Laboratory-2 Report Version-2 (Thursday Afternoon)

IL2212 Embedded Software

Udayan Prabir Sinha
School of Information & Communication Technology
KTH Royal Institute of Technology,
Stockholm, Sweden-10044
upsinha@kth.se

Anirudh Shetty
School of Information & Communication Technology
KTH Royal Institute of Technology,
Stockholm, Sweden-10044
anirudhs@kth.se

***Abstract*—This report summarizes the work done on implementing the required image processing algorithm in typical media streaming scenario on the Nios-2® platform under different sets of constraints. The main focus is on meeting the required throughput while maintaining a low memory footprint.**

I. Introduction.

The report has been divided into four parts based on the various scenarios in which the algorithm has been implemented, at first we discuss briefly about the functionality of the application being implemented. And then we discuss about our bare metal & RTOS implementations on a single Nios-2® core. After that we discuss our bare metal implementations on a multi-core Nios-2® platform. And finally we conclude summarizing the results obtained.

II. Description of the application requirement.

The application is a typical media streaming application which is required to loop over a sequence of RGB images of different sizes found in the SRAM (stored in a .ppm format). It is then supposed to convert each RGB image into a grayscale image, down-size each grayscale image by a factor of two and then scale the intensity values so that up to 16 different intensity levels can be displayed.

Edge-detection is to be performed on each downsized image using 3x3 Sobel filter kernels. The net resulting output is to be scaled and converted to an ASCII image. This ASCII image is to be stored back in SRAM in a format similar to the .pgm format.

Program must have two modes of operation – debug & performance.

III. Single Nios-2® core

A. *Bare metal implementation*

The flowchart of the program flow is shown below:

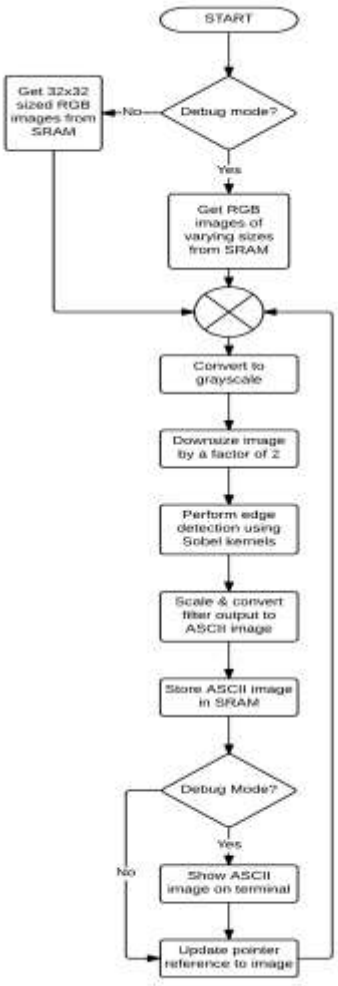

Fig. 1. Flowchart of program flow for bare metal implementation on a single-core Nios-2®

The main functions are mostly combined for fast processing, and the switching between debug and performance modes is achieved with the use of macros. A performance counter is used to measure the execution time for processing 420 images. This is used to calculate the throughput.

The ASCII image is stored back in the SRAM in a format similar to the PGM format except that the ASCII characters are stored instead of gray level values. The square root function used to calculate the net gradient output of the Sobel filters is approximated using Newton-Raphson's method. All other optimizations and approximations are discussed in detail in multi core processor implementation explanation.

B. *RTOS implementation*

The same algorithm shown in Fig. 1 is implemented using MicroC OS-2. The only difference being the additional initialization required to use the OS scheduler.

The entire algorithm is implemented in one single task which is then scheduled using the OS. Use of multiple tasks would only cripple the throughput due to the overhead required in context switching.

IV. MULTI-CORE NIOS-2® - BARE METAL IMPLEMENTATION

Two different programs were developed independently for the multi-core platform. Out of the two, the 2nd implementation exceeds the minimum required throughput of 420 images/second due to the additional optimizations being implemented into it.

The single-core implementations were then optimized based on this 2nd implementation on the multi-core platform.

A. *Implementation-1*

The flow chart for the 1st multi-core implementation of the above algorithm shows the flow of data among the five Nios-2® cores:

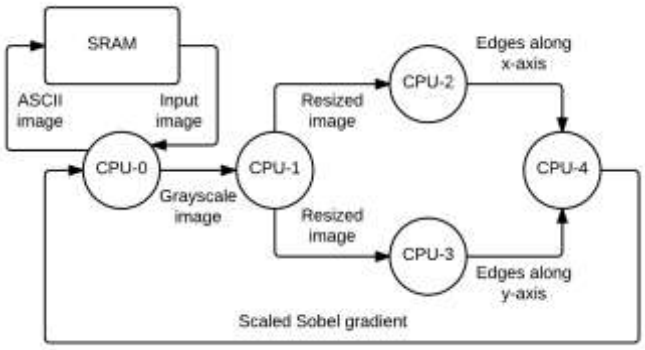

Fig. 2. Flowchart of program flow for bare metal implementation on a multi-core Nios-2® platform

CPU-0: Access the input RGB images stored in the SRAM, converts them to grayscale and stores them in the shared memory. It then takes the scaled gradient image from CPU-4, converts it to an ASCII image and stores it back into the SRAM (and displays it as well if the system is in debug mode).

The following formula is required to convert each RGB pixel to a grayscale one:

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

The above mentioned floating point multiplication has been approximated with bit-shifting improve the throughput.

CPU-1: Downsizes the grayscale image from CPU-0 by a factor of two and stores the result back in the shared memory. A similar bit-shifting operation is used here to optimize the resizing operation.

CPU-2 & CPU-3: Performs Sobel-based edge detection on the downsized grayscale image provide by CPU-1 in the x and y directions respectively and stored their respective results in the shared memory. A generic 2D-convolution algorithm has been used here for edge-detection.

CPU-4: Calculates the net gradient from the filter outputs of CPU-2 & CPU-3, appropriately scales the result and stores the result in shared memory.

The gradient is calculated with the following equation:

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

This equation has been approximated with as explained in [2]:

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

Bit-shifting is used once again to efficiently scale the resulting gradient from the equation above. All the output images from the various CPUs are stored in various segments of the shared memory allowing simultaneous access to the different segments without the possibility of hazards. The access to each segment is controlled with the help of flags (six in total).

Maximum possible shared memory usage:

The maximum possible image size is 40x40.

Output of CPU-0: (40*40) +3 =	1603 bytes
Output of CPU-1: (20*20) +2 =	402 bytes
Output of CPU-2: ((20*20) +2) *2) =	804 bytes
Output of CPU-3: ((20*20) +2) *2) =	804 bytes
Output of CPU-4: (20*20) +2 =	402 bytes
Flags: (6*4) =	24 bytes
	4039 bytes

B. Implementation-2

The flow chart for the 2nd multi-core implementation of the above algorithm shows the flow of data among the five Nios-2® cores:

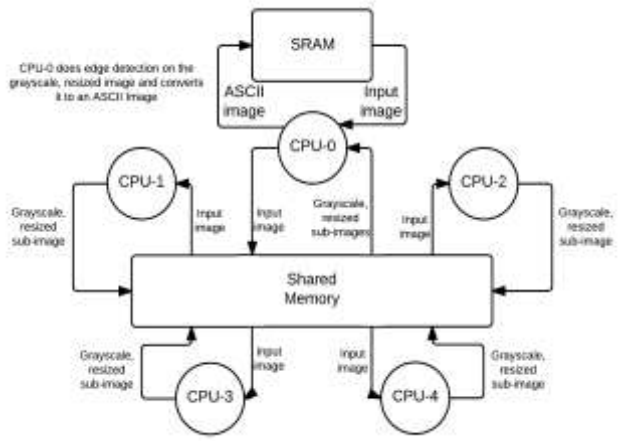

Fig. 3. Flowchart of program flow for optimized bare metal implementation on a multi-core Nios-2® platform

In the second implementation all the values or offsets that can be pre-calculated are written in one common header file called "**offset.h**" and accessed by all 5 CPUs, which makes the code size small and the design easy to implement and understand.

Also the code for common modules between all 5 CPUs, like gray scale conversion etc.., are written in a common .inc file called "**processor.inc**" and it is accessed by all 5 CPUs. And each CPU identifies itself with a unique ID to access the part of code only it is responsible for.

CPU-0: Accesses the input RGB images stored in the SRAM, and stores them in the shared memory. Then the image in shared memory is divided in to 5 parts using memory locations address and size of the images and each of the 5 CPU works on its part of the image and stores it back to shared memory.

CPU-0 signals to other CPUs to start working on the image using flags in shared memory addresses, when it finishes fetching the image. Depending on the size of the picture it is decided whether CPU-0 will also be working on initial processing of the picture or not. It is noticed that keeping CPU idle for a little while is far better than keeping 4 other CPUs idle.

In first part of processing the image is converted to gray scale and simultaneously interpolated.

Once all other CPUs signal completion of their part of processing using flags, CPU-0 then simultaneously converts the interpolated image to ASCII image and stores it back into the SRAM (and displays it as well if the system is in debug mode).

Optimization Here: The Image is transferred to shared memory using integer pointer (4 Bytes) to utilize the full data bus bandwidth of 32 bits for each fetch.

Also the 'for' loop is unrolled to include four such fetches (16 Bytes) in each loop, the unrolling number is calculated based on picture sizes and common dividers of total pixels.

The total loops are also calculated dynamically for each picture based on picture's size.

The grayscale conversion and interpolation is combined to work on all corresponding 12 pixels at a time, and thus each iteration in the for loop jumps over 6 columns of input and two rows of input image and stores final values of the input immediately in the shared memory, which can be accessed by CPU-0 for Sobel filter.

If, S=Shared Memory first address, R = Current Row offset, C=Total Columns. Then the formula for gray scale and interpolation is.

$$Tred = *(S+R) + *(S+R+3) + *(S+R+C) + *(S+R+C+3). \quad (4)$$

$$Tgreen = *(S+R+1) + *(S+R+4) + *(S+R+C+1) + *(S+R+C+4). \quad (5)$$

$$Tblue = *(S+R+2) + *(S+R+5) + *(S+R+C+2) + *(S+R+C+5). \quad (6)$$

$$*(StoreAddress) = ((((Tred<<2)+Tred)>>4) + (((Tgreen<<3) + Tgreen)>>4) + (Tblue>>3))>>2 \quad (7)$$

The conversions are done using only bit shifts.

To utilize the high speed of SRAM and avoid writing and reading from both shared and on chip memory, the Sobel filter is implemented combining ASCII conversions and final image printing in CPU-0 only.

Here the Sobel filter multiplications are broken down to six simple additions/subtractions with or without bit shifts. The result is not saved as only 1 CPU is working on it and directly used in switch statement to find corresponding ASCII character between 0-15 of brightness is stored directly in to SRAM.

If, S=Storing memory first address, R = Current Row offset, C=Total interpolated Columns. Then the formula for Sobel's Gx and Gy are,

$$Gx = *(S+R) - *(S+R+2) + (*(S+R+C)<<1) - (*(S+R+C+2)<<1) + *(S+R+(C<<1)) - *(S+R+(C<<1)+2). \quad (8)$$

Gy = *(S+R) + (*(S+R+1)<<1) + *(S+R+2) - *(S+R+(C<<1)) - (*(S+R+(C<<1)+1)<<1) - *(S+R+(C<<1)+2). (9)

But the values are directly used in Switch statement as Switch (abs(Gx)+abs(Gy)>>6). The shifting is done to convert the values to a spread of 0-15 brightness values.

The Sobel filter was also tested for implementation using 5 CPUs but the requirement of accessing shared memory multiple times, with mandatory SRAM write in the end, and ASCII conversion from shared memory, complexity of the code, all combines to reduce the advantage of multiple CPUs for this particular module. The code still is available in the project in disabled form.

The squaring and square root implementations are approximated using [2] and the loss were minimal as seen in the image output.

Other Optimizations: All the memory accesses and calculations on images are done directly using pointers and pointer arithmetic.

The division of images between CPU means it has to be decided that how many rows each CPU gets for different images to work on. And what is the memory location of the pixel, the CPU should start working on and at what is the memory location of the values the CPU wants to store after calculations. But these values and offsets can be pre calculated to save a tremendous amount of time.

There are 5 CPUs and 5 unique sizes, so 25 combinations of values are to be calculated and also be available to be accessed with ease.

First the unique sizes are converted and represented using numbers 0-4. The conversions used the formula (SizeY-22 >>1), and array with corresponding replacements.

size[actualSizeY-22 >>1]={1, 0,0, 3,0, 2,0,0,0, 4}.

Here we can see size 32 would become size[32-22/2] = size[5]= 2. So value of 32 corresponds to 2 for any future calculations.

Number of rows from image input each CPU would work on based on 5 unique image sizes. Row[CPU][Size].

Row [5][5]={ 0, 6, 0, 4, 0,
6, 4, 8, 6, 10,
6, 4, 8, 6, 10,
6, 4, 8, 6, 10,
6, 4, 8, 6, 10}.

Memory address each CPU should start on based on 5 unique picture sizes. AdressOffset[CPU][Size].

AdressOffset [5][5]={ 0, 0, 0, 0, 0,
0, 576, 0, 480, 0,
432, 960, 768, 1200, 1200,
864, 1344, 1536, 1920, 2400,
1296, 1728, 2304, 2640, 3600}.

Offset of storing address offset each CPU should start storing values, based on 5 unique picture sizes. StoringAddress[cpu][size].

StoringOffset[5][5]={ 0, 0, 0, 0, 0,
0, 48, 0, 40, 0,
36, 80, 64, 100, 100,
72, 112, 128, 160, 200,
108, 144, 192, 220, 300}.

Here the number of rows, and memory offsets for input and output of each CPU for each image is stored in three 2 dimensional arrays of size [5]x[5] . The arrays can be accessed using array[CallingCPU]x[UniqueImageSize], which gives corresponding pre calculated values. The offset values were calculated using a separate C program beforehand, and so need not be calculated while processing the image.

The optimizations are also explained in code using small comments in the code, and the same can be used to understand the code.

CPU-1, CPU-2, CPU-3, CPU-4: Along with CPU-0 these 4 CPUs find gray scale values, and the do interpolation on the Grayscale values simultaneously.

All shared memory accesses are controlled with the help of flags set and reset in pre-decided shared memory locations.

All the output images from the various CPUs are stored in various segments of the shared memory.

1. Maximum possible shared memory usage:

   The maximum possible image size is 40x40.

   Initial Image: (40*40) +3 = 1603 bytes

   Intermediate Results: (20*(20/5)) +2 = 82 bytes

   1685 bytes.

V. CONCLUSION

The following table summarizes the throughput and memory footprint of the code written:

TABLE I. THROUGHPUT & MEMORY FOOTPRINT SUMMARY

	Single-core (Bare metal)	**Single-core (RTOS)**	**Multi-core**	
			1st Implementation.	**2nd Implementation**
Throughput (s^{-1})	245	217	115	1678
SRAM (bytes)	33856	89772	33832	45300
On-chip CPU-1 (bytes)	-	-	1840	5180
On-chip CPU-2 (bytes)	-	-	2048	5180
On-chip CPU-3 (bytes)	-	-	2048	5180
On-chip CPU-4 (bytes)	-	-	1528	5180
On-chip Shared (bytes)	-	-	4039	1685
Total (bytes)	42176	161740	56187	67705

The overhead imposed by the OS scheduler in the RTOS implementation is the primary reason behind the lesser throughput compared to its bare-metal counterpart. Using an RTOS also increases the memory footprint.

The 1st multi-core implementation can be further optimized by using 32-bit pointers instead of 8-bit pointers to access the memory. This will allow full use of the memory bandwidth as the memory is accessed in 4-byte chunks. Furthermore, the number of memory accesses would be four times lesser then.

Also, the RGB-to-gray-conversion and the downsizing operations can be merged into a single set of loops instead of doing them in two separate sets of loops.

This can be further used to divide the image into two parts and then do the RGB-to-gray-conversion and the downsizing operations on both of these parts simultaneously on CPU-0 and CPU-1.

The 2D convolution algorithm running on CPU-2 and CPU-3 can also be optimized by substituting the MAC operations with simple additions and subtractions as done for the 2nd implementation.

The 2nd multi-core implementation's throughput is the maximum I could reach.

2nd multi-core implementation has even more optimizations from its singe core counterpart, which were result of countless iteration on multi core code; hence the improvement in throughput is not linear.

Some of the ASCII images produced by both multi-core implementations have been shown below:

Implementaion 1.

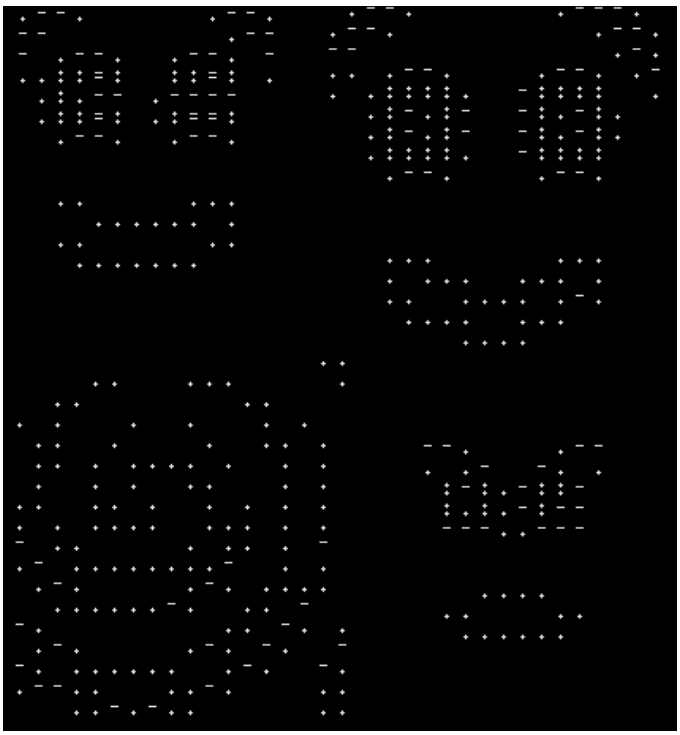

Fig. 4. ASCII images produced by the **1st implementation.**

Implementaion 2.

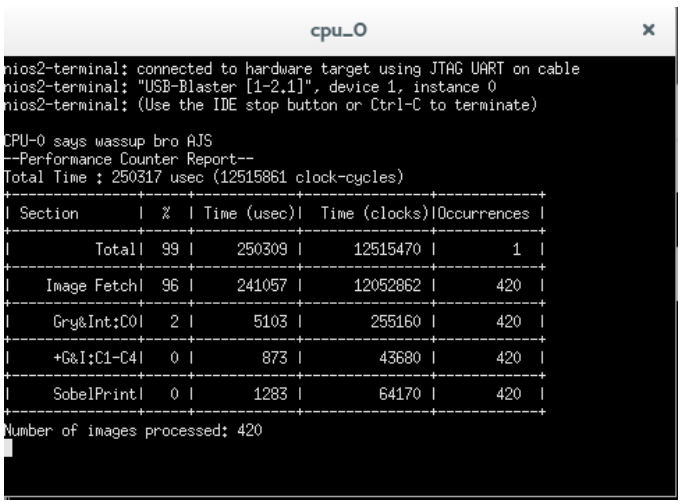

Fig. 5. **2nd Implementation**, Debug Mode 0.

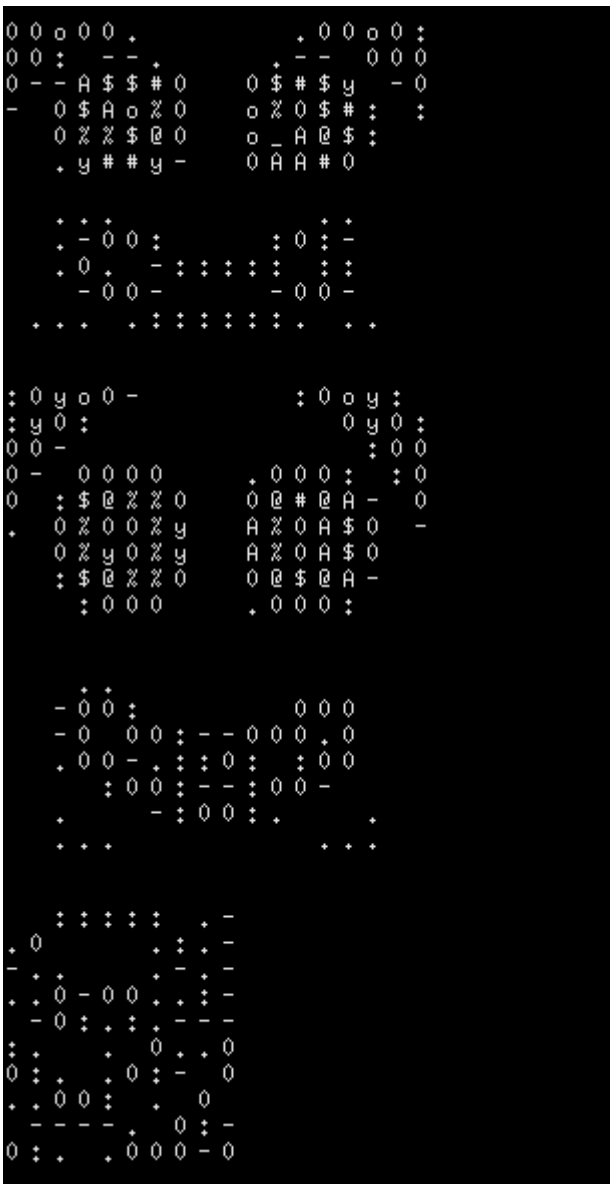

Fig. 6. ASCII images produced by the 2nd implementation.

Fig. 7. ASCII images produced by the 2nd implementation.

REFERENCES

[1] Introduction to the Altera Nios-2 Soft Processor.

[2] V. Kamatchi Sundari & M. Manikandan, P.Prakash, "FPGA IMPLEMENTATION of SOBEL EDGE DETECTOR," International Journal of Advances in Science and Technology, ISBN 2348-5426.

[3] Kieras, David, “C Header File Guidelines”, EECS Dept., University of Michigan

[4] Nios-2 Processor Reference Handbook.

[5] Nios-2 Software Developer’s Handbook.

[6] Embedded Peripherals IP User Guide.

[7] DE2 Development & Education Board, User Manual.