Matrix Multiplication on CPU using SSE

CPEG655 – Final Project

Michael Hutts

Problem Introduction

The goal of this project was to perform matrix multiplication on the CPU entirely via SSE's vector operations. Matrix multiplication is a computationally heavy operation where many simple, identical operations are done repeatedly across multiple arrays of data, bringing out some of the greatest weaknesses of sequential CPU logic. In comparison to pure x86, SSE offers a far better way to handle matrix operations on the CPU. SSE (Streaming SIMD Extensions) is an extension to x86 intended to provide SIMD (single instruction, multiple data), capabilities to processors. Using SSE, vectors of 4 values can be operated on in a single instruction.

Solution Overview

Initial Approach

To start the transformation, I was able to use SSE intrinsics in C++ to transform the original triple-nested loop into a double-nested loop.

```
// function to perform matrix multiplication without vectorization
void matrixMultiply(const float* A, const float* B, float* C, int size) {
    // iterate over each row of matrix A
    for (int i = 0; i < size; i++) {
        // iterate over each column of matrix B
        for (int j = 0; j < size; j++) {
            // initialize the sum for the current position in matrix C
            float sum = 0.0;

            // iterate over each element in the row of A and column of B
            for (int k = 0; k < size; k++) {
                // multiply the corresponding elements and accumulate the result in the sum
                sum += A[i * size + k] * B[k * size + j];
            }

            // store the accumulated sum in the current position of matrix C
            C[i * size + j] = sum;
        }
    }
}
```

Original version of matrix multiplication function

```
// function to perform SSE-based matrix multiplication
void matrixMultiplySSE(float* A, float* B, float* C, int size) {
    // iterate over each row of matrix A
    for (int i = 0; i < size; i++) {
        __m128 rowA, vecB;
        __m128 result = _mm_setzero_ps();
        for (int k = 0; k < 4; k++) {
            // load a single element from the current row of A and fill a vector
            rowA = _mm_set1_ps(A[i * size + k]);
            // load a vector from the current column of B
            vecB = _mm_loadu_ps(B + k * size);

            // multiply the rowA vector with the loaded vecB vector element-wise and add to the result
            result = _mm_add_ps(result, _mm_mul_ps(rowA, vecB));
        }

        // store the result vector to the current position in matrix C
        _mm_storeu_ps(C + i * size, result);
    }
}
```

SSE version of matrix multiplication function

Comparing Initial Results

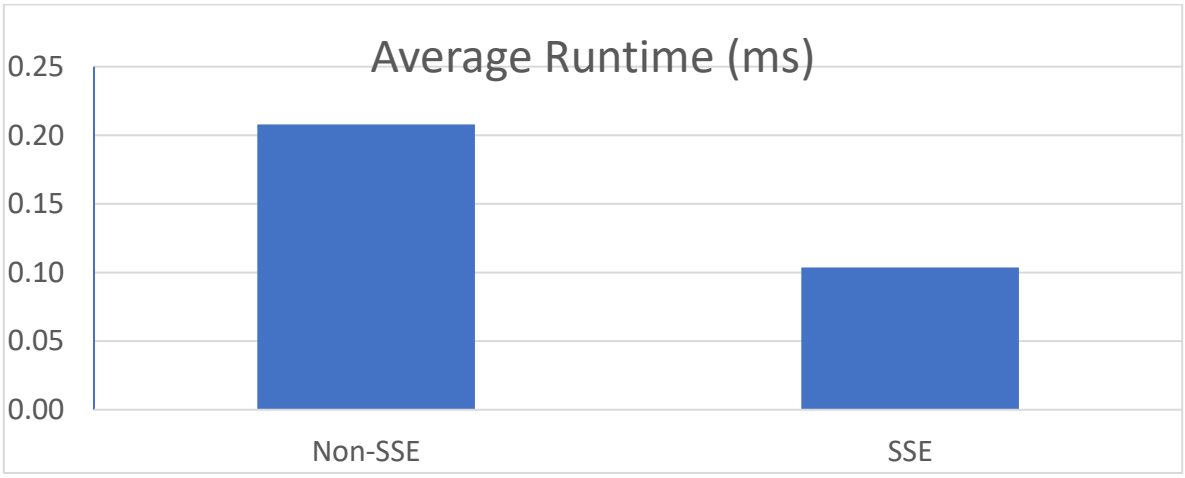
Using RMSE as a benchmark to confirm correctness, two randomized 4x4 matrices were fed to both functions. Resultantly, both functions produced the same matrix C as an output and with an RMSE of 0 between the two C matrices.

```
16:07:00 mike ~/documents/cpeg655/final $ ./matrix_multiplication_sse
Matrix A:
3.52966 3.27929 6.57421 4.09356
4.02743 7.67502 8.70941 5.75692
8.59988 1.32493 8.21583 4.25935
4.43835 7.6059 6.87033 6.13842

Matrix B:
7.63343 4.44275 8.6543 8.87295
5.78655 1.09224 9.39686 7.50227
1.82249 4.08041 3.94084 2.53352
8.27663 7.45234 3.62923 1.80629

SSE-based Matrix Multiplication Result:
91.7815 76.5953 102.126 79.9706
138.676 104.716 162.191 125.779
123.54 104.92 134.712 114.755
141.218 101.805 159.235 124.937

Non-SSE Matrix Multiplication Result:
91.7815 76.5953 102.126 79.9706
138.676 104.716 162.191 125.779
123.54 104.92 134.712 114.755
141.218 101.805 159.235 124.937

Root Mean Square Error (RMSE): 0
```

Running the two functions displays the expected output

Initial Performance

The SSE-based version showed a near-exact 2x improvement in runtime when benchmarking. This was already a satisfying result but there was clearly more on the table to improve.



SSE vs. Non-SSE average runtime over 10,000 runs each

Further Improvement

With a working first version, it was time to dig deeper. Due to the inherent structure of SSE, continuing with the usage of only 4x4 matrices seemed like a logical choice. With that in mind, the other avenue of exploration was performance gains. With much of the data access simplified by removing the innermost loop, there didn't seem to be much to gain from reordering loops in the way you might with a non-SSE function. Switching from IJK order to JIK or JKI would be extremely simple on the non-SSE function, but required dramatic reworking when using SSE due to the dependence on accessing rows and columns in an expected order for the algorithm used. With that in mind, I instead decided that unrolling was the means by which I would experiment with extracting more from SSE. With a 4x4 matrix, unrolling the new inner-most loop was an obvious first choice. When designing the original function, it already seemed like a potential way to write it from the start.

```
// function to perform SSE-based matrix multiplication (unrolled)
void matrixMultiplySSEUnrolled(float* A, float* B, float* C, int size) {
    // iterate over each row of matrix A
    for (int i = 0; i < size; i++) {
        // load column 0 from the current row of A and fill a vector
        __m128 rowA = _mm_set1_ps(A[i * size]);

        // load a vector from the current column of B
        __m128 vecB = _mm_loadu_ps(B);

        // multiply the rowA vector with the loaded vecB vector element-wise
        __m128 result = _mm_mul_ps(rowA, vecB);

        // repeat for column 1
        rowA = _mm_set1_ps(A[i * size + 1]);
        vecB = _mm_loadu_ps(B + (1 * size));
        result = _mm_add_ps(result, _mm_mul_ps(rowA, vecB));
        // repeat for column 2
        rowA = _mm_set1_ps(A[i * size + 2]);
        vecB = _mm_loadu_ps(B + (2 * size));
        result = _mm_add_ps(result, _mm_mul_ps(rowA, vecB));
        // repeat for column 3
        rowA = _mm_set1_ps(A[i * size + 3]);
        vecB = _mm_loadu_ps(B + (3 * size));
        result = _mm_add_ps(result, _mm_mul_ps(rowA, vecB));

        // store the result vector to the current position in matrix C
        _mm_storeu_ps(C + i * size, result);
    }
}
```

Unrolled SSE matrix multiplication function

This change was intuitive to make and only required a few extra lines. The results of this small change were surprisingly noticeable, with more time shaved off the original SSE version.

```
SSE-based Matrix Multiplication Result:
157.99 149.865 203.002 155.16
124.705 128.985 173.016 117.971
96.1138 97.5386 112.141 93.6383
172.261 176.594 207.684 166.255

Average time taken by SSE-based matrix multiplication: 0.108759 microseconds


SSE-based Matrix Multiplication (Unrolled) Result:
157.99 149.865 203.002 155.16
124.705 128.985 173.016 117.971
96.1138 97.5386 112.141 93.6383
172.261 176.594 207.684 166.255

Average time taken by SSE-based matrix multiplication (Unrolled): 0.0837003 microseconds
```

Original vs. Unrolled SSE Functions

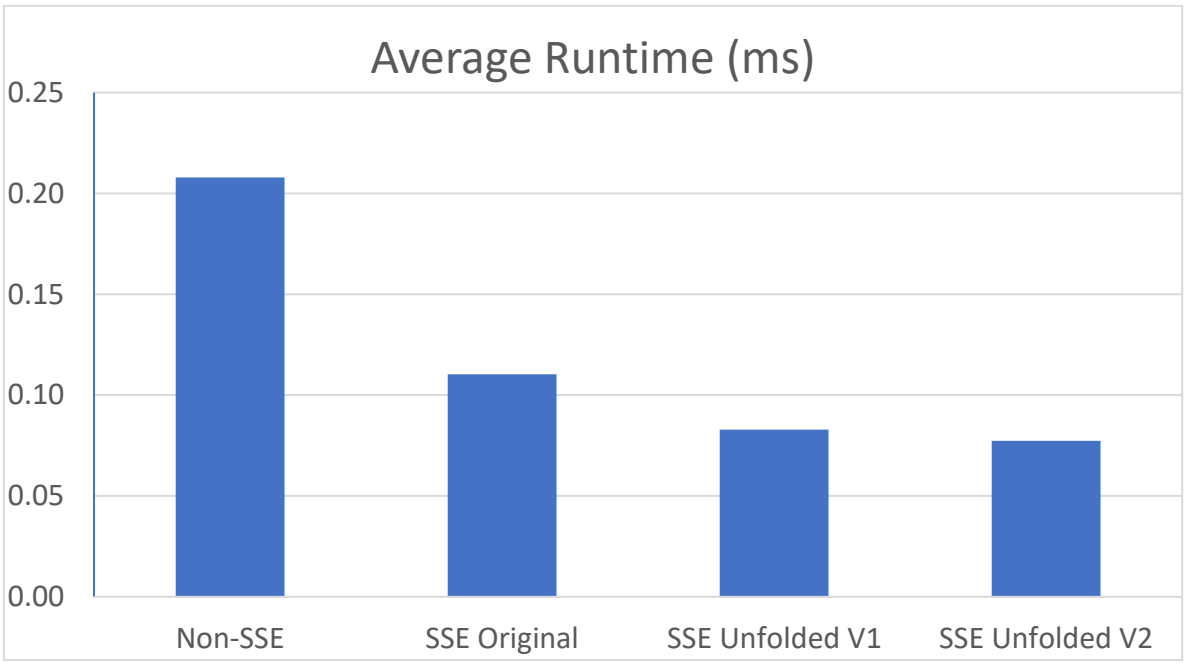
These results made it clear that there was performance to be gained by unrolling these operations. With this in mind, I performed a much more extensive unrolling to remove looping entirely. This required almost 100 lines of code in comparison to less than 40 for the first unrolling, but given the further performance gain this still seemed like a reasonable change in almost any context.

```
SSE-based Matrix Multiplication Result:
163.051 151.996 110.801 171.1
118.778 105.874 78.4746 125.178
111.903 102.203 91.5464 124.238
114.648 113.948 56.2382 122.202

Average time taken by SSE-based matrix multiplication: 0.107363 microseconds


SSE-based Matrix Multiplication (Unrolled) Result:
163.051 151.996 110.801 171.1
118.778 105.874 78.4746 125.178
111.903 102.203 91.5464 124.238
114.648 113.948 56.2382 122.202

Average time taken by SSE-based matrix multiplication (Unrolled): 0.0902908 microseconds


SSE-based Matrix Multiplication (Unrolled Extreme) Result:
163.051 151.996 110.801 171.1
118.778 105.874 78.4746 125.178
111.903 102.203 91.5464 124.238
114.648 113.948 56.2382 122.202

Average time taken by SSE-based matrix multiplication (Unrolled Extreme): 0.0868782 microseconds
```

Complete unrolling gave lesser but still notable gains

Final Results

I discovered that, not only did SSE offer a substantial improvement in performance, but that there was an even greater level of growth possible with further improvement. With the unrolling of each for-loop I found another bump in performance and it seems possible that a reworking of the multiplication algorithm itself could give even greater gains. A 4x4 matrix is the ideal use-case for this approach, but almost any sized matrices should show some kind of performance uplift compared to a standard SISD approach.



All functions' average runtime over 10,000 runs each

Running and Compiling

Environment Details

CPU: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

OS: Windows 10 using WSL v2 (Ubuntu 22.04.2 LTS)

Compiler: g++ (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

Compilation Instructions

To compile with optimal SSE compatibility I used the following command(s):

g++ -march=native -o matrix_multiplication_sse_benchmark matrix_multiplication_sse_benchmark.cpp

g++ -march=native -o matrix_multiplication_sse matrix_multiplication_sse.cpp