# Performance evaluation of MATRIX x MATRIX for floating point arithmetic versus integer arithmetic

Zi Xuan Li

4/15/2024

Professor Izidor Gertner & Professor Albi Arapi

CSC 34200/CSC 34300

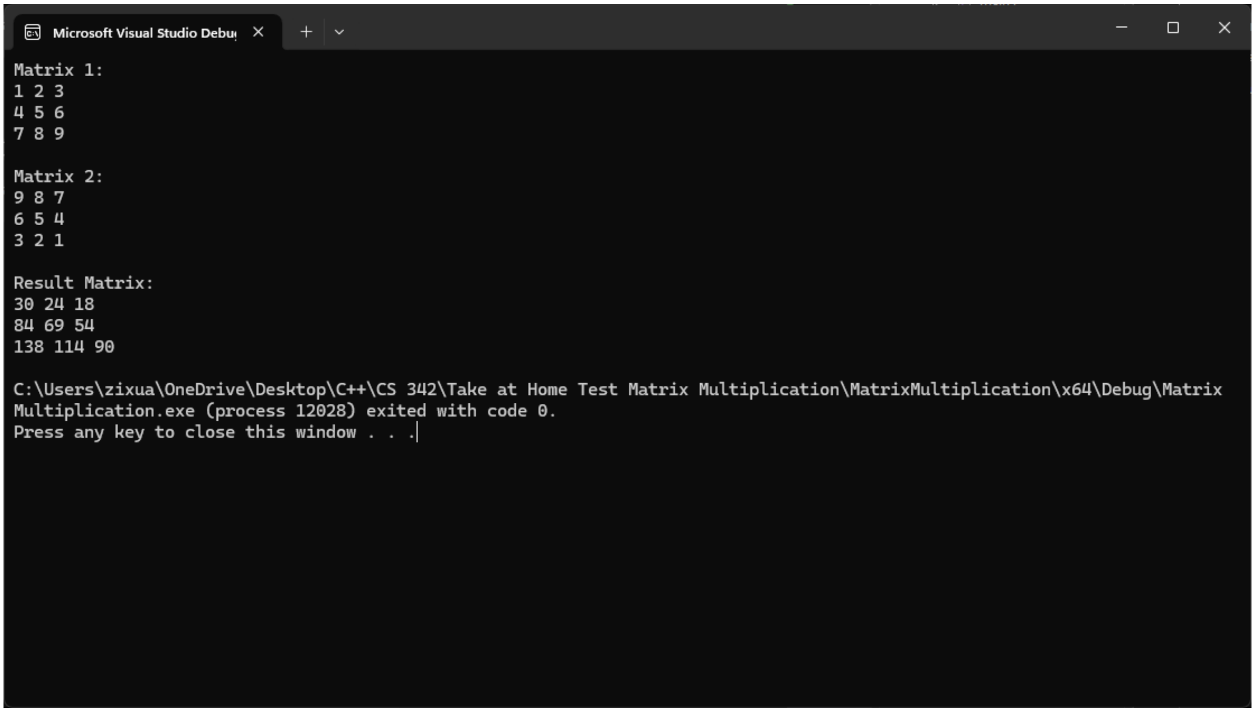
Task #1:

```
#include <stdio.h>

extern int DP(int row[], int column[], int size);

#define N 3 // Size of the matrices (NxN)

// Function to print a matrix
void printMatrix(int matrix[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int mat1[N][N] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int mat2[N][N] = {
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
    };

    int result[N][N];

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int row[N], column[N];

            for (int k = 0; k < N; k++) {
                row[k] = mat1[i][k];
                column[k] = mat2[k][j];
            }

            result[i][j] = DP_asm(row, column, N);
        }
    }

    printf("Matrix 1:\n");
    printMatrix(mat1);

    printf("\nMatrix 2:\n");
    printMatrix(mat2);

    printf("\nResult Matrix:\n");
    printMatrix(result);

    return 0;
}
```

The code found above is the code in the source file main.c which multiplies two square matrixes of size NxN using gcc.

```c
#include "dp.h"

int DP(int row[], int column[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result += row[i] * column[i];
    }
    return result;
}

int DP_asm(int row[], int column[], int size) {
    int result = 0;
    for (int i = 0; i < size; i++) {
        result += row[i] * column[i];
    }
    return result;
}

float DP_float(float row[], float column[], int size) {
    float result = 0.0;
    for (int i = 0; i < size; i++) {
        result += row[i] * column[i];
    }
    return result;
}
```

The code found above is the code in the source file dp.c which contains the function DP(row, column) that is used in main.c



```
Matrix 1:
1 2 3
4 5 6
7 8 9

Matrix 2:
9 8 7
6 5 4
3 2 1

Result Matrix:
30 24 18
84 69 54
138 114 90

C:\Users\zixua\OneDrive\Desktop\C++\CS 342\Take at Home Test Matrix Multiplication\MatrixMultiplication\x64\Debug\Matrix
Multiplication.exe (process 12028) exited with code 0.
Press any key to close this window . . .
```

The image above is the result when the executable created from main.c runs.

Task #2:

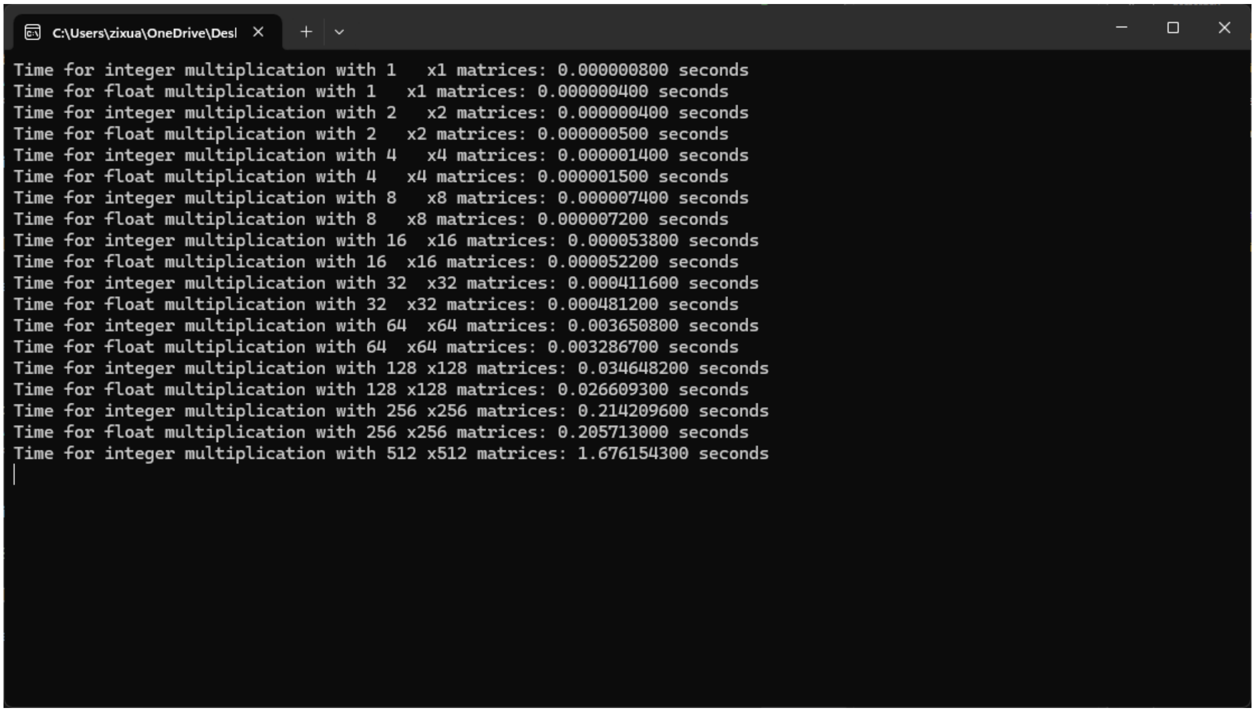
```
#include <chrono>
#include <iomanip>
#include <iostream>
#include <vector>

// Function to multiply two square matrices using integer arithmetic
void matrixMultiplicationInt(const std::vector<std::vector<int>>& mat1,
    const std::vector<std::vector<int>>& mat2,
    std::vector<std::vector<int>>& result) {
    int N = mat1.size();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
                sum += mat1[i][k] * mat2[k][j];
            }
            result[i][j] = sum;
        }
    }
}

// Function to multiply two square matrices using floating-point arithmetic
void matrixMultiplicationFloat(const std::vector<std::vector<double>>& mat1,
    const std::vector<std::vector<double>>& mat2,
    std::vector<std::vector<double>>& result) {
    int N = mat1.size();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += mat1[i][k] * mat2[k][j];
            }
            result[i][j] = sum;
        }
    }
}
int main() {
    std::cout << std::fixed << std::setprecision(9) << std::left;
    for (auto N{ 1 }; N <= 1000; N *= 2) {
        // Integer Arithmetic
        std::vector<std::vector<int>> mat1Int(N, std::vector<int>(N, 1));
        std::vector<std::vector<int>> mat2Int(N, std::vector<int>(N, 2));
        std::vector<std::vector<int>> resultInt(N, std::vector<int>(N, 0));

        const auto startInt = std::chrono::high_resolution_clock::now();
        matrixMultiplicationInt(mat1Int, mat2Int, resultInt);
        const auto endInt = std::chrono::high_resolution_clock::now();

        const std::chrono::duration<double> diffInt = endInt - startInt;

        std::cout << "Time for integer multiplication with " << std::setw(4)
            << N << "x" << N << " matrices: " << diffInt.count() << " seconds\n";
        // Floating-Point Arithmetic
        std::vector<std::vector<double>> mat1Float(N, std::vector<double>(N, 1.0));
        std::vector<std::vector<double>> mat2Float(N, std::vector<double>(N, 2.0));
        std::vector<std::vector<double>> resultFloat(N, std::vector<double>(N,
0.0));

        const auto startFloat = std::chrono::high_resolution_clock::now();
        matrixMultiplicationFloat(mat1Float, mat2Float, resultFloat);
        const auto endFloat = std::chrono::high_resolution_clock::now();

        const std::chrono::duration<double> diffFloat = endFloat - startFloat;

        std::cout << "Time for float multiplication with " << std::setw(4)
            << N << "x" << N << " matrices: " << diffFloat.count() << " seconds\n";
    }

    return 0;
}
```

The code found above is the code in the source file main.cpp that used the chrono library to measure the execution time for different values of NxN matrixes.

Task #3:

```
      .file  "dp.c"
      .text
      .globl DP
      .def   DP;    .scl   2;     .type  32;    .endef
      .seh_proc    DP
DP:
      pushq  %rbp
      .seh_pushreg %rbp
      movq   %rsp, %rbp
      .seh_setframe      %rbp, 0
      subq   $16, %rsp
      .seh_stackalloc    16
      .seh_endprologue
      movq   %rcx, 16(%rbp)
      movq   %rdx, 24(%rbp)
      movl   %r8d, 32(%rbp)
      movl   $0, -4(%rbp)
      movl   $0, -8(%rbp)
      jmp    .L2
.L3:
      movl   -8(%rbp), %eax
      cltq
      leaq   0(,%rax,4), %rdx
      movq   16(%rbp), %rax
      addq   %rdx, %rax
      movl   (%rax), %edx
      movl   -8(%rbp), %eax
      cltq
      leaq   0(,%rax,4), %rcx
      movq   24(%rbp), %rax
      addq   %rcx, %rax
      movl   (%rax), %eax
      imull  %edx, %eax
      addl   %eax, -4(%rbp)
      addl   $1, -8(%rbp)
.L2:
      movl   -8(%rbp), %eax
      cmpl   32(%rbp), %eax
      jl     .L3
      movl   -4(%rbp), %eax
      addq   $16, %rsp
      popq   %rbp
      ret
      .seh_endproc
      .ident "GCC: (Rev2, Built by MSYS2 project) 12.1.0"
```

The above code is the code from the source file dp.s which is compiler generated assembly code from DP.c.



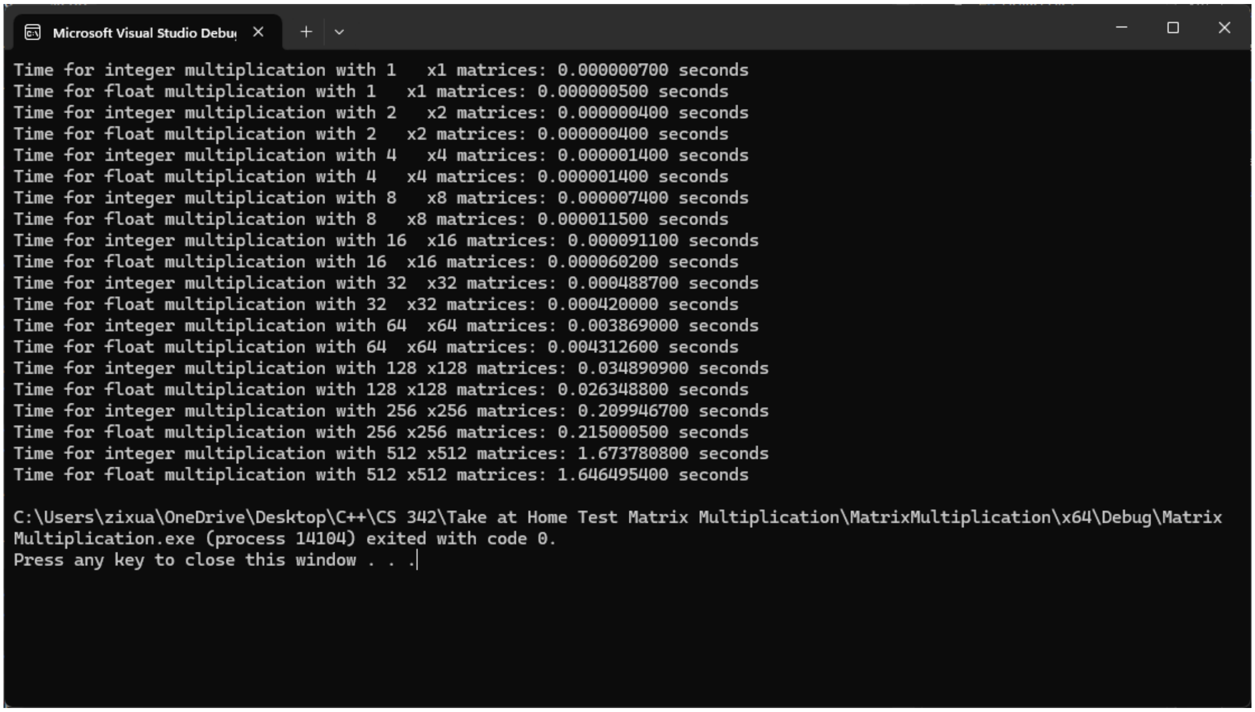
```
Time for integer multiplication with 1   x1 matrices: 0.000000800 seconds
Time for float multiplication with 1   x1 matrices: 0.000000400 seconds
Time for integer multiplication with 2   x2 matrices: 0.000000400 seconds
Time for float multiplication with 2   x2 matrices: 0.000000500 seconds
Time for integer multiplication with 4   x4 matrices: 0.000001400 seconds
Time for float multiplication with 4   x4 matrices: 0.000001500 seconds
Time for integer multiplication with 8   x8 matrices: 0.000007400 seconds
Time for float multiplication with 8   x8 matrices: 0.000007200 seconds
Time for integer multiplication with 16  x16 matrices: 0.000053800 seconds
Time for float multiplication with 16  x16 matrices: 0.000052200 seconds
Time for integer multiplication with 32  x32 matrices: 0.000411600 seconds
Time for float multiplication with 32  x32 matrices: 0.000481200 seconds
Time for integer multiplication with 64  x64 matrices: 0.003650800 seconds
Time for float multiplication with 64  x64 matrices: 0.003286700 seconds
Time for integer multiplication with 128 x128 matrices: 0.034648200 seconds
Time for float multiplication with 128 x128 matrices: 0.026609300 seconds
Time for integer multiplication with 256 x256 matrices: 0.214209600 seconds
Time for float multiplication with 256 x256 matrices: 0.205713000 seconds
Time for integer multiplication with 512 x512 matrices: 1.676154300 seconds
```

The image above is the result of the running the executable when compiling DP in assembly and linking with main.

Task #4:

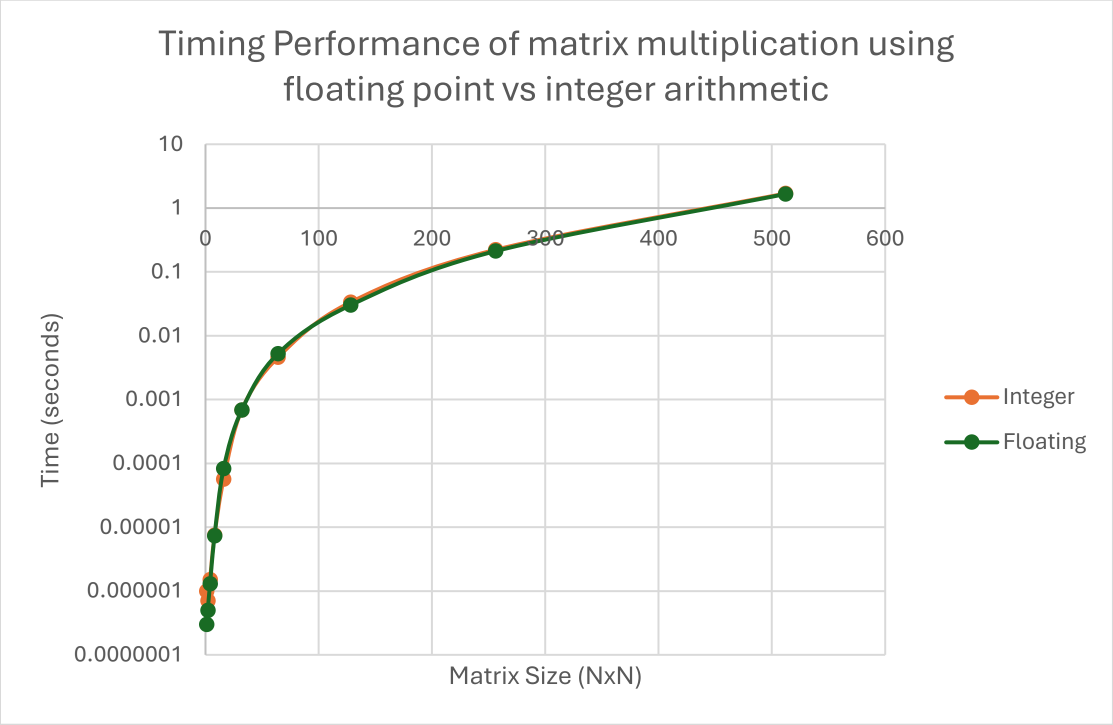
```
.text
.globl DP_asm

DP_asm:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $0, %eax          # Initialize result to 0
    movq    %rdi, %rcx        # Load address of row into rcx
    movq    %rsi, %rdx        # Load address of column into rdx
    movl    %edx, %r8d        # Copy size to r8d

    xorl    %edx, %edx        # Clear edx for multiplication

loop_start:
    movl    (%rcx,%rdx,4), %eax    # Load value from row
    imull   (%rdx,%rsi,4), %eax    # Multiply with value from column
    addl    %eax, %ebp        # Add to result
    incq    %rdx              # Move to next element in row/column
    subq    $1, %r8           # Decrement loop counter
    jnz     loop_start        # Jump to loop_start if counter is not zero

    popq    %rbp
    ret
```

The above code is my attempt and optimizing the assembly code by minimizing the number of instructions.



```
Time for integer multiplication with 1   x1 matrices: 0.000000700 seconds
Time for float multiplication with 1   x1 matrices: 0.000000500 seconds
Time for integer multiplication with 2   x2 matrices: 0.000000400 seconds
Time for float multiplication with 2   x2 matrices: 0.000000400 seconds
Time for integer multiplication with 4   x4 matrices: 0.000001400 seconds
Time for float multiplication with 4   x4 matrices: 0.000001400 seconds
Time for integer multiplication with 8   x8 matrices: 0.000007400 seconds
Time for float multiplication with 8   x8 matrices: 0.000011500 seconds
Time for integer multiplication with 16  x16 matrices: 0.000091100 seconds
Time for float multiplication with 16  x16 matrices: 0.000060200 seconds
Time for integer multiplication with 32  x32 matrices: 0.000488700 seconds
Time for float multiplication with 32  x32 matrices: 0.000420000 seconds
Time for integer multiplication with 64  x64 matrices: 0.003869000 seconds
Time for float multiplication with 64  x64 matrices: 0.004312600 seconds
Time for integer multiplication with 128 x128 matrices: 0.034890900 seconds
Time for float multiplication with 128 x128 matrices: 0.026348800 seconds
Time for integer multiplication with 256 x256 matrices: 0.209946700 seconds
Time for float multiplication with 256 x256 matrices: 0.215000500 seconds
Time for integer multiplication with 512 x512 matrices: 1.673780800 seconds
Time for float multiplication with 512 x512 matrices: 1.646495400 seconds

C:\Users\zixua\OneDrive\Desktop\C++\CS 342\Take at Home Test Matrix Multiplication\MatrixMultiplication\x64\Debug\Matrix
Multiplication.exe (process 14104) exited with code 0.
Press any key to close this window . . .
```

The image above is the timing measurement using dp_asm.s and linking with main to create the executable.


Timing Performance of matrix multiplication using floating point vs integer arithmetic
Time (seconds)
Matrix Size (NxN)
Integer
Floating