IL2212 Embedded Software
Lab 2 Report
Session: Friday Morning

Yuefeng Wu
Embedded Systems
EIT Digital Master School
Personal Number: 921116-6039
Email: yuefeng@kth.se

Renzhi Xing
Embedded Systems
KTH, Royal Institute of Technology
Personal Number: 931011-2025
Email: renzhi@kth.se

***Abstract*—This report is a laboratory record of Lab 2 in IL2212 Embedded Software VT2016. In this report, we will present the implementation procedures of required functions: loading RGB images to on-chip memory, turning the images into gray-scale format, resizing the converted images, performing edge detection algorithm on the resized images and finally saving ASCII images into shared on-chip memory. Additionally, we will give a relatively detailed analysis of the application's throughput and document optimizations.**

I. Introduction

In this laboratory, we are required to implement a image processing algorithm on three systems based on different hardware configurations. On the Bare-Metal single core and μ/C-OS II single core systems, we implemented required algorithm and verified the results, which consolidated our fundamental knowledge of the NIOS II and μ/C-OS II. And additionally we are required to transplant our code onto a multiprocessor system. During the transplantation procedure, we can get a deep insight of multiprocessing, hardware architecture and code optimization.

II. Single Processor Application Description

In the skeleton of the application, a macro was predefined to distinguish debug mode and performance mode. We implemented the debug-mode functions in single processor platforms(both on Bare-Metal and μC-OS II) and the performance mode code on multiprocessor platform.

A. Loading images to shared memory

This function is given by the skeleton. We used the similar coding theme in other functions—using pointers instead of arrays to access and process matrix.

On the other hand, according to the qsys file and the auto-compilation script, the default code space of cpu_0 is SRAM, which means that the default storage location is SRAM. The images are stored in the *image.h* located in SRAM. And the function teaches us to use address to access the shared on-chip memory.

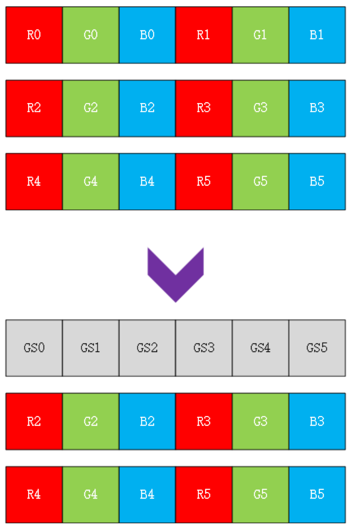

Fig. 1. On-chip memory reuse strategy for gray scale

However, since the access speed of SRAM is significantly slower than the on-chip

B. Converting RGB images into grayscale

The computation formula of the conversion is:

$$Y = 0.3125 \times R + 0.5625 \times G + 0.125 \times B$$

After some experiment, we found that this function consumed a relatively larger amount of storage space and execution time. We currently have made two main optimizations.

- **Reuse of shared on-chip memory**
  According to the memory management strategy, the two dimension array is stored as a line of data blocks. And the gray scale algorithm is to combine three 8 bytes data into one. So we can reuse the RGB image's part of on-chip memory. We drew a simple graph to demonstrate the reusing procedure.
- **Avoiding float computation**
  After several executions, we found that the float computa-

TABLE I
COMPARISON BETWEEN INTEGER AND FLOAT COMPUTATION

Image size	Float(sec)	Integer(sec)
24×24	0.788	0.269
32×22	1.069	0.241
32×22	1.448	0.292
40×28	1.898	0.316
40×40	2.251	0.412

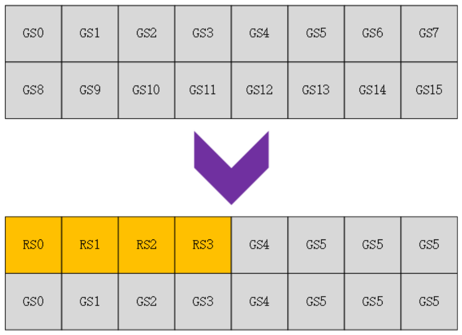

Fig. 2. On-chip memory reuse strategy for resizing

tion required much more CPU time compared to integer. To avoid float computation, we firstly made the RGB value multiplied by 5 9 and 2 respectively and summed them up. Then we divide the sum with 16. All the multiplications and divisions were all replaced by left-shifting and right-shifting.
To indicate the effect the optimization, we built a table for the comparison.

C. Resizing the grayscale images

This function can also reuse the on-chip memory. We used this feature for optimization and drew a simple graph to demonstrate the reuse.

D. Edge detection with Sobel core

Firstly, we made an approximation on the formula of the gradient magnitude. We replaced

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

with

$$G = |G_x| + |G_y|$$

to reduce the computational work. After we made this approximation, we also removed redundant absolute value computation since calling for *abs()* function would also increase execution time.

We divided the image matrix into 9 parts, which is described in the figure. In some cases, G_x or G_y will always be positive or negative, then we removed the calling. For example, at the point $(0, 0)$, the G_x and G_y will always be positive.

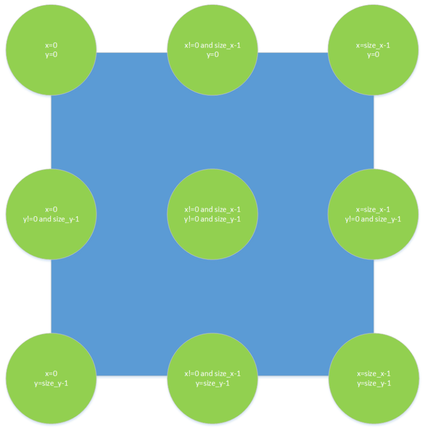

Fig. 3. Parts of the image matrix

TABLE II
SHARED ON-CHIP MEMORY MAPPING

Content	Start(dec)	End(dec)	Size(byte)
RGB image	0	4095	4096
Grayscale image	4096	5119	1024
Resized image	5120	5375	256
ASCII image	6144	6400	256
Semaphore 1-8	7168	7207	32
Image information	8184	8186	3

III. MULTI-PROCESSOR APPLICATION DESCRIPTION

A. Processor Assignment

The final version of application architecture is different from the firstly proposed structure. In this version, we make cpu_0 responsible for loading image from SRAM to shared on-chip memory and storing ASCII image to SRAM. And cpu_1 to cpu_4 are designed to grayscale, resize representative part of the RGB image in shared on-chip memory. Unfortunately, due to the synchronization issue, we eventually assigned the computation of edge detection and ASCII image only to cpu_1. However, this compromise has slight influence on the throughput, which will be talked in detail in the section of *Software Pipeline*.

B. Shared Memory Mapping

In the predefined multiprocessor system, a 8 KByte shared on-chip memory builds a bridge for the communications between five cores. Since it is a part of on-chip memory, its speed of access is much faster than SRAM.

We divided the shared memory into five parts–RGB image, grayscale image, resized image, semaphores and image information. We built a table to indicate the mapping strategy more directly.

C. Synchronization

As mentioned above, we used some part of shared on-chip memory to implement semaphores.

Semaphore is a good mechanism to synchronize threads or cores for accessing the shared resources. In our application, we

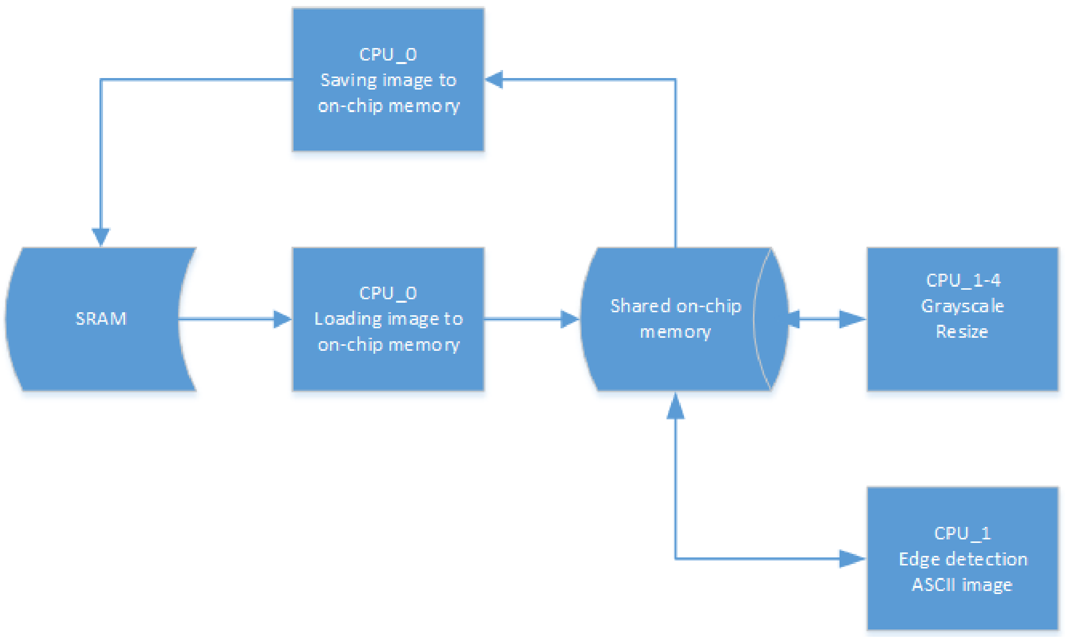

Fig. 4. Multiprocessor application architecture

used semaphores to avoid simultaneous reading and writing to the same part of memory from different cores. For example, if cpu_1 is grayscaling the image in the shared memory and cpu_0 is loading a new image to shared memory at the same time, there will be a inevitable collision that make the generated grayscale image incorrect.

In our application, we designed two kinds of semaphores–*key semaphore and feedback semaphore*.

Key semaphore is the key to start the next procedure. After one precedent procedure is finished and the data required by following procedure is prepared, the precedent procedure sends a key semaphore to the following one to start it. For example, *SEMAPHORE_3* in our application is sent by grayscale procedure to resizing after the loaded image is grayscaled. The key semaphores need to be initialized with 0 in application initialization.

Feedback semaphore is the feedback from following procedure to precedent one, which entitle the precedent procedure to write data to allocated memory. For example, *SEMAPHORE_8* is sent by Sobel value computation procedure to resizing to inform it to continue resizing one new image. The feedback semaphores need to be initialized with 1, otherwise the application won't execute properly.

We drew a interference graph to demonstrate the synchronization mechanism.

D. Software Pipeline

The most time-consuming procedure in our application is loading RGB image to shared memory, since the amount of data is relatively larger and access speed of SRAM is slower than on-chip memory. After optimizations, we finally reduced the execution time of loading image to about 870 μs, which

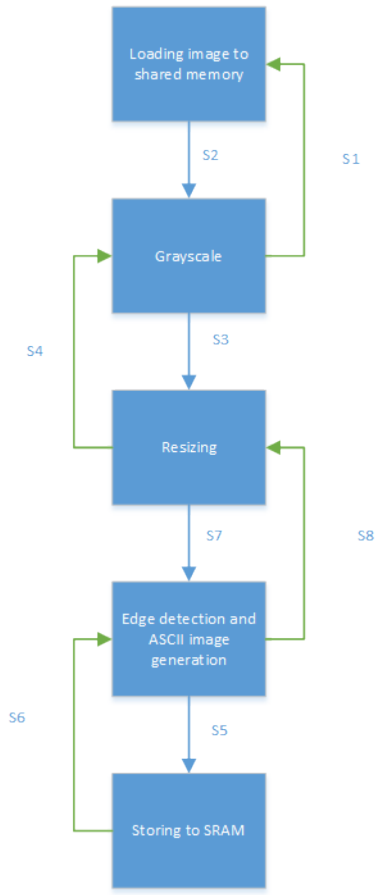
Loading image to shared memory
S2
S1
Grayscale
S4
S3
Resizing
S7
S8
Edge detection and ASCII image generation
S6
S5
Storing to SRAM

Fig. 5. Semaphore interference graph. Blue for key semaphores, green for feedback semaphores.

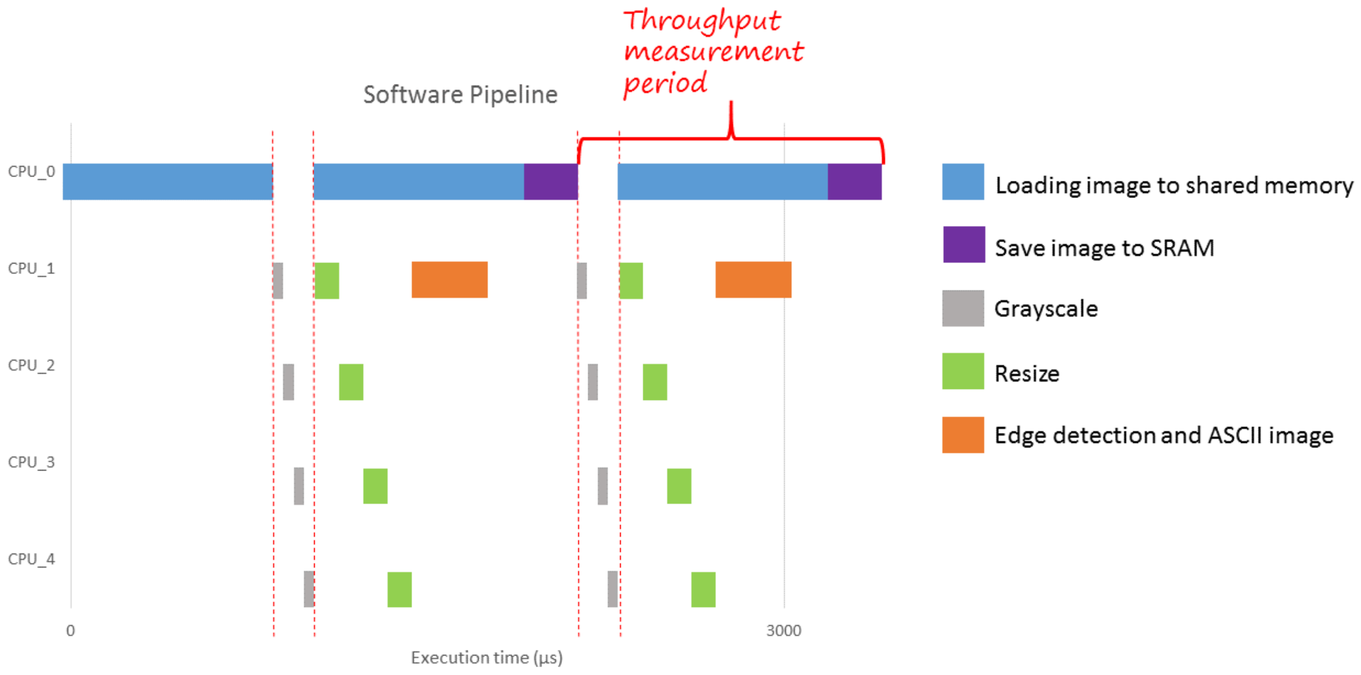

Fig. 6. Software pipeline of the penta-core system

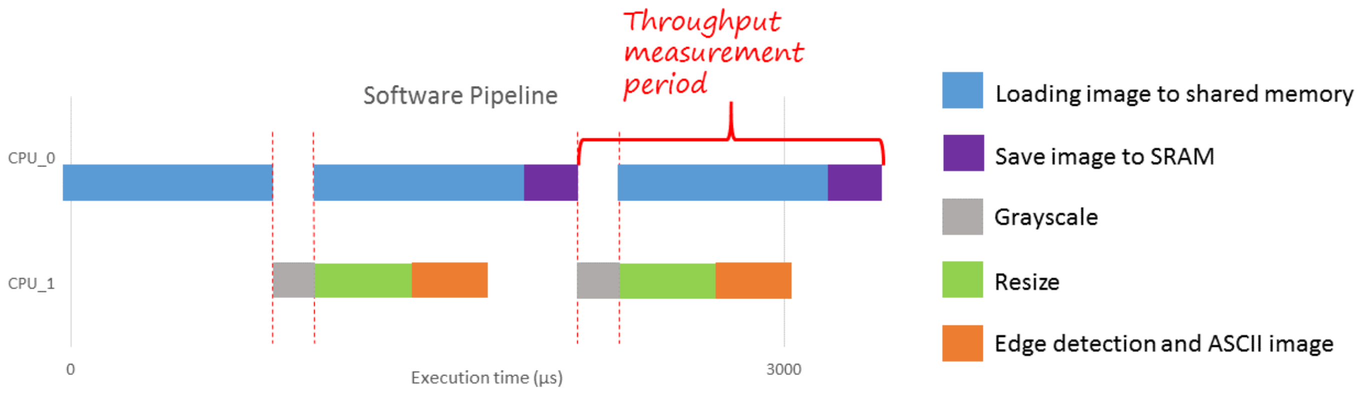

Fig. 7. Software pipeline of the dual-core system

still constitute most part of total execution time. To reduce execution time, we built a software pipeline to assign other jobs to cpu_1-cpu_4. We also drew a graph to demonstrate the pipeline. In the size-based optimized code, we used dual-core system, we also drew the pipeline graph.

As the graph indicated, the cpu_1-cpu_4 will do resizing, compute Sobel value and generate ASCII image while cpu_0 is loading image to on-chip memory. As long as the total execution time of multiple procedures of cpu_1-cpu_4 doesn't exceed the image-loading time of cpu_0, there will be nearly no influence on the final execution time. This is the reason why we eventually assign the Sobel and ASCII image computation only to cpu_1 to avoid synchronization issues.

IV. Final Throughput

We measured the final throughput by calculating the average execution time of three 32×32 images and get its multiplicative inverse. Here is the throughput table. All the code is running in performance mode. And another thing to be mentioned is that the throughputs of single-core applications (μ/C-OS II and bare metal) is based on unoptimized codes.

V. Optimizations

In this section, we only talk about three significant optimizations that cast a huge influence on the total execution time.

TABLE III
FINAL THROUGHPUT

	Single core (RTOS)	Single Core (Bare-Metal)	Multicore (5 cores)	Multicore (2 cores)
Throughput (s^{-1})	**103**	**197**	**1114**	**1114**
SRAM (Byte)	146484	42836	45116	44556
OnChip CPU 1 (Byte)	-	-	2420	2360
OnChip CPU 2 (Byte)	-	-	1696	-
OnChip CPU 3 (Byte)	-	-	1696	-
OnChip CPU 4 (Byte)	-	-	1664	-
Shared OnChip (Byte)	3075	3075	5667	5385
Total Memory (Byte)	149559	45911	58517	52301

A. Utilization of 32bit Data Bus

This optimization significantly reduces the execution time of loading image to shared on-chip memory by more than 2000 μs. The given *sram2sm_p3()* function transmits the image pixels by 8 bit data every time. However, the data bus of the system is 32bit. We defined two integer (32bit) pointers which point to the data in SRAM and shared on-chip memory respectively and transmitted data between the pointers. This optimization cuts down the execution time by 4 times since the iteration of transmission is reduces.

B. Replacing Multiplication and Division

This optimization replaces time-consuming multiplication and division with shifting. CPUs can perform shifting much faster than multiplication and division. Since the Sobel computation contains a lot of multiplication, this optimization has an apparent influence on the final execution time.

C. Macros

We defined a lot of macros to reduce code size and debug conveniently. On the other hand, the well-defined macros can also make the source code more readable.

VI. DISCUSSION AND ANALYSIS

In this section, we will give some discussion and analysis on our final result. The points discussed are questions emerged while developing the applications.

A. Dual-core V.S. Penta-core

From the *Table III* we can see that the final through puts of dual-core and penta-core platforms are the same (actually, the dual-core is a little bit faster if we check the clock circles). The phenomenon can be explained by the access limitation of shared on-chip memory. According to the *Network on Chip* architecture, the data need to be transmitted in ”packages” and the slave (shared on-chip memory) can only accept one package at one time[1]. So even if we hope to let the cpu_1 to cpu_4 grayscale the image in shared on-chip memory simultaneously, they still have to queue to get own turn to read and write the shared memory. So they final throughputs are nearly the same.

B. Different Execution Time Between cpu_0 and cpu_1-4

Firstly, we built a table for the execution time on the single core (cpu_0) of every step. From the table we can see that grayscaling, resizing and edge detecting a 32×32 image need a much longer time than loading it from SRAM to shared on-chip memory.

However, since the default code space for cpu_1-4 is on-chip memory while cpu_0's is SRAM and SRAM performs much slower than on-chip memory, writing and reading a temporary variable can be significantly faster on cpu_1-4. So the procedures can consume less execution time on cpu_1-4 than on cpu_0.

TABLE IV
THE EXECUTION TIME OF PROCEDURES ON CPU_0

Section	Execution time (μsec)
Loading from SRAM	3206
Grayscale	4218
Resizing	1225
Sobel	2733
Total	11382

C. Further Optimization of Loading Image from SRAM

We firstly got a throughput of 954 s^{-1} and the execution time of loading a image from SRAM was about 950 μs. We finally decrease the execution time to around 870 μs by eliminating the function call of *IOWR_32DIRECT()* and using 32-bit data pointers to transmit data directly.

Elimination of function calls can reduce the time consumption because the calls need extra register and memory operations according to the calling conventions of compilers. To process a function call, an update of stack frames is needed, which consumes some time[2].

REFERENCES

[1] A. Corporation, “Applying the benefits of network on a chip architecture to fpga system design.”
[2] A. W. Appel, *Modern compiler implementation in C.* Cambridge university press, 2004.