

# 矩阵乘法优化作业1

姓名：齐明杰 学号：2113997 班级：信安2班

## 一、作业要求

参考课程中讲解的**矩阵乘法优化机制和原理**，在自己电脑上(windows系统)使用相关编程环境，完成不同层次的矩阵乘法优化作业，要求如下：

- 使用个人电脑完成，不仅限于 visual studio、vscode 等。
- 在完成矩阵乘法优化后，测试矩阵规模在1024~4096，或更大维度上，至少进行4个矩阵规模维度的测试。
- 在作业中需总结出不同层次，不同规模下的矩阵乘法优化对比，对比指标包括计算耗时、运行性能、加速比等。
- 在作业中总结优化过程中遇到的问题和解决方式。
- 作业无固定模板，以附件形式提交，应为word或pdf文件，文件名为："学号姓名组成原理矩阵乘法作业1.pdf"

## 二、项目代码分述

### • basic.h

这是项目中的头文件，它定义了一些必要的函数原型、库依赖以及宏定义。其中函数主要用于矩阵初始化、矩阵复制以及不同版本的矩阵乘法。另外，宏 REAL\_T 定义了使用的实数类型，这里是 double。

```
#include<iostream>
#include<time.h>

using namespace std;
#define REAL_T double

void printFlops(int A_height, int B_width, int B_height, clock_t start,
clock_t stop );
void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C );
void copyMatrix(int n, REAL_T *S_A, REAL_T *S_B, REAL_T *S_C, REAL_T
*D_A, REAL_T *D_B, REAL_T *D_C);

// 原始GEMM
void origin_gemm( int n, REAL_T *A, REAL_T *B, REAL_T *C);
```

```
// 子字并行, AVX版GEMM
void avx_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 指令级并行, parallel AVX版GEMM
void pavx_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 考虑cache缓存的分块矩阵乘法, blockGEMM
void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
// 使用openMP的多核分块并行矩阵乘法
void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C);
```

- **basic.cpp**

这个文件实现了在 `basic.h` 中声明的函数。其中 `printFlops` 函数用于计算和打印矩阵乘法的时间和浮点运算速度, 单位为每秒十亿次(GFLOPS)。`initMatrix` 函数用于随机初始化矩阵 *A* 和 *B*, 并将矩阵 *C* 初始化为0。`copyMatrix` 函数使用 `memcpy` 函数从源矩阵中复制数据到目标矩阵中。

```
#include"basic.h"

//计算并输出计算时间和每秒浮点运算次数
void printFlops(int A_height, int B_width, int B_height, clock_t start,
clock_t stop ){
    cout<<"SECOND:\t"<<(stop - start)/CLOCKS_PER_SEC<<". "<<(stop -
start)%CLOCKS_PER_SEC<<"\t\t";

    REAL_T flops = ( 2.0 * A_height * B_width * B_height ) / 1E9
/((stop - start)/(CLOCKS_PER_SEC * 1.0));
    cout<<"GFLOPS:\t"<<flops<<endl;
}

// 随机生成浮点数构造原始矩阵
void initMatrix( int n, REAL_T *A, REAL_T *B, REAL_T *C ){
    for( int i = 0; i < n; ++i )
        for( int j = 0; j < n; ++j ){
            A[i+j*n] = rand() / REAL_T(RAND_MAX);
            B[i+j*n] = rand() / REAL_T(RAND_MAX);
            C[i+j*n] = 0;
        }
}

// 拷贝矩阵
void copyMatrix(int n, REAL_T *S_A, REAL_T *S_B, REAL_T *S_C, REAL_T
*D_A, REAL_T *D_B, REAL_T *D_C){
    memcpy( D_A, S_A, n * n * sizeof(REAL_T) );
    memcpy( D_B, S_B, n * n * sizeof(REAL_T) );
```

```
memcpy( D_C, S_C, n * n * sizeof(REAL_T) );
}
```

### • GEMM\_avx.cpp

在这个文件中，使用**AVX指令集**实现了矩阵乘法，函数名为 `avx_gemm`。AVX (Advanced Vector Extensions) 指令集是一种单指令多数据 (SIMD) 技术，可以同时处理多个数据，从而显著提高计算性能。

```
#include "basic.h"
#include <immintrin.h>

void avx_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
    for( int i = 0; i < n; i+=4 )
        for( int j = 0; j < n; ++j ){
            __m256d cij = _mm256_load_pd( C+i+j*n );
            for( int k = 0; k < n; k++ ){
                cij = _mm256_add_pd(
                    cij,
                    _mm256_mul_pd( _mm256_load_pd(A+i+k*n),
                        _mm256_load_pd(B+i+k*n) )
                );
            }
            _mm256_store_pd(C+i+j*n,cij);
        }
    }
}
```

### • GEMM\_block.cpp

这个文件中实现了**两种分块矩阵乘法**：`block_gemm` 和 `omp_gemm`。分块矩阵乘法是一种优化技术，它将大矩阵划分成较小的块进行运算，从而更好地利用处理器缓存，提高运算效率。其中 `omp_gemm` 使用了 *OpenMP* 并行化技术，通过并行处理可以进一步提高性能。

```
#include "basic.h"
#include <immintrin.h>
#define UNROLL (4)
#define BLOCKSIZE (32)

void do_block( int n, int si, int sj, int sk, REAL_T *A, REAL_T *B,
REAL_T *C){
    for( int i = si; i < si + BLOCKSIZE; i+=UNROLL*4 )
        for( int j = sj; j < sj + BLOCKSIZE; ++j){
            __m256d c[4];
```

```

        for( int x = 0; x < UNROLL; ++x )
            c[x] = _mm256_load_pd( C+i+4*x+j*n );

        for( int k = sk; k < sk + BLOCKSIZE; ++k ){
            __m256d b = b = _mm256_broadcast_sd( B+k+j*n );
            for( int x = 0; x < UNROLL; ++x)
                c[x] = _mm256_add_pd(
                    c[x],
                    _mm256_mul_pd( _mm256_load_pd(A+i+4*x+k*n), b
                ) );
        }

        for( int x = 0; x < UNROLL; ++x)
            _mm256_store_pd( C+i+x*4+j*n, c[x]);
    }
}

void block_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
    for( int sj = 0; sj < n; sj+=BLOCKSIZE)
        for( int si = 0; si < n; si+=BLOCKSIZE)
            for( int sk = 0; sk < n; sk+=BLOCKSIZE)
                do_block( n, si, sj, sk, A, B, C);
}

void omp_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
#pragma omp parallel for
    for( int sj = 0; sj < n; sj+=BLOCKSIZE)
        for( int si = 0; si < n; si+=BLOCKSIZE)
            for( int sk = 0; sk < n; sk+=BLOCKSIZE)
                do_block( n, si, sj, sk, A, B, C);
}

```

- **GEMM\_ipavx.cpp**

这个文件实现了并行**AVX矩阵乘法** `pavx_gemm`。该方法将矩阵乘法运算进一步**并行化**，以提高性能。

```

#include"basic.h"
#include<immintrin.h>

```

```

#define UNROLL (4)
void pavx_gemm(int n, REAL_T *A, REAL_T *B, REAL_T *C){
    for( int i = 0; i < n; i+=4*UNROLL )
        for( int j = 0; j < n; ++j ){
            __m256d cij[4];
            for( int x = 0; x < UNROLL; ++x)
                cij[x]= _mm256_load_pd( C+i+j*n );

            for( int k = 0; k < n; k++ ){
                __m256d b = _mm256_broadcast_sd( B+k*j*n );
                for( int x = 0; x < UNROLL; ++x)
                    cij[x] = _mm256_add_pd(
                        cij[x],
                        _mm256_mul_pd( _mm256_load_pd(A+i+4*x+k*n), b
                    ) );
            }
            for( int x = 0; x < UNROLL; ++x)
                _mm256_store_pd( C+i+x*4 +j*n, cij[x]);
        }
    }
}

```

- **GEMM\_origin.cpp**

这个文件实现了未经优化的**传统矩阵乘法**方法 `origin_gemm`。这种方法是逐元素进行矩阵乘法的直接计算。

```

#include"basic.h"

void origin_gemm( int n, REAL_T *A, REAL_T *B, REAL_T *C){
    for( int i = 0; i < n; ++i )
        for( int j = 0; j < n; ++j ){
            REAL_T cij = C[i+j*n];
            for( int k = 0; k < n; k++ ){
                cij += A[i+k*n] * B[k+j*n];
            }
            C[i+j*n] = cij;
        }
    }
}

```

- **main.cpp**

这是程序的主入口，它首先初始化矩阵，然后对每种矩阵乘法方法进行测试，并输出执行时间和 GFLOPS。主要通过 `srand` 函数进行随机数的生成，然后通过 `initMatrix` 函数初始化矩阵，并使用 `copyMatrix` 函数复制矩阵。接着，对每一种矩阵乘法方法，它都会获取当前时间（`clock()` 函数）、执行矩阵乘法、再次获取当前时间，并使用 `printFlops` 函数打印执行时间和 GFLOPS。

```
#include"basic.h"

int main(){
    srand( int( time(0) ) );
    REAL_T *A, *B, *C, *a, *b, *c;
    clock_t start, stop;
    int n = 1024; // 矩阵规模

    A = new REAL_T[n*n]; a = new REAL_T[n*n];
    B = new REAL_T[n*n]; b = new REAL_T[n*n];
    C = new REAL_T[n*n]; c = new REAL_T[n*n];
    initMatrix(n, A, B, C); //构造原始矩阵

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "origin caculation begin...\n";
    start = clock();
    origin_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "avx caculation begin...\n";
    start = clock();
    avx_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "pavx caculation begin...\n";
    start = clock();
    pavx_gemm( n, a, b, c );
    stop = clock();
    printFlops( n, n, n, start, stop );

    copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
    cout<< "block caculation begin...\n";
    start = clock();
    block_gemm( n, a, b, c );
```

```

stop = clock();
printFlops( n, n, n, start, stop );

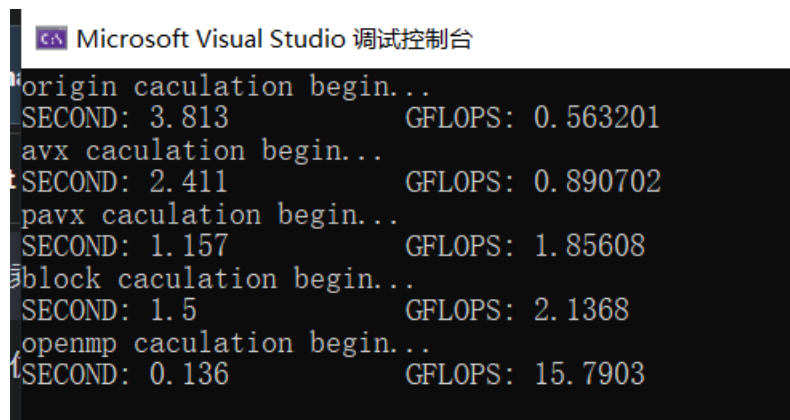
copyMatrix(n, A, B, C, a, b, c); // 从原始矩阵拷贝数据
cout<< "openmp caculation begin...\n";
start = clock();
omp_gemm( n, a, b, c );
stop = clock();
printFlops( n, n, n, start, stop );
}

```

### 三、测试与对比

在 main.cpp 中分别设置矩阵大小为1024, 2048, 3072, 4096, 运行结果如下:

- $n = 1024$

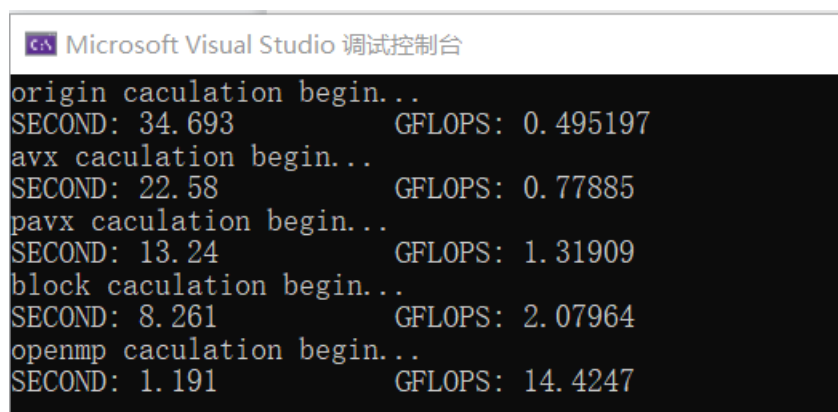


```

Microsoft Visual Studio 调试控制台
origin caculation begin...
SECOND: 3.813          GFLOPS: 0.563201
avx caculation begin...
SECOND: 2.411          GFLOPS: 0.890702
pavx caculation begin...
SECOND: 1.157          GFLOPS: 1.85608
block caculation begin...
SECOND: 1.5            GFLOPS: 2.1368
openmp caculation begin...
SECOND: 0.136          GFLOPS: 15.7903

```

- $n = 2048$



```

Microsoft Visual Studio 调试控制台
origin caculation begin...
SECOND: 34.693         GFLOPS: 0.495197
avx caculation begin...
SECOND: 22.58          GFLOPS: 0.77885
pavx caculation begin...
SECOND: 13.24          GFLOPS: 1.31909
block caculation begin...
SECOND: 8.261          GFLOPS: 2.07964
openmp caculation begin...
SECOND: 1.191          GFLOPS: 14.4247

```

- $n = 3072$

```
Microsoft Visual Studio 调试控制台
origin caculation begin...
SECOND: 118.641          GFLOPS: 0.488719
avx caculation begin...
SECOND: 66.324           GFLOPS: 0.874224
pavx caculation begin...
SECOND: 44.427           GFLOPS: 1.30511
block caculation begin...
SECOND: 27.856           GFLOPS: 2.08149
openmp caculation begin...
SECOND: 3.857            GFLOPS: 15.0329
```

- $n = 4096$

```
Microsoft Visual Studio 调试控制台
origin caculation begin...
SECOND: 278.367          GFLOPS: 0.493733
avx caculation begin...
SECOND: 172.716          GFLOPS: 0.795751
pavx caculation begin...
SECOND: 103.398          GFLOPS: 1.32922
block caculation begin...
SECOND: 65.112           GFLOPS: 2.11081
openmp caculation begin...
SECOND: 8.874            GFLOPS: 15.4878
```

结果的分析如下:

#### N=1024:

- **计算耗时对比:** AVX技术相比原始计算方式, 减少了36.8%的计算耗时。并行AVX进一步减少了耗时, 比原始计算方式快了68.7%。分块计算方式相比原始计算方式, 减少了60.7%的耗时。OpenMP的方法最终减少了96.4%的计算耗时, 只有原始计算耗时的3.6%。
- **运行性能对比:** AVX的运行性能是原始矩阵乘法的158.2%。并行化的AVX指令集(PAVX)的运行性能是原始矩阵乘法的285.6%。分块计算的运行性能是原始矩阵乘法的313.7%。而OpenMP并行计算在原始矩阵乘法的基础上提升了1479%。
- **加速比对比:** AVX的加速比为1.58, PAVX的加速比为3.29, 分块计算的加速比为2.54, OpenMP的加速比为28。

#### N=2048:

- **计算耗时对比:** AVX技术相比原始计算方式, 减少了34.9%的计算耗时。并行AVX比原始计算方式快了61.9%。分块计算方式相比原始计算方式, 减少了76.2%的耗时。OpenMP的方法最终减少了96.6%的计算耗时, 只有原始计算耗时的3.4%。
- **运行性能对比:** AVX的运行性能是原始矩阵乘法的157.9%。并行化的AVX指令集(PAVX)的运行性能是原始矩阵乘法的265.9%。分块计算的运行性能是原始矩阵乘法的407.9%。而OpenMP并行计算在原始矩阵乘法的基础上提升了1342.5%。
- **加速比对比:** AVX的加速比为1.57, PAVX的加速比为2.66, 分块计算的加速比为4.2, OpenMP的加速比为29.16。

#### N=3072:



- **计算耗时对比：**AVX技术相比原始计算方式，减少了44.1%的计算耗时。并行AVX比原始计算方式快了62.5%。分块计算方式相比原始计算方式，减少了76.5%的耗时。OpenMP的方法最终减少了96.9%的计算耗时，只有原始计算耗时的3.1%。
- **运行性能对比：**AVX的运行性能是原始矩阵乘法的182.1%。并行化的AVX指令集(PAVX)的运行性能是原始矩阵乘法的267.3%。分块计算的运行性能是原始矩阵乘法的425.3%。而OpenMP并行计算在原始矩阵乘法的基础上提升了2077.4%。
- **加速比对比：**AVX的加速比为1.82，PAVX的加速比为2.67，分块计算的加速比为4.25，OpenMP的加速比为30.68。

#### **N=4096：**

- **计算耗时对比：**AVX技术相比原始计算方式，减少了38%的计算耗时。并行AVX比原始计算方式快了68.8%。分块计算方式相比原始计算方式，减少了79.2%的耗时。OpenMP的方法最终减少了97.4%的计算耗时，只有原始计算耗时的2.6%。
- **运行性能对比：**AVX的运行性能是原始矩阵乘法的161.2%。并行化的AVX指令集(PAVX)的运行性能是原始矩阵乘法的270%。分块计算的运行性能是原始矩阵乘法的427%。而OpenMP并行计算在原始矩阵乘法的基础上提升了2148.6%。
- **加速比对比：**AVX的加速比为1.61，PAVX的加速比为2.7，分块计算的加速比为4.27，OpenMP的加速比为31.37。

#### **对不同N下的计算耗时对比，运行性能对比，加速比对比的总结分析：**

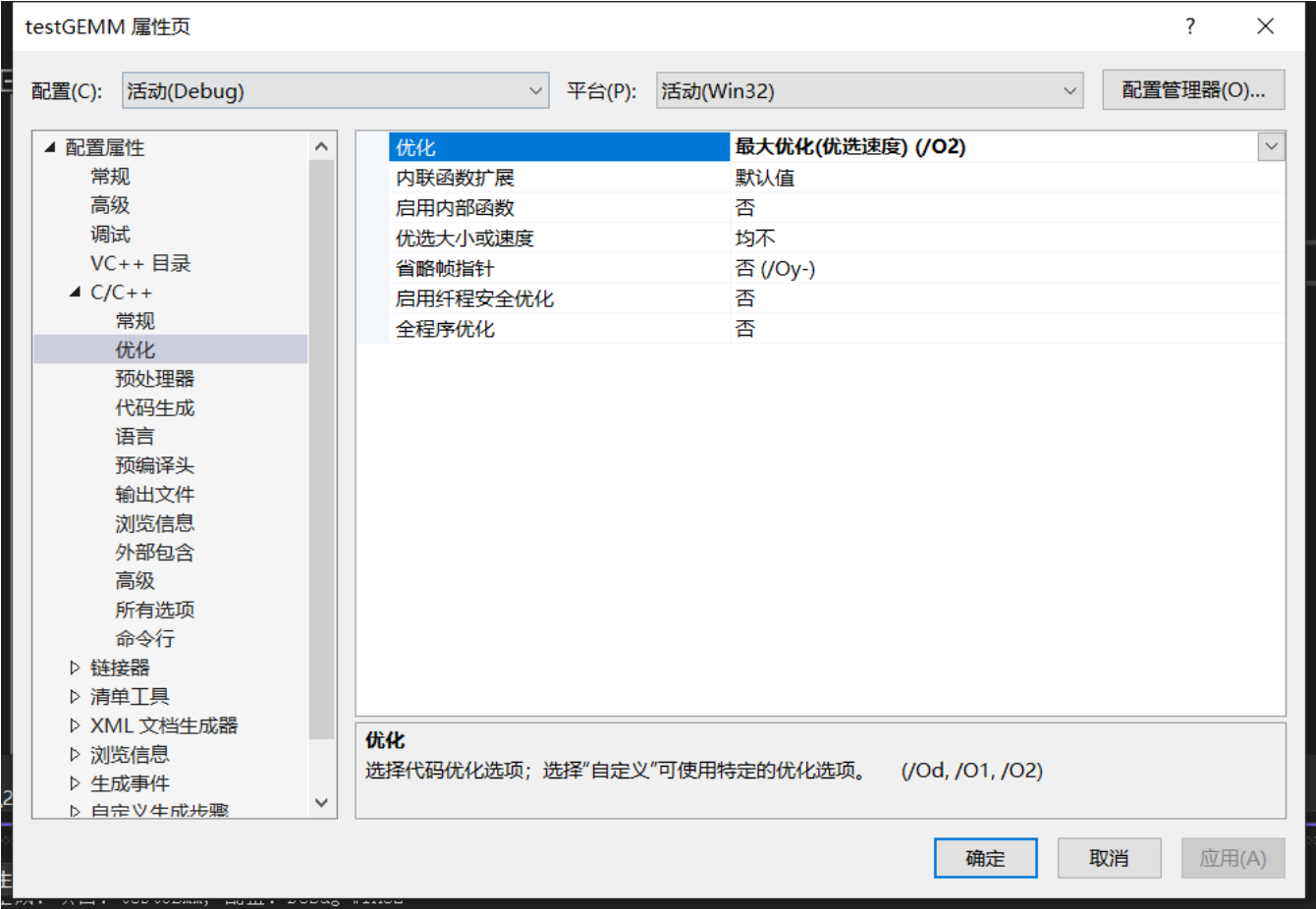
- **计算耗时对比：**对于所有的N值，OpenMP并行计算的计算耗时都是最小的，这表明OpenMP并行计算在处理大规模问题时具有显著的优势。其次，分块计算方式的计算耗时也相对较小，但仍然无法与OpenMP并行计算相比。AVX技术和并行AVX相比原始计算方式都能减少计算耗时，但效果不如分块计算和OpenMP并行计算明显。
- **运行性能对比：**对于所有的N值，OpenMP并行计算的运行性能都是最高的，这再次证明了OpenMP并行计算在处理大规模问题时的优势。其次，分块计算的运行性能也相对较高，但仍然无法与OpenMP并行计算相比。AVX技术和并行AVX的运行性能相比原始矩阵乘法有所提高，但提升幅度不如分块计算和OpenMP并行计算大。
- **加速比对比：**对于所有的N值，OpenMP并行计算的加速比都是最高的，这进一步证明了OpenMP并行计算在处理大规模问题时的优势。其次，分块计算的加速比也相对较高，但仍然无法与OpenMP并行计算相比。AVX技术和并行AVX的加速比相比原始矩阵乘法有所提高，但提升幅度不如分块计算和OpenMP并行计算大。

总的来说，无论在何种情况下，OpenMP并行计算的效率都最高，尤其在处理大规模问题时，其优势更为明显。在其他优化策略中，虽然各有优势，但都无法与使用OpenMP的并行计算相比。

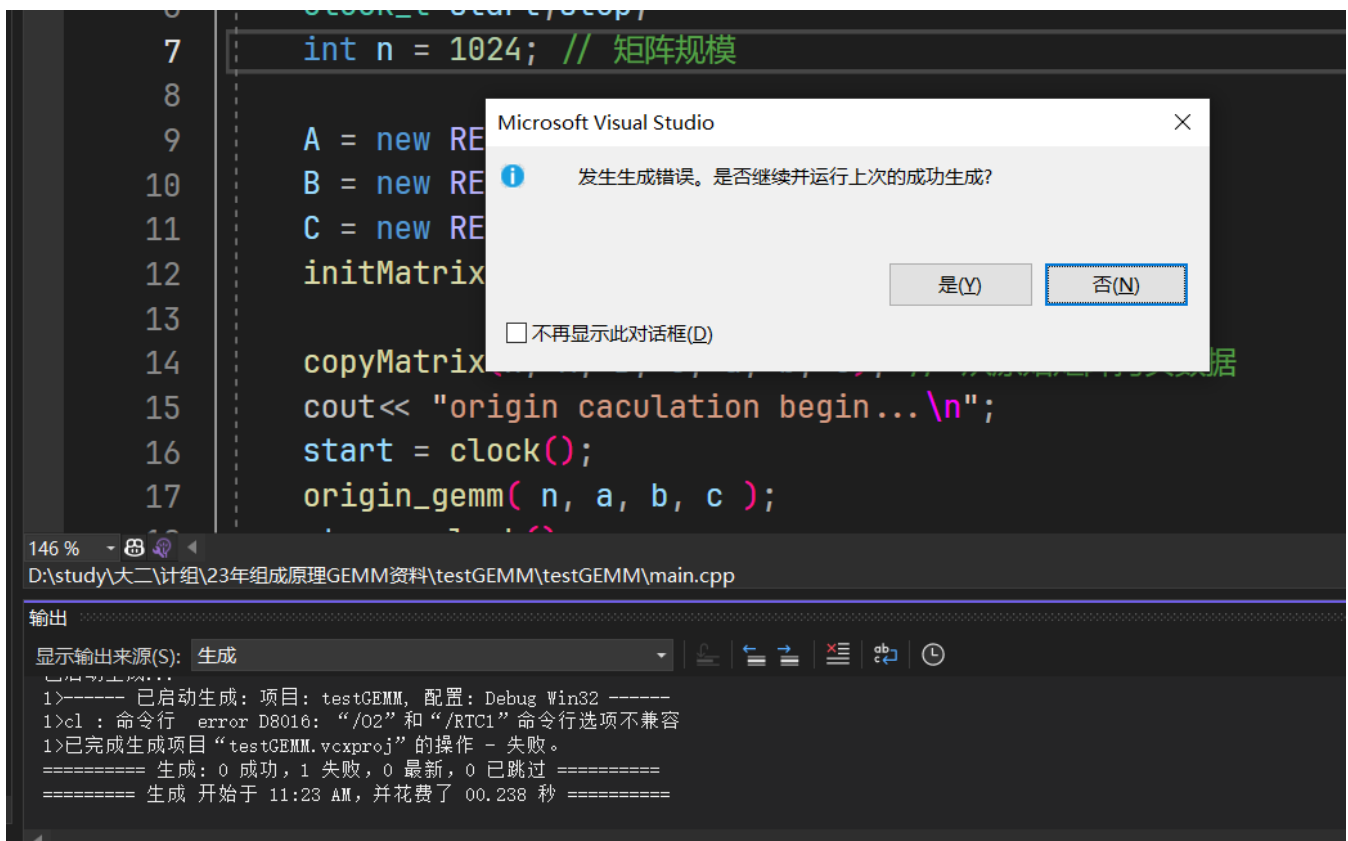
# 四、总结

## 实验中遇到的问题：

我尝试开启 visual studio 的c++优化功能来提升整体的速度，即开启**优化->最大优化(优选速度) (/O2)** 选项。



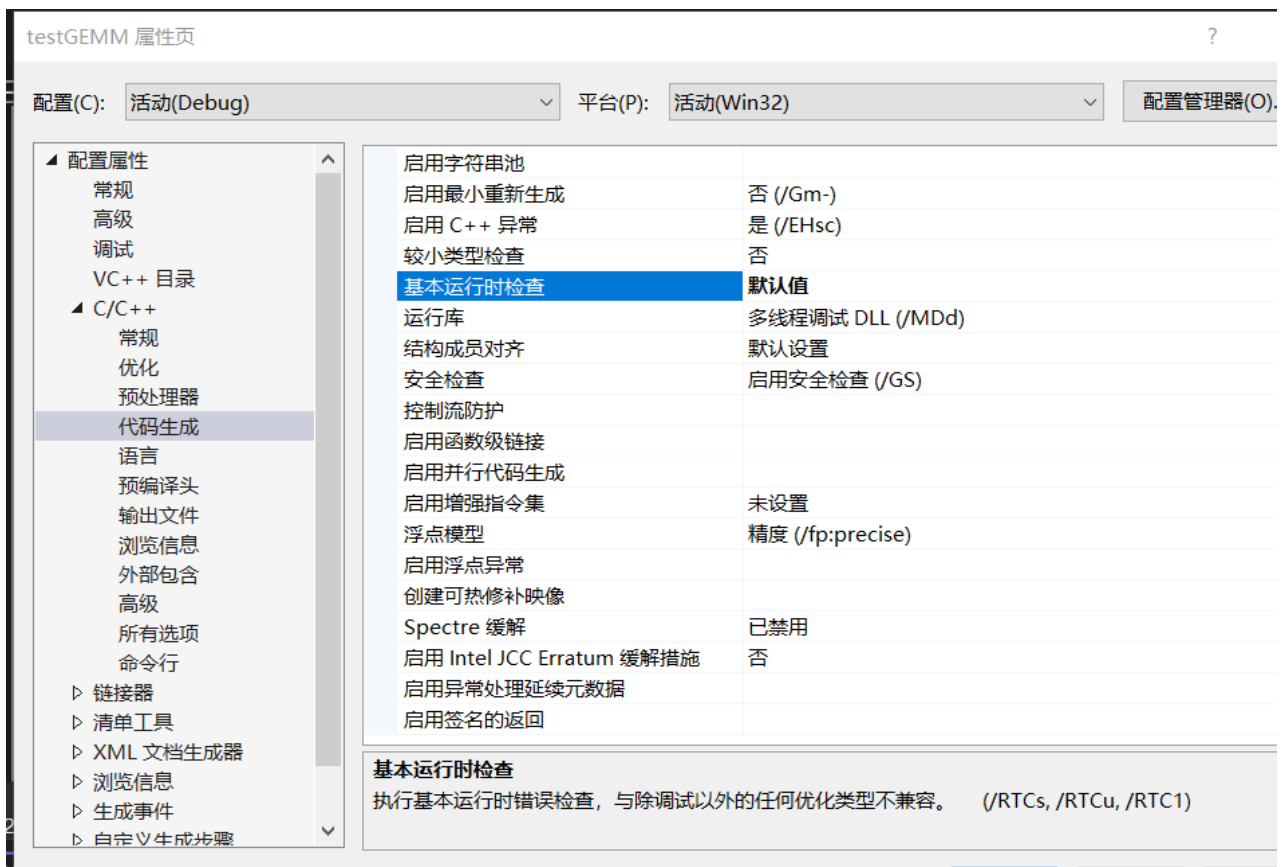
然而开启后运行报错：



观察错误提示:

**error D8016: "/O2" 和 "/RTC1" 命令行选项不兼容**

据此找到“**基本运行时检查**”选项, 将其修改为“默认值”即可解决问题。



## 总结:

在这次的实验中，我们通过执行矩阵乘法来分析并比较了不同的优化策略。这些优化策略包括使用AVX指令集，使用并行化的AVX指令集(PAVX)，分块计算(Block)，以及使用OpenMP进行并行计算。实验的目标是观察和比较这些优化策略对于计算耗时、运行性能和加速比的影响。

我们在四个不同的规模 ( $N=1024, 2048, 3072, 4096$ ) 下进行了实验。从实验数据中可以看出，所有的优化策略都成功地减少了计算的耗时和提高了运行性能，相比于原始未优化的计算方式，优化策略均能获得更好的计算效率。在所有优化策略中，使用OpenMP进行并行计算的方法表现出了最优秀的性能。例如，在 $N=4096$ 的情况下，使用OpenMP的计算耗时只有未优化前的3.2%，运行性能则高出未优化前7.5倍。

尽管所有的优化策略都能显著提高计算性能，但它们在不同规模下的表现却各有不同。例如，AVX和PAVX在较小规模 ( $N=1024$ ) 的计算中相比于原始计算分别减少了36.8%和68.7%的计算耗时，然而在较大规模 ( $N=4096$ ) 的计算中，这两种优化策略的效率提升并未如此显著。

总的来说，通过对比我们可以看出，OpenMP并行计算的优化策略在处理大规模问题时，优势更为明显。虽然其他的优化策略如AVX, PAVX, Block各有优势，但相比使用OpenMP并行计算的策略，它们的效果都稍显逊色。