

CENG 331

Computer Organization

Fall 2017-2018

Optimization Homework

Due date: 24 January 2018, Wednesday, 23:30

1 Objectives

Deep learning contains many functions that are memory intensive. Therefore optimization of memory intensive functions are quite important, in order to be able to work on large datasets, which most of the time required for higher success rates for deep learning. In this homework, we will consider two of this functions : **Matrix Multiplication**, which multiplies two matrices, and **Convolution**, which is used for blurring, sharpening, embossing, edge detection etc. in image processing, and extracting features from data in deep learning.

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 will 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

Matrix Multiplication

The naive approach for matrix multiplication is given in `kernels.c` as

```
void naive_matrix_multiplication(int dim,int *src, int *src2,int *dst) {
    int i,j,k;

    for(i = 0; i < dim; i++)
        for(j = 0; j < dim; j++) {
            dst[j*dim+i]=0;
            for(k = 0; k < dim; k++)
                dst[j*dim+i] = dst[j*dim+i] + src[j*dim+k]*src2[i+k*dim];
        }
}
```

where the arguments to the procedure are pointers to the destination (`dst`) and two sources (`src,src2`) matrices, as well as the matrix size $N(\text{dim})$. Overall what this function does is multiplying `src` with `src2` and obtaining `dst` (`dst = src*src2`)

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

Convolution

Convolution is used to modify and capture the spatial characteristics in both image processing and deep learning. A convolution is done by selecting a region on your main matrix and multiplying the values in that region by another smaller matrix, which is called as kernel.

For example, lets say you have a source matrix (`src`) and kernel matrix (`ker`)

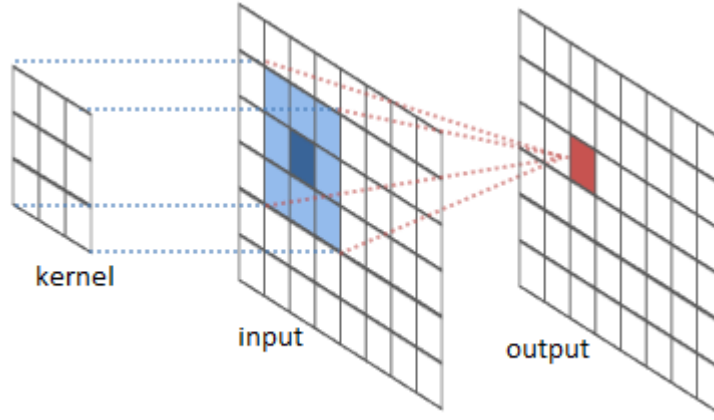
$$\text{src} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$\text{ker} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Your output matrix (`dst`) will look like

$$\text{dst} = \begin{bmatrix} 1*a + 2*b + 4*c + 5*d & 2*a + 3*b + 5*c + 6*d \\ 4*a + 5*b + 7*c + 8*d & 5*a + 6*b + 8*c + 9*d \end{bmatrix}$$

In normal Image processing convolution kernel first rotated, and after that it applied on the source image. However, you are not required to do that in this homework. For additional information and examples about convolution operation click on [here](#).



The naive approach for Convolution is given in `kernels.c` as

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

    for(i = 0; i < dim-8+1; i++)
        for(j = 0; j < dim-8+1; j++) {
            dst[j*dim+i] = 0;
            for(k = 0; k < 8; k++)
                for(l = 0; l < 8; l++) {
                    dst[j*dim+i] = dst[j*dim+i] +src[(j+l)*dim+(i+k)]*ker[l*dim+k];
                }
        }
}
```

where the arguments to the procedure are pointers to the destination (`dst`), source (`src`) and kernel(`ker`) matrices, as well as the matrix size N (`dim`). You will only be using top left most 8×8 region of `ker` matrix for your kernel.

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

Performance measures

The main performance measure is *CPE* or *Cycles per Element*. If a function takes C cycles to run for an matrix 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

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

Assumptions

Assume that N is a multiple of 32 and the dimensions of kernel as 8×8 . For `convolution`, dimensions of destination matrix can be determined by $N - K + 1$ where K is equal to dimension size of the kernel. Your code must run correctly for all such values of N , but we will measure its performance only for $N = \{32, 64, 128, 256, 512\}$.

4 Infrastructure

We provided support code to test the correctness of your implementations and measure their performance. The exact details of each part of the assignment is described in the following subsections.

Versioning

You will be writing many versions of the `matrix_multiplication` and `convolution` functions. In order to compare the performance of different versions, you can register your functions by using `register_conv_functions()` and `register_matrix_multiplication_functions()`. For example, `register_conv_functions()` defined in `kernels.c` as,

```
void register_conv_functions() {
    add_conv_function(&convolution, convolution_descr);
}
```

This function contains one or more calls to `add_conv_function`. In the above example, `add_conv_function` registers the function `convolution` along with a string `convolution_descr` which is an ASCII description of what the function does. ASCII description of the functions can be at most 256 characters long.

Driver

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

```
unix> make driver
```

Recompile driver each time you change the code in `kernels.c`. Command to test your implementation:

```
unix> ./driver
```

The `driver` run in four different modes:

- *Default mode*, in which all versions of your implementation are executed.
- *Autograder mode*, in which only the `convolution()` and `matrix_multiplication()` functions are executed. This is the mode that we will run driver with, when we are grading your implementations.
- *File mode*, in which only versions that are mentioned in an input file are executed.
- *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 versions that you'd like to test by using the *file mode*.

If you run `driver` without any arguments it will work in *Default mode*. Other modes and options can be specified by command-line arguments to `driver`, as listed below:

- `-g` : Run only `convolution()` and `matrix_multiplication()` 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* for each 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 Convolution (50 points)

In this part, you will optimize `Convolution` to achieve as low 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 `Convolution` generates the output shown below:

```
unix> ./driver
```

```
ID: eXXXXXXX
```

```
Name: Fatih Can Kurnaz
```

```
Email: eXXXXXXX@ceng.metu.edu.tr
```

```
conv: Version = naive_conv: Naive baseline implementation:
Dim           32      64      128      256      512      Mean
Your CPEs     213.1   284.5   321.2   344.1   355.1
Baseline CPEs 213.1   283.2   322.6   342.9   354.6
Speedup       1.0     1.0     1.0     1.0     1.0     1.0
```

Optimizing Matrix Multiplication (50 points)

In this part, you will optimize `Matrix Multiplication` to achieve as low 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 `Matrix Multiplication` generates the output shown below:

```
unix> ./driver
```

```
ID: eXXXXXXX
```

```
Name: Fatih Can Kurnaz
```

```
Email: eXXXXXXX@ceng.metu.edu.tr
```

```
Multip: Version = Naive_matrix_multiplication: Naive baseline implementation:
```

Dim	32	64	128	256	512	Mean
Your CPEs	146.2	302.1	637.7	1367.5	3370.7	
Baseline CPEs	150.0	302.8	638.7	1376.9	3348.4	
Speedup	1.0	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 `convolution` and `matrix_multiplication` will each count for 50% of your grade respectively. The grade 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 matrices of other sizes. As mentioned earlier, you may assume that the matrix dimension is multiple of 32.
- **Speed-up:** For each part(`Convolution` and `Matrix Multiplication`) the 3 student with the lowest amount of CPE's (highest speedup) will receive 50 points for their 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 because of CPU status, run `driver` many times and take only the best scores into consideration. While evaluating we will run your codes many times in a closed environment ,and only your best speed-ups will be taken into account.

6 Regulations

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