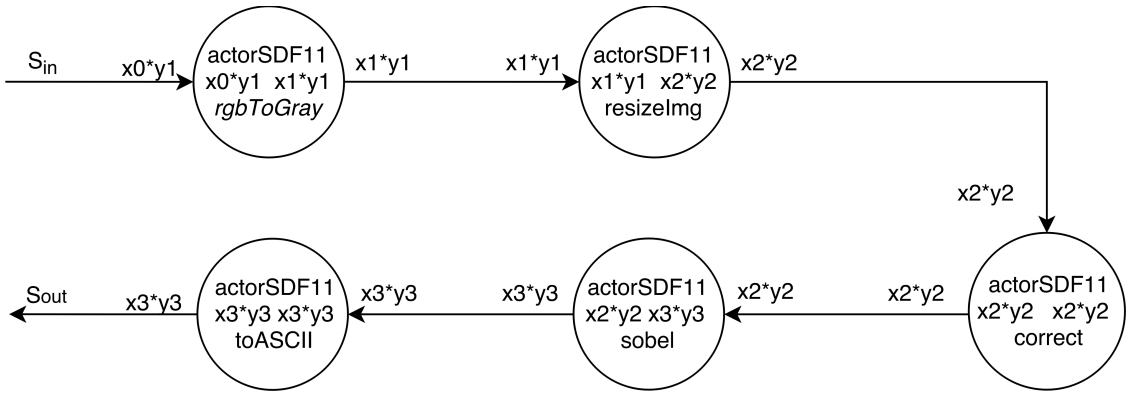

Figure 2. Parameterized Synchronous Data Flow of Image Processing

$$\begin{bmatrix} y_{11} & y_{21} & y_{31} & y_{41} & \cdots \\ y_{12} & y_{22} & y_{32} & y_{42} & \cdots \\ y_{13} & y_{23} & y_{33} & y_{43} & \cdots \\ y_{14} & y_{24} & y_{34} & y_{44} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \mapsto \begin{bmatrix} y'_{11} = \frac{y_{11}+y_{21}+y_{12}+y_{22}}{4} & y'_{21} = \frac{y_{31}+y_{41}+y_{32}+y_{42}}{4} & \cdots \\ y'_{12} = \frac{y_{13}+y_{23}+y_{14}+y_{24}}{4} & y'_{22} = \frac{y_{33}+y_{43}+y_{34}+y_{44}}{4} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \quad (2)$$

5. Implementation

5.1. Bare Metal

For the bare-metal implementation, basic procedures are called sequentially. The sequence is rgbToGray, resizeImg, brightCorrect, sobel and at last printAscii. RgbToGray will be called from main() and a pointer to images will be passed as an argument. Then at the end of every basic functions, the next function will be called and the pointer to the processed image will be passed on the argument.

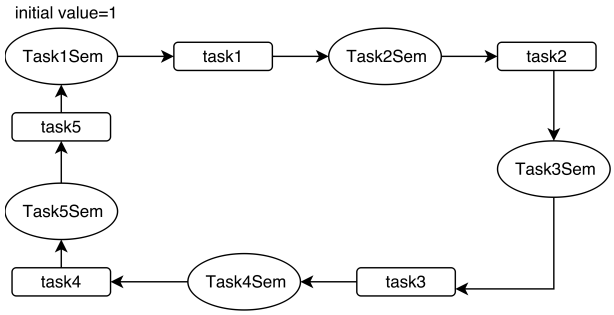

Figure 3. RTOS Implementation

5.2. RTOS

For RTOS implementation, we separate it to be five tasks: grayscale function, resize function, brightness correct function, sobel function and printASCII function. Every basic function will be put on seperate tasks. Hence, there are 5 tasks. Tasks communicate with each other with 5 semaphores. Semaphores are used to control the execution order of tasks and access of shared on-chip memories.According to the initial order of the application, we set the semaphore of task RgbToGray to 1 initially, and successively pend its own semaphore and post the semaphore of the subsequent task to continue processing.RTOS implementation can be shown as Figure 3.

5.3. Multi-core

Multi-core application also has five parts like single core, and we implement grayscale function and store the ASCII back to the SRAM in CPU0, because only the CPU0 can get access to the SRAM, so it can get the RGB image from SRAM and store ASCII back to the SRAM. We also implement resize function and brightness correct function in CPU1, sobel function function in CPU2, and printASCII function in CPU3. On flow chart below, we show how it works.

We divide the shared memory into four independent sections and each section is responsible for one picture. The size of each section is fixed and it is 2048 byte.So we can store the grayscale dataset at the beginning of each section and it will occupy $32 * 32 + 3 = 1027$ byte of the shared memory space, since the first three byte always store the size of X, the size of Y and the maximum value of the grayscale. It is followed by image dataset after resize and brightness correct function and it will take up $16 * 16 + 3 = 259$ byte of of the shared memory space. The next part is dataset after sobel function and it accounts for $14 * 14 + 3 = 199$ byte of the shared memory space. The last part of each section is ASCII storage and it contains the same shared memory space as the dataset processed by sobel function.

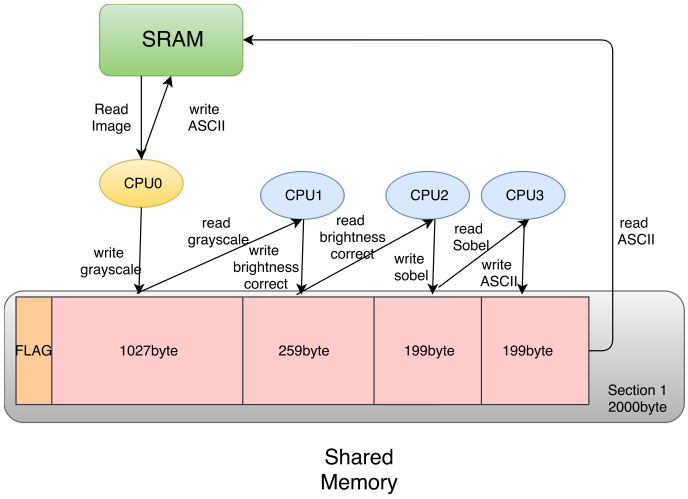

Figure 4. Mapping Plan

Therefore, in total the size of each section which is being used is $1027 + 259 + 199 + 199 = 1684$ byte which is less than 2048 allocated for each section.

And Apart from these four sections, there is also a one-byte flag used to store the current state of the process. The flag contains eight binary bit, every two of them forms a counter which can calculate the number of image being disposed. 00 stands for the smallest number zero and 11 stands for the largest number three. And the first two bits represents the counter relating to grayscale function. The remaining six bits represents brightness correctness counter, sobel counter and printASCII counter respectively. The schedule of multi-cores implementation is showed below.

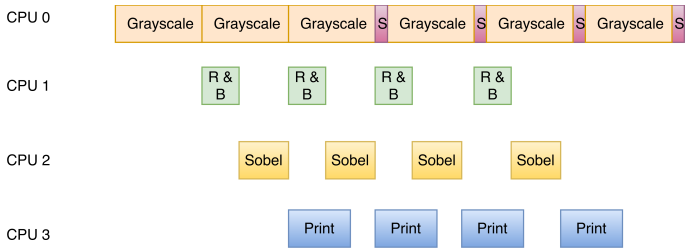

Figure 5. Multi-core Schedule

6. Performance Optimization

First, We optimize sobel operator, when it comes to square root, we did not use $G = \sqrt{G_x^2 + G_y^2}$, we use $G = |G_x| + |G_y|$ instead. Because square root function can consume a large amount of time, we use absolute values to do the approximation. This approximation is correct because in this project, this approximation results to an maximum error of 74.68 on grayscale values when $|G_x| = |G_y| = 127.5$. In this case, the approximation result for G is 255 while it should be 180.31 using square roots. Before sobel optimization, the soble function accounts for nearly half of the total execution time. And it just takes up 17 percent of the total execution time after optimization. It is much faster to compute and thus optimize the performance.

Second, when converting RGB picture to greyscale, we use the approximation of use binary shift instead of multiplication and division. Because multiplication and division cost 3 times more time than binary shift. For example, if we use multiplication and division to calculate the equation of, it should be like this $G = (5 * R + 9 * G + 2 * B)/16$, and the performance of that is 30 pictures per second. However, if authors use binary shift like this,$G = (R >> 2) + (R >> 4) + (G >> 1) + (G >> 4) + (B >> 3)$, and the performance will increase significantly, 180 pictures per second. So we need to avoid multiplication and division as much as possible and use binary shift function instead.

Third, we hoist the common statement from the loop to avoid repeat computation of the similar statement in the loop.

Fouth, we use int array to store char values that are on shared onchip memory to SRAM. For example, in this project, a char is 1 byte and an int is 4 bytes, which means reading or writing one integers are equivalent to reading or writing 4 chars. This can reduce the number of reading or writing, thereby increasing the Throughput.

TABLE 1. SINGLE CORE MEASUREMENTS

	Single Core (RTOS)	Single Core (Bare Metal)
Throughput[s-1]	106	112
SRAM(Byte)	157244	106820

7. Measurement and Analysis

The project is conducted on DE2 board. There are three implementations: single core without RTOS, single core with RTOS and multi-core. The performance and memory footprint of these three implementation are measured.

7.1. Single Core

Throughputs and footprints of two implementation are shown in Table 1. Performance counter reports are shown in Table 2 and Table 3, where Table 2 is table for bare metal implementation and Table 3 is table for RTOS implementation. The total percentage does not equal to 100 percent, because some time is spent on context switch. The result suggests that for single core, RTOS implementation is less efficient than bare metal. It is because RTOS has too many overheads for running tasks, which slows down the speed.

7.2. Multi-core

Throughputs and footprints of two implementation are shown in Table 4. Both throughput and footprints meets the design constraints which are 320 or higher throughput and 45KB or less footprints.

TABLE 2. BARE METAL PERFORMANCE

Section	%	Time (sec)	Time (clock)	Occurrences
rgbToGray	30.7	0.00273	136490	1
resizeImg	10.7	0.00095	47600	1
brightCorrect	6.31	0.00056	28028	1
sobel	23.6	0.00210	104858	1
printAscii	28.3	0.00252	125820	1
Total time	100	0.00888	444171	1

TABLE 3. RTOS PERFORMANCE

Section	%	Time (sec)	Time (clock)	Occurrences
rgbToGray	30.2	0.00283	141605	1
resizeImg	10.5	0.00099	49350	1
brightCorrect	3.52	0.00033	16501	1
sobel	24.9	0.00233	116642	1
printAscii	27.7	0.00260	129792	1
Total time	100	0.00937	468572	1

7.3. Evaluations

The multi-core implementation has the highest throughput which is almost three times the figure of single core implementations. In theory, the throughput of multi-core implementation can be 5 times that of single core implementations. But because of our mapping, calculations are not distributed equally to every core. The throughput of multi-core implementation is decided by cpu_0 which is mapped to finish the convertion of rgb to grayscale because it takes longest time.

Single core implementations can easily ensure the sequence of executing procedures. For bare metal implementaion, because C language is a imperative language, which means functions will be executed sequentially. For RTOS implementation, some tools can be used, for example semaphores, to communicate between tasks, thereby ensuring the sequence of execution. When it comes to multi-core, CPUs have to communicate via shared memory. Techniques such as flags and fifos should be use to ensure a correct output, which means there will be overheads for communications that can slow down the execution speed. There are also problems when communicating via shared memory, the data corruption. Data on shared memory can be changed by more than one processor. If the programme is not well written, there will be incorrect results.

For resource consumption, bare metal implementation consumes least because it has only one core and use less memory. RTOS implementation use more resources than bare metal implementation because it needs time for communication between tasks and more overheads for OS. The multi-core implementation consumes most resources because it uses 4 cores and 4 memory sections with only 3 times the throughput of single core implementations. Therefore, for economical consideration, we should choose bare metal implementation.

TABLE 4. MULTI-CORE MEASUREMENTS

	Single Core (RTOS)	Single Core (Bare Metal)	Multi-processor
Throughput[s-1]	106	112	323
SRAM[Byte]	157244	106820	23772
OnChip CPU 1 [bytes]	–	–	3244
OnChip CPU 2 [bytes]	–	–	3100
OnChip CPU 3 [bytes]	–	–	2096
OnChip CPU 4 [bytes]	–	–	1856
OnChip Shared [bytes]	1027	0	6737
Total Memory [bytes]	157244	106820	40805

8. conclusion

In conclusion, compared with single core implementation, multi-core implementation can achieve much better throughput. But in the mean time, it requires more footprint and hardware resourses. Therefore we have to consider this trade-off option when implementing image processing algorithm into the hardware architecture.

References

[1] Kent Orthner, *Applying the Benefts of Network on a Chip Architecture to FPGA System Design*. Intel.

Personal Contributions

Jiaqi Li

Implement brightness correction and sobel functions.
Conduct and debug RTOS implementation.
Write contents of Procedures for Image Processing and Measurement and Analysis for the report.
Use latex to format the report.
Optimize the multi-core implementation to meet design constrains.

Guanghao Guo

Implement grayscale, resize and printAscii functions.
Conduct and debug bare-metal implementation.
Write contents of Hardware Architecture, Concurrent Processes Networks for the report.
Draw graphs for the report. Transplant single core codes to the multi-core implementation