A concurrent data-flow application on a multiprocessor based on FPGA

Lab session 2: Feb 22

Peng Jiaqiao
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: jiaqiao@kth.se

Li Sujie
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: sujie@kth.se

***Abstract*—This report presents implementation of image processing in different ways. The image processing procedure contains converting into gray-scale format, resizing, correcting brightness, applying sobel algorithm and storing the image into on-chip memory in ASCII format. The possible optimization, throughput and memory footprint are also discussed. In addition, this report gives detailed analysis of the architectural features and limitation of the multiprocessor.**

Keywords—architecture; multiprocessor; on-chip memory.

I. INTRODUCTION

In this project, we are required to implement an image processing procedure on an FPGA, the application was implemented respectively on a Bare-Metal single core without system; on a single core using RTOS; and on the multiprocessor without operating system. For the multiprocessor application, it uses concurrent process network, in which several tasks can be processed at the same time, instead of waiting for only one processor. So multiprocessor application should be able to greatly enhance processing speed and consequently have higher throughput.

While implementing the applications, we also measure the throughput and memory footprint, then compare and analyze the difference between these implementations. In addition, we optimize the code to make the application have shorter processing time, so that it can satisfy the requirement that throughput should be higher than 320 images per second for 32 ×32pixel images.

II. IMAGE PROCESSING

The image processing procedure consists of several steps to eventually save the modified image into shared on-chip memory in the format of ASCII. After implementing the application, we verify the result and measure the throughput and memory footprint to make sure the application meets the requirement. The specific steps are as follow:

A. Loading and converting RGB images to grayscale

First of all, we load an image from *image.h* file, images in this file have already been stored in the format of RGB value. In order to convert it into grayscale image, we can use this formula:

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

But one problem with this formula is that the calculation takes long time, so we can modify the formula to reduce the processing time while getting the same result.

Let's assume the size of original image is $3X * Y$, then after the grayscale step, the image size becomes $X * Y$.

B. Resizing images

Resizing function reduces the size of grayscale images by using this formula:

$$y_{00} = \frac{y_{00} + y_{01} + y_{10} + y_{11}}{4} \quad (2)$$

After resizing function, the size of grayscale images is resized to $X/2 * Y/2$.

C. Correcting brightness

This step firstly detects the maximum brightness level and minimum brightness level, then check the difference between the two values. If the difference is no lager than 127, the image brightness would be multiplied by 2, 4, 8 or 16 depending on the diffecence.

This step doesn't change the size of images. It remains $X/2 * Y/2$.

D. Sobel

By using Sobel operator and formula, we detect the edge of images, but again the original formula consumes long time, we will modify it in the optimizing procedure.

This step changes the size of images to $X/2\text{-}2 * Y/2\text{-}2$.

E. Converting to ASCII

The final step is to convert the images into ASCII format, then we save them in SRAM and also print them out.

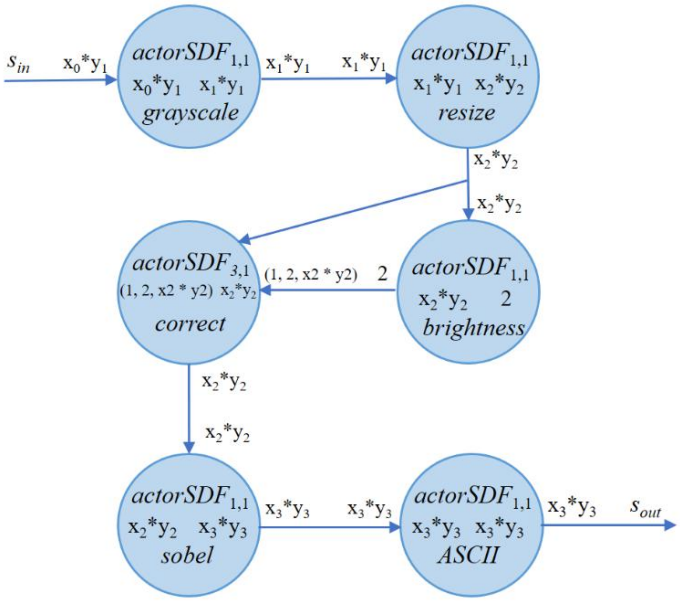

Figure 1 Synchronous Data Flow

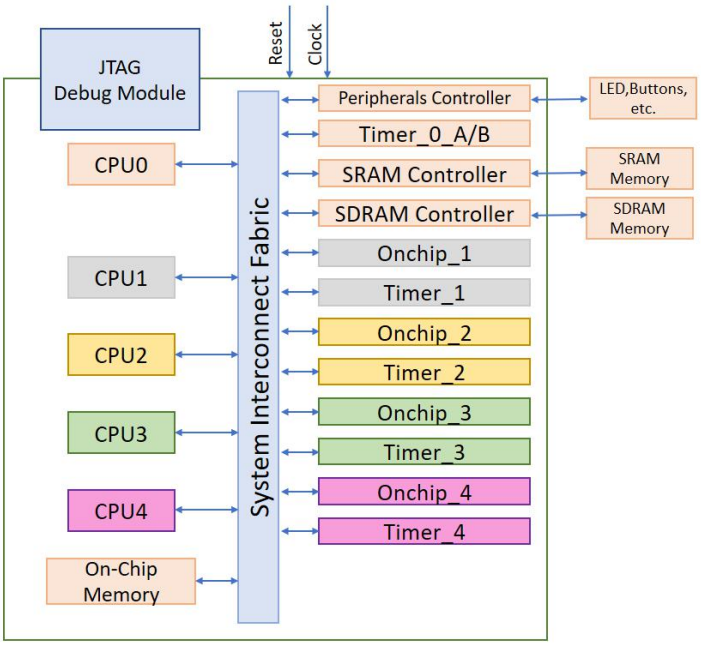

Figure 2 Platform Architecture

III. THE PLATFORM ARCHITECTURE

Nios II uses the Avalon switch fabric as the interface to its embedded peripherals. Compared to a traditional bus in a processor-based system, which lets only one bus master access the bus at a time, the system interconnect fabric allows multiple masters operate simultaneously by using a slave-side arbitration scheme [1].

The platform architecture is shown as Figure 1. The communication between components is realized by system interconnect fabric. As shown in the diagram, the chip has global reset and clock. CPU0 is connected to timer_0_A, timer_0_B, shared on-chip memory, SRAM controller, SDRAM controller and some other peripherals controllers which further control LED, buttons, etc. CPU1-4 are connected respectively to their JTAG module, on-chip memory and timer through system interconnect fabric.

IV. IMPLEMENTATION ON A SINGLE CORE

To compare and analyze different implementation, we implemented the application on a single processor respectively using the MicroC/OS-II (RTOS) and without RTOS (bare-metal implementation).

A. Without Real-time Operating Systems(Bare-metal)

For the bare metal implementation, the application is implemented without real-time operating system, so no task is used in this case. When running the program, firstly function grayscale is called in main, then after the processing, next function get called at the end of last function. When function ASCII finish, the result will be saved and printed out.

B. Using RTOS (MicroC/OS-II)

For RTOS implementation, each function was processed in different individual tasks. The task executing order was realized by using several semaphores for different tasks. When one task is executed, it decreases its own semaphore by 1 at the beginning of the task and increases semaphore of next task by 1 at the end of the task. When the last task is processed, semaphore of first task is increased by 1 at the end of the last task.

Result of 2 single core applications is listed as follow:

TABLE I. MEASUREMENTS

	Single Core (RTOS)	Single Core (Bare Metal)
Throughput [s-1]	0.4488	0.4638
SRAM [bytes]	183648	149868

V. IMPLEMENTATION ON MULTI-CORE

The flowchart in Figure 3 shows the flow of data among the five Nios-2® cores:

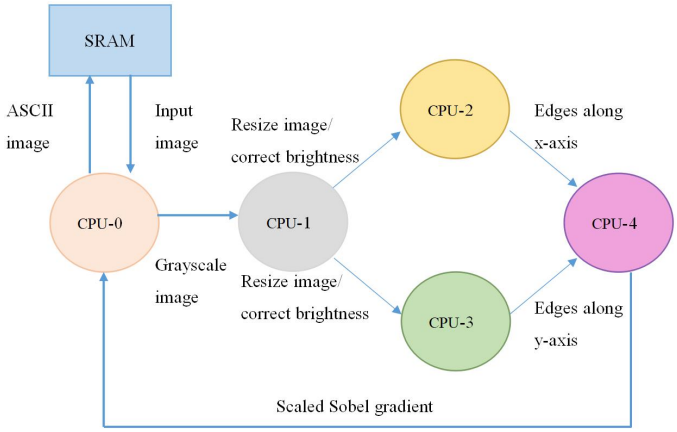

Figure 3 Flowchart of multicore implementation

A. *CPU_0*

Accesses the images stored as input in the SRAM, and converts them from RGB format to grayscale and stores them in the shared memory. And converts the sobel image produced by CPU-4 to the ASCII images like RGB format, storing them in the SRAM.

In IV we found step grayscale takes a long time. So we use bit-shifting replacing the floating point multiplication mentioned in formula (1).

B. *CPU_1*

Resizes and corrects the brightness of the grayscale image produced by CPU_0 and stores the result back in the shared memory.

C. *CPU-2 & CPU-3*

Respectively perform Sobel edge detection in x and y dimensions, using the downsized images from CPU_1.

D. *CPU-4:*

Calculates the Sobel gradient based on outputs of CPU-2 & CPU-3 and stores the result in shared memory.

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

The initial gradient formula (2) costs a long time so we uses formula (3) to approximate it.

$$|G| = |G_x| + |G_y| \quad (3)$$

Result of 2 single core applications is listed as follow:

TABLE II. MULTICORE MEASUREMENTS

	Single Core (RTOS)	Single Core (Bare Metal)	Multiprocessor
Throughput (s^{-1})	0.4488	0.4638	226.3498
SRAM (bytes)	183648	149868	44360
Onchip CPU 1 (bytes)			2408
Onchip CPU 2 (bytes)			2684
Onchip CPU 3 (bytes)			2656
Onchip CPU 4 (bytes)			2360
Onchip shared (bytes)			1886
Total Memory (bytes)			56354

REFERENCES

[1] Nios II Processor features, https://www.altera.com/products/processors/features.html.

Personal Contribution

Sujie Li

Implemented individure functions for image processing;

Implemented the application on single core (bare metal and RTOS);

Wrote about SDF, platform architecture and single core implementation of the report.

Jiaqiao Peng

Implemented the application on multiprocessor;

Optimized code to meet requirement;

Wrote about multicore implementation, optimization and conclusion of the report.