Embedded Software

Lab 2

Tanoh Henry Gertrude

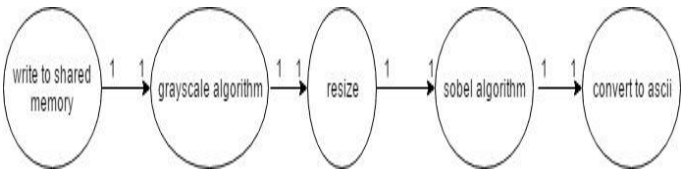
Farbod Haselzadeh

I. INTRODUCTION

This report summarizes our work concerning the laboratory 2 of the Embedded Software course. The lab2 consists of developing a streaming application software. We are asked to develop the application on three different architectures (Single Core with RTOS, Single core without OS, Multicore without OS). Also in this laboratory, we had to meet a throughput constraint and a memory footprint constraint.

A. Single Core Solution With RTOS

i. Implementation & Data Flow



The token unit in this data flow graph is picture.
Each node consumes one picture and release one picture.
We mapped every node to a function. The functions are scheduled sequentially. For the sobel algorithm, we do not use arrays, the values of the kernel matrix are hard coded, we use shift operations for multiplication and division. For the square root operation, at this step, we use the normal square operation.
We could have use a pipeline implementation with multiples tasks

ii. Results

The throughput of the RTOS solution is lower than the single bare solution. We think it is because of the overhead generated by the operating systems, i.e the handling of tasks and semaphores.
Our throughput is 0.1441 s^-1.

B. Single Core without OS Solution

i. Implementation & Data Flow

The dataflow graph is the same as the rtos solution.

The token unit in this data flow graph is picture.
Each node consumes one picture and release one picture.
We mapped every node to a function. The functions are scheduled sequentially. For the sobel algorithm, we do not use arrays, the values of the kernel matrix are hard coded, we use shift operations for multiplication and division. For the square root operation, at this step, we use the normal square operation.

ii. Results

The application on the single bare is faster than the RTOS solution but is still low. We have identified that the grayscale operation and the Sobel operator, especially the square root function are the bottlenecks of the application. The throughput reached is the performance mode. Our application did not work at the first session. We will discuss these optimization in the multiprocessor architecture.
The throughput is 0.1402s-1

C. *Multicore Solution without OS*

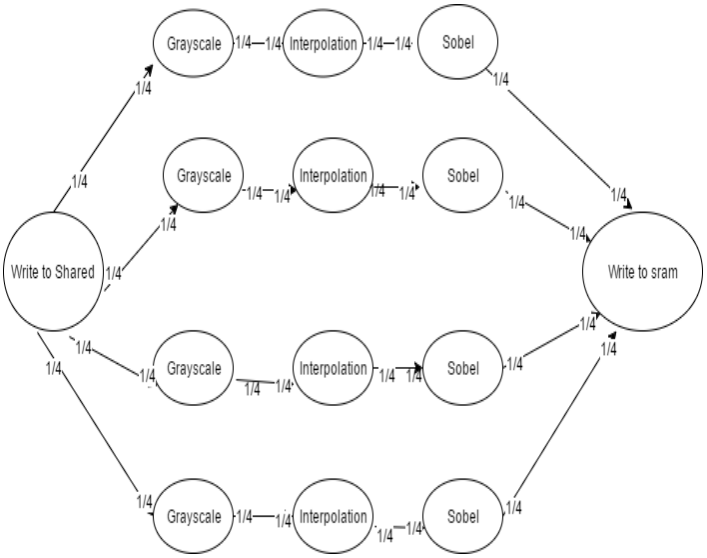
i. Observation

These observations have been made after some optimizations, which will be detailed later in the document.

Sram copy	0.00105s
Grayscale	0.00428s
Inter	0.00144s
Sobel	0.01480s
Write	0.0000s

As one can deduct from the table, the bottleneck of the application are the grayscale operation and the sobel algorithm.

ii. Data Flow Graph



iii. Implementation

a. Scheduling & Mapping

The data flow graph unit is a picture.
We split the tasks grayscale, interpolation and sobels into four tasks. In addition, we can conclude from our data flow graph that the buffer size is one picture. Therefore, we can have two pictures in the shared memory. The shared memory has a size of 8192 bytes. We mapped each channel to one core.
See next page.

iv. Design Methodoly

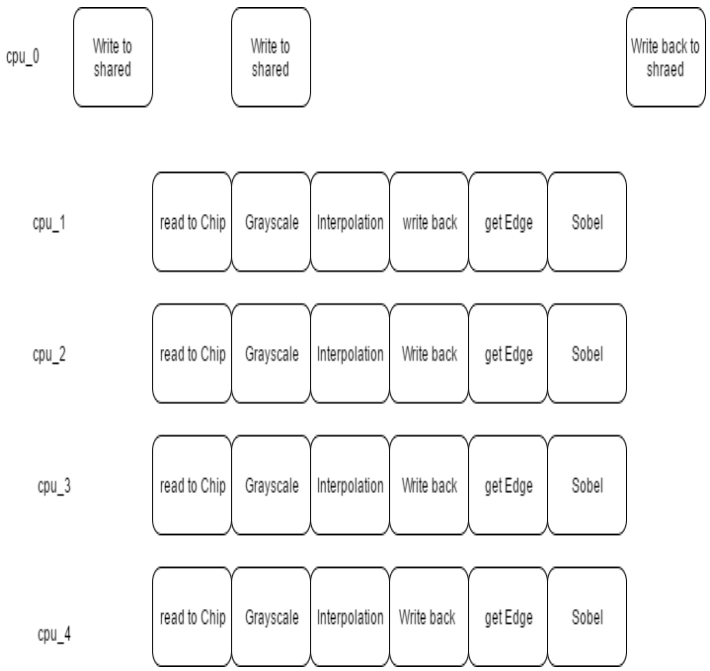
a. Hardware choices

We have noticed a couple of interesting facts:

- The Avalon interface allows multiple master to read/write data to a slave with data corruption, of course at different locations. This is done by slave side arbitration.
- The Avalon interface allows simultaneous multi-master, which allows cou_1, cpu_2, cpu_3, cpu_4 to access their slave while the cpu_0 is writing the shared memory.
- It is faster if cpu_1, for example, works on his on_chip memory because it will be the only one accessing the slave.
- The Nios architecture allows to write data by 4 bytes.

b. Software choices

We split the picture into four quarters. We have one main cpu with sram memory and four cores with a chip memory. They can all communicate through the shared memory. The cpu_1, cpu_2, cpu_3, cpu_4 work on their on chip. We do not have measurements to prove this solution is faster, but with the altera specifications of the Avalon interface, we concluded that it is faster.
We have four synchronizations steps, as one can see in the next diagram.

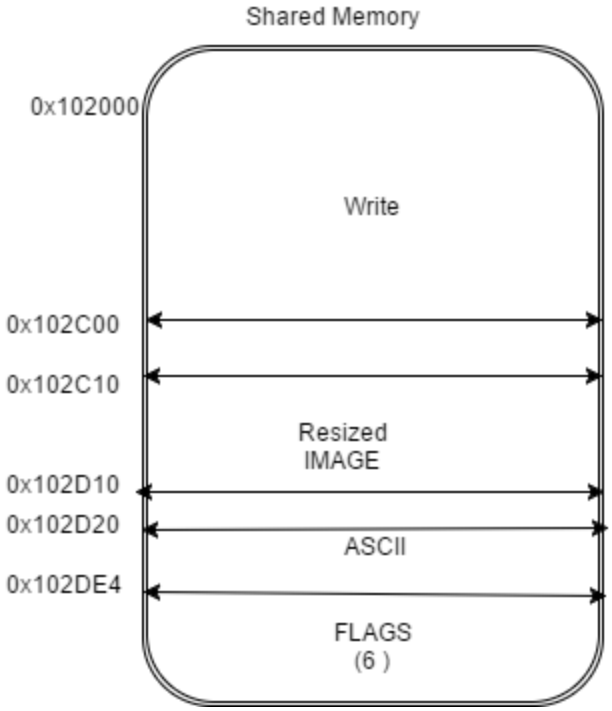


The boxes' size does not represent the time needed to perform these actions.

The synchronization is done by using flags, which stored in the shared memory. The cpu_0 writes always at the same address for the picture. The cpu_1, cpu_2, cpu_3, cpu_4 write back always at the same address. The cpu_0 read the picture back to sram at this address.

We have one flag for the cpu_0 to inform the other cpu that a picture is ready, 4 flags for cpu_1, cpu_2, cpu_3, cpu_4 to inform each other for getting the edge back, 4 flags for cpu_1, cpu_2, cpu_3, cpu_4 to inform cpu_0 that a picture is ready.
The wait is implemented by a while loop reading the flags.
We can exploit parallelism because we allow cpu_1, cpu_2, cpu_3, cpu_4 to work on the quarter of picture while we write to shared memory another picture.

Shared Memory



c. Optimizations

- Write to shared memory: we write by 4 bytes. Use of the attribute aligned((4))
- In grayscale operation: we have used bit shift instead multiplying with float. We are aware that we lose some precision.
- We use bit shift for operations.
- Implementation of square root function: we calculated the maximum value of the gradient after sobel, applied square root and splitted this value into 16. We then matched the value to the corresponding interval, then to the asci character. We also avoid the multiplication of gx and gy in the sobel operator.
- Flag O3

d. Results

We implemented our application only for performance mode because

- We struggled with the fact that with other images, we will need the size data making our data not compact, so not easy to split.
- Also we struggled to perform the single bare/rtos solution so we decided to go for the performance in order to meet the lab deadline. We strongly believe that it is possible to do it.

At this step, our application is not fully functional for the multiprocessor. We are stucked at getting the matching edges for each cpu.

e. Limitations & future optimizations

- Implementation of the debug mode
- The approach requires getting the edges for the sobel, one solution could be that the cpu_0 performs grayscale and interpolation. So we could have 20 images in the shared memory. This way, we eliminate the need of synchronization, and the pipeline will have a better throughput overtime.
- To have a better throughput, maybe the best, we could write two pictures in the shared memory, one after one (32*32*3 + 32*32*3 = 6144). We release cpu_1..4 after the first picture. Cpu_1..4 need this space to work ((16*16 + 14*14) * 2 = 904). We still have space in the shared memory to define our flags. We believe that with this architecture, we eliminate the time cpu_1..4 wait for cpu_0 to finish writing a picture and consume the previous ascii picture because they could store the picture just done at one of the spot for asci images.

D. Throughput and Memory footprint

	RTOS	Bare-Metal	Multicore
Throughput(s-^1)	0.14 41	0.1402	0.00164
SRAM(bytes)	143 KB	125 KB	20KB
OnChip CPU 1			2820 B
OnChip CPU 2			2828 B
OnChip CPU 3			2852 B
OnChip CPU 4			2836 B
OnChip (Shared)			3556 B
Total			34892 B

E. Conclusion

This lab was quite challenging because it required a lot of knowledge of the architecture, good knowledge of pointers and a methodology for software design. As students, we were quite shocked by the time taken to process a floating point, even though in the Nios we do not have a floating-point unit.