NC State University
Department of Electrical and Computer Engineering
ECE 461/561: Spring 2019 (Dr. Dean)
Project #1: Speed Analysis and Optimization

by

<< SALONI SHAMBHUWANI >>

NCSU Honor Pledge: "I have neither given nor received unauthorized aid on this test or assignment."
Student's electronic signature: __Saloni Shambhuwani____
(sign by typing your name)
Course number: _____ECE-561_____

Introduction

In this project I tried to optimize the execution speed of a program which decodes a JPEG image and displays it on the LCD. The base code provided had a speed of 1228 sample counts and on optimization without hashing enable reduced to **434 sample count**. That is the code now runs about 3x times faster. More optimizations can be achieved by improving NextdecodeMCU and huffDecode functions since they are the top 2 functions with highest sample counts. The function which were touched and optimized are: pjpeg_load_from_memory, LCD_Start_Rectangle, LCD_Write_Rectangle_Pixel, LCD_24S_Write_Data, LCD_24S_Write_Command, GPIO* settings for PDOR and Hashing function.

Execution Time Optimization (in order performed, including dead ends (ineffective attempts))

Below is the table of all the optimizations performed to get as close to the expected sample count.(Sample counts with and without hash are both shown):

Step	Otimization description	#Sample count with hash	#sample ount without hash
1	Starter code	1228	
2	compiler to optimize at level 3 (maximum) for time	1050	1014
3	Removing bx and by loop(inner most loops)(Replacing LCD_Plot_Pixel with LCD_Start_Rectangle and LCD_)	558	517
4	PDOR->PCOR/PSOR	505	469
5	Utilising for R,G,B 32-bit space in register(Parallely sending 4 pixels)	488	440
6	Changing all divide (to the power of 2)to shift operations	488	440
7	No-Inlining / Inlining	494	446
8	Initializing b1 and b2 in hashing code and modifying hashing code to validate pixels without exceeding size limit(32KB)	479	440
9	Creating LCD_24S_write_data(my function) to set LCD_D_NC_POS only once for all write calls	477	435
10	Removing	476	434

	GPIO_SetBit(LCD_D_NC_POS) from write data function and adding it at the end of write command function		
11	No Inline for stuffChar()	471	429

Following are the answer to the questions.

Q1 What optimization did you perform? Why did you expect it to help, and by approximately how much?

Q2 How much time did implementing the optimization correctly take? This includes planning, coding, testing, debugging, etc.

Q3 What were the before and after execution sample counts? How much did the performance actually improve (speed-up factor)? If the amount of improvement was unexpected, examine and explain.

Optimization #1: Starter code

The starter code gave 1228 sample count with good image hash and had the areas where it could be optimized. To the starter code multiple optimizations were implemented and new functions were return to hit the performance as optimum as 434 sample count. The most dominating functions in the starter code were LCD_24S_Write_Data, decodeNextMCU and LCD_Plot_Pixel. So optimizing on these functions and making compiler to optimize (if it isn't) was the first thought.

Optimization #2: Compiler to optimize at level -O3 (maximum) for time.

Setting the compiler to optimize at level 3 (maximum) for time and adding the --fpmode=fast option to compiler options helped to get the performance of 1050 sample counts with hash and 1014 sample count without hash. There is still a lot of work (specially loops) the compiler is not optimizing. This could be seen by adding –remarks in command line options. It though was expected that compiler could optimize more.

It took about 2 mins to implement the optimization and there was reduction of about 100 samples. Below are the samples before and after this optimization.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
-O3 optimization for time	1228	1014	17.42

Optimization #3: Removing bx and by loop(inner most loops)

For a full-screen JPEG, 320*240 = 76,800 pixels are computed and plotted. So, optimization is the obvious one (it was discussed in class-hot loops) and definitely 4 nested loops for writing pixels is a lot of work. So here is where I started. This optimization is expected to give performance, say, about 30-35% improved (not more than this) because even though the two inner most loops are removed the LCD_Write_Rectangle_Pixel still was in the loop of count bx_limit*by_limit. The multiplication of count of two inner most loops. But since there is reduction in LCD write calls the performance was expected to improve and it did.

The time it took to understand and analyze was the longest, about 12 hrs. But the performance improvement was unbelievable. It gave 517 sample counts without hash. Which means almost double the speed, below table shows that.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
Removing bx and by loop(inner most loops)	1014	517	49.01

This was the most difficult optimization because it required the complete understanding on how pixels are being written. With the hash enable it was easier to understand if the right image is being printed. With hash after optimization the number of sample count is 558.

Optimization #4: PDOR->PCOR/PSOR

In this optimization, `FPTC->PDOR &= ~LCD_DATA_MASK; FPTC->PDOR |= (cmd & 0xff) << LCD_DB8_POS;` this code was changed to do FPTC->PCOR = LCD_DATA_MASK; FPTC->PSOR = (cmd & 0xff) << LCD_DB8_POS; This is to clear the bit using PCOR instead of loading the pin with negated value and Setting the bit using PSOR. Thus the performance is improved since it's no longer loading the value but setting and resetting the bit. The time it took to implement is 2 mins and 2 mins to understand the functionality previously. Over-all nearly 10-15 mins for this optimization and the performance was really good. About 50 samples down. Below is the sample count before and after the optimization.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
PDOR->PCOR/PSOR	517	469	9.2

Optimization #5: Utilising for R,G,B 32-bit space in register(Parallely sending 4 pixels)

Instead of using only 8 bits of register to send the pixel to write on LCD, optimization was made to use all 32 bits of register to send 4 pixels at a time. This was expected to give more than 80 sample count reduction. The implementation time was about 7 hrs including implementation, debugging and analysis.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
Utilising for R,G,B 32-bit space in register(Parallely sending 4 pixels)	469	440	6.18

Optimization #6: Changing all divide (to the power of 2)to shift operations

For this optimization I changed divide operation by the shift operation to reduce load for compiler. There was few cycles reduction expected but the sample count remains the same. So it seems like compiler is already doing these optimization. Time required to implement this was nearly 3-5 mins.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
Changing all divide (to the power of 2)to shift operations	440	440	0

Optimization #7: No-Inlining

I sequentially made these three functions noinline : decodeNextMCU, LCD_24S_Write_Command, LCD_24S_Write_Data. For first two functions there was no change in sample count on making third function less inline sample count increased. It was expected since LCD_24S_Write_Data is called quite a lot of times and compiler is doing the optimization already here.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
No-Inlining	440	446	-1.36

Optimization #8: Initializing b1 and b2 in hashing code and modifying hashing code to validate pixels without exceeding size limit(32KB)

I initialized b1 and b2 values & hashing code every time I was writing them on LCD after parallel pixel write. This gave error while build that the code exceeded the limit of 32KB. Following changes were made to resolve this:

Commented UART and I2C code.

Changed the hashing functionality to accept all 4 pixels in one go and validate 8 bit by 8 bit.

And allowed hashing to accept all 32 bits and validated 8 bits at a time.

This optimization was not expected to give much of a change in code performance but better and good hash and it turned up as expected. The code exceed limit was resolved and with good hash the perfornce improved a bit.

Below is the old and new sample counts with hashing enabled as well as old and new sample counts with hashing disabled:

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
Initializing b1 and b2 in hashing code and modifying hashing code to validate pixels without exceeding size limit(32KB) (without hash)	446	440	1.36

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
Initializing b1 and b2 in hashing code and modifying hashing code to validate pixels without exceeding size limit(32KB) (with hash)	494	479	3.036

Optimization #9: Creating LCD_24S_write_data(my function) to set LCD_D_NC_POS only once for all write calls in LCD_Write_Rectangle_Pixel_mycode.

Looking at the object code of LCD_24S_Write_Data I found that the GPIO PIN 12 on port C is set every time Write function is called. This task could be done once and data can be written serially after that. I created my own function to Write data which instead of doing set->write->set->reset does only write->set->reset and the 1st set is performed only once. The set and reset after write is to latch the data so that needs to be done every time we write the data. It

took about half an hour to analyze and 5 mins to implement. Expect reduce in samples was nearly 7-8 and I get reduction of 5 cycles.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
Creating LCD_24S_write_data(my function) to set LCD_D_NC_POS only once for all write calls in LCD_Write_Rectangle_Pixel_mycode.	440	435	1.136

Optimization #10: Removing GPIO_SetBit(LCD_D_NC_POS) from write data function and adding it at the end of write command function.

Overall in the code there are more write data function call than write command. Write command tends to reset the LCD_D_NC_POS which is set every time by write data function every time. So setting back the LCD_D_NC_POS the same bit before exiting from write command is expected to help as there are quite a lot of Write Data calls. It was expected to reduce the cycles by 3-4 at least but it only reduced the sample count by one. Time taken to implement was 1-2 mins.

Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
-O3 optimization for time	435	434	0.229

Optimization #11: No Inline for stuffChar() function.
On making stuffChar() less inline the sample count decreased by count of 5. It was not expecte to give any improvement as such but on trial runs this function turns out to give 5 sample count decrement.

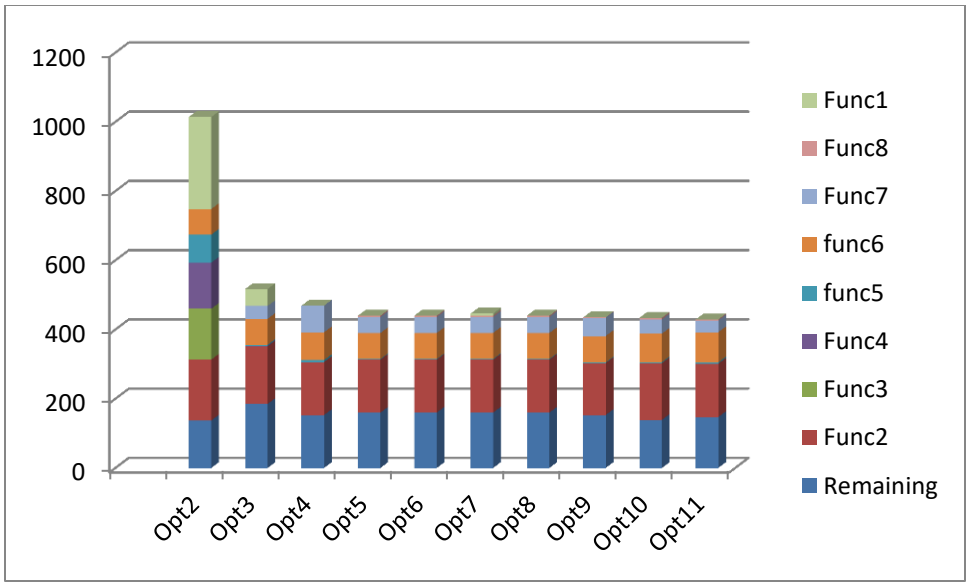
Step #Optimization	Samples Before Optimization	Samples after Optimization	%improvement
-O3 optimization for time	434	429	1.152

Analysis and discussion of results

Q1. Create a summary stacked column chart showing the execution profile for each successful optimization step, as shown here. To make the data easier to understand, put the five2 functions with the most samples at the top of each column, in order of decreasing initial sample count. Group the remaining functions into a single item (blue, “Remaining Functions”).

Optimization	Func1	Func2	Func3	Func4	func5	func6	Func7	Func8	Remaining
Opt2	266	176	147	133	81	74	0	0	137
Opt3	49	166	0	2	3	75	37	0	185
Opt4	0	153	0	0	6	80	77	0	153
Opt5	0	153	0	0	2	75	46	4	160
Opt6	0	153	0	0	2	75	46	4	160
Opt7	6	153	0	0	2	75	46	4	160
Opt8	0	153	0	0	2	75	46	4	160
Opt9	0	150	0	0	2	76	53	1	153
Opt10	0	164	0	0	3	82	41	5	139
Opt11	0	154	0	0	4	87	34	3	147

Func1	LCD_24S_Write_data
Func2	decodeNextMCU
Func3	LCD_Plot_Pixel
Func4	LCD_24S_Write_command
Func5	pjeg_load_from_memory
Func6	huffdecode
Func7	LCD_Write_Rectangle_Pixel_mycode
Func8	LCD_Start_Rectangle



Q2. Which optimizations improved performance the most? Which offered the best improvement/development time?

Answers:

Optimization#3 : Removing bx and by loop(inner most loops)(Replacing LCD_Plot_Pixel with LCD_Start_Rectangle and LCD_)

The above optimization gave most performance but it also took about 12 hrs of implementation. It improved the speed almost 2x times.

Also, Optimization#2 the obvious optimization itself gave about 17% improvement.

Optimization#4 PDOR->PCOR/PSOR

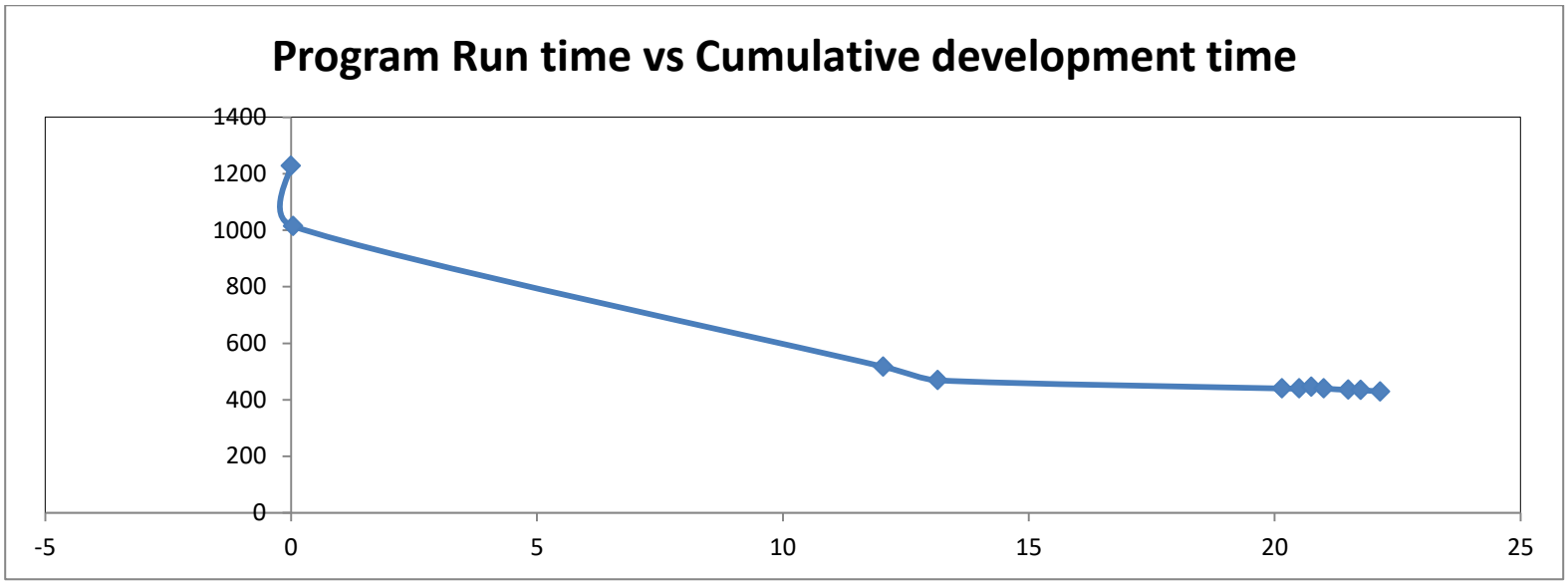
The above optimization was the next best and first to take least time for optimization. It gave about the difference of 50 samples onoptimization.

Q3. Plot the program execution time or sample count (Y) as a function of development time (X).

X axis shows the development time and Y axis shows the Program run-time(sample time)

Cumulative Development Time	Program Run time(Sample count)
0	1228

0.04	1014
12.04	517
13.15	469
20.15	440
20.5	440
20.75	446
21	440
21.5	435
21.75	434
22.15	429



Lessons learned in this project, and how you might do things differently next time

- I started implementation of optimisation3 without planning how I do then after almost waiting an hour I made control flow graph of original code and tried to optimize as required. I will from next understand the flow first and then attack.