ECE 5755 Modern Computer Systems and Architecture, Fall 2023
Assignment 3: Implementing Sparse Multiplication

1. Introduction

This lab assignment will build off the work you completed in the previous lab assignment. In particular, you will be exploring the efficiency gains in your kernel when you implement **sparse matrix multiplication**

2. Materials

Use all of the same materials from Lab 1. We are still using the course virtual machine for this (and all future) labs. Make sure to make a copy of your original lab 1 code or use a version control tool like git so that you have a known good state to revert to. You will add an additional function `matmul_sparse` which implements a version of `matmul` that exploits sparsity in matrix inputs to improve efficiency.

3. Compressed Sparse Row (CSR) Format

Many matrices are often sparse, meaning they have many zero-valued terms. Since zero multiplied by anything is still zero, multiplications with zero can be effectively skipped. A sparse representation of a matrix removes the zeros and stores only the non-zero values and their locations. The compressed sparse row (CSR) format of a 2D matrix uses three one-dimensional arrays or vectors to represent the matrix.

The first array stores all the non-zero values of the matrix in row-major order, i.e. values from lower-indexed rows precede values from higher-indexed rows, and values from the same row are stored in order from lower column index to higher column index. The second array is the row-index array which stores the cumulative number of non-zero values after each row. The row-index array is of length m + 1 for a matrix with m rows. The first value in the row-index array is always 0, the second value of the row-index array is the cumulative number of non-zero values from the first row, the third value of the row-index array is the cumulative number of non-zero values from the first and second rows, etc. The third array is the column-index array which stores the column index of each of the non-zero values in the values array. Its length is equal to the number of non-zero values.

Below is a simple example from https://en.wikipedia.org/wiki/Sparse_matrix

$$\begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

Its representation in CSR format is:

```
Values: {10, 20, 30, 40, 50, 60, 70, 80}
Row-Index Array: {0, 2, 4, 7, 8}
Col-Index Array: {0, 1, 1, 3, 2, 3, 4, 5}
```

4. Objective

The objective of this lab is to implement a better matrix-multiplication algorithm that takes advantage of matrix sparsity, and **to analyze the performance gains of your algorithm using microarchitectural analysis techniques**. You are tasked with implementing the following function:

```
float **matmul_sparse(float **A, float **B, int A_rows, int A_cols,
int B_rows, int B_cols) {

    float **result;

    /**** 1. Create CSR format of input matrix ****/

    /**** 2. Perform matrix multiplication on CSR format of input
matrix ****/

    // When profiling, only loop over part 2.

    return result;
}
```

`matmul_sparse` will comprise two parts: 1. Creating a compressed-sparse row (CSR) representation of the input matrix, and 2. Performing matrix multiplication on the CSR matrix. `matmul_sparse` should return the same results as `matmul` and `matmul_blocking`, i.e. the output matrix as a multidimensional array. Parts 1 and 2 are

kept within the same `matmul_sparse` function so that the interface is the same as that of `matmul` and `matmul_blocking`.

The function should be put in `kernel/matrix_ops.c`. Don't forget to put the function declaration in the header file (`kernel/matrix_ops.h`) as well so the lab will compile. You'll then want to swap out your original matrix multiplication implementation for this one in your tests to verify that it's actually working, i.e. call `matmul_sparse` instead of `matmul` inside `tests/test_matrix_ops.c`

When testing for runtime, you can use the Linux `time` command; make sure to take the `real` time which is the wall-clock time. Moreover, as with profiling, make sure you are looping over the function you want to profile so that the runtime of the entire program is dominated by the function you are profiling/timing. **In the case of sparse matrix multiply, only loop over part 2, i.e. the part where convolution is performed on the CSR matrix.** You can consider part 1. creating the CSR representation as part of set up code. Test with a variety of input sizes; **make sure you have some inputs with high sparsity and some inputs with low sparsity.**

For Lab 3, you are only required to submit `matrix_ops.c` and `matrix_ops.h` with `matmul_sparse` and any additional helper functions you choose to use inside those two files.

For this lab, we will not be grading your testing procedure, only the correctness of `matmul_sparse`. Functional correctness grading will be done with an internal grading program similar to the testing programs you were expected to submit for Lab 1. We will also be reading your code to grade for correctness in implementing tiling/blocking, so please make sure your code is readable (use comments to clarify your code).

Your profiling process must meet the following requirements:

- Use `toplev.py` from `pmu-tools` for Top-down analysis data
- Profile only the core the program is running on
- Runtime of the program-under-test should be dominated by the function-under-test (you can accomplish this by looping over your function many times)

The specific commands you use for profiling and testing and how you write your program-under-test is up to you as long as your process meets the above requirements. One option is to use the existing testing infrastructure under `tests/`.

5. Report

The report will be **3-pages max** but can be shorter if you do not need all three pages. Report requirements/prompts:

- Include a horizontal segmented bar chart for the Top-down analysis of some of the different test inputs of `matmul_sparse` you tried. In the same bar chart, for the same set of inputs, also include the Top-down analysis data for `matmul` from Lab 1 to serve as a baseline.
- For the same inputs, collect their performance data (runtime) on both `matmul_sparse` and `matmul` from Lab 1 and include it in a table in the report.
- Explain in detail your profiling process.
- With reference to computer architecture concepts and the Top-down analysis data, explain the performance differences between the different inputs and between native `matmul` and `matmul_sparse`.
- Discuss any issues you had during the lab and your debugging process.

5. Deliverables

Please submit your assignment on Canvas by **Thursday, Oct. 19, 9:00 pm**. You are expected to submit three files total: **matrix_ops.c, matrix_ops.h,** and your report in a file named **lab3.pdf.**

The lab will be marked out of a **total of 100 marks.**

- **40 marks** for `matmul_sparse()`
    - 20 marks for functional correctness (computes correct outputs for given set of inputs)
    - 20 marks for correct implementation of sparse matmul
- **60 marks** for report
    - 20 marks for Top-down analysis data
    - 10 marks for performance (runtime) data
    - 30 marks for discussion (see prompts in 4. Report)