Middle East Technical University | Department of Computer Engineering

CENG 331

Computer Organization

Fall '2022-2023

Performance Lab Homework

Deadline: 9 January 2023, Monday, 23:59

1 Objectives

This assignment deals with optimizing memory-intensive code. Image processing and machine learning tasks have matrix operations that offer many examples of functions that can benefit from optimization. In this homework, we will consider two matrix operations used often in machine learning (Convolution Neural Networks[1] specifically) and rarely in image processing.

The first function is the convolution operation that calculates the element-wise multiplication and summation of a section of the image with a filter (or kernel) and changes that section by striding the filter across the image. This is done for all depth values of the image. Normally filter size, striding step size and filter counts are adjusted to specific requirements. In this homework all of these parameters are fixed and there is only a single filter. Finally, channel values of an image are summed together so that each filter creates only a single matrix.

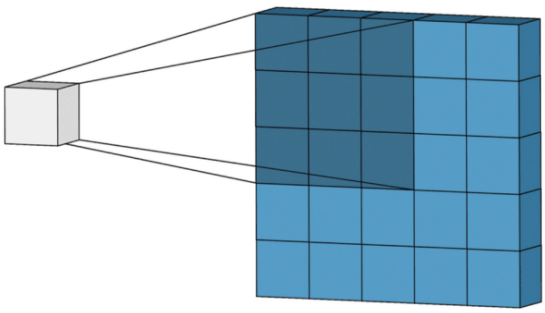

Figure 1: Example Convolution

For the convolution function, we will consider an image and a filter to be represented as a two-dimensional pixel matrix M and K, where $M_{i,j}$ denotes the value of (i, j)th pixel in the image and $K_{i,j}$ are the filter values. Pixel values are triples of red, green, and blue (RGB) values. Filter size is fixed to

[1] https://en.wikipedia.org/wiki/Convolutional_neural_network

be $(8, 8)$ and striding step size is 1. 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 `convolution` operation is implemented by multiplying filter size sections of the image with the filter values and summing the results. The filter then strode across the image. The general formula of the resulting matrix for the depth-wise convolution operation can be seen below:

$$\mathtt{R_{ij}} = \sum_{\mathtt{red,green,blue}} \sum_{\mathtt{k=0}}^{7} \sum_{\mathtt{l=0}}^{7} \mathtt{M_{(i+k)(j+l)}} \times \mathtt{K_{kl}}$$

where i and j values range from 0 to $N-8-1$ and `M` is the source image, `K` is the kernel and `R` is the resulting matrix.

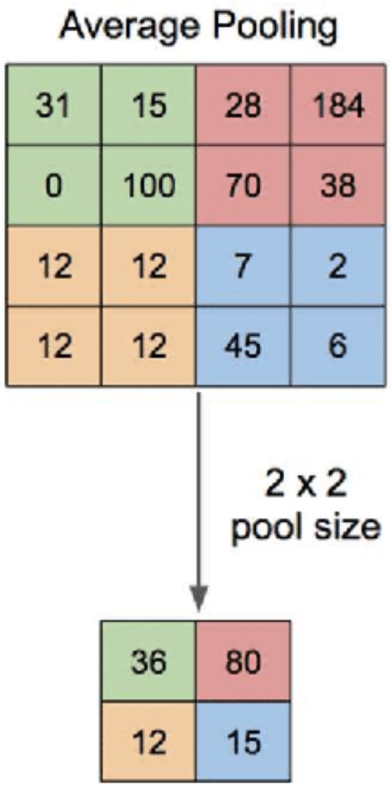

Figure 2: Example Average Pooling

Second function is called average pooling that reduces the size of the images by taking the average of a section. This way each section converted to a single pixel. The block sizes can change depending on the application. In this homework you will be using $(2, 2)$ sized sections for this function. The formula can be seen below:

$$R_{ij} = \frac{\sum_{k=0}^{1} \sum_{l=0}^{1} M_{(i+k)(j+l)}}{4}$$

where `M`, is the source matrix and `R` is the image.

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 your team. **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.

Convolution

The convolution function takes as input a source image `src`, filter image `ker` and returns the convoluted result in the destination image `dst`. Here is the part of an implementation:

```
void naive_conv(int dim, pixel *src, pixel *ker, unsigned *dst) {
    int i,j,k,l;

    for(i = 0; i < dim-8+1; i++)
        for(j = 0; j < dim-8+1; j++) {
            dst[RIDX(i, j, dim)] = 0;
            for(k = 0; k < 8; k++)
                for(l = 0; l < 8; l++) {
                    dst[RIDX(i, j, dim)] += src[RIDX((i+k),(j+l), dim)].red *
                    ker[RIDX(k, l, 8)].red;
                    dst[RIDX(i, j, dim)] += src[RIDX((i+k),(j+l), dim)].green *
                    ker[RIDX(k, l, 8)].green;
                    dst[RIDX(i, j, dim)] += src[RIDX((i+k),(j+l), dim)].blue *
                    ker[RIDX(k, l, 8)].blue;
                }

        }
}
```

Your task is to optimize `convolution` to run as fast as possible. This code is in the file `kernels.c`.

Average Pooling

The average pooling function takes as a single image matrix `src` and returns the pooled result in the destination matrix `dst`. Here is the implementation:

```
for(i = 0; i < dim/2; i++)
        for(j = 0; j < dim/2; j++) {
            dst[RIDX(i, j, dim/2)].red = 0;
            dst[RIDX(i, j, dim/2)].green = 0;
            dst[RIDX(i, j, dim/2)].blue = 0;
            for(k = 0; k < 2; k++) {
                for (l = 0; l < 2; l++) {
                    dst[RIDX(i, j, dim/2)].red += src[RIDX(i*2 + k, j*2 + l, dim)].red;
                    dst[RIDX(i, j, dim/2)].green += src[RIDX(i*2 + k, j*2 + l, dim)].green;
                    dst[RIDX(i, j, dim/2)].blue += src[RIDX(i*2 + k, j*2 + l, dim)].blue;
                }
            }
            dst[RIDX(i, j, dim/2)].red /= 4;
            dst[RIDX(i, j, dim/2)].green /= 4;
            dst[RIDX(i, j, dim/2)].blue /= 4;
    }
```

Your task for this function is to optimize average pooling function to cross a threshold

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

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

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.

Test case	1	2	3	4	5	
Method N	32	64	128	256	512	Geom. Mean
Naive `convolution` (CPE)	277.0	358.6	404.1	427.8	439.5	
Optimized `convolution` (CPE)	172.3	224.1	252.8	267.7	275.7	
Speedup (naive/opt)	1.6	1.6	1.6	1.6	1.6	1.6
Method N	32	64	128	256	512	Geom. Mean
Naive `average pooling` (CPE)	6.9	6.6	6.5	6.5	6.5	
Optimized `average pooling` (CPE)	3.0	3.0	3.0	3.1	3.0	
Speedup (naive/opt)	2.3	2.2	2.2	2.1	2.2	2.2

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

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 `convolution` and `average_pooling` 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_conv_functions() {
    add_conv_function(&convolution, convolution_descr);
}

void register_average_pooling_functions() {
    add_average_pooling_function(&average_pooling, average_pooling_descr);
}
```

These functions contains one or more calls to `add_conv_function` and `add_average_pooling_function`. In one of the examples above,
`add_conv_function` registers the function `convolution` along with a string `convolution_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. Average pooling 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 `convolution()` and `average_pooling()` 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 `convolution()` and `average_pooling()` 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 `team` in `kernels.c` with information(IDs, names) about you or your teammates if you have any.

5 Assignment Details

Optimizing Convolution (70 points)

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

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

```
unix> ./driver
Team Name: TEAM
ID1: eXXXXXX
Name1: NAME

conv: Version = naive_conv: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       277.8   358.3   403.8   426.8   439.5
Baseline CPEs   277.0   358.0   404.0   427.0   439.0
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

Optimizing Average Pooling (30 points)

In this part, you will optimize `average_pooling` to achieve as low a CPE as possible to cross a speed-up threshold. 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 `average_pooling`) generates the output shown below:

```
unix> ./driver
Team Name: TEAM
ID1: eXXXXXX
Name1: NAME

...

...


Avpol: Version = Naive Average Pooling: Naive baseline implementation:
Dim             32      64      128     256     512     Mean
Your CPEs       6.9     6.6     6.5     6.5     6.5
Baseline CPEs   6.8     6.6     6.5     6.5     6.5
Speedup         1.0     1.0     1.0     1.0     1.0     1.0
```

Some advice. Look at the assembly code generated for the `convolution` and `average_pooling`. 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.
- Convolution: You will get 30 points for implementation of `convolution` if it is correct and achieve mean speed-up threshold of 1.6. The team that achieves the biggest speed-up will get 70 points. Other grades will be scaled between 30 and 70 according to your speed-up. You will not get any partial credit for a correct implementation that does below the threshold.

- Average Pooling: You will get 30 points if your implementation is correct and achieve a mean speed-up of 2.2. 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.