# 中山大学计算机院本科生实验报告

**（2024 学年秋季学期）**

课程名称：高性能计算程序设计　　**批改人：**

| 实验 | **通用矩阵乘法** | 专业（方向） | **信息与计算科学** |
|---|---|---|---|
| 学号 | 22336313 | 姓名 | **郑鸿鑫** |
| Email | zhenghx57@mail2.sysu.edu.cn | 完成日期 | 2024/9/16 |

## 1.实验目的

本实验的目的是通过实现和比较 C、Python 和 Java 语言中的通用矩阵乘法算法，深入理解不同编程语言在处理数值计算任务时的性能表现。我们旨在探索编译器优化对 C/C++ 程序性能的具体影响，并学习如何在 Python 和 Java 中实现高效的数值算法。此外，实验还将引导我们了解浮点性能的测量方法，并计算程序的 GFLOPS，从而评估程序的计算效率。

## 2.实验测试平台参数：

| 项目 | 参数 |
|---|---|
| 微体系结构 | Tiger Lake（11th Gen in tel Core i5-11320H） |
| 时钟频率 | 3.20GHz |
| 处理器数目 | 1 |
| 处理器核心数 | 4 |
| 超线程 | 2（每个核心支持两个线程，共 8 线程） |
| 浮点数计算单元 | 8 个双精度浮点运算每个周期<br>16 个单精度浮点运算每个周期 |
| Cache-line 大小 | 64B |
| L1-icache | 80KB 12-way set associative |
| L1-dcache | 48KB 12-way set associative |
| L2-cache | 1.25MB per core |
| L3-cache | 8MB |

单精度浮点峰值性能估计：

$$Peak = (3.20 \times 10^9) \times 4 \times 16 = 204.8GFLOPS$$

双精度浮点峰值性能估计：

$$Peak = (3.20 \times 10^9) \times 4 \times 8 = 102.4GFLOPS$$

## 3. 实验过程和核心代码

- **编写 C 语言代码:**

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

// Build a matrix of size m x n
float** build_Matrix(int m, int n) {
    float** A = (float**)malloc(m * sizeof(float*));
    for (int i = 0; i < m; i++) {
        A[i] = (float*)malloc(n * sizeof(float));
    }
    return A;
}

// Fill the matrix with a fixed value or a random value
void fill_Matrix(int m, int n, float** A, float value) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = value; // Assign a fixed value or generate a random one
        }
    }
}

// Matrix multiplication for float matrices
void matrix_multiply(float** A, float** B, float** C, int m, int n, int k) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i][j] = 0.0f;
            for (int p = 0; p < n; p++) {
                C[i][j] += A[i][p] * B[p][j];
            }
        }
    }
}

// Print the matrix with float values
void print_Matrix(float** matrix, int m, int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%f ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int m = 2048, n = 2048, k = 2048;

    float** A = build_Matrix(m, n);
```

```
    float** B = build_Matrix(n, k);
    float** C = build_Matrix(m, k);

    // Fill matrices with random float values for testing
    printf("Filling Matrix A (2048x2048) with random values\n");
    fill_Matrix(m, n, A, (float)(rand() % 100)); // Fill A with random float values
    printf("Filling Matrix B (2048x2048) with random values\n");
    fill_Matrix(n, k, B, (float)(rand() % 100)); // Fill B with random float values

    clock_t start = clock();
    matrix_multiply(A, B, C, m, n, k);
    clock_t end = clock();

    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    // Uncomment this line if you want to print the matrix (can be very large)
    // print_Matrix(C, m, k);

    printf("Matrix multiplication completed in %f seconds.\n", time_spent);
    printf("Press Enter to continue...");
    _getch();  // 等待用户按下一个键
    printf("\nContinuing...");

    // Free allocated memory
    for (int i = 0; i < m; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);

    return 0;
}
```

- 编写 python 代码实现：

```
import time
import random

def build_matrix(m, n):
    """创建一个 m x n 的矩阵，并初始化为零。"""
    return [[0.0 for _ in range(n)] for _ in range(m)]  # 使用 0.0 来初始化浮点数

def fill_matrix(m, n, matrix):
    """用随机数填充矩阵。"""
    for i in range(m):
        for j in range(n):
            matrix[i][j] = random.uniform(0, 99)  # 使用 random.uniform 生成浮
```

```
点数

def matrix_multiply(A, B):
    """执行矩阵乘法 A * B。"""
    m, n, k = len(A), len(A[0]), len(B[0])
    C = build_matrix(m, k)  # 确保 C 矩阵也使用浮点数初始化
    for i in range(m):
        for j in range(k):
            C[i][j] = 0.0  # 初始化为浮点数 0.0
            for p in range(n):
                C[i][j] += A[i][p] * B[p][j]
    return C

def print_matrix(matrix):
    """打印矩阵。"""
    for row in matrix:
        print(' '.join(map(lambda x: f"{x:.2f}", row)))  # 格式化输出为两位小
数的浮点数

def main():
    m, n, k = 2048, 2048, 2048

    A = build_matrix(m, n)
    B = build_matrix(n, k)
    C = build_matrix(m, k)

    # Fill matrices with random values for testing
    print("Filling Matrix A (2048x2048) with random values")
    fill_matrix(m, n, A)

    print("Filling Matrix B (2048x2048) with random values")
    fill_matrix(n, k, B)

    start_time = time.time()
    C = matrix_multiply(A, B)
    end_time = time.time()

    time_spent = end_time - start_time

    print(f"Matrix multiplication completed in {time_spent:.6f} seconds.")
    input("Press Enter to continue...")

if __name__ == "__main__":
    main()
```

- 编写 Java 代码实现：

```
import java.util.Random;
import java.util.Scanner;

public class Project1_j {
    // Build a matrix of size m x n with float elements
    public static float[][] buildMatrix(int m, int n) {
```

```
        float[][] matrix = new float[m][n];
        return matrix;
    }

    // Fill the matrix with random float values between 0.0 and 99.99
    public static void fillMatrix(int m, int n, float[][] matrix) {
        Random rand = new Random();
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i][j] = rand.nextFloat() * 100; // Random float values
between 0.0 and 99.99
            }
        }
    }

    // Matrix multiplication for float matrices
    public static float[][] matrixMultiply(float[][] A, float[][] B, int m, int
n, int k) {
        float[][] C = buildMatrix(m, k);
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < k; j++) {
                C[i][j] = 0.0f;
                for (int p = 0; p < n; p++) {
                    C[i][j] += A[i][p] * B[p][j];
                }
            }
        }
        return C;
    }

    // Print the matrix with float values
    public static void printMatrix(float[][] matrix, int m, int n) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int m = 2048, n = 2048, k = 2048;

        // Build matrices A, B, and C
        float[][] A = buildMatrix(m, n);
        float[][] B = buildMatrix(n, k);
        float[][] C = buildMatrix(m, k);

        // Fill matrices A and B with random float values
        fillMatrix(m, n, A);
        System.out.println("Filling Matrix A (2048x2048) with random values");
        fillMatrix(n, k, B);
        System.out.println("Filling Matrix B (2048x2048) with random values");
```

```
        // Measure time for matrix multiplication
        long startTime = System.currentTimeMillis();
        C = matrixMultiply(A, B, m, n, k);
        long endTime = System.currentTimeMillis();

        double timeSpent = (endTime - startTime) / 1000.0;
        System.out.println("Matrix multiplication completed in " + timeSpent
+ " seconds.");

        // Wait for user input to continue
        Scanner scanner = new Scanner(System.in);
        System.out.println("Press Enter to continue...");
        scanner.nextLine();  // Wait for user to press Enter
    }
}
```

- C 语言最终版本代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>
#include <omp.h>

// Build a matrix of size m x n
float** build_Matrix(int m, int n) {
    float** A = (float**)malloc(m * sizeof(float*));
    for (int i = 0; i < m; i++) {
        A[i] = (float*)malloc(n * sizeof(float));
    }
    return A;
}

// Fill the matrix with a fixed value or a random value
void fill_Matrix(int m, int n, float** A, float value) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = value; // Assign a fixed value or generate a random one
        }
    }
}

// Block matrix multiplication for float matrices
void block_matrix_multiply(float** A, float** B, float** C, int m, int n, int
k, int block_size) {
    // 初始化结果矩阵 C 为零
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < k; j++) {
            C[i][j] = 0.0f;
        }
    }

    // 分块矩阵乘法
```

```
    #pragma omp parallel for collapse(3)
    for (int ii = 0; ii < m; ii += block_size) {
        for (int jj = 0; jj < k; jj += block_size) {
            for (int pp = 0; pp < n; pp += block_size) {
                // 对每个分块进行计算
                for (int i = ii; i < (ii + block_size < m ? ii + block_size :
m); i++) {
                    for (int p = pp; p < (pp + block_size < n ? pp + block_size :
n); p++) {
                        for (int j = jj; j < (jj + block_size < k ? jj + block_size :
k); j++) {
                            C[i][j] += A[i][p] * B[p][j];
                        }
                    }
                }
            }
        }
    }
}

// Print the matrix with float values
void print_Matrix(float** matrix, int m, int n) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%f ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int m = 2048, n = 2048, k = 2048;
    int block_size = 128; // 分块大小可以调整，譬如 32, 64, 128 等

    float** A = build_Matrix(m, n);
    float** B = build_Matrix(n, k);
    float** C = build_Matrix(m, k);

    // Fill matrices with random float values for testing
    printf("Filling Matrix A (2048x2048) with random values\n");
    fill_Matrix(m, n, A, (float)(rand() % 100)); // Fill A with random float
values
    printf("Filling Matrix B (2048x2048) with random values\n");
    fill_Matrix(n, k, B, (float)(rand() % 100)); // Fill B with random float
values

    clock_t start = clock();
    block_matrix_multiply(A, B, C, m, n, k, block_size);
    clock_t end = clock();

    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
```

```
    // Uncomment this line if you want to print the matrix (can be very large)
    // print_Matrix(C, m, k);

    printf("Block Matrix multiplication completed in %f seconds.\n",
time_spent);
    printf("Press Enter to continue...");
    _getch();  // 等待用户按下一个键
    printf("\nContinuing...");

    // Free allocated memory
    for (int i = 0; i < m; i++) {
        free(A[i]);
        free(B[i]);
        free(C[i]);
    }
    free(A);
    free(B);
    free(C);

    return 0;
}
```

## 4. 实验结果

在本次实验中，3 种语言实现的程序都是随机生成 2048x2048 的矩阵（每个位置都为 0 到 99 的单精度浮点随机数）进行乘法，并打印计算过程中花费的时间，我们在命令行对 3 个程序测试结果如下：

C 语言结果：

```
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 85.810000 seconds.
Press Enter to continue...
```

Python 语言结果：

```
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> py Project1_p.py
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 1061.942493 seconds.
Press Enter to continue...
```

Java 语言结果：

```
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> javac Project1_j.java
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> java -Xint Project1_j.java
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 251.158 seconds.
Press Enter to continue...
```

表 1 不同版本的运行时间及浮点性能

| 版本 | 实现 | 运行时间(s) | 相对加速比(相对前一版本) | 绝对加速比(相对版本1) | 浮点性能(GFLOPS) | 达到峰值性能的百分比 |
|---|---|---|---|---|---|---|
| 1 | Python | 1061.94 | 1 | 1 | 0.01618 | 0.0079% |
| 2 | Java | 251.16 | 1 | 1 | 0.06845 | 0.0334% |
| 3 | C | 85.81 | 1 | 1 | 0.1998 | 0.0976% |
| 4 | +调整循环顺序 | 25.35 | 3.39 | 3.39 | 0.6770 | 0.3306% |
| 5 | +编译优化 | 2.63 | 9.64 | 32.63 | 6.5323 | 3.1896% |
| 6 | +多核并行 | 0.64 | 4.11 | 134.08 | 26.8435 | 13.1072% |
| 7 | +分块矩阵 | 0.45 | 1.42 | 190.69 | 38.1775 | 18.64% |

由于三种语言采取相同实现方式的情况下，C 语言的运行时间最短，所以为了方面实验测试时间，我们后续所有的优化都在 C 语言的代码中进行修改。而各种优化手段对于三种语言的优化效果应该也是大同小异，故以 C 语言为例进行后续实验，结果已经全部呈现在表 1，详细的实验过程如下：

对于通过改变循环顺序来提升性能，表 2 将列出使用 C 语言遍历所有情况的运行时间，而表 1 则只将运行时间最短的一种放在表中计算浮点性能。

**表 2 不同循环顺序的运行时间**

| 循环顺序 | 运行时间 |
|---|---|
| i,j,k | 57.63 |
| i,k,j | 25.11 |
| j,i,k | 33.65 |
| j,k,i | 75.53 |
| k,i,j | 25.35 |

| | |
|---|---|
| k,j,i | 70.88 |

```
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Cyclic Sequence: i,j,k
Matrix multiplication completed in 57.632000 seconds.
Press Enter to continue...
```

```
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Cyclic Sequence: j,i,k
Matrix multiplication completed in 33.647000 seconds.
Press Enter to continue...
```

```
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Cyclic Sequence: k,i,j
Matrix multiplication completed in 25.350000 seconds.
Press Enter to continue...
```

```
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Cyclic Sequence: i,k,j
Matrix multiplication completed in 25.108000 seconds.
Press Enter to continue...
```

```
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Cyclic Sequence: j,k,i
Matrix multiplication completed in 75.528000 seconds.
Press Enter to continue...
```

```
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Cyclic Sequence: k,j,i
Matrix multiplication completed in 70.877000 seconds.
Press Enter to continue...
```

对于通过编译优化来提升性能，表 3 将列出使用 C 语言 5 种不同级别的编译优化的运行时间，而在表 1 中只将运行时间最短的一种放在表中计算浮点性能。

**表 3 不同编译等级的运行时间**

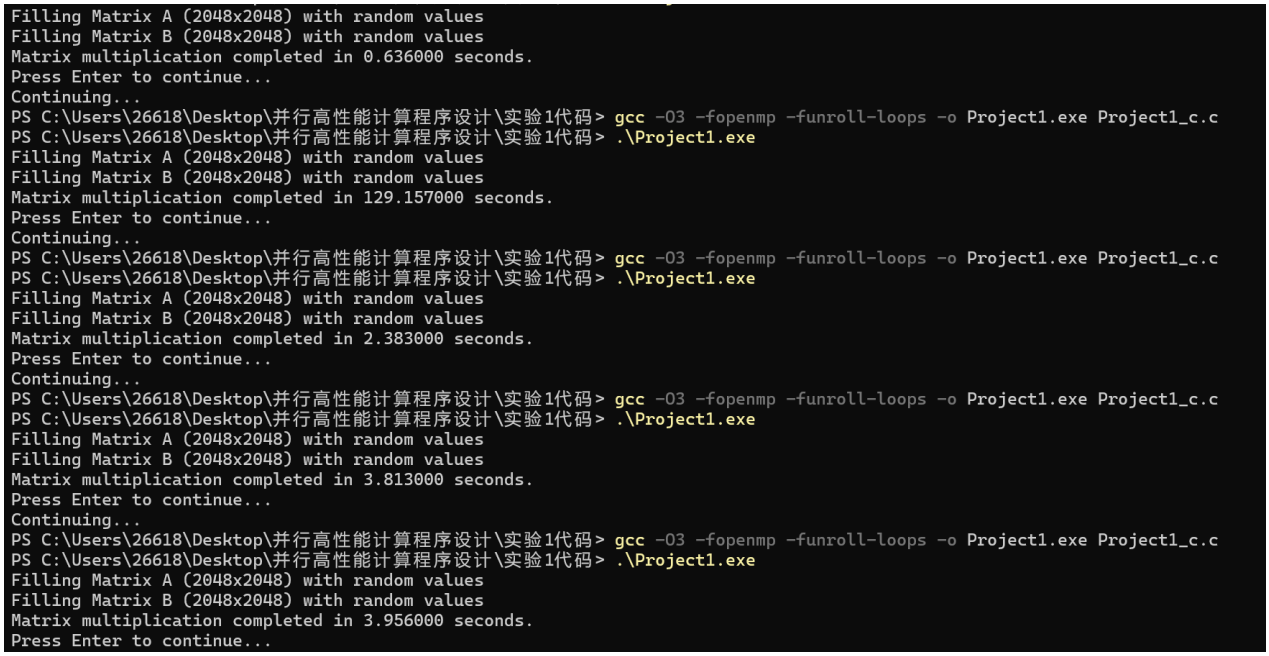
| 编译等级 | 运行时间 |
|---|---|
| O0 | 27.37 |
| O1 | 7.06 |
| O2 | 4.37 |
| O3 | 2.70 |
| O3+funroll-loops | 2.63 |

```
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O0 -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 27.374000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O1 -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 7.058000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O2 -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 4.368000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 2.698000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Matrix multiplication completed in 2.627000 seconds.
Press Enter to continue...
Continuing...
```

对于通过多核并行来提升性能，表 4 将列出使用 C 语言对不同层循环进行并行化的运行时间，而在表 1 中只将运行时间最短的一种放在表中计算浮点性能。

**表 4 不同的循环层进行并行化的运行时间**

| 并行化的循环 | 运行时间 |
|---|---|
| i 所在循环层 | 0.64 |
| j 所在循环层 | 129.16 |
| k 所在循环层 | 2.38 |
| i 和 j 所在循环层 | 3.81 |
| i 和 j 和 k 所在循环层 | 3.96 |




对于通过分块矩阵来提升性能，表 5 将列出使用 C 语言对不同大小的分块进行矩阵乘法的运行时间，而在表 1 中只将运行时间最短的一种放在表中计算浮点性能。

**表 5 不同大小的分块矩阵的运行时间**

| 块的大小 | 运行时间 |
|---|---|
| 4 | 2.16 |
| 8 | 1.08 |
| 16 | 0.74 |
| 32 | 0.61 |
| 64 | 0.50 |
| 128 | 0.45 |
| 256 | 0.47 |

```
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 2.164000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 1.083000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 0.737000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 0.613000 seconds.
Press Enter to continue...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 0.495000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 0.451000 seconds.
Press Enter to continue...
Continuing...
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> gcc -O3 -fopenmp -funroll-loops -o Project1.exe Project1_c.c
PS C:\Users\26618\Desktop\并行高性能计算程序设计\实验1代码> .\Project1.exe
Filling Matrix A (2048x2048) with random values
Filling Matrix B (2048x2048) with random values
Block Matrix multiplication completed in 0.470000 seconds.
Press Enter to continue...
```

## 5. 实验感想

在这次实验中，我深刻体会到了并行计算在提升程序性能方面的重要性。通过实现和比较 C、Python 和 Java 语言中的通用矩阵乘法算法，我不仅加深了对这些语言特性的理解，还学习到了如何在实际应用中优化算法以提高计算效率。在实验过程中，我首先注意到了 C 语言在未经优化的情况下，其执行效率已经明显高于 Python 和 Java。这让我意识到底层语言在处理这类密集型计算任务时的天然优势。随后，通过调整循环顺序、编译优化、多核并行以及分块矩阵等优化手段，我进一步观察到了程序性能的显著提升。特别是多核并行优化，它让我深刻认识到在现代多核处理器上，合理利用并行计算资源对于提升程序性能的重要性。在实验中，我尝试了对不同层循环进行并行化处理，发现并行化处理可以显著减少运行时间，提高浮点运算的性能。例如，通过对 i 所在循环层进行并行化，运行时间从原来的 85.81 秒降低到了 0.64 秒，这是一个巨大的提升。这让我意识到，并行计算是实现高性能计算的关键技术之一。