CS 141 Final Project
Spring 2019

Due May 8th, 2019, 11:59PM

Overall Goal

For your CS 141 final project, you will use the knowledge you have gained over the course of the class to implement, analyze, and optimize matrix multiplication. This will draw upon your work on digital logic design, processor micro-architecture, and how high-level software is mapped to hardware.

Description

1. Modifying your MIPS processor (20 points)

- Implement support for matrix multiplication in your MIPS-style processor.
    - Support for a multiply instruction (extend ALU for this, R-type, mul).
    - Write MIPS assembly for an 8x8 matrix multiplication.
    - We provide an example processor that supports the same instructions as the book. Please see the readme there for additional details, though you may use the processor you wrote instead of the example one.

2. Software Optimization of Matrix Multiplication (30 points)

- Given a baseline (naive) matrix-multiplication implementation (in C), run and report the execution time for varying sizes of input and output matrices.
    a. Please consider only square matrices of the following sizes: 8x8, 32x32, 256x256, 512x512, 1024x1024
- Implement tiled matrix-multiplication (in C).
    a. Vary the tile sizes and matrix sizes, and report the execution time for each case.

3. Written Analysis & Summary (50 points)

- Write a report (up to 5 pages including figures, references, and extra credit) targeting an ML developer. Convince them that optimizing matrix-multiplications is important and something they should consider when building AI-enabled systems.
- Provide results (tables, plots, etc.) for the execution time of the naive matrix multiplication at the provided sizes (8x8, 32x32, 256x256, 512x512, 1024x1024).
- Discuss why the baseline matrix multiplications are insufficient.
- Discuss your tiled matrix multiplication implementation.
- Provide results (tables, plots, etc.) for the execution time of the tiled matrix multiplication at the provided sizes (8x8, 32x32, 256x256, 512x512, 1024x1024)
- Analyze the effectiveness of tiled matrix multiplication as you vary the tile size.

Deliverables

Please submit the following via Canvas by May 8th, 2019 at 11:59PM:

- Project file for modified version of processor
- zip file of all source code for your matrix multiplication implementation & related scripts
- pdf of written analysis

Collaboration

This is primarily an individual project. For the Verilog component **only (1)**, you may discuss with your partner and use code from your existing processor. The software optimization of matrix multiplication (2) and final written analysis (3) should be completely **individually**. Feel free to leverage any and all materials from CS141, and please cite any external resources.

Provided Files

1. Modifying your MIPS processor

- You can extend your MIPS processor or, if you prefer or had issues with your processor, please contact us and we will provide you with a simpler, working starting point.

2. Software Optimization of Matrix Multiplication

- main.c
    a. Implementation of matrix-multiplication for specified input matrix sizes
    b. The following functions are provided for you:

i. `naivemm(float * a, float * b, float * c, int out_rows, int out_cols, int in_cols)`: This function computes the matrix multiplication of two input matrices.
ii. `generate_rand_input(int rows, int cols)`: This function takes the desired number of rows (R) and columns (C) of the input matrix and returns a RxC matrix of random values.
iii. `print_mat(float * mat, int rows, int cols)`: This function prints a given matrix (useful for debugging).
iv. `generate_output(int rows, int cols)`: This function allocates and initializes the output matrix
v. `compare_matrices(float * mat1, float * mat2, int rows, int cols)`: Given two matrices, this function checks that every element is equal (useful for debugging).

- compile.sh
    a. Compiles main.c
- run.sh
    a. Runs the compiled code
    b. Reports execution time

Extra Credit (up to 20 points)

In addition to optimizing matrix multiplication using tiling, there are many other ways to improve performance. For extra credit, propose, implement, and analyze results for additional optimizations. As a starting point, we have given you some ideas below that you can work with.

1. **Reduced Precision Data Type**: One of the reasons we have to tile matrix multiplication in order to improve performance is that the matrices do not fit in caches. One way to accomplish this is by reducing the size of the matrices by using smaller data types. The code provided uses floating point 32-bit values. Explore how the performance varies if you use int32, int16, and int8 data types.
2. **Sparse Matrix Multiplication:** In neural networks, sometimes the input matrices will be sparse (many values are 0). Can you design an optimization that does particularly well for this case? If so, implement it, analyze the performance, and explain why it performs better for sparse matrices.
3. **Parameterize matrix-multiplication sizes:** Your tiled matrix multiplication implementation may have been tailored for the specific matrix sizes provided (i.e., 8x8, 32x32, 256x256, 512x512, 1024x1024). Please implement a version that considers tiling for an arbitrary matrix-multiplication sizes between 32x32 and 1024x1024. Here you are free to assume specifics about the memory system on your computer, such as the L1, L2, and L3 cache sizes. Show results for all matrix sizes, including non-square matrices,

and convince us that your implementation just as good as or outperforms the naive implementation in all cases.

If you choose to do extra-credit, please make it apparent what you have implemented in your write-up with pointers to the code. Extra credit will be given for doing any one of the tasks listed above, or for ones that you come up with! Implementing more than one extra-credit will give you more points.