Lab3

学号： N/A

姓名： N/A

专业： 计算机科学与技术

Overview

The GEMM algorithm, which stands for General Matrix Multiplication, is a fundamental algorithm in linear algebra. It is used to multiply two matrices and produce a third matrix. The GEMM algorithm is widely used in many fields, such as machine learning, computer vision, and scientific computing. The performance of the GEMM algorithm is critical for the performance of the whole system.

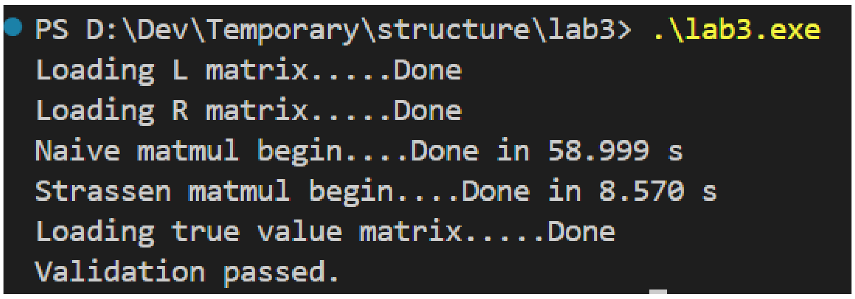
There has been a lot of research on optimizing the GEMM algorithm. Its time complexity has been reduced from $O(n^3)$ to $O(n^{\log_2 7})$ by the Strassen algorithm, and is being further optimized until now. There are also general methods to optimize the GEMM algorithm, such as parallel computing, cache optimization, and instruction optimization.

In this lab, we will implement the GEMM algorithm and optimize it using various techniques except for parallel computing. We will focus on cache optimization and instruction optimization. We will analyze the performance of the optimized GEMM algorithm and compare it with the naive GEMM Implementation.

Implementation

Algorithm

In practice, only the naive algorithm and the Strassen algorithm are used to implement the GEMM algorithm, because other algorithms are either too complex to implement or not efficient in constant factors. In this lab, we tested the naive algorithm and the Strassen algorithm. The result is shown in the figure:



```
PS D:\Dev\Temporary\structure\lab3> .\lab3.exe
Loading L matrix.....Done
Loading R matrix.....Done
Naive matmul begin....Done in 58.999 s
Strassen matmul begin....Done in 8.570 s
Loading true value matrix.....Done
Validation passed.
```

Figure 1: Test of the Strassen Algorithm

The Strassen algorithm is faster than the naive algorithm when the matrix size is large enough. However, the Strassen algorithm has three major drawbacks:

- The Strassen algorithm needs frequent memory allocation and deallocation, which is time-consuming. Besides, for large matrices, the memory allocation may fail.
- The Strassen algorithm can only be used when the matrix size is a power of 2. If not, the Strassen algorithm needs to pad the matrix with zeros, introducing extra overhead in memory and computation.
- The Strassen algorithm is neither cache-friendly nor SIMD-friendly. The Strassen algorithm accesses the memory in a non-sequential way, causing cache misses and eliminating the possibility of using SIMD instructions.

Therefore in this lab, we only focus on optimizing the naive algorithm.

Cache Friendliness

Cache optimization is essential for the performance of the GEMM algorithm. The GEMM algorithm is memory-bound, which means that the performance of the GEMM algorithm is limited by the memory bandwidth. The cache optimization can reduce the number of cache misses and improve the performance of the GEMM algorithm.

Matrix Transposition

The naive GEMM algorithm accesses the L matrix in a row-major way and the R matrix in a column-major way. The row-major access of the L matrix is cache-friendly, but the column-major access of the R matrix is not. We can improve the cache performance by transposing the R matrix and accessing it in a row-major way.

Another benefit of matrix transposition is that it makes both L and R matrices contiguous in memory. The contiguous memory access allows the usage of SIMD instructions, which can further improve the performance of the GEMM algorithm.

However, the matrix transposition itself needs to access the memory in a non-sequential way and causes cache misses. We can improve the cache performance of the matrix transposition by using the tiling technique.

Matrix Tiling

In the previous chapter, we have mentioned that the matrix transposition needs to access the memory in a non-sequential way and causes cache misses. We can improve the cache performance by using the tiling technique.

The tiling technique divides the matrix into small tiles and transposes each tile separately. The size of the tile is chosen to fit into the cache. By transposing the matrix tile by tile, we can reduce the number of cache misses and improve the cache performance.

It is worth noting that the performance of the matrix tiling depends on the size of the tile. If the size of the tile is too small, the overhead of the matrix tiling may outweigh the benefit of cache optimization. If the size of the tile is too large, the matrix tiling cannot fit into the cache and causes cache misses.

The size of the tile is a trade-off between cache optimization and computation overhead. In practice, the size of the tile is chosen to by experiments.

Instruction Parallelism

The instruction-level parallelism is essential for the performance of the GEMM algorithm. Though the GEMM algorithm is mainly memory-bound, the number of arithmetic operations is still large. The instruction-level parallelism can improve the performance of the GEMM algorithm by performing multiple operations in parallel.

SIMD Instruction

Modern CPUs have SIMD instructions, which can perform multiple operations in parallel. The SIMD instructions can improve the performance of the GEMM algorithm by performing vectorized operations.

A element of the result matrix is calculated by the dot product of a row of the L matrix and a row of the transposed R matrix. The dot product can be calculated by the SIMD instructions. In this lab, we use the AVX2 instruction set, the `__m256` data type, and the `_mm256` intrinsic functions to perform the SIMD operations.

Loop Unrolling

In the tiled transpose algorithm and vector dot product algorithm, we can further improve the performance by loop unrolling. Loop unrolling reduces the overhead of the loop control and allows the CPU to use more registers for the computation.

However, loop unrolling may need compiler support to effectively utilize the registers, by renaming the registers and scheduling the instructions.

Language and Compiler

We implemented the GEMM algorithm in C and compiled it with the GCC compiler. We can use language features like `restrict` to tell the compiler that the pointers do not alias, and use `inline` or `__attribute__((always_inline))` to inline the functions.

The `__builtin_unreachable` intrinsic function is used to tell the compiler that the code should never be reached. This can help the compiler to optimize the code by removing the unnecessary branches. For example, by setting `if(size % 8 != 0) __builtin_unreachable();`, the compiler can optimize the code by removing the unnecessary corner cases.

Besides, we can use the `-O3` or `-Ofast` optimization level to enable the most aggressive optimizations. The results are shown both with and without the `-Ofast` optimization level.

Code

Function `vec_dot`

The function `vec_dot` calculates the dot product of two vectors. The two vectors are a row of the L and a row of the transposed R matrix.

The function uses the AVX2 instruction set and the `_mm256` intrinsic functions to perform the SIMD operations. The function is inlined to reduce the overhead of the function call, though the compiler may not inline the function if the function is too large.

Loop unrolling is used to reduce the overhead of the loop control and allow the CPU to use more registers for the computation. The loop unrolling factor is chosen to be 8, as our CPU has 8 AVX2 registers (ymm0-ymm7).

```
inline static int vec_dot(const int *restrict a, const int *restrict b, int
size)
{
  // Loop unroll 8 times
  // Initialize the sums to zero
  __m256i sum_0 = _mm256_setzero_si256();
  __m256i sum_1 = _mm256_setzero_si256();
  __m256i sum_2 = _mm256_setzero_si256();
  __m256i sum_3 = _mm256_setzero_si256();
  __m256i sum_4 = _mm256_setzero_si256();
  __m256i sum_5 = _mm256_setzero_si256();
  __m256i sum_6 = _mm256_setzero_si256();
  __m256i sum_7 = _mm256_setzero_si256();

  // The remaining size after vectorized computation
  int remaining_size = size;

  while (remaining_size >= 64)
  {
    // Load 8 256-bit vectors from memory
    // Each vector contains 8 32-bit integers
    __m256i va_0, va_1, va_2, va_3, va_4, va_5, va_6, va_7;
    __m256i vb_0, vb_1, vb_2, vb_3, vb_4, vb_5, vb_6, vb_7;
    va_0 = _mm256_loadu_si256((__m256i *)a);
    va_1 = _mm256_loadu_si256((__m256i *)(a + 8));
    va_2 = _mm256_loadu_si256((__m256i *)(a + 16));
    va_3 = _mm256_loadu_si256((__m256i *)(a + 24));
    va_4 = _mm256_loadu_si256((__m256i *)(a + 32));
    va_5 = _mm256_loadu_si256((__m256i *)(a + 40));
    va_6 = _mm256_loadu_si256((__m256i *)(a + 48));
    va_7 = _mm256_loadu_si256((__m256i *)(a + 56));
    vb_0 = _mm256_loadu_si256((__m256i *)b);
    vb_1 = _mm256_loadu_si256((__m256i *)(b + 8));
    vb_2 = _mm256_loadu_si256((__m256i *)(b + 16));
    vb_3 = _mm256_loadu_si256((__m256i *)(b + 24));
    vb_4 = _mm256_loadu_si256((__m256i *)(b + 32));
    vb_5 = _mm256_loadu_si256((__m256i *)(b + 40));
    vb_6 = _mm256_loadu_si256((__m256i *)(b + 48));
    vb_7 = _mm256_loadu_si256((__m256i *)(b + 56));
    // Multiply and accumulate
    sum_0 = _mm256_add_epi32(sum_0, _mm256_mullo_epi32(va_0,vb_0));
    sum_1 = _mm256_add_epi32(sum_1, _mm256_mullo_epi32(va_1,vb_1));
    sum_2 = _mm256_add_epi32(sum_2, _mm256_mullo_epi32(va_2,vb_2));
    sum_3 = _mm256_add_epi32(sum_3, _mm256_mullo_epi32(va_3,vb_3));
    sum_4 = _mm256_add_epi32(sum_4, _mm256_mullo_epi32(va_4,vb_4));
    sum_5 = _mm256_add_epi32(sum_5, _mm256_mullo_epi32(va_5,vb_5));
    sum_6 = _mm256_add_epi32(sum_6, _mm256_mullo_epi32(va_6,vb_6));
    sum_7 = _mm256_add_epi32(sum_7, _mm256_mullo_epi32(va_7,vb_7));
    // Move to the next 64 elements
    a              += 64;
    b              += 64;
    remaining_size -= 64;
  }

  // Extract the sums from the vectors
  int result = 0;
  for (int i = 0; i < 8; ++i)
  {
    result += ((int *)&sum_0)[i];
    result += ((int *)&sum_1)[i];
    result += ((int *)&sum_2)[i];
    result += ((int *)&sum_3)[i];
    result += ((int *)&sum_4)[i];
    result += ((int *)&sum_5)[i];
    result += ((int *)&sum_6)[i];
    result += ((int *)&sum_7)[i];
  }

  // Calculate the remaining elements
  for (int i = 0; i < remaining_size; ++i)
    result += a[i] * b[i];

  return result;
}
```

Function `tiled_transpose`

The function `tiled_transpose` transposes the matrix by tiles. The size of the tile is chosen by experiments. The experiment results show that the algorithm has the best performance when the size of the tile is 32 or 64.

The function also untilizes the loop unrolling technique to reduce the overhead of the loop control and allow the CPU to use more registers for the computation. A drawback is that the code is less readable and maintainable.

```
inline static int *tiled_transpose(const int *const restrict matrix)
{
  // The returned matrix is allocated on the heap
  int *new_matrix = (int *)malloc(R_ROW * R_COL * sizeof(int));

  int i, j;
  for (i = 0; i <= R_ROW - TILE_SIZE; i += TILE_SIZE)
  {
    for (j = 0; j <= R_COL - TILE_SIZE; j += TILE_SIZE)
    {
      // General case: transpose a tile of size TILE_SIZE x TILE_SIZE
      const int *tile     = matrix + i * R_COL + j;
      int       *new_tile = new_matrix + j * R_ROW + i;

      // Loop unroll 4 x 4 times
      for (int k = 0; k < TILE_SIZE; k += 4)
        for (int l = 0; l < TILE_SIZE; l += 4)
        {
          new_tile[l * R_ROW + k]           = tile[k * R_COL + l];
          new_tile[(l + 1) * R_ROW + k]     = tile[k * R_COL + l+ 1];
          new_tile[(l + 2) * R_ROW + k]     = tile[k * R_COL + l+ 2];
          new_tile[(l + 3) * R_ROW + k]     = tile[k * R_COL + l+ 3];
          new_tile[l * R_ROW + k + 1]       = tile[(k + 1) *R_COL + l];
          new_tile[(l + 1) * R_ROW + k + 1] = tile[(k + 1) *R_COL + l + 1];
          new_tile[(l + 2) * R_ROW + k + 1] = tile[(k + 1) *R_COL + l + 2];
          new_tile[(l + 3) * R_ROW + k + 1] = tile[(k + 1) *R_COL + l + 3];
          new_tile[l * R_ROW + k + 2]       = tile[(k + 2) *R_COL + l];
          new_tile[(l + 1) * R_ROW + k + 2] = tile[(k + 2) *R_COL + l + 1];
          new_tile[(l + 2) * R_ROW + k + 2] = tile[(k + 2) *R_COL + l + 2];
          new_tile[(l + 3) * R_ROW + k + 2] = tile[(k + 2) *R_COL + l + 3];
          new_tile[l * R_ROW + k + 3]       = tile[(k + 3) *R_COL + l];
          new_tile[(l + 1) * R_ROW + k + 3] = tile[(k + 3) *R_COL + l + 1];
          new_tile[(l + 2) * R_ROW + k + 3] = tile[(k + 3) *R_COL + l + 2];
          new_tile[(l + 3) * R_ROW + k + 3] = tile[(k + 3) *R_COL + l + 3];
        }
    }

    // Corner case: the column size is not a multiple of TILE_SIZE
    const int *tile     = matrix + i * R_COL + j;
    int       *new_tile = new_matrix + j * R_ROW + i;
    for (int k = 0; k < TILE_SIZE; k += 4)
      for (int l = 0; l < R_COL - j; ++l)
      {
        new_tile[l * R_ROW + k]     = tile[k * R_COL + l];
        new_tile[l * R_ROW + k + 1] = tile[(k + 1) * R_COL + l];
        new_tile[l * R_ROW + k + 2] = tile[(k + 2) * R_COL + l];
        new_tile[l * R_ROW + k + 3] = tile[(k + 3) * R_COL + l];
      }
  }

  // Corner case: the row size is not a multiple of TILE_SIZE
  for (j = 0; j <= R_COL - TILE_SIZE; j += TILE_SIZE)
  {
    const int *tile     = matrix + i * R_COL + j;
    int       *new_tile = new_matrix + j * R_ROW + i;
    for (int k = 0; k < R_ROW - i; ++k)
      for (int l = 0; l < TILE_SIZE; l += 4)
      {
        new_tile[l * R_ROW + k]       = tile[k * R_COL + l];
        new_tile[(l + 1) * R_ROW + k] = tile[k * R_COL + l + 1];
        new_tile[(l + 2) * R_ROW + k] = tile[k * R_COL + l + 2];
```

```c
        new_tile[(l + 3) * R_ROW + k] = tile[k * R_COL + l + 3];
      }
  }

  // Corner case: the row size and the column size are both not
  //              multiples of TILE_SIZE
  const int *tile     = matrix + i * R_COL + j;
  int       *new_tile = new_matrix + j * R_ROW + i;
  for (int k = 0; k < R_ROW - i; ++k)
    for (int l = 0; l < R_COL - j; ++l)
      new_tile[l * R_ROW + k] = tile[k * R_COL + l];

  return new_matrix;
}
```

Function

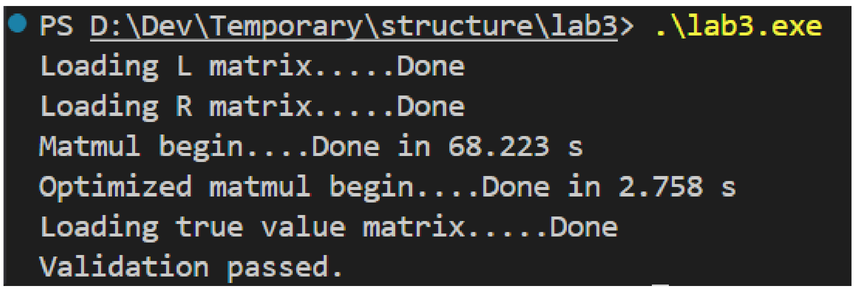
The function `optimized_matmul_impl` is the optimized implementation of the GEMM algorithm. The function first transposes the R matrix by tiles, then calculates each element of the result matrix by the dot product of a row of the L matrix and a row of the transposed R matrix.

```c
void optimized_matmul_impl(const int *const restrict L, const int *const
restrict R, int *const restrict result)
{
  int *R_T = tiled_transpose(R);
  for (int i = 0; i < L_ROW; ++i)
    for (int j = 0; j < R_COL; ++j)
      result[i * R_COL + j] = vec_dot(L + i * L_COL, R_T + j * R_ROW, L_COL);
  free(R_T);
}
```

Results

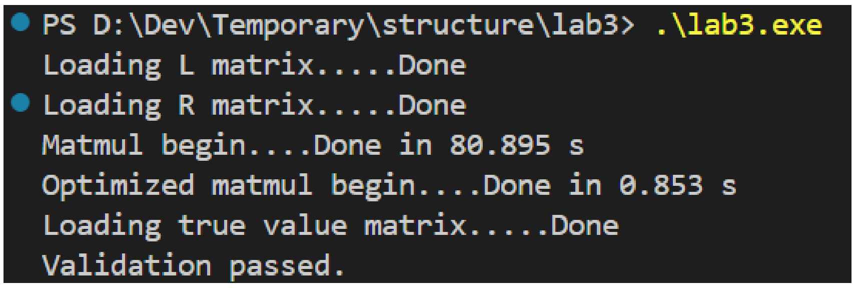
Matrix 2048×2048

The result is shown in the figure:



```
PS D:\Dev\Temporary\structure\lab3> .\lab3.exe
Loading L matrix.....Done
Loading R matrix.....Done
Matmul begin....Done in 68.223 s
Optimized matmul begin....Done in 2.758 s
Loading true value matrix.....Done
Validation passed.
```

Figure 2: Matrix Size 2048×2048



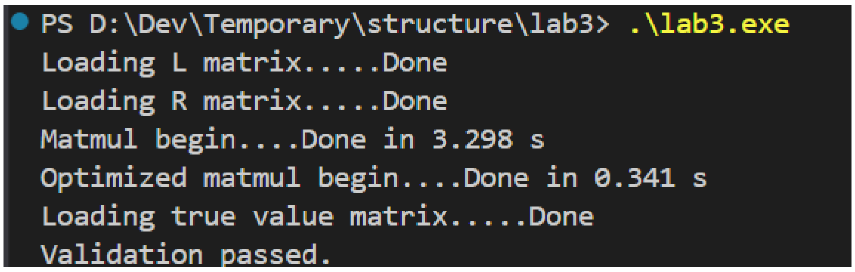
```
PS D:\Dev\Temporary\structure\lab3> .\lab3.exe
Loading L matrix.....Done
Loading R matrix.....Done
Matmul begin....Done in 80.895 s
Optimized matmul begin....Done in 0.853 s
Loading true value matrix.....Done
Validation passed.
```

Figure 3: Matrix Size 2048 × 2048 with `-Ofast`

The implemented GEMM algorithm is about 24.7 times faster than the naive GEMM algorithm. If we enable the `-Ofast` optimization level, the implemented GEMM algorithm is about 94.8 times faster than the naive GEMM algorithm.

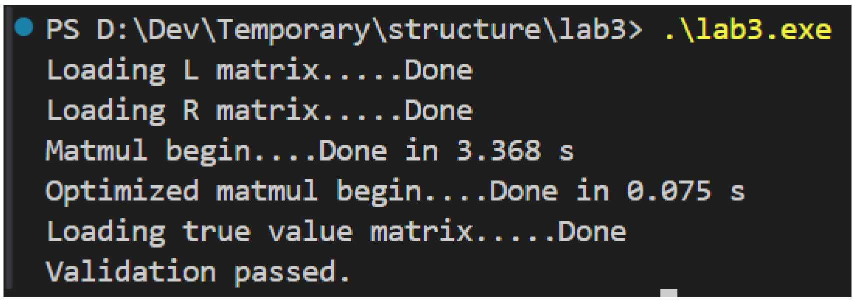
Matrix 1024 × 1024

The result is shown in the figure:



```
PS D:\Dev\Temporary\structure\lab3> .\lab3.exe
Loading L matrix.....Done
Loading R matrix.....Done
Matmul begin....Done in 3.298 s
Optimized matmul begin....Done in 0.341 s
Loading true value matrix.....Done
Validation passed.
```

Figure 4: Matrix Size 1024 × 1024



```
PS D:\Dev\Temporary\structure\lab3> .\lab3.exe
Loading L matrix.....Done
Loading R matrix.....Done
Matmul begin....Done in 3.368 s
Optimized matmul begin....Done in 0.075 s
Loading true value matrix.....Done
Validation passed.
```

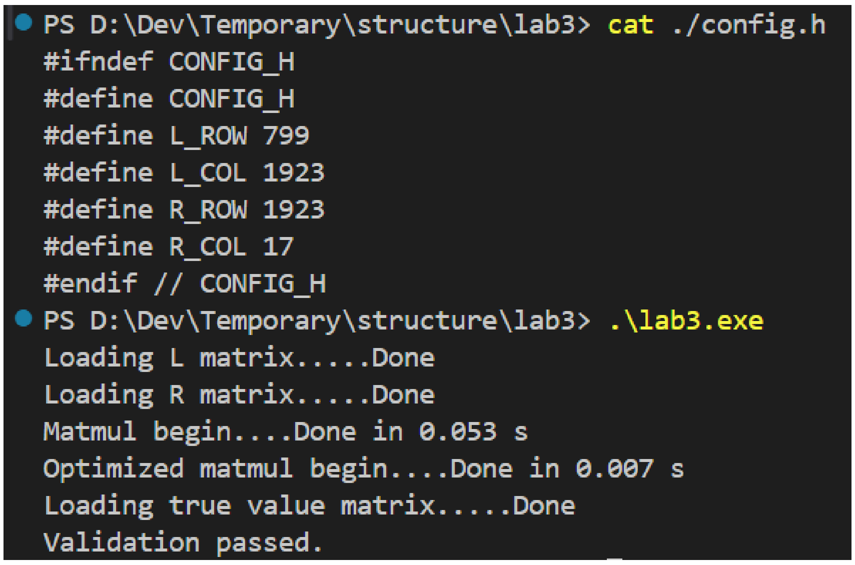
Figure 5: Matrix Size 1024 × 1024 with `-Ofast`

The implemented GEMM algorithm is about 9.67 times faster than the naive GEMM algorithm. If we enable the `-Ofast` optimization level, the implemented GEMM algorithm is about 44.9 times faster than the naive GEMM algorithm.

With the size of the matrix increasing, the performance of the implemented GEMM algorithm is increasingly better than the naive GEMM algorithm.

Matrix with non-power-of-2 size

The result is shown in the figure:



```
PS D:\Dev\Temporary\structure\lab3> cat ./config.h
#ifndef CONFIG_H
#define CONFIG_H
#define L_ROW 799
#define L_COL 1923
#define R_ROW 1923
#define R_COL 17
#endif // CONFIG_H
PS D:\Dev\Temporary\structure\lab3> .\lab3.exe
Loading L matrix.....Done
Loading R matrix.....Done
Matmul begin....Done in 0.053 s
Optimized matmul begin....Done in 0.007 s
Loading true value matrix.....Done
Validation passed.
```

Figure 6: Matrix with non-power-of-2 size

This test is to show the correctness of the implemented GEMM algorithm when the size of the matrix is arbitrary.

Performance Graph

When both matrices are suqare with the same size, the relationship between the performance of the two algorithms and the size of the matrix is shown in the figure:

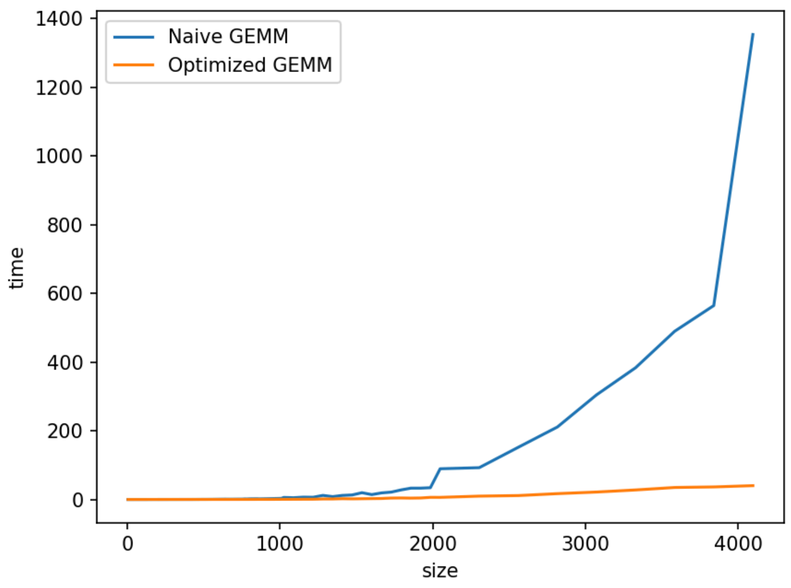


Figure 7: Performance Graph (without `-Ofast`)

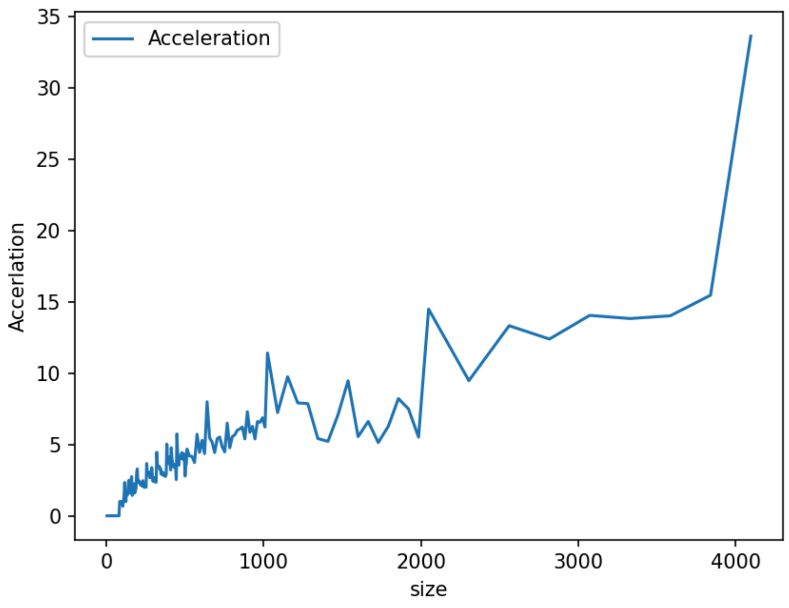


Figure 8: Acceleration Ratio (without `-Ofast`)

Conclusion

In this lab, we implemented the GEMM algorithm and optimized it using various techniques including:

- Algorithm choice: we tested the naive algorithm and the Strassen algorithm, and chose the naive algorithm for optimization.
- Cache optimization: we transposed the R matrix by tiles to improve the cache performance.

- Instruction optimization: we used the AVX2 instruction set and the `_mm256` intrinsic functions to perform the SIMD operations.
- Loop unrolling: we used loop unrolling to reduce the overhead of the loop control and allow the CPU to use more registers for the computation.
- Compiler optimization: we used the `-Ofast` optimization level to enable the most aggressive optimizations.

The results show that the implemented GEMM algorithm is significantly faster than the naive GEMM algorithm. The implemented GEMM algorithm is about 24.7 times faster than the naive GEMM algorithm when the matrix size is 2048×2048. If we enable the `-Ofast` optimization level, the implemented GEMM algorithm is about 94.8 times faster than the naive GEMM algorithm.

The performance of the implemented GEMM algorithm is increasingly better than the naive GEMM algorithm with the size of the matrix increasing. This is because the cache misses and page faults go exponentially with the size of the matrix, and the cache optimization and instruction optimization can significantly reduce the number of memory misses.

In the lab, we learned the importance of cache optimization and instruction optimization for the performance of the GEMM algorithm. We also learned that algorithm is not the only factor that affects the performance, and the cache optimization and instruction optimization can significantly improve the performance of the GEMM algorithm.