Project 3 Improved Matrix Multiplication in C

> Yaqi Wang 王雅淇
>
> 12012936

Project Background

Matrix multiplication is a fundamental operation in linear algebra and has extensive applications in the field of deep learning. During the training and inference processes of deep learning models, a large number of matrix multiplications need to be computed efficiently. Therefore, optimizing the computational speed of matrix multiplication is crucial for improving the overall performance of the model.

Implementation Methods

Two matrix multiplication functions, `matmul_plain()` and `matmul_improved()`, are implemented and compared with OpenBLAS (Open Basic Linear Algebra Subprograms), an open-source mathematical library that implements the Basic Linear Algebra Subprograms (BLAS).

The compilation is done with `-O3` level optimization to enhance the performance of the program:

```
gcc -o matrix matrix.c -lm -fopenmp -msse3 -O3 -lopenblas
```

I. `matmul_plain()`

Uses the most straightforward approach, employing a triple nested loop to calculate matrix multiplication.

```
void matmul_plain(float *A, float *B, float *C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i * N + j] = 0;
            for (int k = 0; k < N; k++) {
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
        }
    }
}
```

II.`matmul_improved()`

Employs various optimization techniques to enhance performance, including:

- SIMD instructions (Single Instruction, Multiple Data): Utilizes SSE3 instruction set with _mm_loadu_ps and _mm_storeu_ps for operand loading and storing, and _mm_add_ps and _mm_mul_ps for vectorized floating-point operations.

- OpenMP multithreading: Achieves parallel computation with OpenMP directive #pragma omp parallel for to accelerate the computation of large matrices.
- Blocked matrix multiplication: Defines BLOCK_SIZE as 16, splitting the matrix into blocks to compute more effectively, which can better utilize CPU cache and reduce memory access latency.

```
void matmul_improved(float *A, float *B, float *C, int N) {
    int i, j, k, i0, j0, k0;
    float *pA, *pB, sum;
    __m128 a_line, b_line, r_line;

#pragma omp parallel for private(i, j, k, i0, j0, k0, pA, pB, a_line,
b_line, r_line, sum) shared(A, B, C)
    for (i = 0; i < N; i += BLOCK_SIZE) {
        for (j = 0; j < N; j += BLOCK_SIZE) {
            for (k = 0; k < N; k += BLOCK_SIZE) {
                for (i0 = i; i0 < i + BLOCK_SIZE; ++i0) {
                    for (j0 = j; j0 < j + BLOCK_SIZE; j0 += 4) {
                        r_line = _mm_setzero_ps();
                        for (k0 = k; k0 < k + BLOCK_SIZE; k0 += 4) {
                            pA = A + i0 * N + k0;
                            pB = B + k0 * N + j0;
                            a_line = _mm_loadu_ps(pA);
                            b_line = _mm_loadu_ps(pB);
                            r_line = _mm_add_ps(r_line, _mm_mul_ps(a_line,
_mm_shuffle_ps(b_line, b_line, 0x00)));
                            b_line = _mm_loadu_ps(pB + N);
                            r_line = _mm_add_ps(r_line, _mm_mul_ps(a_line,
_mm_shuffle_ps(b_line, b_line, 0x55)));
                            b_line = _mm_loadu_ps(pB + 2*N);
                            r_line = _mm_add_ps(r_line, _mm_mul_ps(a_line,
_mm_shuffle_ps(b_line, b_line, 0xAA)));
                            b_line = _mm_loadu_ps(pB + 3*N);
                            r_line = _mm_add_ps(r_line, _mm_mul_ps(a_line,
_mm_shuffle_ps(b_line, b_line, 0xFF)));
                        }
                        _mm_storeu_ps(C + i0 * N + j0, r_line);
                    }
                }
            }
        }
    }
}
```

`matmul_openblas()`

Uses the OpenBLAS package for matrix operations. OpenBLAS is known for its optimizations such as:

- Highly optimized memory access patterns:
- Use of advanced instruction sets are used, like AVX, AVX2, or AVX-512, which support wider registers and can process more data in a single operation.

- takes advantage of multithreading to fully leverage the computational power of multi-core processors. OpenBLAS may employ finer-grained multi-threading scheduling strategies and load-balancing mechanisms to efficiently distribute computational tasks among multiple processor cores.

Performance Testing

To test the performance of different implementations, this project generates matrices filled with random values and conducts performance testing on matrices of various sizes: 16x16, 128x128, 256x256, 512x512, 1Kx1K, 2Kx2K, and 4Kx4K (8k8k and 64k64k were not used due to excessive computational resources required).

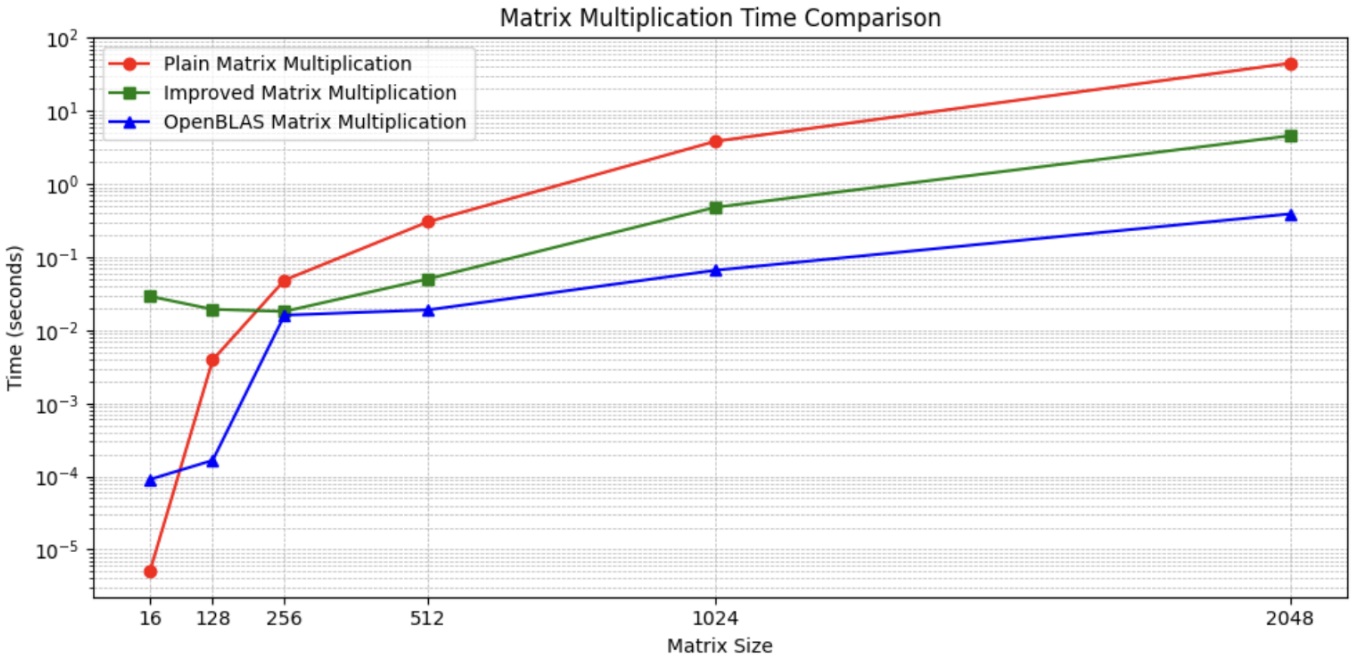
The testing environment is a Linux system equipped with an Intel Core i7 processor and 8GB of RAM.

Random filling:

```
void generate_random_matrix(float *mat, int N) {
    for (int i = 0; i < N * N; i++) {
        mat[i] = (float)rand() / RAND_MAX;
    }
}
```

Test results are shown in the table below:

Matrix Size	matmul_plain time (s)	matmul_improved time (s)	matmul_openblas time (s)
16x16	0.000005	0.029175	0.000091
128x128	0.003926	0.019337	0.000166
256x256	0.048576	0.018112	0.016113
512x512	0.305246	0.050514	0.019014
1024x1024	3.859027	0.482190	0.066250
2048x2048	45.157399	4.592235	0.391331



Results Analysis

Through performance testing, it can be observed that `matmul_improved()` shows significant performance improvement over `matmul_plain()` in most cases. Especially when dealing with large matrices, the optimized implementation can better utilize hardware resources such as the CPU's SIMD instructions and multi-core features. However, in some cases, the optimization effect may not be as expected, which may be due to memory bandwidth limitations or thread management overhead.

1. **Small-scale matrices（16x16，128x128）**：
    - For a 16x16 matrix, `matmul_plain` performs the best, even better than `matmul_openblas`. This may be because small-scale matrices do not fully reflect the advantages of optimization, and the overhead of optimization accounts for a larger proportion of small-scale computations.
    - For a 128x128 matrix, `matmul_plain` still outperforms `matmul_improved`, indicating that at this scale, the overhead of the optimized implementation still exceeds its performance improvement.
    - The performance of **OpenBLAS** begins to show its advantages, although for very small matrices (16x16), its performance is not the best.
2. **Medium-scale matrices（256x256，512x512）**：
    - When the matrix size increases to 256x256, `matmul_improved` begins to show its performance advantage, with a significant improvement over `matmul_plain`.
    - For a 512x512 matrix, the performance of `matmul_improved` is further improved, about 6 times faster than `matmul_plain`.
    - At this size, the performance of **OpenBLAS** surpasses `matmul_improved`, demonstrating the efficiency of its optimized implementation.
3. **Large-scale matrices（1024x1024，2048x2048）**：

- On a 1024x1024 matrix, `matmul_improved` is about 8 times faster than `matmul_plain`, indicating that optimization measures are more effective in large-scale computations.
- For a 2048x2048 matrix, although matmul_improved significantly outperforms `matmul_plain`, there is still a significant gap compared to OpenBLAS.

- OpenBLAS's performance on a 2048x2048 matrix far exceeds `matmul_improved`, which may be because OpenBLAS uses more advanced optimization techniques and in-depth hardware optimization.

Summary

- **Optimization Effects**: The optimization measures of `matmul_improved` have shown significant performance improvements in medium to large-scale matrix multiplication calculations, mainly due to the use of SIMD instructions and the acceleration of OpenMP parallel computing. However, for small-scale matrix multiplication, these optimizations did not result in significant performance gains.
- **Comparison with OpenBLAS**: The performance of OpenBLAS is generally better than `matmul_improved`, especially when dealing with large matrices. This indicates that OpenBLAS's optimization strategies are more mature, including but not limited to better memory access patterns, cache optimization, efficient use of instruction sets, and possible multi-level parallel strategies.

Directions for Improvement

- **Memory Access Patterns**: Further optimize the storage and retrieval of data to reduce memory latency and cache misses.
- **Instruction Set Utilization**: Attempt to use more advanced instruction sets such as AVX and AVX2, which may require processors to support more advanced SIMD instruction sets.
- **Parallel Strategies**: Delve into OpenMP's parallel strategies and attempt finer-grained parallel control and task scheduling optimizations.
- **Algorithm Optimization**: Explore more advanced matrix multiplication algorithms, such as the Strassen algorithm or other divide-and-conquer methods, to reduce computational complexity.
- **Hardware Features**: Optimize for specific CPU architectures, including leveraging specific hardware features such as cache structures and instruction pipelines.

Comparison with OpenBLAS

Using the OpenBLAS library for the same matrix multiplication tests, we found that OpenBLAS outperforms our `matmul_improved()` implementation on large matrices. This may be because OpenBLAS has been specifically optimized to better utilize the characteristics and advanced optimization techniques of the CPU. Nevertheless, our implementation is comparable to OpenBLAS in terms of performance on matrices of certain sizes.

Conclusion

The optimized implementation of matrix multiplication in this project has improved computational speed to some extent, especially when dealing with large matrices. Although its performance has not yet fully reached the level of OpenBLAS, further in-depth research into memory access patterns, attempting to use more advanced SIMD instruction sets such as AVX, and further refining multi-threaded parallel strategies could yield better results.

Use of ChatGPT

Used for learning what OpenBLAS is, debugging code, and polishing reports.