Middle East Technical University Department of Computer Engineering

CENG 331

Computer Organization

Fall '2023-2024

The Performance Lab

Due Date : January 6, Saturday, 2024, 23:59
Late Policy : January 7, Sunday, 2024, 23:59 (with 10 points penalty)

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 **Batched Matrix Multiplication** and **Point Reflection**. Your objective in this homework is to optimize these functions as much as possible by using the concepts you have learned in class.

Point reflection is a geometric transformation that involves flipping an object or a set of points across a fixed point known as the 'center of reflection'. In the context of a matrix, this transformation mirrors the matrix along its center. Visually, point reflection on a matrix is akin to creating a symmetrical image by mirroring points along the x and y axes, both passing through the center of the matrix. This process results in a flipped representation on the opposite side of the center, as shown in Figure 1.

Figure 1: The point reflection

The second function, batched matrix multiplication, is used for neural network computations and operates on tensors and matrices. In this context, a tensor is treated as a batched matrix of size $N \times N \times N$, where the first dimension represents the batch dimension. The operand matrix is of size $N \times N$. The function performs matrix multiplication for each matrix in the tensor with the operand matrix, and the resulting matrix is obtained by summing the products of these individual matrix multiplications. The tensor is effectively a collection of matrices processed in a batch-wise manner, contributing to the final result. The formula of the resulting matrix can be seen below:

$$M_{jk} = \sum_{i=0}^{n} \sum_{l=0}^{n} B_{ijl} * A_{lk}$$

where M, B and A represents the resulting matrix, the batched matrices, and the operand matrix, respectively. For clarity, when the described batched matrix multiplication is executed for a tensor (batched matrix) of the dimensions $n \times m \times k$ and an operand matrix of size $k \times j$, the resulting matrix will have dimensions $m \times j$.

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

Point Reflection

The `reflect` function takes two matrices; source `src`, and destination `dst`. Here is the implementation:

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

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

Here RIDX is a macro defined as follows:

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

See the file defs.h for this code.

Batched Matrix Multiplication with Sum Reduction

The `batched_mm` function takes as two matrices `b_mat`, `mat` and returns the batched matrix multiplication after sum reduction over batch dimension in the destination matrix `dst`. Here is the implementation:

```
void naive_batched_mm(int dim, int *b_mat, int *mat, int *dst) {
    int i,j,k,l;

    for (i = 0; i < dim; i++) {
```

```
        for (j = 0; j < dim; j++) {
            for (k = 0; k < dim; k++) {
             if (i == 0){
                 dst[RIDX(j, k, dim)] = 0;
             }
             for (l = 0; l < dim; l++){
                    dst[RIDX(j, k, dim)] += b_mat[RIDX(i*dim+j, l, dim)]
                                                    * mat[RIDX(l, k, 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 5 different values of N. All measurements were made on the the department computers (ineks).
The ratios (speed-ups) 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 speed-ups 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

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

$$R_{batched\_mm} = \sqrt[3]{R_{32} \times R_{64} \times R_{128}}$$

Test case	1	2	3	4	5	
Method N	32	64	128	256	512	Geom. Mean
Naive reflect (CPE)	2.7	2.6	2.7	2.7	2.8	
Optimized reflect (CPE)	0.9	0.9	0.9	1.0	1.0	
Speedup (naive/opt)	3.0	2.9	2.9	2.8	2.8	2.9
Method N	32	64	128			Geom. Mean
Naive batched_mm (CPE)	4812.3	19340.0	85466.2			
Optimized batched_mm(CPE)	427.0	1633.4	7964.9			
Speedup (naive/opt)	11.3	11.8	10.7			11.3

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

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 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 `batched_mm` and `reflect` 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_batched_mm_functions(){
    add_batched_mm_function(&batched_mm, batched_mm_descr);
}
void register_reflect_functions(){
    add_reflect_function(&reflect, reflect_descr);
}
```

This function contains one or more calls to `add_batched_mm_function` and `add_reflect_function`. In one of the examples above, `add_batched_mm_function` registers the function `batched_mm` along with a string `batched_mm_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. Point reflection 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 `batched_mm()` and `reflect()` 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 `batched_mm()` and `reflect()` 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 Point Reflection (30 points)

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

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

Point Reflect: Version = Naive Point Reflection: Naive baseline implementation:
Dim             32     64     128    256    512    Mean
Your CPEs       2.7    2.7    2.7    2.7    2.6
Baseline CPEs   2.7    2.6    2.7    2.7    2.8
Speedup         1.0    1.0    1.0    1.0    1.1    1.0
```

Optimizing Batched Matrix Multiplication (70 points)

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

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

Batched MM: Version = naive_batched_mm: Naive baseline implementation:
Dim               32        64          128         Mean
```

```
Your CPEs          4812.3    19270.9     85466.2
Baseline CPEs      4812.0    19340.0     85426.0
Speedup            1.0       1.0         1.0         1.0
```

Some advice. Look at the assembly code generated for the `reflect` and `batched_mm`. 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 matrices of other sizes. As mentioned earlier, you may assume that the matrix 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.
- Point Reflection: You will get 30 points if your implementation is correct and achieve a mean speed-up of 2.9. There is no scaling in this function.
- Batched Matrix Multiplication: You will get 50 points for implementation of `batched_mm` if it is correct and achieve mean speed-up threshold of 11.3. The team that achieves the highest speed-up will get 70 points. Other grades will be scaled between 50 and 70 according to your speed-up. You will not get any partial credit for a correct implementation that does below the threshold.
  NOTE: In order to have an idea of what will be your grade, you should know about speed-ups of other students. Therefore, sharing your highest speed-ups is highly recommended.
- 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 two people**. However, you can also do it alone.

Submission

Submissions will be 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.