Project Performance

Due: October 2014

1 Introduction

Baby Daniel wants to play with his toys. First he wants solve a square grid puzzle, and then he wants to play with his other toys. Uh oh! The other toys edges are too sharp! Because Daniel is easily distracted you need to help solve the puzzle and smooth the edges of the toys as quickly as possible!

2 Assignment

The purpose of this assignment is to utilize various techniques that can be used to optimize the speed of C code. You will be optimizing the performance of two C functions, `transpose` and `smooth`, which transposes a two-dimensional array and smooths a two-dimensional array respectively.. For each of these functions, a baseline implementation is provided; your task is to identify and implement ways to improve the performance of those implementations.

2.1 Installing the Stencil

To get started, run the following command in a terminal:

```
cs033_install performance
```

This will install the stencil code to your *course/cs033/performance* directory. The only files you need to edit (and only files you should edit) are *transpose.c* and *smooth.c*.

2.2 `transpose`

```
void transpose_naive(int n, int mat[][n], int res[][n]) {
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            res[j][i] = mat[i][j];
}
```

`transpose` creates a new matrix `res` whose columns are the rows of `mat` and whose rows are the columns of `mat`.

For example, the transpose of $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$ is $\begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$.

Unlike this matrix, however, all matrices passed to your `transpose` function are square and have dimension that is a multiple of 64.

2.3 Smooth

You will also be writing code to perform a *smoothing*[1] operation on a square image whose dimensions are divisible by 16. The basic algorithm for smoothing is fairly simple; a baseline implementation is given below:

```
void smooth_naive(pgm_image_t *src, pgm_image_t *dst)
{
    int i, j;
    int dim = src->x;
    for (i = 0; i < dim; i++) {
      for (j = 0; j < dim; j++) {
        dst->img[IDX(i, j, src->x)] = avg(src->x, i, j, src->img);
      }
    }
}
```

The function `avg()` computes the average of all pixels in the 3x3 grid centered at the pixel `i, j`; code for it is given below:

```
unsigned int avg(int dim, int i, int j, unsigned int *img)
{
  int ii, jj;
  unsigned int sum = 0;
  unsigned int num = 0;
  for(ii = max(i-1, 0); ii <= min(i+1, dim-1); ii++) {
    for(jj = max(j-1, 0); jj <= min(j+1, dim-1); jj++) {
            sum += img[IDX(ii, jj, dim)];
            num++;
    }
  }
  return (unsigned int)(sum/num);
}
```

The second half of this assignment is to write a new function, `smooth`, that optimizes the performance of `naive_smooth`.

3 Support Code

Provided for you are five files: a *Makefile*, *transpose_main.c*, *smooth_main.c*, *transpose_demo*, and *smooth_demo*. The *.c* files contain `main` functions that will run your `transpose` and `smooth` code respectively, when built by the *Makefile*:

[1] The result of this smooth operation is a new image that contains the important parts of the original image, with reduced noise or other rapid shifts in image intensity.

```
./transpose <dim> [<check>]
./smooth <input file> <output file> [<check>]
```

If the optional `<check>` argument, these programs will compare the output of your code against the output of the corresponding naive solution. Consequently, *do not edit the naive versions of these functions*. High performance is meaningless if your program output is not correct, and editing the naive functions could cause your code to appear correct even if it is not. The *transpose_demo* and *smooth_demo* are optimized executables to which you can compare your implementation's running time. In additon, you are provided with a collection of images in the folder pics, on which you can test/time your smooth implementation.

4 Hardware Information

For this project, knowing some of the details of the hardware you will be running your code on can be important. While working, keep in mind that the L1 caches of the Sunlab computers have 64 cache lines per way and that each cache line is 64 bytes long. The caches have 8 ways each, for a total L1 cache size of 32KB. Review the Memory Hierarchy lectures for more information about the structure of caches.

5 Grading

You will graded on this project according to the following:

- **Functionality**: your program produces correct output, and does not crash for any reason. It does not terminate due to a segmentation fault or floating point exception. Note that you do not have to perform any error-checking on the arguments passed to the `transpose` or `poly` functions — this is done for you by the support code.

  As with previous C assignments, there will also be a "Code Correctness" component of your grade.

  Since correct functionality is provided to start with, and not the primary goal of this assignment, this will not comprise a large fraction of your grade.

- **Performance**: your programs run quickly. Your score in this category will be inversely proportional to how fast your program runs, i.e. faster programs will receive a higher score. However, you will not receive any performance points if your code does not produce correct output.

  You will receive full performance points for each operation if they run as quickly as the provided demo. However, this does *not* mean that if your code does not run as quickly you will not receive full performance points.

6 Handing In

To hand in your project, run the command

```
cs033\_handin performance
```

from your project working directory.

At a minimum, your handin must include any *.c* and *.h* files containing code that you have written, and a *README*.

If you wish to change your handin, you can do so by re-running the handin script. Only the most recent handin will be graded.