Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall '2019-2020

Performance HW

Due date: 29 December 2019, Sunday, 23.59

1 Objectives

Many algorithms require the application of a basic operation repeatedly. Hence, their performance depends on the performance of those basic operations. In this homework we will consider two image processing operations, namely **Sobel Filter** and **Mirror**, which applies a vertical symmetry to an image.

Your objective in this homework is to optimize these functions as much as possible by using the concepts you have learned in class.

2 Specifications

Start by copying `Optimization.tar` to a protected directory in which you plan to do your work. Then give the command: `tar xvf Optimization.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

Sobel

Sobel is a filtering operation which is used to find the edges in an image. Sobel filter has two 3x3 kernels: x-direction kernel and y-direction kernel but we will use only x-directional kernel which operates as follows. We move the appropriate kernel over the input image, computing the value for one pixel and then shifting one pixel to the right. An example of sobel filter operation in x direction is given bellow;

For example, let's say you have a source matrix (src) and kernel matrix (ker);

$$src = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots \\ a_{21} & a_{22} & a_{23} & \dots \\ a_{31} & a_{32} & a_{33} & \dots \\ . & . & . & . \\ . & . & . & . \end{bmatrix}$$

$$ker = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Your output matrix (dst) will look like;

$$dst = \begin{bmatrix} a_{11} * (-1) + a_{12} * (0) + a_{13} * (1) + a_{21} * (-2) + a_{22} * (0) + a_{31} * (-1) + a_{32} * (0) + a_{33} * (1) & ... \\ ... & \\ ... & \end{bmatrix}$$

The example below shows the kernel (ker) matrix being shifted the top left portion of the source matix (src) represented by the green outline.

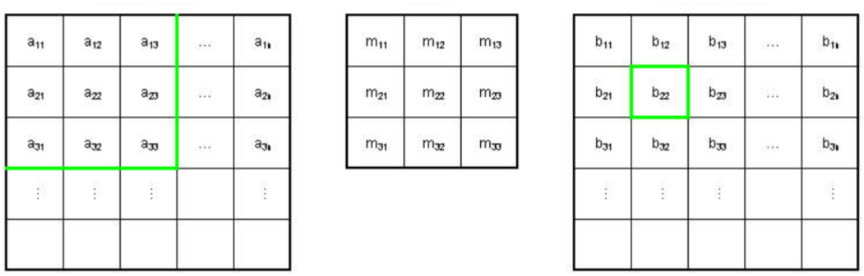

Figure 1: Sobel Convolution Sample

The naive approach for Sobel is given in kernels.c as;

```
void naive_sobel(int dim,int *src, int *dst) {
    int i,j,k,l;
    int ker[3][3] = {{-1, 0, 1},
                     {-2, 0, 2},
                     {-1, 0, 1}};
    for(i = 0; i < dim; i++)
        for(j = 0; j < dim; j++) {
   dst[j*dim+i]=0;
            if(!((i == 0) || (i == dim-1) || (j == 0) || (j == dim-1))){
            for(k = -1; k <= 1; k++)
                for(l = -1; l <= 1; l++) {
          dst[j*dim+i]=dst[j*dim+i]+src[(j + l)*dim+(i + k)]*ker[l+1][k+1];
                }
      }
    }
}
```

where the arguments of the procedure are pointers to the destination (dst) and source (src) matrix, as well as the matrix size N (dim). Your task is to rewrite this code and minimize its CPE, by using techniques like code motion, loop unrolling and blocking.

Mirror

The naive approach for mirror is given in kernels.c as

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

    for(j = 0; j < dim; j++)
        for(i = 0; i < dim; i++) {
            dst[RIDX(j,i,dim)]=src[RIDX(j,dim-1-i,dim)];

        }
}
```

where the arguments of the procedure are pointers to the destination (dst) and a source (src) matrix, as well as the matrix size N (dim). The code given above takes the mirror of the image on the vertical axis, and store the result in the destination (dst) matrix. For example, lets say you have a source matrix (src);

$$src = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Your output matrix (dst) will look like;

$$dst = \begin{bmatrix} a_{13} & a_{11} & a_{12} \\ a_{23} & a_{21} & a_{22} \\ a_{33} & a_{31} & a_{32} \end{bmatrix}$$

Your task is to rewrite this code and minimize its CPE, by using techniques like code motion, loop unrolling and blocking.

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.

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 $N = \{32, 64, 128, 256, 512\}$ are R_{32} R_{64}, R_{128}, R_{256}, and R_{512} then we compute the overall performance as

$$R = \sqrt[5]{R_{32} \times R_{64} \times R_{128} \times R_{256} \times R_{512}}$$

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 `Sobel` and `Mirror` 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_sobel_functions() {
    add_sobel_function(&naive_sobel, naive_sobel_descr);
    add_sobel_function(&sobel, sobel_descr);
    /* ... Register additional test functions here */
}
```

This function contains one or more calls to `add_sobel_function`. In the above example, `add_sobel_function` registers the function `sobel` along with a string `sobel_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 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 `sobel()` and `mirror()` functions are run. This is the mode we will run in when we use the driver to grade your hand in.
- *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 `sobel()` and `mirror()` 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.

Student Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about you(student name, student id, student email).

5 Assignment Details

Optimizing Sobel (60 points)

In this part, you will optimize `sobel` 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 `sobel`) generates the output shown below:

```
unix> ./driver
Student Name: Busra AKARSU
Student Id: eXXXXXXX

Sobel: Version = naive_sobel: Naive baseline implementation:
Dim             32     64      128   256   512   Mean
Your CPEs       40.1   42.1    43.5  53.0   55.2
Baseline CPEs   40.0   42.3    43.6  53.0   55.2
Speedup         1.0    1.0     1.0   1.0    1.0   1.0
```

Optimizing Mirror (40 points)

In this part, you will optimize `mirror` 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 `mirror`) generates the output shown below:

```
unix> ./driver
Student Name: Busra AKARSU
Student Id: eXXXXXXX

Mirror: Version = naive_mirror: Naive baseline implementation:
Dim             32     64    128    256    512   Mean
Your CPEs       2.6    2.3   2.2    2.2    2.2
Baseline CPEs   2.3    2.3   2.2    2.2    2.2
Speedup         0.9    1.0   1.0    1.0    1.0   1.0
```

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.
- It must solely belong to you.
- **Important:** Please, work on department inek machines. Because, all CPE values has been configured according to inek's CPU and we will evaluate your codes on inek machines.

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

Evaluation

Your solutions for `sobel` and `mirror` will each count for 40% and 60% of your grade respectively. 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.
- Note that you should only modify `dst` pointer. If you modify `src` pointer or exceed the limits of `src` pointer, you will not get any credit.
- Speed-up: For each part(Sobel and Mirror) the 3 student with the lowest amount of CPE's (highest speedup) will receive 60 and 40 points for their each implementation. Rest of the grades will be scaled accordingly, based on their standing in between highest cpe score of the top 3 student and base line cpe count.
- 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 `outer` function will be taken into account.

Regulations

- **Programming Language:** You must use ANSI C.
- **Submission:** Submission will be done via ODTUClass. Submit a single c source file named kernels.c which will be modified version of supplied kernels.c source file.
- **Late Submission:** Late submission is not allowed.
- **Cheating: We have zero tolerance policy for cheating.** People involved in cheating will be punished according to the university regulations.
- **Newsgroup:** You must follow ODTUClass for discussions and possible updates on a daily basis. In order to have an idea of what will be your grade, you should know about speedups of other students. Therefore, sharing your highest speedups is highly recommended.