CMPE-380: Applied Programming

Laboratory Exercise 08

2 Dimensional Arrays, & Performance measurement

Table of Contents

Pre-Lab – 10 pts 3

Timing 3

Linux /usr/bin/time 4

Size 4

2 Dimensional Arrays 5

More Efficient Matrix Allocation/Deallocation 5

Interactive Exercises – 30 pts 6

Timing 6

Assignment – 60 pts 9

Objectives 9

Setup 9

Preliminary Step 9

Analysis and Optimization 10

Grading Criteria 13

Laboratory Grading Sheet 14

2 –D Arrays, Timing

Pre-Lab – 10 pts

In this lab session, we will review 2 dimensional array performance improvements and more details on program measurement. Please review and understand the class notes on "Time measurement", "Memory use" and "Performance tuning". This lab will assume you are familiar with the topics covered in class.

Timing

It is difficult to measure time and memory accurately, because, most of the time and memory measurement methods are either too intrusive, too slow or require code modification. Typical CPU clocks have a resolution of 1ms but processor instructions have a typical execution time of 7 ns, so small changes to program performance will be impossible to measure without looping. The only way we can measure small improvements is to execute the same code over and over and divide by the iterations. This is why you wrote **Timers.h**.

Linux /usr/bin/time

The Linux OS provides OS level timing tool that can be easily used for gross (rough) performance estimates. **/usr/bin/time -p** (-p posix format) can be used to measure the total time spent by a program form the point of view of the operating system. /usr/bin/time reports three-time measurements as shown:

The GNU/Linux `time`

- The GNU/Linux function **`time`** (**`/usr/bin/time`**) reports ***three timing measurements***

1. ***Real Time (RT):*** Actual time elapsed as measured by an external clock (*e.g.*, your watch)
2. ***User Time (UT):*** Amount of time the ***program is actually executing*** (programs spend some of their "real time" waiting to be executed)
3. ***System Time (ST):*** Amount of time spent executing operating systems code *on the program's behalf*

   Note: $RT \geq UT + ST$

Size

Linux provides a binary code size tool which will report on the internal structure of a program, identifying the amount of memory used by variables and code. The output has the following structure:

- ***text segment* (.text):** contains all ***machine code***
- ***data segment* (.data):** contains all statically initialized variables with non-zero values
- ***bss segment* (.bss): b**lock **s**torage **s**tart, contains statically allocated data variables that are uninitialized or initialized to zero

2 Dimensional Arrays

More Efficient Matrix Allocation/Deallocation

In lab 6, exercise **lab_c.c** you implemented the memory free function for a simple, row by row, 2D memory array program. The lab 6 **lab_c.c** example allocated N rows of memory length L, one row at a time. Review the memory allocation and free functions from the exercise of lab 6.

You can allocate and release the memory of a dynamic array in a more efficient way than one row at a time. Rather than allocate N rows of memory length L, malloc one giant memory block of (N*L) bytes and use pointer manipulation to break the giant block into functional pieces. The malloc() function is very slow so this approach results in significant performance improvements for large arrays. The prelab file **lab_a.c** is provided and implements the smarter allocation approach. Document your answers in **prelab.txt**.

1. Copy your implementation of lab 6, exercise **lab_c.c** and **Timers.h** into the prelab directory for this lab. Contact your professor if your **lab_c.c** or **Timers.h** file, at least a day before your lab, if your files don't work.
2. "Comment out" your code from lab 6 that grew the array from 3x5 to 8x5 by using an "#if 0 #endif" command.
3. Explain the difference between the allocation and deallocation approaches of lab 6 **lab_c.c** and this example **lab_a.c**.
4. Build your **lab_c.c** and **lab_a.c** with (gcc -g -std=c99) and verify it is functioning .
5. Run valgrind (***valgrind --tool=memcheck --leak-check=yes --track-origins=yes***) against both applications and verify that the code has no issues. Redirect the valgrind output to your **prelab.txt** file

Interactive Exercises – 30 pts

Timing

In this section you will modify your prelab **lab_a.c** and **lab_c.c** 2D array implementations to measure their performance and size using Timers.h and Linux command line features. You will then write code to implement and measure Honer's factorization.

1) Copy the lab_a.c and lab_c.c files from your prelab into the exercise directory
2) Copy Timers.h from a previous lab. Contact your TA or professor if you don't have a functioning Timers.h file.
3) Add the **Timers.h** include file to both applications.
4) Modify the array sizes of both applications to be macro driven.

   E.g: **rows=ROWS, cols=COLS**; You will pass the actual value on the compiler command line with: **-DROWS=10000 -DCOLS= 100**
5) Use the EN_TIME macro variable to "comment out" the array initialization and printing routines in both applications. We only want to measure our malloc/free changes and nothing else.
6) Use the value "**REPEAT_NUM**" for your looping limit. You will pass the actual value on the compiler command line with: **-DREPEAT_NUM=1000**
7) Add the required Timer.h macros to both applications. You should print both elapsed time and time per iteration values.
8) Compile the programs with:

   gcc -g -DEN_TIME -DREPEAT_NUM=1000 -std=c99 -DROWS=10000 -DCOLS=100 lab_a.c -o lab_a

   gcc -g -DEN_TIME -DREPEAT_NUM=1000 -std=c99 -DROWS=10000 -DCOLS=100 lab_c.c -o lab_c

9) Run both programs using the command line "time" feature: e.g **time ./lab_a.**

10) Which implementation is faster? How much faster (%)? Does the data returned from the "time" command line feature correlate with your Timers.h macro value? Document your answers in **exercise.txt**.

11) Re compile both programs with: **-DREPEAT_NUM=2** and name the output **lab_a2** and **lab_c2.**
12) Run valgrind (***valgrind --tool=memcheck --leak-check=yes --track-origins=yes***) against both binaries to show that you have not introduced any memory leaks. Append your valgrind output in **exercise.txt** .
13) Run both programs using the command line "size" feature: e.g **size ./lab_a.** Append the size output in **exercise.txt .** Is one version significantly larger?

14) Re compile both programs **without -DEN_TIME** and **with -DROWS=3 -DCOLS=3** and name the output **lab_a3** and **lab_c3**
15) Run both programs and redirect the output into **exercise.txt**. Are both outputs the same? Did the initialization and print routines run?

16) Create a new file called lab_d.c which will be used to implement and time Honer's factorization code versus alternatives. Code the following polynomial:

f = 4.4x^4 - 3.3X^3 + 2.2X^2 - 1.1x + 6.0

Using:

A) Traditional multiplications (e.g x*x)

B) Power functions (e.g pow(x,2))

C) Using Honer's factorization.

17) Your timing repeat value should utilize a macro called **REPEAT_NUM** with a value of **100000000.**

18) Your code should clearly identify which implementation is being measured and print the result of the calculation. E.g.:

```
Honer's performance, ans 14088.747140
Elapsed CPU Time (time) = 0.203125 sec.
Elapsed CPU Time per Iteration (time, 100000000) =  2.03e-09 sec.

Mult performance, ans 14088.747140
Elapsed CPU Time (time) = 0.296875 sec.
Elapsed CPU Time per Iteration (time, 100000000) =  2.97e-09 sec.

Pow performance, ans 14088.747140
Elapsed CPU Time (time) = 5.28125 sec.
Elapsed CPU Time per Iteration (time, 100000000) =  5.28e-08 sec.
```

19) Use the following to compile and link your code. **"-O0"** is critical!

gcc -g -O0 lab_d.c -DEN_TIME -lm -o lab_d

20) Run the program and redirect the output into **exercise.txt**. Are all the results identical? Which implementation is fastest?

Note: If you reuse the same timing variable be sure to call RESET_TIMER()

Contact your lab TA to have them review your lab_a.c, lab_c.c, lab_d.c and exercise.txt files.

Assignment – 60 pts

Objectives

To measure program performance and analyze the impact of optimization techniques in C programs.

Setup

The file **results.txt** is provided to assist you in understanding the delivery requirements. Copy your timing enabled code **hw6.c** as **hw8.c**. Copy **Timers.h** and **data.txt** from the previous assignment (lab 6), you will not need the CPP code. The file **hw8.c** implements a least-squares linear fitting program, with timing features included, in C . This program takes *input from a file* whose name is passed as a *command-line argument* and are designed to accommodate a *variable number of data points*. The file **data.txt** contains the data points to be used as input to the programs *for testing*.

Note that you will need the **Timers.h** which includes your implementations of the timing macros.

Preliminary Step

For this assignment we will use the following compiler options as the base case:

gcc -Wall -std=c99 -pedantic -g -DMOVE_IO_CLOSE -O1 hw8c.c -o hw8c

Note: During the assignment we will change the compiler optimization level from -O0 to -O3, inclusive.

Use the Implementation of Instrumentation Macros and Instrumentation for Timing from the previous assignment to do the analysis and optimization below. Furthermore, for timing of the C program take into consideration the requirements explained in your previous assignment.

1. Your timing should *measure and report execution time* of ***data input*** and ***calculations*** separately. Use two different timers, one for data input and another for calculations. Your data should include the total elapsed time and the per loop time for data input and calculations.
2. Run all your measurement code three times without modification. You will find that data input performance is HIGHLY variable. This variability significantly impacts our ability to "tune"

our software because we will have issues identifying the value of our changes versus normal system performance noise. Use the average data for three runs for your calculations. You must clearly identify and provide your raw data in your submission.

Analysis and Optimization

1. First analyze the C program **hw8c.c** for ***speed and size***. For speed you will use Timing macros. Generate data for all compiler optimization levels (-O0 to -O3). For size use **ls -l** and **size** (refer to the manual pages aka: "man size" for more details about these commands.) Be sure to include the output from each of your execution speed runs and your ls –l and size in your TAR file as **ls.txt** and **size.txt**, respectively with clear documentation on what is being reported. As always, run each test 3 times, use the average for calculations and clearly provide your raw data.

2. Copy the **hw8c.c** and call it **hw8c_opt1.c**. Modify the program to replace the malloc()/copy/free() memory resizing features in AddPoint() with a more efficient realloc() based approach. Compare your new implementations performance and size against the original implementation for all compiler optimization levels (-O0 to -O3) and sizes. The software must print out a message indicating the realloc() implementation.

3. Copy the **hw8c.c** and call it **hw8c_opt2.c**. Modify the program to "loop unroll" the coefficient calculation in CalculateCoefficients() by TWO. Only use the temporary sum variables in the original code:

```
/* Declare and initialize sum variables */
double S_XX = 0.0;
double S_XY = 0.0;
double S_X  = 0.0;
double S_Y  = 0.0;
```

   Again, compare using all compiler optimization levels and sizes. The software must print out a message indicating the simple loop unrolling implementation.

4. Copy the **hw8c_opt2.c** and call it **hw8c_opt3.c**. Modify the program to "loop unroll" the coefficient calculation in CalculateCoefficients() by TWO, but this time use a unique variable for each sub calculation (double the number of sum variables). Again, compare using all compiler optimization levels and sizes . The software must print out a message indicating the more complex loop unrolling implementation.
   What happens to the execution performance, why?

Makefile:

You must include a makefile with: all, mem, test, help and clean targets. Due to the number of binaries you will build, you do NOT need to compile to object (.o) format first. Compile directly to a binary.

"all" should build all your timing enabled code (hw8c.c, hw8c_opt1.c, hw8c_opt2.c and hw8c_opt3.c) against EACH optimization level (-O0 to -O3). Append an "_<level> to the end of the build to identify optimization 0,2,3. Optimization level -O1 should NOT have an _1. In addition, you should also build a "mem" build with timing disabled, at -O1 and labeled "_mem". Build list

hw8c, hw8c_0, hw8c_2, hw8c_3, hw8c_mem

hw8c_opt1, hw8c_opt1_0, hw8c_opt1_2, hw8c_opt1_3, hw8c_opt1_mem

hw8c_opt2, hw8c_opt2_0, hw8c_opt2_2, hw8c_opt2_3, hw8c_opt2_mem

hw8c_opt3, hw8c_opt3_0, hw8c_opt3_2, hw8c_opt3_3 ,hw8c_opt3_mem

Your dependency should be based on the raw .c file names and then just build all the other optimization levels.

"test" should run each of your performance test targets, with the data.txt and redirect all the output to a single file called "out.txt". A line indicating what command is being executed must be included in "out.txt" prior to the actual execution of each command to clearly indicate what is being tested.

"mem" should run valgrind using the standard parameters, with timing disabled at -O1 for your hw8c.c, hw8c_opt1.c, hw8c_opt2.c and hw8c_opt3.c. No memory leaks or access errors are permitted.

"clean" should remove ALL build files.

Analysis

This assignment generates substantial data and is primed for extensive analysis. You should provide a table for all your averaged data. You only need to consider the per loop performance, not the overall times.

Sample data table: (your values may be different)

	hw8c_0	hw8c	hw8c_2	hw8c_3
Data timer per iteration	0.08030000	0.01860000	0.01810000	0.01780000
Calclation timer per iteration	0.00003250	0.00000700	0.00000700	0.00000700
ls -l	16824	20360	20784	23400
text (code) size	5136	3623	3687	4047
data size	620	612	612	612
	hw8c_opt1_0	hw8c_opt1	hw8c_opt1_2	hw8c_opt1_3
Data timer per iteration	0.00230000	0.00220000	0.00230000	0.00230000
Calclation timer per iteration	0.00003250	0.00000750	0.00000650	0.00000700
ls -l	16784	20040	20440	22384
text (code) size	5082	3593	3593	3841
data size	628	620	620	620
	hw8c_opt2_0	hw8c_opt2	hw8c_opt2_2	hw8c_opt2_3
Data timer per iteration	0.09240000	0.01870000	0.01770000	0.01790000
Calclation timer per iteration	0.00002950	0.00000700	0.00000700	0.00000700
ls -l	16928	20560	21016	21736
text (code) size	5520	3751	3823	4039
data size	620	612	612	612
	hw8c_opt3_0	hw8c_opt3	hw8c_opt3_2	hw8c_opt3_3
Data timer per iteration	0.11200000	0.01870000	0.01790000	0.01770000
Calclation timer per iteration	0.00002400	0.00000600	0.00000500	0.00000550
ls -l	17024	21032	21432	22152
text (code) size	5648	3815	3887	4103
data size	620	612	612	612

What specific conclusion can you draw from the data? Are there any general conclusions you can reach? Was hw8c_opt3.c much faster than hw8c_opt2.c? Does adding a few extra

temporary variables help performance? What is the impact of the compiler optimization levels on binary size and speed? Does a higher level always help?

Once you are done create a <lastName>_hw8.tar (lastName is your last name) file and submit.

Grading Criteria

1. (18 points) Code implemented properly, code generates proper results.
    a. (6 points) hw8c_opt1.c
    b. (6 points) hw8c_opt2.c
    c. (6 points) hw8c_opt3.c
2. (6 points) The make file contains all the required features and operates properly.
3. (6 points) All the raw output data is provided and clearly labeled and identified.
4. (30 points) Analysis
    a. (6 points) Includes accurate and fully populated data matrix
    b. (6 points) Compare data times, calculation times, file size, code size and data size across each optimization level for each implementation (hw8.c. hw8c_opt1.c, hw8c_opt2.c and Hw8_opt3.c)
    c. (6 points) Compare the data times, calculation times, file size, code size and data size between hw8c.c and hw8c_opt1.c
    d. (6 points) Compare the data times, calculation times, file size, code size and data size between hw8c.c and hw8c_opt2.c
    e. (6 points) Compare the data times, calculation times, file size, code size and data size between hw8c_opt2.c and hw8c_opt3.c

Student Name: _____

Laboratory Grading Sheet

Lab 08 - 2 Dimensional Arrays, Timing

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: 2D array code	2		
Pre-Lab: prelab.txt	8		
Interactive Exercises: lab_a.c and lab_c.c code	15		
Interactive Exercises: leaks	2		
Interactive Exercises: performance questions	7		
Interactive Exercises: time questions	4		
Interactive Exercises: size questions	2		
Total	40		

You must turn this signed sheet in at the end of lab to receive credit!