CSCE 569 Assignment 1

Chris McKinney

This full document is available at https://tachibanatech.com/litdoc/569/assign1.full.pdf in PDF format and https://tachibanatech.com/litdoc/569/assign1.full.d/_book/ in HTML format.

A simplified document is available at https://tachibanatech.com/litdoc/569/assign1.simple.pdf in PDF format.

The source is available via git at https://tachibanatech.com/litdoc/569/csce569assign1.git.

The source is also available at https://tachibanatech.com/litdoc/569/csce569assign1.tar.gz with the pre-tangled target source files and pre-compiled PDF (the PDF generation dependencies are many).

Compiling the target executables requires GCC, OpenMP, and OpenCV.

Tangling the target source files requires Literate (http://literate.zbyedidia.webfactional.com/), Python 3 (https://python.org/), and Bottle (https://bottlepy.org/).

Contents

Matrix-Matrix Multiplication

Chris McKinney

Matrix-Matrix Multiplication

1. General Matrix Multiplication

For an $n \times k$ matrix $\mathbf{A}$ and a $k \times m$ matrix $\mathbf{B}$, where $\mathbf{C} = \mathbf{AB}$:

$$C_{ij} = \sum_{w=1}^{k} A_{iw} B_{wj}$$

This is straightforward to implement in C, but can also be optimized with some simple techniques to increase performance by an order of magnitude. Also, a note: instead of one-indexing as is common in mathematics, C uses zero-indexing, which is what will be used for the remainder of this document.

Initially I implemented the multiplication in the order used in the provided `mm.c`: the variables for the `for` loops, from outer to inner, went `i`, `j`, `w`. One advantage of having the `w` loop be the innermost is that the output matrix does not need to be zeroed before entering the main loop, as each element of the output is full computed by the `w` loop before moving on to the next element. However, zeroing the output matrix is only $O(n^2)$ with a relatively small coefficient, and as such makes little impact on the overall performance.

Moving the `j` loop to be the innermost enables several compiler optimizations that, at `-O3`, interact to give an order of magnitude improvement in performance. I did enough testing to determine that it is neither a single mode of optimization (as represented by GCC's `-f` flags), nor a linear combination of optimization modes, but an interaction between several. `-O3` enables 103 `-f` flags, so I am not currently going to write a script to test all 10 nonillion (10^{31}) combinations.

{Multiply 1}

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        C[i*M + j] = 0.0f;
    }
}
for (i = 0; i < N; i++) {
    for (w = 0; w < K; w++) {
        for (j = 0; j < M; j++) {
            C[i*M + j] += ACCESS_A(i,w) * BT[w*M + j];
        }
    }
}
```

Used in sections 4, 4, 4 and 4

Note that accessing `A` uses `ACCESS_A`, but `B` is not accessed. Instead, there is `BT`. The optimization described above only works if `B` is row-major, and so if `B` is column-major, `BT` is its transpose. As with zeroing `C`, this is only $O(n^2)$ with a relatively small coefficient, and as such does not itself add much time to the calculation, even though it requires dynamic memory allocation. The performance improvement from the GCC optimization is just so great that it makes tradeoffs like this worth it.

{Transpose B 1}

```
REAL *BT = malloc(sizeof(REAL)*K*M);
for (w = 0; w < K; w++) {
    for (j = 0; j < M; j++) {
        BT[w*M + j] = ACCESS_B(w,j);
    }
}
```

Used in sections 4 and 4

Otherwise `BT` is just a pointer to `B`. The `ACCESS_A` and `ACCESS_B` macros are set based on the element order of `A` and `B`, either row-major or column-major.

2. Access to Row-Major Matrices

For row-major matrices, the row number i needs to be multiplied by the row length (that is, the number of columns), as the row number is essentially the number of rows that need to be "skipped" to reach the correct row. Then the column number j is added to index into that row. For example, the index for element $(2, 3)$ in the following 4×6 matrix is $2 \cdot 6 + 3 = 15$.

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \end{pmatrix}$$

So the index of element (i, j) in an $n \times m$ row-major matrix is $i \cdot m + j$. In C:

{Row-Major A 2}

```
#define ACCESS_A(I,J) (A[(I)*K + (J)])
```

Used in sections 4 and 4

{Row-Major B 2}

```
#define ACCESS_B(I,J) (B[(I)*M + (J)])
```

Used in sections 4 and 4

3. Access to Column-Major Matrices

For column-major matrices, the column number j needs to be multiplied by the column length (that is, the number of rows), as the column number is essentially the number of columns that need to be "skipped" to reach the correct column. Then the row number i is added to index into that column. For example, the index for element $(2, 3)$ in the following 4×6 matrix is $2 + 3 \cdot 4 = 14$.

$$\begin{pmatrix} 0 & 4 & 8 & 12 & 16 & 20 \\ 1 & 5 & 9 & 13 & 17 & 21 \\ 2 & 6 & 10 & 14 & 18 & 22 \\ 3 & 7 & 11 & 15 & 19 & 23 \end{pmatrix}$$

So the index of element (i, j) in an $n \times m$ column-major matrix is $i + j \cdot n$. In C:

{Col-Major A 3}

```
#define ACCESS_A(I,J) (A[(I) + (J)*N])
```

Used in sections 4 and 4

{Col-Major B 3}

```
#define ACCESS_B(I,J) (B[(I) + (J)*K])
```

Used in sections 4 and 4

4. Bringing It Together

The function itself basically just switches between copies of the multiplication routine with different definitions for `ACCESS_A` and `ACCESS_B`. After each copy of the multiplication routine, it is probably prudent to undefine the macros:

{Undefine Macros 4}

```
#undef ACCESS_A
#undef ACCESS_B
```

Used in sections 4, 4 and 4

Here is the function. As you can see, if just switches on the orders of the input matrices, transposing B if necessary and multiplying:

{MM Implementation 4}

```
// C[N][M] = A[N][K] * B[K][M]
void mm(int N, int K, int M, REAL *A, REAL *B, REAL *C,
        int A_rowMajor, int B_rowMajor) {
    int i, j, w;
    if (A_rowMajor && B_rowMajor) {
        {Row-Major A, 2}
        {Row-Major B, 2}
        REAL *BT = B;
        {Multiply, 1}
        {Undefine Macros, 4}
    } else if (A_rowMajor && !B_rowMajor) {
        {Row-Major A, 2}
        {Col-Major B, 3}
        {Transpose B, 1}
        {Multiply, 1}
        free(BT);
        {Undefine Macros, 4}
    } else if (!A_rowMajor && B_rowMajor) {
        {Col-Major A, 3}
        {Row-Major B, 2}
        REAL *BT = B;
        {Multiply, 1}
        {Undefine Macros, 4}
    } else {
        {Col-Major A, 3}
        {Col-Major B, 3}
        {Transpose B, 1}
        {Multiply, 1}
        free(BT);
        {Undefine Macros, 4}
    }
}
```

Used in section 6

5. Testing

This program produces a file with the generated inputs and outputs, so that the calculation's validity can be tested by an outside program. The file is three ints: N, K, and M, stored in the native format, and then the two input matrices, also in native format, and then all the output matrices in native format. The file is meant for immediate checking and debugging on the same machine, not results storage.

{Testing 5}

```
// Start the debug file
FILE *debug_file = fopen(".mm.debug", "w");
if (debug_file == NULL) {
    fprintf(stderr, "Could not open file for writing: .mm.debug");
} else {
    int buf[3] = { N, K, M };
    fwrite(buf, sizeof(int), 3, debug_file);
    fwrite(A, sizeof(REAL), N*K, debug_file);
    fwrite(B, sizeof(REAL), K*M, debug_file);
    fflush(debug_file);
}
```

Added to in section 5 Used in section 6

The program uses the provided `read_timer` to test the performance of `mm` for each of the permutations of matrix orders.

{Testing 5} +=

```
// Note: neested functions are a GNU C extension.
double time_mm(int A_rowMajor, int B_rowMajor) {
    // Time mm
    double elapsed_mm = read_timer();
    mm(N, K, M, A, B, C, A_rowMajor, B_rowMajor);
    elapsed_mm  = (read_timer() - elapsed_mm);
    // Output matrix to debug file
    if (debug_file != NULL) {
        fwrite(C, sizeof(REAL), N*M, debug_file);
        fflush(debug_file);
    }
    // Print time
    printf("mm (%d, %d):\t\t\t%4f\t%4f\n", A_rowMajor, B_rowMajor,
            elapsed_mm * 1.0e3, (double)M*N*K / (1.0e6 * elapsed_mm));
    fflush(stdout);
    return elapsed_mm;
}

// Do the timing
time_mm(1, 1);
time_mm(1, 0);
time_mm(0, 1);
time_mm(0, 0);
```

Added to in section 5 Used in section 6

Finally, if there was an error with the debug file, set the exit code to `5`.

{Testing 5} +=

```
if (debug_file == NULL || ferror(debug_file)) {
    status = 5;
}
```

Added to in section 5 Used in section 6

6. Base Code

This is roughly based on the provided `mm.c`, but a lot has been removed.

{**mm.c** 6}

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/timeb.h>
#include <omp.h>

/* read timer in second */
double read_timer() {
    struct timeb tm;
    ftime(&tm);
    return (double)tm.time + (double)tm.millitm / 1000.0;
}

/* read timer in ms */
double read_timer_ms() {
    struct timeb tm;
    ftime(&tm);
    return (double)tm.time * 1000.0 + (double)tm.millitm;
}

#define REAL float
#define VECTOR_LENGTH 512

/* initialize a vector with random floating point numbers */
void init(REAL *A, int N) {
    int i;
    for (i = 0; i < N; i++) {
        A[i] = (double)drand48();
    }
}

{MM Implementation, 4}

{Main, 6}
```

{Main 6}

```
int main(int argc, char **argv) {
    int N = VECTOR_LENGTH;
    int M = N;
    int K = N;
    double elapsed; /* for timing */
    if (argc < 4) {
        fprintf(stderr, "Usage: mm [<N(%d)> <K(%d)> <M(%d)>]\n", N, K, M);
        fprintf(stderr, "\t Example: ./mm %d %d %d (default)\n", N, K, M);
    } else {
        N = atoi(argv[1]);
        K = atoi(argv[2]);
        M = atoi(argv[3]);
    }
    REAL *A = malloc(sizeof(REAL)*N*K);
    REAL *B = malloc(sizeof(REAL)*K*M);
    REAL *C = malloc(sizeof(REAL)*N*M);

    srand48((1 << 12));
    init(A, N*K);
    init(B, K*M);

    /* you should add the call to each function and time the execution */
    printf( "================================================="
            "=====================\n");
    printf("\tC[%d][%d] = A[%d][%d] * B[%d][%d] without OpenMP\n", N, M, N, K, K, M);
    printf( "-------------------------------------------------"
            "---------------------\n");
    printf("Performance:\t\t\tRuntime (ms)\t MFLOPS \n");
    printf( "-------------------------------------------------"
            "---------------------\n");
    fflush(stdout);

    int status = 0;

    {Testing, 5}

    free(A);
    free(B);
    free(C);
    return status;
}
```

Matrix Multiplication Checker

Chris McKinney

Matrix Multiplication Checker

1. Code

{**checker.py** 1}

```
import array
import numpy as np
import struct
import sys

def check_mm():
    correct = True
    with open('.mm.debug', 'rb') as f:
        head = f.read(struct.calcsize('3i'))
        n, k, m = struct.unpack('3i', head)
        a, b, c_rr, c_rc, c_cr, c_cc = [array.array('f') for x in range(6)]
        a.fromfile(f, n*k)
        b.fromfile(f, k*m)
        a_r = np.reshape(a, (n, k), order='C')
        b_r = np.reshape(b, (k, m), order='C')
        a_c = np.reshape(a, (n, k), order='F')
        b_c = np.reshape(b, (k, m), order='F')
        c_rr.fromfile(f, n*m)
        c_mm = np.reshape(c_rr, (n, m), order='C')
        c_np = a_r @ b_r
        if (c_mm != c_np).any():
            print('mm rr failed')
            correct = False
        else:
            print('mm rr passed')
        c_rc.fromfile(f, n*m)
        c_mm = np.reshape(c_rc, (n, m), order='C')
        c_np = a_r @ b_c
        if (c_mm != c_np).any():
            print('mm rc failed')
            correct = False
        else:
            print('mm rc passed')
        c_cr.fromfile(f, n*m)
        c_mm = np.reshape(c_cr, (n, m), order='C')
        c_np = a_c @ b_r
        if (c_mm != c_np).any():
            print('mm cr failed')
            correct = False
        else:
            print('mm cr passed')
        c_cc.fromfile(f, n*m)
        c_mm = np.reshape(c_cc, (n, m), order='C')
```

```
        c_np = a_c @ b_c
        if (c_mm != c_np).any():
            print('mm cc failed')
            correct = False
        else:
            print('mm cc passed')
    return correct

def check_matvec():
    print('matvec checking not yet implemented')
    return False

def show_usage():
    print('Usage: {} [mm | matvec]*'.format(sys.argv[0]))
    raise SystemExit(4)

def main():
    status = 0
    if sys.argv == 1:
        show_usage()
    for arg in sys.argv[1:]:
        if arg == 'mm':
            if check_mm():
                print('mm output correct')
            else:
                print('mm output incorrect')
                status |= 1
        elif arg == 'matvec':
            if check_matvec():
                print('matvec output correct')
            else:
                print('matvec output incorrect')
                status |= 2
        else:
            show_usage()
    raise SystemExit(status)

if __name__ == '__main__':
    main()
```

Plotter

Chris McKinney

Plotter

1. Introduction

This Octave script (Yes, Octave specifically. From my googling, it seems MATLAB doesn't have a reasonable way to handle command line arguments. Octave has `argv`.) plots output from the Fortran programs.

2. Command-Line Argument Handling

The script takes $4p + c$ command line arguments, where c is the total number of curves to plot, and p is the number of separate plots on which to plot them. Usage:

```
plotter.m [ ( XPLOT | YPLOT ) <filename> <xlabel> <ylabel> [ <legend> ]... ]...
```

- `XPLOT` and `YPLOT` starts a new plot with the specified dependent axis.
- `<filename>` is the image file to output to.
- `<xlabel>` and `<ylabel>` specify the axis labels.
- `<legend>` is a legend entry. `NOP` will skip records.

{Usage 2}

```
function usage_and_exit
    fprintf(stderr(), '%s [ ( XPLOT | YPLOT ) <filename> ', program_name());
    fprintf(stderr(), '<xlabel> <ylabel> [ <legend> ]... ]...\n\n');
    fdisp(stderr(), '`XPLOT` and `YPLOT` starts a new plot with the specified dependent axis.');
    fdisp(stderr(), '<filename> is the image file to output to.');
    fdisp(stderr(), '<xlabel> and <ylabel> specify the axis labels.');
    fdisp(stderr(), '<legend> is a legend entry. `NOP` will skip records.');
    exit(1);
endfunction
```

Used in section 4

Here the script loops through the arguments and builds up the figures as the relevant arguments are read. If there is a fatal error later in the argument list, plots earlier in the list should be unaffected, since they have already been output. `read_fortran_line` is implemented in the next section.

{Argument Handling 2}

```
args = argv(); % Cell array of arguments
independent = read_fortran_line(); % Read independent values
i_in_plot = 1; % Keeps track of which field we're on
dependent_axis = ''; % 'X' or 'Y'
filename = '';
x_label = '';
y_label = '';
legend_array = {}; % Cell array of legend items
for i = 1:(nargin+1)
    if i == nargin+1 || strcmp(args{i}, 'XPLOT') || strcmp(args{i}, 'YPLOT')

        {Prepare New Figure, 2}
```

```
        elseif i == 1

            fdisp(stderr(), 'ERROR: The first argument must be either XPLOT or YPLOT.');
            usage_and_exit();

        elseif i_in_plot == 2

            filename = args{i};

        elseif i_in_plot == 3

            x_label = args{i};

        elseif i_in_plot == 4

            y_label = args{i};

        else

            {Plot Curve, 2}

        endif

        i_in_plot = i_in_plot + 1;
    end
```

Used in section 4

When a new figure is created, the previous one is saved if it exists, and the figure-specific variables are reset.

{Prepare New Figure 2}

```
    if i > 1
        if i_in_plot < 4
            fdisp(stderr(), 'ERROR: Early termination of plot specification.');
        endif
        % Apply saved figure adornments
        xlabel(x_label);
        ylabel(y_label);
        legend(legend_array);
        printf('SAVED %s\n', filename);
        saveas(gcf(), filename); % Output current figure to file
    endif
    if i == nargin+1
        break % End of arguments
    endif
    % Create new invisible plot, set current
    figure('visible', 'off', 'paperposition', [0.25, 2.5, 9.0, 6.0]);
    hold on;
    % Reset for new plot
    i_in_plot = 1;
    dependent_axis = args{i}(1); % First character of the arg ('X' or 'Y')
    filename = '';
    legend_array = {};
```

Plotting a curve is as simple as reading the dependent values and calling `plot`.

{Plot Curve 2}

```
if strcmp(args{i}, 'NOP')
    read_fortran_line(); % Skip
else
    legend_array{length(legend_array) + 1} = args{i}; % Append to legend
    dependent = read_fortran_line(); % Read dependent values
    %color = prism(i_in_plot - 4)((i_in_plot - 4),1:3);
    color = sqrt(rainbow(8)(mod(i_in_plot-5, 8) + 1, 1:3) * 0.9);
    yellowness = color(1) + color(2) - color(3) - 0.5;
    if yellowness > 1
        color /= yellowness;
    endif
    if dependent_axis == 'X'
        plot(dependent, independent, 'color', color);
    else
        plot(independent, dependent, 'color', color);
    endif
endif
```

3. Standard Input Handling

The script takes in $c + 1$ lines of space-separated reals in scientific notation. The first is for the independent variable array, and the subsequent lines are for the dependent variable arrays to plot. **Important:** the first value of each line is ignored.

{Input Handler 3}

```
function values = read_fortran_line
    line = fgetl(stdin()); % Retrieve the next line of standard input
    values = sscanf(line, '%e'); % Find all reals in line
    values = values(2:length(values)); % Remove first value
endfunction
```

Used in section 4

4. File Outline

{**plotter.m** 4}

{Usage, 2}

{Input Handler, 3}

{Argument Handling, 2}

Automated Script

Chris McKinney

Automated Script

1. Code

This is just a little script to generate the plots.

{**launcher.sh** 1}

```
set -x
set -e
cd "$(dirname "$0")/.."
function checked_mm {
    ( bin/mm "$1" "$1" "$1" | tee ".mm.$1.out" ) && script/checker.py mm
    status=$?
    mv .mm.debug ".mm.$1.debug"
    return $status
}
checked_mm 1024 #&& checked_mm 2048
status=$?
echo "Status: $status"
exit $status
```

Contents

.0.assign1.info

This simple document is available at https://tachibanatech.com/litdoc/569/assign1.simple.pdf in PDF format.

The full document is available at https://tachibanatech.com/litdoc/569/assign1.full.pdf in PDF format and https://tachibanatech.com/litdoc/569/assign1.full.d/_book/ in HTML format.

TODO