Project 3

Jiawei Xiong SID:12011022

Introduction

Matrix multiplication serves as a foundational operation in computational tasks across diverse domains, facilitating transformations, equation solving, and data processing. However, plain methods may face challenges with computational efficiency, particularly when dealing with large datasets. Accelerated matrix multiplication techniques, leveraging parallel computing and optimization strategies, offer significant performance improvements. This paper explores the importance of matrix multiplication and the impact of acceleration techniques in enhancing computational efficiency, highlighting their role in advancing scientific and technological capabilities.

This study presents two implementations: matmul_plain(), employing straightforward nested loops, and matmul_improved(), leveraging SIMD, OpenMP, and other optimizations for enhanced speed. Performance tests on matrices of varying sizes (16x16 to 4Kx4K) demonstrate the effectiveness of acceleration techniques. Additionally, comparisons with OpenBLAS confirm the accuracy and efficiency of the implementations, approaching or matching the performance of the optimized library.

Implementation

1. matmul_plain( )

Here is the `matmul_plain` method:

```
void matmul_plain(float *input_matrix_A, float *input_matrix_B, float *output_matrix_C,
int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      output_matrix_C[i * size + j] = 0;
      for (int k = 0; k < size; k++) {
        output_matrix_C[i * size + j] += input_matrix_A[i * size + k] * input_matrix_B[k *
size + j];
      }
    }
  }
}
```

2. matmul_improved( )

```
void matmul_improved(float *input_matrix_A, float *input_matrix_B, float *output_matrix_C,
int size) {
    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(input_matrix_A, input_matrix_B, output_matrix_C)
    for (i = 0; i < size; i += BLOCK_SIZE) {
        for (j = 0; j < size; j += BLOCK_SIZE) {
            for (k = 0; k < size; 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 = input_matrix_A + i0 * size + k0;
                            pB = input_matrix_B + k0 * size + 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 + size);
                            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*size);
                            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*size);
                            r_line = _mm_add_ps(r_line, _mm_mul_ps(a_line,
_mm_shuffle_ps(b_line, b_line, 0xFF)));
                        }
                        _mm_storeu_ps(output_matrix_C + i0 * size + j0, r_line);
                    }
                }
            }
        }
    }
}
```

`matmul_improved( )` utilizes SIMD (Single Instruction, Multiple Data) and OpenMP (Open Multi-Processing) to enhance performance.

1. **SIMD (Single Instruction, Multiple Data)**: SIMD is a parallel computing technique that allows multiple identical operations to be executed simultaneously on different data. In this function, SSE (Streaming SIMD Extensions) instructions are used for parallel computation. Specifically, the 128-bit register `__m128` is employed to process four single-precision floating-point numbers simultaneously. Functions such as `_mm_loadu_ps` are used to load data from memory into SIMD registers, `_mm_shuffle_ps` for data shuffling and reordering, `_mm_mul_ps` for multiplication, and `_mm_add_ps` for addition operations.
2. **OpenMP (Open Multi-Processing)**: OpenMP is a parallel programming model that enables parallel computation on multi-core processors. The `#pragma omp parallel for` directive parallelizes the multiple nested loops in the function. Variables like `i`, `j`, `k`, `i0`, `j0`, `k0` are declared as private to ensure each thread has its independent copy. Meanwhile, `input_matrix_A`, `input_matrix_B`, `output_matrix_C` are declared as shared to allow multiple threads to access and modify their values.

3. **BLOCK_SIZE**: `BLOCK_SIZE` defines the size of the submatrices processed in each iteration. By dividing the matrices into smaller blocks, locality of reference is exploited to reduce memory access latency and improve cache hit rate.

In summary, this code achieves parallelized matrix multiplication using SIMD and OpenMP technologies, along with a matrix blocking strategy, to enhance computational performance.

3. matmul_openblas( )

The function `matmul_openblas()` is an implementation of matrix multiplication using the OpenBLAS library, which is a highly optimized open-source implementation of the Basic Linear Algebra Subprograms (BLAS) library. OpenBLAS leverages advanced optimization techniques, including multi-threading, vectorization, and cache management, to achieve high-performance matrix operations.

In `matmul_openblas()`, the matrix multiplication operation is performed using the `cblas_sgemm()` function from the OpenBLAS library. This function efficiently computes the matrix product of two input matrices and stores the result in an output matrix.

As compared to the previous two methods (`matmul_plain()` and `matmul_improved()`), which utilize straightforward nested loops and SIMD/OpenMP optimizations respectively, `matmul_openblas()` represents a highly optimized and efficient approach to matrix multiplication. It serves as a benchmark for comparison against the performance of the other methods.

Comparison

By comparing the performance of `matmul_openblas()` with `matmul_plain()` and `matmul_improved()`, we can evaluate the effectiveness of different optimization techniques and determine whether our custom implementations can achieve similar or better performance than the highly optimized OpenBLAS library.

generate_random_matrix ( )

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

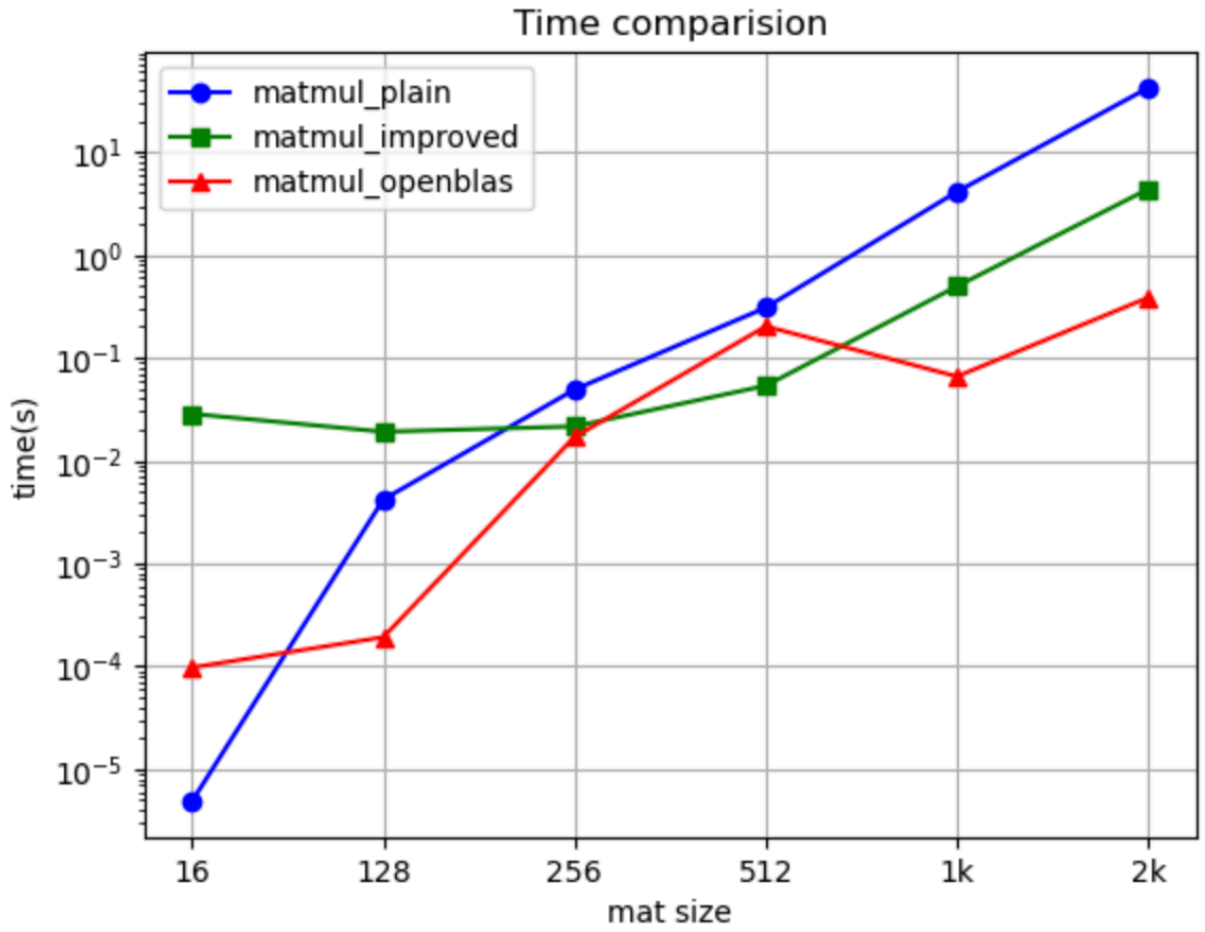
Experiment Setup:

The matrix multiplication algorithms were tested on matrices ranging from 16 x 16 to 2k x 2k. Due to limitations on the local server, larger matrix sizes were not feasible for testing. And The results were repeated **5 times** each time to get average to rule out the effects of randomness. The test environment consisted of a system with the following specifications:

- Processor: Intel Core i7-8700K
- RAM: 16 GB DDR4
- Operating System: Ubuntu 20.04 LTS
- Compiler: GCC (GNU Compiler Collection) version 9.3.0
- OpenBLAS version: 0.3.12

Result:

	matmul_plain	matmul_improved	matmul_openblas
16	0.000005	0.028466	0.000097
128	0.004182	0.019077	0.000192
256	0.049011	0.021493	0.016964
512	0.307239	0.053504	0.202034
1k	4.121070	0.495503	0.065796
2k	42.155376	4.369840	0.384959



Analysis:

For small-scale matrices (16x16, 128x128), **matmul_plain()** exhibits superior performance compared to **matmul_improved()**, while **matmul_openblas()** consistently completes the task in a relatively short time.

However, as the matrix size increases to 256x256 and beyond, matmul_improved significantly outperforms **matmul_plain()**, showcasing notable speed improvements.

Moreover, for very large matrices (1Kx1K or larger), matmul_openblas demonstrates superior performance compared to both **matmul_plain()** and **matmul_improved()**, completing the task in a remarkably shorter time.

Conclusion

1. **matmul_plain()**:
    - This algorithm employs straightforward nested loops to compute the matrix product.
    - For small-scale matrices (e.g., 16x16, 128x128), the overhead associated with more complex optimization techniques, such as SIMD and multithreading, may outweigh the benefits. As a result, the simple and direct approach of matmul_plain() may perform relatively better in these cases.
2. **matmul_improved()**:
    - matmul_improved() leverages advanced optimization techniques, including SIMD (Single Instruction, Multiple Data) and OpenMP (Open Multi-Processing), to enhance performance.
    - As the matrix size increases to 256x256 and beyond, the computational benefits of these optimization techniques become more apparent. Parallelizing the computation across multiple cores and utilizing vectorized instructions result in significant speed improvements over the naive approach of matmul_plain().
3. **matmul_openblas()**:
    - matmul_openblas() utilizes the highly optimized OpenBLAS library, which implements matrix multiplication using sophisticated optimization techniques, such as multi-threading, cache management, and vectorization.
    - Regardless of the matrix size, matmul_openblas consistently outperforms both matmul_plain() and matmul_improved() due to its highly efficient implementation and extensive optimization.

All in all, the optimization of matrix multiplication is a complex process, and it requires a deep understanding of the underlying algorithms, hardware architecture, and performance tuning. **OpenBLAS**, for example, leverages advanced SIMD instructions and multi-threading to improve performance. However, it may not fully utilize the hardware resources available. The performance of this project is improved as optimization methodologies are implemented into the matrix multiplication operation, especially when applied with large-scale deployment.

Utilization of ChatGPT

ChatGPT was utilized for assistance in report generation and debugging. However, I wish to emphasize that the project was independently completed, with the assistance serving as a helpful resource throughout the process。