Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall '2020-2021

Performance Lab Homework

Deadline: 17 January 2021, Sunday, 23:59

1 Objectives

This assignment deals with optimizing memory intensive code. Image processing and matrix operations offer many examples of functions that can benefit from optimization. In this homework, we will consider one image processing operation and one matrix operation.

First function is the bokeh operation that blurs the background to bring the subject of an image to focus. To achieve this effect, it uses a filter that marks the focused pixels. The unmarked pixels are averaged with their immediate neighbours for the blur effect.

For the bokeh function, we will consider an image and a filter to be represented as a two-dimensional matrix M, where $M_{i,j}$ denotes the value of (i, j)th pixel or integer of M. Pixel values are triples of red, green, and blue (RGB) values. Filter values are integers 1 for focused and 0 out of focus pixels. We will only consider square images. Let N denote the number of rows (or columns) of an image or a matrix. Rows and columns are numbered, in C-style, from 0 to $N-1$.

The `bokeh` operation is implemented by replacing every pixel that is out of focus with the average of all the pixels around it (in a maximum of 3×3 window centered at that pixel). The general formula of the resulting matrix for the bokeh operation can be seen below:

$$\texttt{M[x][y]} = \begin{cases} \frac{\sum_{\texttt{i}=-1}^{2} \sum_{\texttt{j}=-1}^{2} \texttt{M1[x+i][y+j]}}{9}, & \textbf{if } 0 < x < N-1 \ \ \& \ \ 0 < y < N-1 \\ \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=-1}^{2} \texttt{M1[x+i][y+j]}}{6} & \textbf{if } x = 0 \ \ \& \ \ 0 < y < N-1 \\ \frac{\sum_{\texttt{i}=-1}^{2} \sum_{\texttt{j}=0}^{2} \texttt{M1[x+i][y+j]}}{6} & \textbf{if } 0 < x < N-1 \ \ \& \ \ y = 0 \\ \frac{\sum_{\texttt{i}=-1}^{1} \sum_{\texttt{j}=-1}^{2} \texttt{M1[x+i][y+j]}}{6} & \textbf{if } x = N-1 \ \ \& \ \ 0 < y < N-1 \\ \frac{\sum_{\texttt{i}=-1}^{2} \sum_{\texttt{j}=-1}^{1} \texttt{M1[x+i][y+j]}}{6} & \textbf{if } 0 < x < N-1 \ \ \& \ \ y = N-1 \\ \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=0}^{2} \texttt{M1[x+i][y+j]}}{4} & \textbf{if } x = 0 \ \ \& \ \ y = 0 \\ \frac{\sum_{\texttt{i}=-1}^{1} \sum_{\texttt{j}=0}^{2} \texttt{M1[x+i][y+j]}}{4} & \textbf{if } x = N-1 \ \ \& \ \ y = 0 \\ \frac{\sum_{\texttt{i}=0}^{2} \sum_{\texttt{j}=-1}^{1} \texttt{M1[x+i][y+j]}}{4} & \textbf{if } x = 0 \ \ \& \ \ y = N-1 \\ \frac{\sum_{\texttt{i}=-1}^{1} \sum_{\texttt{j}=-1}^{1} \texttt{M1[x+i][y+j]}}{4} & \textbf{if } x = N-1 \ \ \& \ \ y = N-1 \end{cases}$$

Second function is called hadamard product that allows for the multiplication of two matrices in an element-wise manner. The formula can be seen below:

$$Mij = M1_{ij} * M2_{ij}$$

where `M1`, `M2` are the source matrices and `M` is the resulting matrix.

2 Specifications

Start by copying `perflab-handout.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf perflab-handout.tar`. 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`. The `driver.c` program is a driver program that allows you to evaluate the performance of your solutions. Use the command `make driver` to generate the driver code and run it with the command `./driver`.

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

3 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.

Bokeh

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

```
void naive_bokeh(int dim, pixel *src, short *flt, pixel *dst) {

    int i, j;

    for(i = 0; i < dim; i++) {
        for(j = 0; j < dim; j++) {
            if ( !flt[RIDX(i, j, dim)] )
                dst[RIDX(i, j, dim)] = avg(dim, i, j, src);
            else
                dst[RIDX(i, j, dim)] = src[RIDX(i, j, dim)];
        }
    }
}
```

The function `avg` returns the average of all the pixels around the $(i, j)th$ pixel. Your task is to optimize `bokeh` (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 bokeh some other way.) This code (and an implementation of `avg`) is in the file `kernels.c`.

Test case	1	2	3	4	5	
Method N	64	128	256	512	1024	Geom. Mean
Naive `bokeh` (CPE)	54.5	54.5	54.0	54.4	54.7	
Optimized `bokeh` (CPE)	41.1	41.1	41.1	41.1	41.3	
Speedup (naive/opt)	1.3	1.3	1.3	1.3	1.3	1.3
Method N	64	128	256	512	1024	Geom. Mean
Naive `hadamard` (CPE)	1.8	1.7	1.6	1.6	2.2	
Optimized `hadamard` (CPE)	1.3	1.3	1.4	1.4	2.4	
Speedup (naive/opt)	1.4	1.3	1.2	1.2	0.9	1.2

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

Hadamard

The hadamard function takes as two matrices `src1`, `src2` and returns the multiplied result in the destination matrix `dst`. Here is the implementation:

```
void naive_hadamard(int dim, int *src1, int *src2, int *dst) {

    int i, j;

    for(i = 0; i < dim; i++)
        for(j = 0; j < dim; j++)
            dst[RIDX(i, j, dim)] = src1[RIDX(i, j, dim)] * src2[RIDX(i, j, dim)];
}
```

Performance measures

Our main performance measure is *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. Table 1 summarizes the performance of the naive implementations shown above and compares it against an optimized implementation. Performance is shown for for 5 different values of N. All measurements were made on the the department computers (ineks).

The ratios (speedups) of the optimized implementation over the naive one will constitute a *score* of your implementation. To summarize the overall effect over different values of N, we will compute the *geometric mean* of the results for these 5 values. That is, if the measured speedups for $N = \{64, 128, 256, 512, 1024\}$ are R_{64}, R_{128}, R_{256}, R_{512}, and R_{1024} then we compute the overall performance as

$$R = \sqrt[5]{R_{64} \times R_{128} \times R_{256} \times R_{512} \times R_{1024}}$$

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

Versioning

You will be writing many versions of the `bokeh` and `hadamard` 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 functions:

```
void register_bokeh_functions() {
   add_bokeh_function(&bokeh, bokeh_descr);
}

void register_hadamard_functions() {
   add_hadamard_function(&hadamard, hadamard_descr);
}
```

This function contains one or more calls to `add_bokeh_function` and `add_hadamard_function`. In one of the examples above,
`add_bokeh_function` registers the function `bokeh` along with a string `bokeh_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. Hadamard works the same way.

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 `bokeh()` and `hadamard()` functions are run. This is the mode we will use when grading your solution.
- *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 `bokeh()` and `hadamard()` 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 struct in `kernels.c` with information about your team (team name, student names, student ids).

5 Assignment Details

Optimizing Bokeh (70 points)

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

For example, running driver with the supplied naive version (for `bokeh`) generates the output shown below:

```
unix> ./driver
Teamname: Team
Member 1: Student Name
ID 1: eXXXXXXX

Bokeh: Version = naive_bokeh: Naive baseline bokeh with filter:
Dim             64      128     256     512     1024    Mean
Your CPEs       54.9    54.8    54.8    54.7    54.7
Baseline CPEs   54.5    54.5    54.0    54.4    54.7
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

Optimizing Hadamard (30 points)

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

For example, running driver with the supplied naive version (for `hadamard`) generates the output shown below:

```
unix> ./driver
Teamname: Team
Member 1: Student Name
ID 1: eXXXXXXX

...

...


Hadamard: Version = naive_hadamard_product: The naive baseline version of hadamard product of t
Dim             64      128     256     512     1024    Mean
Your CPEs       2.0     1.7     1.7     1.6     2.4
Baseline CPEs   1.8     1.7     1.6     1.6     2.2
Speedup         0.9     1.0     1.0     1.0     0.9     1.0
```

Some advice. Look at the assembly code generated for the `bokeh` and `hadamard`. Focus on optimizing the inner loop (the code that gets repeatedly executed in a loop) using the optimization tricks covered in class.

Coding Rules

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

- It must be in ANSI C. You may not use any embedded assembly language statements.
- It must not interfere with the time measurement mechanism. You will also be penalized if your code prints any extraneous information.

You can only modify code in `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in this file.

Evaluation

- 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.
- Note that you should only modify `dst` pointer. If you modify input pointers or exceed their limits, you will not get any credit.
- Bokeh: You will get 40 points for implementation of `bokeh` if it is correct and achieve mean speed-up threshold of 1.3. The team that achieves the biggest speed-up will get 70 points. Other grades will be scaled between 40 and 70 according to your speed-up. You will not get any partial credit for a correct implementation that does below the threshold.
- Hadamard: You will get 30 points if your implementation is correct and achieve a mean speed-up of 1.2. There is no scaling in this function.

- Since there might be changes of performance regarding to CPU status, test the same code many times and take only the best into consideration. When your codes are evaluated, your codes will be tested in a closed environment many times and only your best speed-up of functions will be taken into account.
- Optional: In this assignment, you have an option to create a team up to three people. However, you can also do it alone.

Submission

Submissions are done via ODTUClass. You can only submit `kernels.c` function. Therefore make sure all of your changes are only on this file. Any member of the team can submit the file. Do **NOT** submit the same file by different members of the team. One submission is enough.