Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall '2024-2025

The Performance Lab

Due Date : January 9, Thursday, 2024, 23:59

1 Objectives

This assignment focuses on optimizing memory-intensive code, with examples drawn from image processing and matrix operations. Specifically, we will work on optimizing two functions: **Normalization** and **Kronecker Product**. Your goal is to apply the techniques learned in class to achieve the most efficient implementations possible for these functions.

Normalization is a statistical process that comes in various forms. For this assignment, we will use min-max scaling. Given a source matrix M, normalization involves finding the minimum and maximum values of M and scaling each element according to the formula:

$$M^{'} = \frac{M - M_{min}}{M_{max} - M_{min}}$$

Here, $M^{'}$ is the resulting normalized matrix, M_{min} is the minimum value of M, and M_{max} is the maximum value of M.

The Kronecker Product is an operation that involves two matrices. Let the operand matrices be of sizes $N \times N$ and $M \times M$. The operation scales the second matrix by each element of the first matrix, producing a larger matrix by concatenating these scaled matrices. The resulting matrix has a size of $N \cdot M \times N \cdot M$.

An example of the Kronecker Product is shown below:

$$\begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 10 & 12 & 14 \\ 0 & 0 & 0 & 16 & 18 & 20 \\ 0 & 0 & 0 & 22 & 24 & 26 \\ 5 & 6 & 7 & 15 & 18 & 21 \\ 8 & 9 & 10 & 24 & 27 & 30 \\ 11 & 12 & 13 & 33 & 36 & 39 \end{bmatrix}$$

Here, each element of the first matrix scales the second matrix, and the results are arranged to form the final matrix.

Your task is to implement and optimize these two functions, ensuring efficient memory usage and computation speed.

2 Specifications

Begin by copying the file `perflab-handout.tar` to a secure directory where you intend to work. Extract the contents by running the following command:

```
tar xvf perflab-handout.tar
```

This will unpack several files into your directory. Among these, the only file you will modify and submit is `kernels.c`.

The `driver.c` file serves as a driver program to evaluate the performance of your solutions. To compile the driver, use the command:

```
make driver
```

Once compiled, you can run the driver program with:

```
./driver
```

In the `kernels.c` file, you will find a C structure named `team`. Make sure to fill in the required identifying information about yourself in this structure **immediately** to avoid forgetting later.

3 Implementation Overview

Normalization

The `normalize` function takes two matrices; source `src`, and destination `dst` and is implemented as:

```
void naive_normalize(int dim, float *src, float *dst) {
    float min, max;
    min = src[0];
    max = src[0];

    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            if (src[RIDX(i, j, dim)] < min) {
                min = src[RIDX(i, j, dim)];
            }
            if (src[RIDX(i, j, dim)] > max) {
                max = src[RIDX(i, j, dim)];
            }
        }
    }

    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            dst[RIDX(i, j, dim)] = (src[RIDX(i, j, dim)] - min) / (max - min);
        }
    }
}
```

, where RIDX is a macro defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

See the file `defs.h` for this code.

Kronecker Product

The `kronecker_product` function takes as two matrices `mat1`, `mat2` and returns the Kronecker Product in the destination matrix `prod` as shown below:

```
void naive_kronecker_product(int dim1, int dim2, float *mat1, float *mat2, float *prod) {
    for (int i = 0; i < dim1; i++) {
        for (int j = 0; j < dim1; j++) {
            for (int k = 0; k < dim2; k++) {
                for (int l = 0; l < dim2; l++) {
                    prod[RIDX(i, k, dim2) * (dim1 * dim2) + RIDX(j, l, dim2)] =
                                mat1[RIDX(i, j, dim1)] * mat2[RIDX(k, l, dim2)];
                }
            }
        }
    }
}
```

Performance Measures

Our primary performance metric is **CPE** (Cycles Per Element). If a function takes C cycles to process a matrix of size $N \times N$, the CPE is calculated as:

$$CPE = \frac{C}{N^2}$$

Table 1 summarizes the performance of the naive implementations discussed above and compares them to an optimized implementation. Performance is reported for six different values of N. Note that, for `kronecker_product` function, the size of the `prod` matrix is N. All measurements were conducted on the department computers (inek's).

The **speed-ups** (ratios) achieved by the optimized implementation compared to the naive one will determine your implementation's score. To summarize the overall performance across different values of N, we compute the **geometric mean** of the speed-up ratios. Specifically, if the speed-ups for $N = \{32, 64, 128, 256, 512, 1024\}$ are R_{32}, R_{64}, R_{128}, R_{256}, R_{512}, and R_{1024} the overall performance is calculated as:

$$R = \sqrt[6]{R_{32} \cdot R_{64} \cdot R_{128} \cdot R_{256} \cdot R_{512} \cdot R_{1024}}$$

Assumptions

To make life easier, you can assume that N is a multiple of **32** and for the `kronecker_product` function the dimensions of the two operand matrices are multiples of **4** and their product equals N. Your code must run correctly for all such values of N, but we will measure its performance only for the values shown in Table 1.

Test case	1	2	3	4	5	6	
Method N	32	64	128	256	512	1024	Geom. Mean
Naive normalize (CPE)	7.7	7.5	7.4	7.3	7.3	7.3	
Optimized normalize (CPE)	6.7	6.7	6.7	6.7	6.7	6.8	
Speedup (naive/opt)	1.1	1.1	1.1	1.1	1.1	1.1	1.1
Method N	32	64	128	256	512	1024	Geom. Mean
Naive kronecker_product (CPE)	6.2	5.7	6.1	5.6	5.6	5.4	
Optimized kronecker_product (CPE)	2.3	1.7	2.1	1.7	1.8	1.6	
Speedup (naive/opt)	2.7	3.4	2.9	3.4	3.2	3.4	3.1

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

4 Infrastructure

Support code is provided to verify the correctness of your implementations and measure their performance. This section explains how to use this infrastructure. Detailed instructions for each part of the assignment are described in the following section.

Note: The only source file you will modify is `kernels.c`.

Versioning

You will be implementing multiple versions of the `kronecker_product` and `normalize` functions. To facilitate performance comparisons among these versions, we provide a function registration mechanism.

For example, the provided `kernels.c` file includes the following functions:

```
void register_normalize_functions(){
    add_normalize_function(&normalize,normalize_descr);
}
void register_kronecker_product_functions(){
    add_kronecker_product_function(&kronecker_product, kronecker_product_descr);
}
```

These functions register one or more implementations of `add_normalize_function` and `add_kronecker_product_function`. For example, the call to `add_kronecker_product_function` registers the function `kronecker_product` along with a string `kronecker_product_descr`, which is a brief ASCII description of the implementation. You can refer to `kernels.c` to learn how to create these string descriptions.

The string description is limited to a maximum of 256 characters.

Driver

Your code will be linked with provided object files to create a driver binary. To compile the driver, execute:

```
    unix> make driver
```

Each time you modify `kernels.c`, you will need to re-run this command to rebuild the driver.

To test your implementations, run the driver with:

```
    unix> ./driver
```

The `driver` can be run in four different modes:

- *Default mode* runs all registered versions of your implementations.
- *Autograder mode* runs only the `kronecker_product()` and `normalize()` functions. This is the mode used for grading.
- *File mode* executes only the versions specified in an input file.
- *Dump mode* outputs a one-line description of each registered version to a text file. You can edit this file to select specific versions to test in file mode. You can also specify whether to exit after dumping the file or proceed to run your implementations.

If run without any arguments, the driver defaults to **Default Mode**, executing all registered versions. Other modes and options can be specified using the following command-line arguments:

`-g` : Runs only `kronecker_product()` and `normalize()` functions (*Autograder Mode*).

`-f <funcfile>` : Executes only those versions specified in `<funcfile>` (*File Mode*).

`-d <dumpfile>` : Dumps the names of all versions to `<dumpfile>` (*Dump Mode*).

`-q` : Quits immediately after creating the dump file. Typically used with -d. For example, to generate a dump file and quit:

```
     ./driver -qd <dumpfile>
```

`-h` : Displays the command line usage.

5 Assignment Details

Optimizing Normalization (40 points)

In this part, you will optimize `normalize` to minimize its CPE. To test your implementation, compile `driver` and run it with the appropriate arguments.

For instance, running the driver with the provided naive implementation of `normalize` produces the following output:

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

Normalize: Version = naive_normalize: Naive baseline implementation:
```

```
Dim             32      64      128     256     512     1024    Mean
Your CPEs       7.5     7.5     7.4     7.3     7.3     7.3
Baseline CPEs   7.7     7.5     7.4     7.3     7.3     7.3
Speedup         1.0     1.0     1.0     1.0     1.0     1.0     1.0
```

Optimizing Kronecker Product (60 points)

In this part, you will optimize `kronecker_product` to minimize its CPE. To test your implementation, compile `driver` and run it with the appropriate arguments.

For instance, running the driver with the provided naive implementation of `kronecker_product` produces the following output:

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

Kronecker Product: Version = Naive Kronecker Product: Naive baseline implementation:
Dim             32      64      128     256     512     1024    Mean
Your CPEs       6.2     5.6     5.8     5.5     5.6     5.4
Baseline CPEs   6.2     5.7     6.1     5.6     5.6     5.4
Speedup         1.0     1.0     1.0     1.0     1.0     1.0     1.0
```

Hint. Look at the assembly code generated for the `normalize` and `kronecker_product`. Focus on optimizing the inner loop (the code 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 the 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 fail! This includes implementations that work correctly for the provided test sizes but fail for matrices of other sizes. As stated earlier, you may assume that the resultant matrix dimensions are multiples of 32.
- You are allowed to modify only the `dst` and the `prod` pointer. Any modifications to input pointers or attempts to access data beyond their limits will result in **no credit**.

- Normalization: You will earn **30 points** if your implementation is correct and achieves a mean speed-up of 1.1. The student that achieves the highest speed-up will receive **40 points**. Other grades will be scaled between 30 and 40 based on your speed-up. **No partial credit** will be awarded for correct implementations that fall below the threshold.
- Kronecker Product: You will earn **40 points** for a correct implementation of `kronecker_product` if it achieves a mean speed-up threshold of 3.1. The student that achieves the highest speed-up will receive **60 points**. Other grades will be scaled between 40 and 60 based on your speed-up. **No partial credit** will be awarded for correct implementations that fall below the threshold.

  NOTE: To gauge your potential grade, it is recommended to share your highest speed-ups with other students. This will give you a sense of the competition.
- Since CPU performance can fluctuate, test your code multiple times and consider only the best result. During evaluation, your code will be tested in a controlled environment multiple times, and only the best speed-up for each function will be used.

Submission

Submissions must be made via ODTUClass. You should only submit the `kernels.c` file. Ensure that all your changes are contained within this file.