ECE454, Fall 2011
Homework2: Memory Performance
Assigned: Sept 24th, Due: Oct 9th, 11:59PM

Mike Delorme (`mdelorme@eecg.toronto.edu`) is the TA for this assignment.

1 Introduction

After great success with your first client, OptsRus has a second client: an image processing software firm that also requires improved performance to stay competitive. After diligently profiling and analyzing their code, you have decided that the first function to optimize is a matrix rotation function called `rotate`, which rotates an image counter-clockwise by 90°. You have also discovered that the main opportunity for improving `rotate` is to improve its cache/memory performance.

In this code an image is represented as a two-dimensional matrix M, where $M_{i,j}$ denotes the value of (i, j)th pixel of M. Pixel values are triples of red, green, and blue (RGB) values. **We will only consider square images**. Let N denote the number of rows (or columns) of an image. Rows and columns are numbered, in C-style, from 0 to $N - 1$.

Given this representation, the `rotate` operation can be implemented quite simply as the combination of the following two matrix operations:

- *Transpose*: For each (i, j) pair, $M_{i,j}$ and $M_{j,i}$ are interchanged.
- *Exchange rows*: Row i is exchanged with row $N - 1 - i$.

This combination is illustrated in Figure 1.

2 Implementation Overview

Data Structures

The core data structure deals with image representation. A `pixel` is a struct as shown below:

```
typedef struct {
   unsigned short red;   /* R value */
   unsigned short green; /* G value */
   unsigned short blue;  /* B value */
} pixel;
```

As can be seen, RGB values have 16-bit representations ("16-bit color"). An image `I` is represented as a one-dimensional array of `pixel`s, where the (i, j)th pixel is `I[RIDX(i,j,n)]`. Here `n` is the dimension of the image matrix, and `RIDX` is a macro defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

See the file `defs.h` for this code.

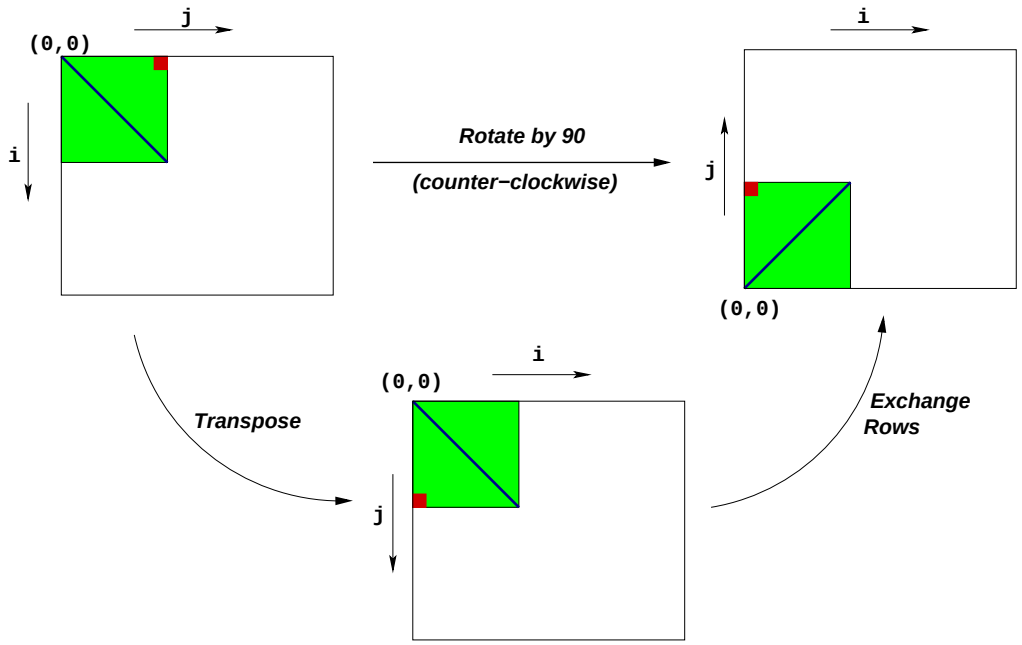

Figure 1: Rotation of an image by 90° counterclockwise

Test case	1	2	3	4	5	6	7	
Method N	64	128	256	512	1024	2048	4096	Geom. Mean
Naive `rotate` (CPE)	5.1	6.3	10.7	15.1	20.5	102.2	114.1	
Optimized `rotate` (CPE)	7.3	7.6	8.0	10.7	14.7	32.6	34.9	
Speedup (naive/opt)	0.7	0.8	1.3	1.4	1.4	3.1	3.3	1.5

Table 1: CPEs and Ratios for Optimized vs. Naive Implementations

Rotate

The following C function computes the result of rotating the source image `src` by 90° and stores the result in destination image `dst`. `dim` is the dimension of the image.

```
void naive_rotate(int dim, pixel *src, pixel *dst) {
  int i, j;
  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
      dst[RIDX(dim-1-j,i,dim)] = src[RIDX(i,j,dim)];
  return;
}
```

The above code scans the rows of the source image matrix, copying to the columns of the destination image matrix. **Your task** is to rewrite this code to make it run as fast as possible using techniques like tiling, loop unrolling and code motion. The performance of naive_rotate is your baseline. Please see the file `kernels.c` for this code.

3 Performance measures

We will be measuring performance using one performance metric *CPE* or *Cycles per Element*, which measures the real world performance of your function. If it takes C cycles to run for an image of size $N \times N$, the CPE value is C/N^2.

Table 1 summarizes the performance of the naive implementation shown above and compares it against an optimized implementation, as measured on a "ugXXX" machine. Remember that you can get specific information about the construction of the memory system of the "ugXXX" machines via `lstopo` and `/sys/devices/...` as described in the memory_performance lecture.

The score of your implementation is the geometric mean of the CPE-speedups of the different dimension arrays.

Perf Tool

To gain insight into the cache behavior of your implementation, you can use the recently-released `perf` infrastructure to access the hardware performance counters of the processor. For example, to output the first-level cache misses generated by your program `foo` you would execute:

```
perf stat -e L1-dcache-load-misses foo
```

You can view a listing of all performance counters that you can monitor by running:

```
perf list
```

Note that you can monitor multiple counters at once by using multiple `-e` options in one command line. `perf` has many other features that you can learn more about by browsing:

```
perf --help
```

For example, you can consider monitoring TLB misses or other more advanced events. A small write-up on `perf` is available here:

```
http://www.pixelbeat.org/programming/profiling/
```

Unfortunately there is not a lot of documentation on `perf` yet as it is so new, but the "help" information is clear.

Assumptions

To make life easier, you can assume that N is a multiple of 32. Your code must run correctly for all such values of N, but we will measure its performance only for the 5 values shown in Table 1.

4 Setup

Start by copying the `hw2.tar.gz` file from UG shared directory
`/cad2/ece454f/hw2/hw2.tar.gz`
into a protected directory within your **UG** home directory.

Then run the command:
`tar xzvf hw2.tar.gz`
This will cause a number of files to be unpacked into the directory.
The **ONLY** file you will be modifying and handing in is `kernels.c`. You should **NOT** modify other files.

Looking at the file `kernels.c`, you'll notice a C structure `team` into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**

```
team_t team = {
    "group1",           /* Team name */
    "AAA BBB",      /* First member full name */
    "AAA@nowhere.edu",  /* First member email address */
    "",                   /* Second member full name (leave blank if none) */
    ""                    /* Second member email addr (leave blank if none) */
};
```

We have provided support code to help you test the correctness of your implementations and measure their performance. This section describes how to use this infrastructure. The exact details of the assignment is described in the following section.

Note: The *only* source file you will be modifying is `kernels.c`.

4.1 Solution Versions

You will be writing many versions of the `rotate` routines. To help you compare the performance of all the different versions you've written, we provide a way of "registering" functions.

For example, the file `kernels.c` that we have provided you contains the following function:

```
void register_rotate_functions() {
   add_rotate_function(&rotate, rotate_descr);
}
```

This function contains one or more calls to `add_rotate_function`. In the above example, `add_rotate_function` registers the function `rotate` along with a string `rotate_descr` which is an ASCII description of what the function does. See the file `kernels.c` to see how to create the string descriptions. This string can be at most 256 characters long.

4.2 Drivers

The source code you will write will be linked with object code that we supply into the driver binary, i.e. `driver_cpe`. To create the driver, simply run the following command:

```
unix> make all
```

You will need to re-make the driver *each* time you change the code in `kernels.c`.

CPE Driver

This `driver_cpe` driver uses the `rdtsc` instruction (as described briefly in lecture) to measure the number of clock cycles it takes to execute your rotate() functions. To test your implementations, run the command:

```
unix> ./driver_cpe
```

`driver_cpe` can be run in four different modes:

- *Default mode*, in which all versions of your implementation are run.
- *Autograder mode*, in which only the `rotate()` function is run. This is the mode we will run in when we use this driver to grade your handin.
- *File mode*, in which only versions that are mentioned in an input file are run.
- *Dump mode*, in which a one-line description of each version is dumped to a text file. You can then edit this text file to keep only those versions that you'd like to test using the *file mode*. You can specify whether to quit after dumping the file or if your implementations are to be run.

If run without any arguments, `driver_cpe` will run all of your versions (*default mode*). Other modes and options can be specified by command-line arguments to `driver_cpe`, as listed below:

`-g` : Run only the `rotate()` function (*autograder mode*).

`-f <funcfile>` : Execute only those versions specified in `<funcfile>` (*file mode*).

`-d <dumpfile>` : Dump the names of all versions to a dump file called `<dumpfile>`, *one line* to a version (*dump mode*).

`-q` : Quit after dumping version names to a dump file. To be used in tandem with `-d`. For example, to quit immediately after printing the dump file, type `./driver_cpe -qd dumpfile`.

`-h` : Print the command line usage.

We recommend you to start with **default mode**. For example, running CPE driver with the supplied naive_rotate version under default mode generates the output shown below:

```
unix> ./driver_cpe
Teamname: group1
Member 1: AAA BBB
Email 1: AAA@nowhere.edu

Rotate: Version = rotate: Current working version:
Dim             64      128     256     512     1024    2048    4096    Mean
Your CPEs       5.1     6.3     10.7    15.1    20.5    102.2   114.1
Baseline CPEs   5.1     6.4     10.7    15.3    20.4    102.3   114.1
Speedup         1.0     1.0     1.0     1.0     1.0     1.0     1.0     1.0

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    2048    4096    Mean
Your CPEs       5.1     6.3     10.7    15.1    20.5    102.2   114.1
Baseline CPEs   5.1     6.4     10.7    15.3    20.4    102.3   114.1
Speedup         1.0     1.0     1.0     1.0     1.0     1.0     1.0     1.0

Summary of Your Best Scores:
  Rotate: 1.0 (naive_rotate: Naive baseline implementation)
```

Note: Here, Baseline CPEs is according to the performance of `naive_rotate()` on a test run that we performed; don't fret if it is slightly different than what you are seeing.

4.3 Coding Rules

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You should not use any embedded assembly language statements.
- It must not interfere with the time measurement mechanism.
- You will be penalized if your code prints any extraneous information.
- You cannot modify the default compilation flags in the Makefile (they are set to optimize at -O2).

You can only modify code in `kernels.c`.

4.4 Team Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about your team (group name, team member names and email addresses). A maximum number of students in a team is **2**.

5 Evaluation

You should optimize `rotate()` to achieve as low a CPE as possible. You should compile `driver_cpe` and then run them with the appropriate arguments to test your implementations.

Your solutions for `rotate()` will count for 100% of your grade. Your grade will be calculated as follows:

- *Correctness and Effort: 20 points.*
- *Performance: 80 points.*
- *Total: 100 points.*

The score for each will be based on the following:

- Effort: Show your work! The homework infrastructure is set up for you to easily save multiple solutions in `kernels.c`. Be sure to evaluate and save your intermediate solutions as you optimize. Don't submit a `kernels.c` containing only your single/final solution, it should contain all of your intermediate solutions too.
- Correctness: You will get NO CREDIT for buggy code that causes the driver to complain! This includes code that correctly operates on the test sizes, but incorrectly on image matrices of other sizes. As mentioned earlier, you may assume that the image dimension is a multiple of 32.
- Performance: We will evaluate the performance using CPE scores of your `rotate()` function. The best known solution (so far) gives a mean CPE of speedup 3.0, which is worth a 80/80 for performance. See if you can beat it! A CPE speedup of 2.0 will be worth at least 55/80 for performance, higher speedups will be worth more than that.

Hint. Look at the assembly code generated for the `rotate`, e.g. using

```
unix> objdump -d your-binary-file-name | less
```

Focus on optimizing the **inner loop** (the code that gets repeatedly executed in a loop) using the optimization tricks, like blocking, loop unrolling and loop-invariant code motion, covered in class.

6 Logistics

You should work in a group of up to **two** people in solving the problems for this assignment. Any clarifications and revisions to the assignment will be posted on the course Web page.

7 Submission

When you have completed the lab, you will hand in two files, `kernels.c`, that contains your solution, and a one paragraph report in plain text. In the report, you should describe your final solution, and why it performs well in relation to what is happening in the processor architecture. Here is how to hand in your solution:

Submit your assignment by typing
```
submitece454f 2 kernels.c
submitece454f 2 report
```
on one of the UG machines.

- Make sure you have included your identifying information in the team struct in `kernels.c`.
- Make sure that the `rotate()` function corresponds to your **fastest** implemnentations, as this is the only function that will be tested when we use the driver to grade your assignment.
- Remove any extraneous print statements.