CS 247, Winter 2013
Optional Extra Credit Project 5: Code Optimization with SSE Instructions
Posted: March 6th
Due: March 20th, 11:30PM

Michael Meehan

March 6, 2013

1 Introduction

Project 4 had you investigate various ways you could improve the performance of the smooth and rotate operations by trying to write cache-friendly code. This project will start where that one left off. In this project we will try to improve the performance of the smooth and rotate operations by using the SIMD capabilities of the core i7 processor.

The authors of your text provide various "Web Asides" to extend the topics covered in your text. These can be found at http://csapp.cs.cmu.edu/public/waside.html. The Web Aside titled MEM:BLOCKING provided information that you will have found useful in performing project 4. The Web Aside we will refer to for project 5 is called OPT:SIMD. This article explains how to achieve greater parallelism with SIMD instructions. The objective of project 5 is to rework the code from project 4 utilizing the SIMD capabilities of the core i7 processor to achieve an even greater speedup.

The narrative below is a reworking of project 4.

This assignment deals with optimizing memory intensive code. Image processing offers many examples of functions that can benefit from optimization. In this lab, we will consider two image processing operations: `rotate`, which rotates an image counter-clockwise by $90°$, and `smooth`, which "smooths" or "blurs" an image.

For this lab, we will consider an image to be 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 $N-1-i$.

The `smooth` operation is implemented by replacing every pixel value with the average of all the pixels around it (in a maximum of 3×3 window centered at that pixel). The values of pixels `M2[1][1]` and `M2[N-1][N-1]` are given below:

$$\texttt{M2[1][1]} = \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=0}^{2} \texttt{M1[i][j]}}{9}$$

$$\texttt{M2[N-1][N-1]} = \frac{\sum_{\texttt{i}=N-2}^{N-1} \sum_{\texttt{j}=N-2}^{N-1} \texttt{M1[i][j]}}{4}$$

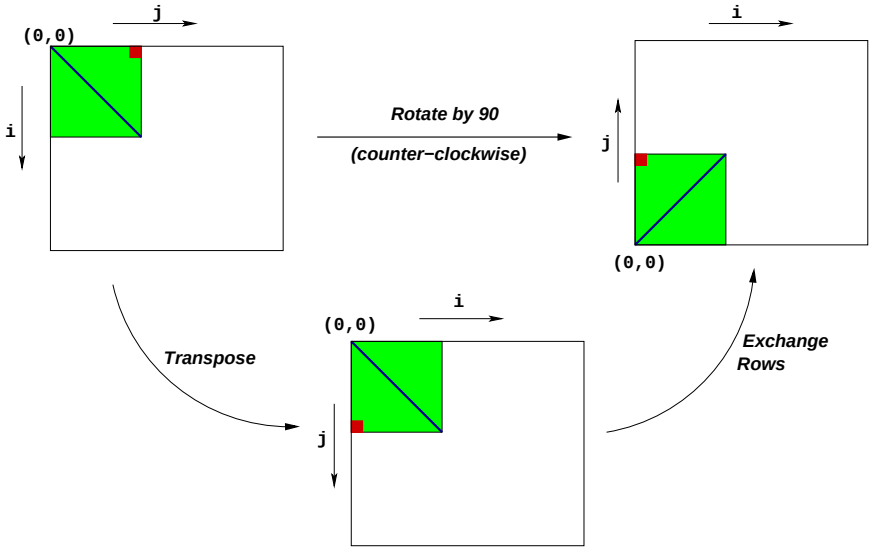

Figure 1: Rotation of an image by $90°$ counterclockwise

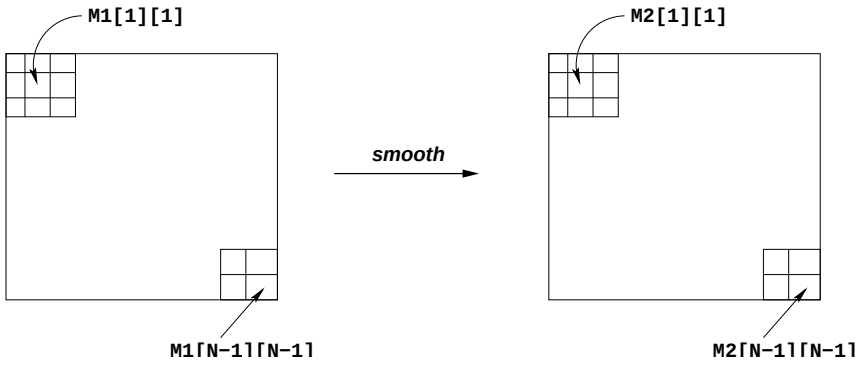

Figure 2: Smoothing an image

2 Logistics

You may work in a group of up to three people for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Moodle page.

3 Hand Out Instructions

> You can start with the code you produced for project 4 to begin building this project or you can go to the Project 4 Perflab tar Download link and start from a fresh copy.

Since you will be working in a group change the information in the file `kernels.c` in the C structure `team` to reflect the information about yourself and your group members. **Do this right away so you don't forget.**

4 Implementation Overview

Data Structures

The core data structure deals with image representation. A `pixel` is a struc 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.

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 the code you produced for project 4 using SSE SIMD instructions to make it run as fast as possible. You do not need to abandon the general techniques you may have employed in the project 4 version like code motion, loop unrolling and blocking. You will need to re-think them in order to be able to take full advantage of the SIMD operations on the core i7.

See the file `kernels.c` for this code.

Smooth

The smoothing function takes as input a source image `src` and returns the smoothed result in the destination image `dst`. Here is part of an implementation:

```
void naive_smooth(int dim, pixel *src, pixel *dst) {
  int i, j;

  for(i=0; i < dim; i++)
    for(j=0; j < dim; j++)
      dst[RIDX(i,j,dim)] = avg(dim, i, j, src); /* Smooth the (i,j)th pixel */

  return;
}
```

The function `avg` returns the average of all the pixels around the `(i,j)`th pixel. Your task is to optimize `smooth` (and `avg`) to run as fast as possible. (*Note:* The function `avg` is a local function and you can get rid of it altogether to implement `smooth` in some other way.)

This code (and an implementation of `avg`) is in the file `kernels.c`.

Performance measures

Our main performance measure is still *CPE* or *Cycles per Element*. If a function takes C cycles to run for an image of size $N \times N$, the CPE value is C/N^2.

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.

5 Infrastructure

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 each part of the assignment is

described in the following section.

Note: Depending on the implementation approach you chose you may be only modifying the file `kernels.c` or you may be creating a separate assembly language file to be linked with `kernels.c`.

Versioning

Just as you did in project 4 you can write many versions of the `rotate` and `smooth` routines and compare the performance of all the different versions you've written using the "registering" functions provided.

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.

A similar function for your smooth kernels is provided in the file `kernels.c`.

Driver

The source code you will write will be linked with object code that we supply into a `driver` binary. To create this binary, you will need to execute the command

```
unix> make driver
```

You will need to re-make driver each time you change the code in `kernels.c`. To test your implementations, you can then run the command:

```
unix> ./driver
```

The `driver` 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()` and `smooth()` functions are run. This is the mode we will run in when we use the 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` will run all of your versions (*default mode*). Other modes and options can be specified by command-line arguments to `driver`, as listed below:

`-g` : Run only `rotate()` and `smooth()` functions (*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 -qd dumpfile`.

`-h` : Print the command line usage.

Team Information

Important: Before you start, you should fill in the struc in `kernels.c` with information about yourself and your group members (name and email address). You may have up to three members in a group.

6 Assignment Details

Optimizing Rotate (50 points)

In this part, you will optimize `rotate` to achieve as low a CPE as possible. You should compile `driver` and then run it with the appropriate arguments to test your implementations.

Optimizing Smooth (50 points)

In this part, you will optimize `smooth` to achieve as low a CPE as possible.

Some advice. Look at the assembly code generated for the `rotate` and `smooth`. Focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class. The `smooth` is more compute-intensive and less memory-sensitive than the `rotate` function, so the optimizations are of somewhat different flavors.

Coding Rules

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

- Unlike project 4 which restricted you to using only ANSI C, you may use any combination of GNU C and embedded assembly language statements as well as externally assembled routines. In particular you can use the GNU extensions to C that provide vector data declarations and operations as described in the OPT:SIMD Web Aside. You should remember that many of the SSE instructions impose a very strict alignment requirement on memory operands. They require that any data being read from memory into an XMM register, or written from an XMM register to memory, satisfy a 16-byte alignment.
- As in project 4 your code must not interfere with the time measurement mechanism. You will also be penalized if your code prints any extraneous information.

You can modify code in `kernels.c` any way you wish. You are allowed to define macros, additional global variables, and other procedures in this files.

Evaluation

Your solutions for `rotate` and `smooth` will each count for 50% of your grade. The score for each will be based on the following:

- 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.
- CPE: You will get full credit for your implementations of `rotate` and `smooth` if they are correct and achieve mean CPEs above thresholds S_r and S_s respectively. You will get partial credit for a correct implementation that does better than the supplied naive one.

  I will decide later what the full credit thresholds S_r and S_s are going to be and post them on Moodle. For partial credits we will use a linear scale, with about a 40% minimum if you made a serious attempt to actually try to solve the lab.

7 Hand In Instructions

When you have completed this project, you will hand in one or more files that contain your solution. If all your code is in kernels.c just submit that. If you created and external assembly language files include them as well. If you required changing the make file for building the driver to include your external assembly language files include your modified make file as well.

- Make sure you have included your identifying information in the team struc in `kernels.c`.
- Make sure that the `rotate()` and `smooth()` functions correspond to your fastest implementations, as these are the only functions that will be tested when we use the driver to grade your assignment.
- Remove any extraneous print statements.
- To submit your work, copy `kernels.c` to a file named `your-login-id-kernels.c.`, submit it to Moodle via the link called Project 5 Submission. The name of the file should be the EXACT string `login-id-kernels.c` where `login-id` is your Computer Science department login ID.
- Remember you can submit the files to Moodle as many times as you like before the deadline and only the last one submitted will be graded.

There are three different approaches you might chose to take in trying to use the SSE instructions to speed up your code.

You can write assembly language statements as a function compatible with C calling conventions in a separate file and then assemble and link the external function to you C program.

You can do what is called in-line assembly where you embed assembly language statements in your C code.

You can use extensions to the C language provided in GNU C to write code that will cause the gcc compiler to emit code that utilizes the SSE instructions.

I will be adding more details about the three different approaches you may take to work on this but I wanted to go ahead and get the initial idea posted. If you are thinking about doing this project the first thing you need to do is read the Web Aside on SIMD instructions.

Good luck!