Digital Image Processing on a Multiprocessor

Kalyani Jayaprakash
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: kalyani@kth.se

Rucha Thorat
Embedded Systems
KTH Royal Institute of Technology
Stockholm, Sweden
Email: thorat@kth.se

***Abstract*—The paper focusses on the implementation of a concurrent data-flow image processing application on multiprocessor, single processor (with and without RTOS). The main focus is to exploit parallelism in the architecture to increase the throughput of application. For the analysis various optimizations were carried out for the same image processing application. The objective of this study is to get maximum throughput and minimum memory footprint.**

I. Introduction

A Nios II processor system is equivalent to a microcontroller or computer on a chip that includes a processor and a combination of peripherals and memory on a single chip. The given architecture for the laboratory corresponds to a Nios II multiprocessor with five CPUs. The CPUs are connected to the shared onchip memory and CPU 0 has access to other pheriperals. All these interconnections are achieved via avalon interconnect fabric and the communication between the shared memory and CPUs are secured using mutex logic. There are a total of four mutex available in the architecture and two fifo buffers which is shared between all CPUs. While the processor core has separate instruction and data buses, the overall Nios II processor system might present a single, shared instruction/data bus to the outside world. The outside view of the Nios II processor system depends on the memory and peripherals in the system and the structure of the system interconnect fabric. The data and instruction master ports never cause a gridlock condition in which one port starves the other. For highest performance, assign the data master port higher arbitration priority on any memory that is shared by both instruction and data master ports[1]. Every CPU has inernal timer pheriperals and only CPU 0 is connected to external pheriperals like buttons, leds, switches and seven segment display. If any of the processor has to communicate to external harware pheriperals, it can be achieved through CPU 0 because only CPU 0 can acess pheriperals. Each CPU has a data master and the data master port is an Avalon-MM master port that connects to data memory and peripherals via system interconnect fabric[2].

II. Image Processing Application

The image processing application developed focusses on processing a cycle of 10 different images to get continous input. The images are stored in the header images.h which is added in the code for CPU_0 The application developed bascially does the following functions:

1. Load the image from the SRAM
2. Perform the gray scale application on the image
3. The gray scaled image is converted to a compact image by decreasing the image size by one fourth.
4. Sobel operator algorithm is used to carry out edge detection in the resized image
5. ASCII operation to convert the pixels into ASCII character is done and the final output is displayed on terminal

The application runs two different modes: Debug and Performance

In Debug mode, images of different size are processed and after the complete processing of a image the output is displayed in the respective CPU terminal. Before taking the next image a delay is given so that the user can note down the performance of the first run. The delay is needed in the case of a linux system due to the lack of scrolling functionality in output xterm terminal.

In performance mode, different 32 x 32 images are processed and the execution time, throughput etc is noted down after 420 iterations.

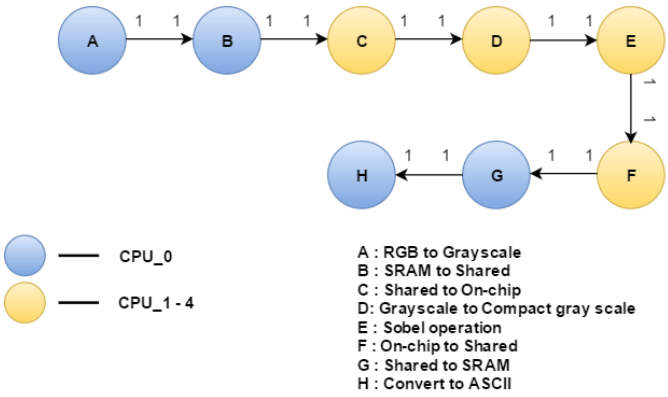

Fig. 1. Dataflow diagram of the application

A. Design Methodology for multiprocessor

Figure 1 represents the dataflow diagram of the implemented application. The functionality is divided between CPU_0 and other CPUs for processing single image.

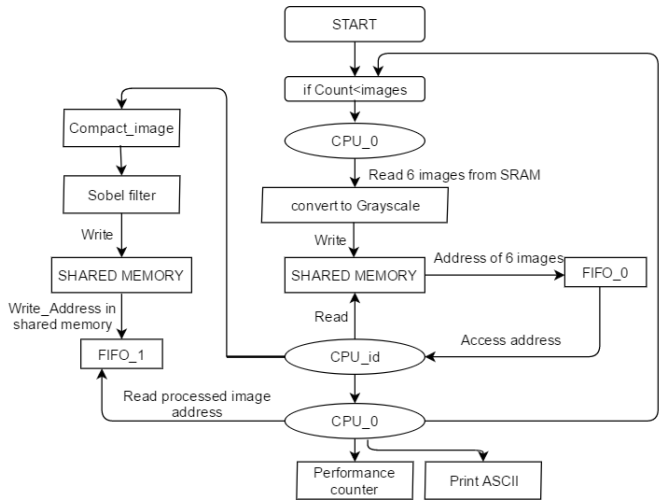

Fig. 2. Flow of application in performance mode in multiprocessor

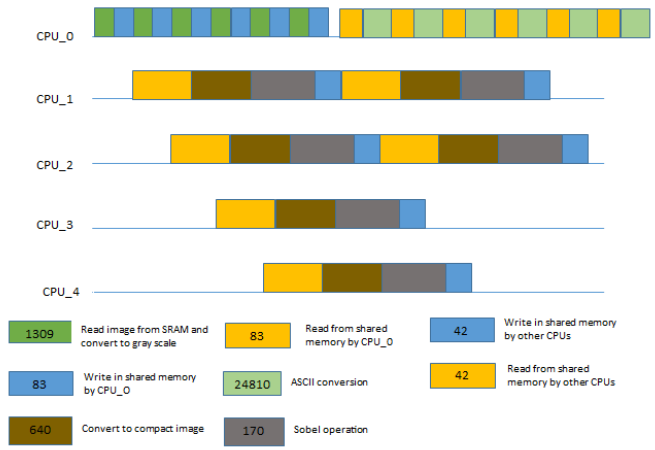

Fig. 3. Scheduling diagram(time in microseconds)

1) Performance Mode: For bringing in the concept of parallelism, we made use of all five processors. As shown in Figure 2 CPU_0 reads 6 images from SRAM, convert them to gray scale and store in shared memory. The start location of each image in shared memory is put in FIFO_0. After writing each image, whichever CPU gets access starts processing the image. When other CPUs get access, they read address location in FIFO and fetch gray image from shared memory. Then the image is converted to compact image and passed on to sobel operation. Once sobel edge detection is done the image is stored back in shared memory and address of processed image is pushed to FIFO_1. In this way, all the other CPUs work independently by grabbing different images from shared memory and process it and write back to shared memory. CPU_0 will be continously checking the status of FIFO_1 and when it is not empty, it grabs the address, reads processed image from shared memory, store in SRAM and convert to ASCII. The performance of the application is considered without ASCII printing in terminal. So when the CPU_0 function is called once, at a time 6 images are processed.

2) Debug Mode: In Debug mode,because we have to process images of different sizes, CPU_0 stores 3 images at a time in shared memory instead of 6 images. Remaining functionalities are same as in performance mode.

B. Profiling and Time behaviour

Scheduling diagram in Figure 3 represents the timing schedule for each CPU for 1 cycle. CPU 0 reads 6 images from SRAM, converts into grayscale and writes in shared memory, and other CPU's start processing as soon as they find an image in shared memory and store the processed image in shared memory. Once images have written, CPU 0 starts reading the processed images one by one. And the the cycle repeats for next 6 images. The CPU 0 keeps writing to the shared memory in it's idle time and since more than 4 images are written at a time, no CPU starvs for image i.e., next image is available for other CPUs to start processing. The time is calculated for all operations in microseconds.

	Single core (RTOS)	Single core (Bare metal)	Multicore
Throughput (s-1)	174	179	473
SRAM (bytes)	145196	128012	45716
On Chip CPU_1(bytes)	_	_	3332
On Chip CPU_2(bytes)	_	_	3332
On Chip CPU_3(bytes)	_	_	3332
On Chip CPU_4(bytes)	_	_	3332
On Chip shared(bytes)	0	0	7248
Total memory (bytes)	145196	128012	66292

Fig. 4. Memory footprint and Throughput for 32 X 32 images processed per second

III. Memory footprint and Throughput

The application is run on three different processor designs and the performance in each is analysed.We analysed the size of compiled codes using the .elf file. In Figure 4, the throughput is calculated considering only 32x32 images and it doesn't consider time taken for printing ASCII.

IV. Code optimization and approximation techniques

Code size is always a concern for embedded systems developers, because there is a cost associated with the memory device that stores code. The ability to control and reduce code size is important in controlling this cost.Code size plays an important role in controlling the cost of memory device that stores the code. The HAL environment is designed to include only those features that we request, minimizing the code footprint. From the given scripts, we studied and analysed the optimizations used. We rechecked if they are really required, and also, what other optimizations can possibly be added.

1. In sobel operation, to calculate gradiant magnitude instead of using square root function- sqrt(Gx^2 + Gy^2), we adopted modulus function- $|Gx| + |Gy|$.

2. Instead of multiplication, left shift operation and instead of division right shift has been performed. For example: while

converting RGB to grayscale, instead of standard formula as (R+G+B)/3, we approximated to (R+G+B)>>1. Also, while multiplying with gradiant matrices in sobel we have used <<1 for multiplication by 2.

3. For all read/write operations to/from the shared memory, the data is stored/read as 8 bytes instead of 1 byte. For example: while copying images from SRAM to shared memory, we have copied 8 bytes at a time by typecasting the pointer to the shared memory.

4. Selection of data types are specific according to the optimum requirement. For storing the pixel values, arrays of type alt_u8 were used. On the other hand the variables defined for image size, for looping were of alt_u32.

5. Loop unrolling technique has been used wherever possible. We have reduced the iterations of forloop by loop unrolling during grayscale conversions, shared memory operations, etc.

6. Instead of comparing value in forloop with larger numbers, we have compared the value with zero because it is less time consuming. For example: for(i=0;i<n;i++) was replaced with for(i=n;i>0;i- -)

V. Conclusion

The digital image processing application is implemented on multiprocessor in three different ways and the performance in terms of throughput and memory footprint is analysed. The possible optimizations that could be performed on the application is found out and performance is optimized.

References

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

[2] Nios II Architecture, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf