report

Implementation

a. How do you implement the matrix multiplication (both version)?

as shown,

standard version:

```
inline void MMUL() noexcept
{
    for (int x = 0; x < n; ++x)
        for (int y = 0; y < m; ++y)
            for (int z = 0; z < k; ++z)
                C[x][y] += A[x][z] * B[z][y];
}
```

blocked version:

```
inline void MMUL() noexcept
{
    for (int x = 0; x < n; x += BLOCK)
        for (int y = 0; y < m; y += BLOCK)
            for (int z = 0; z < k; z += BLOCK)
            {
                for (int xx = x; xx < std::min(n, x + BLOCK); ++xx)
                    for (int yy = y; yy < std::min(m, y + BLOCK); ++yy)
                        for (int zz = z; zz < std::min(k, z + BLOCK);
++zz)
                            C[xx][yy] += A[xx][zz] * B[zz][yy];
            }
}
```

b. How do you verify the correctness of your program?

the standard matrix multiplication program is 100% correct (has been verified by online judge), so i take it as standard program.

then i wrote a test data generator, put the test data into standard program, generating standard answers.

use `diff` command to find the difference between standard answers and other version's answer.

if `diff` detect difference, use another program to calculate the error.

c. Why the blocked version is correct? Answer by explaining the mathematics or program's calculation steps.

block matrix multiplication says:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,k} \\ A_{2,1} & \dots & & \\ \dots & \dots & & \\ A_{n,1} & \dots & & \end{bmatrix}$$

$$B = \begin{bmatrix} B_{1,1} & B_{1,2} & \dots & B_{1,m} \\ B_{2,1} & \dots & & \\ \dots & \dots & & \\ B_{k,1} & \dots & & \end{bmatrix}$$

the multiplication AB can be performed blockwise.

$C_{q,r} = \sum_{i=1}^{k'} A_{q,i} B_{i,r}$

for example:

$$A = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

blocked:
$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$, where $A_{1,1} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $A_{1,2} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$, $A_{2,1} = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}$, $A_{2,2} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$

$$A^2 = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1}A_{1,1} + A_{1,2}A_{2,1} & A_{1,1}A_{1,2} + A_{1,2}A_{2,2} \\ A_{2,1}A_{1,1} + A_{2,2}A_{2,1} & A_{2,1}A_{1,2} + A_{2,2}A_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 3 \end{bmatrix}$$

the blocked version operates like the mathematics.

Experiment

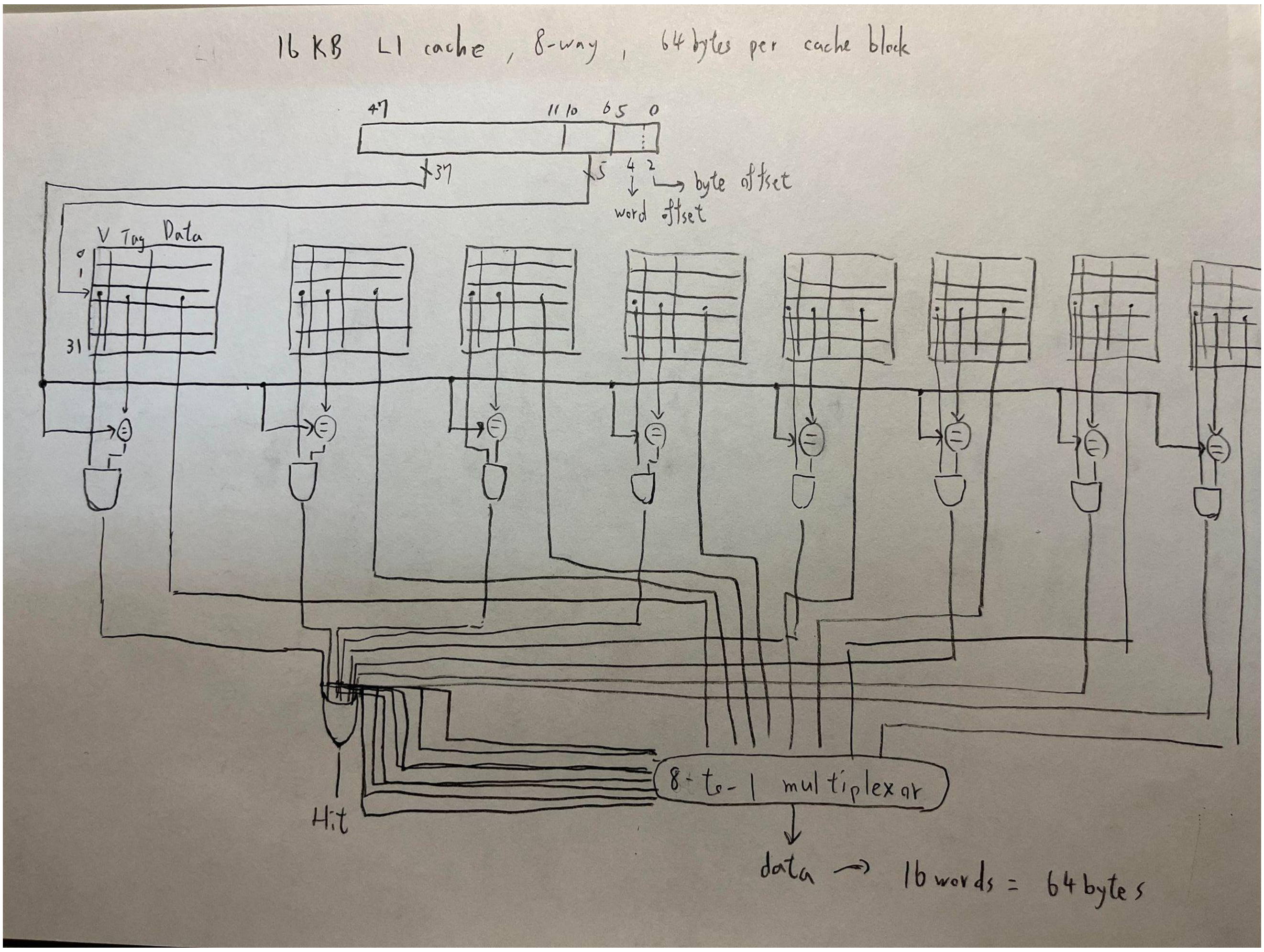
environment:

- hardware
  - CPU: AMD Ryzen 7 PRO 4750U with Radeon Graphics @ 1.70 GHz
  - RAM: DDR4 3200 MHz
  - Cache:
    - L1: 8 $*$ 32 KB, 8-way
    - L2: 8 $*$ 512 KB, 8-way

- L3: 2 * 4 MB, 16-way
- software
  - compiler version: g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
  - compile command:
    - `g++ -Wall -OX -c matrix.cpp`
    - `g++ -Wall -OX -DBLOCK=XX -c blocked-matrix.cpp`
    - `g++ -Wall -OX -DBLOCK=XX -c blocked-cache-matrix.cpp`
    - `g++ -msse -msse2 -msse3 -mssse3 -msse4 -msse4a -msse4.1 -msse4.2 -Wall -OX -c vector-matrix.cpp`
  - OS: Linux emilia 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux

a.

my L1 cache is 32 KB, so the cache below is 16 KB.
the index counts is $16 \times 2^{10}/(8 \times 64) = 2^5$, so 5 bits is for index.
the cache line is 64 byte, namely 16 words, so the width of word offset is 4-bit, and the byte offset is 2-bit.
the left bit is for tag, which is $48 - 5 - 4 - 2 = 37$-bit.



b.

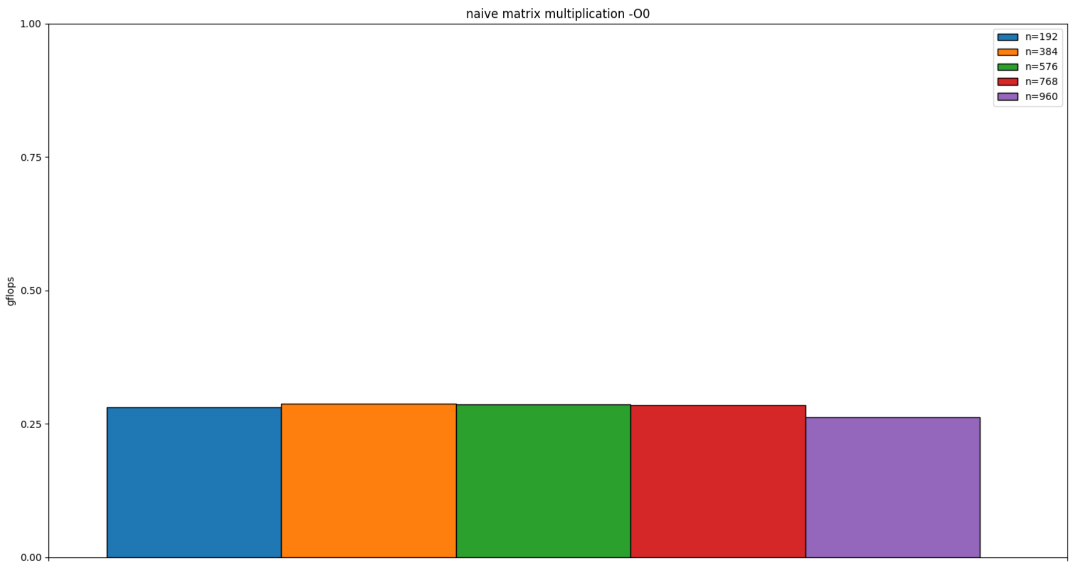
the test data have different sizes, whose $n = k = m = 192, 384, 576, 768, 960$, produced by the geneator.

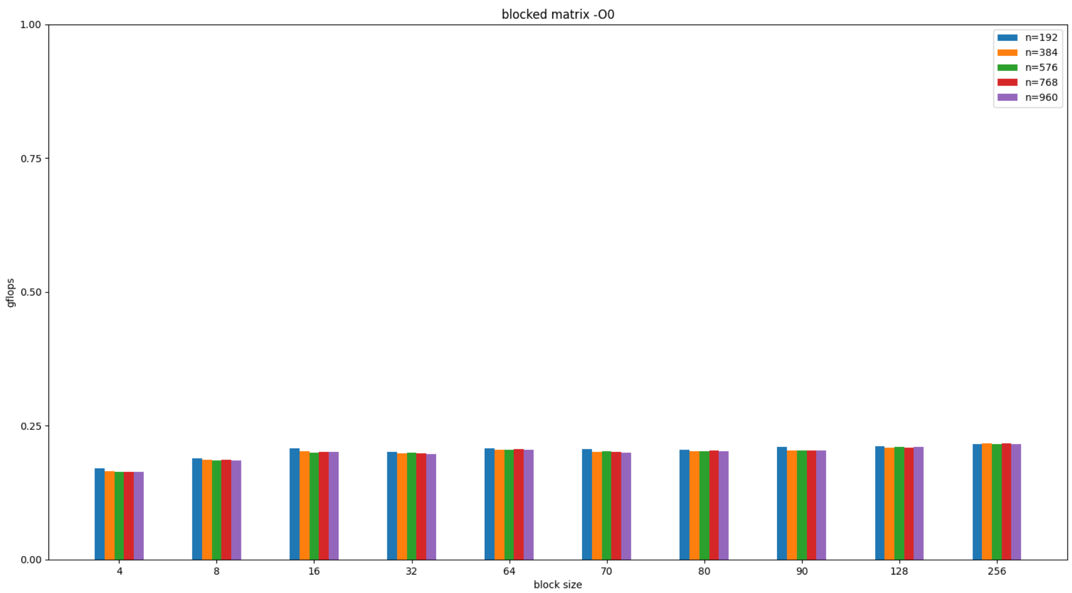
partial code of geneator:

```
std::random_device rd;
std::default_random_engine eng(rd());
std::uniform_real_distribution<float> unif(-1000, 1000);

n = k = m = 192 * CNT;
std::cout << n << ' ' << k << ' ' << m << '\n';
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < k; ++j)
        std::cout << unif(eng) << ' ';
    std::cout << '\n';
}
for (int i = 0; i < k; ++i)
{
    for (int j = 0; j < m; ++j)
        std::cout << unif(eng) << ' ';
    std::cout << '\n';
}
```

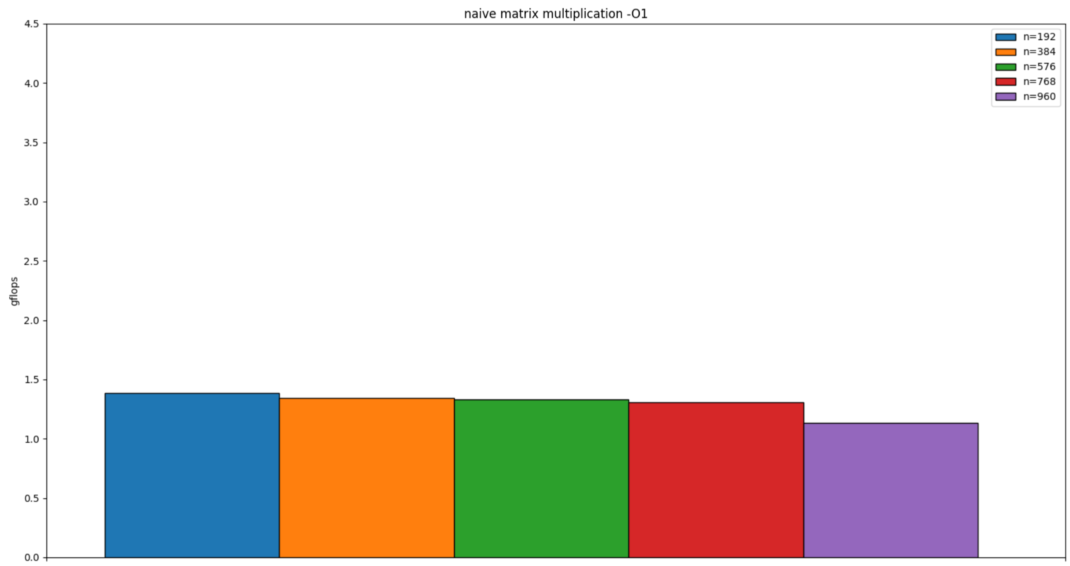
c.
matrix multiplication programs will run 10 times on every test data (n=192,384,576,768,960), every optimization flags (O0,O1,O2,O3,Ofast), every block size (4,8,16,32,64,70,80,90,128,256), and record the execution time of `MMUL()`.
later, a python program calculates average execution time, maximum/minimum execution time, standard deviation of execution time, and average GFLOPS.

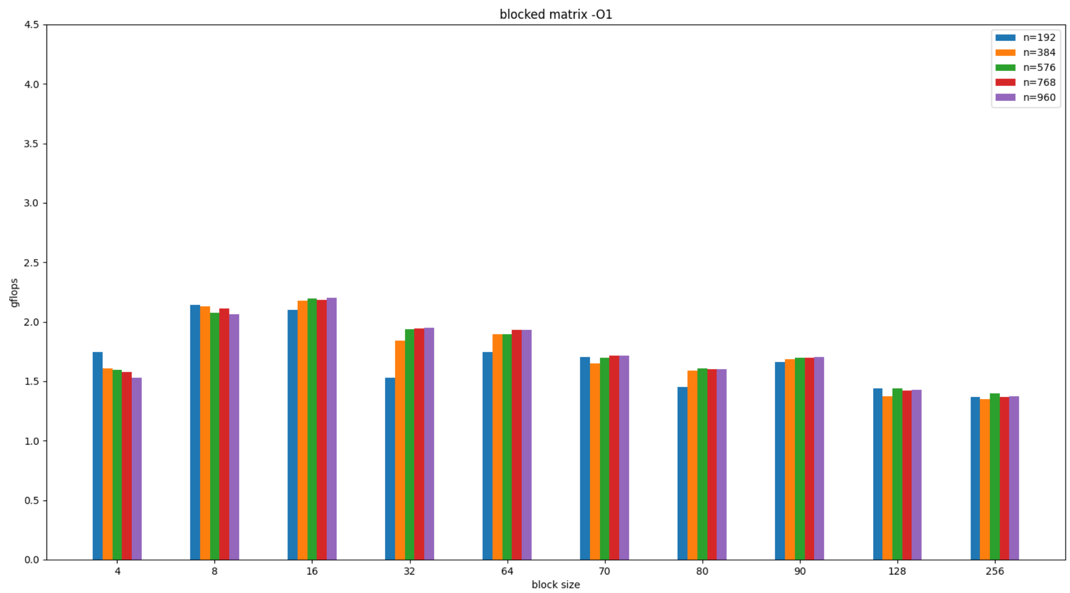
GFLOPS of standard version and blocked version in -O0:



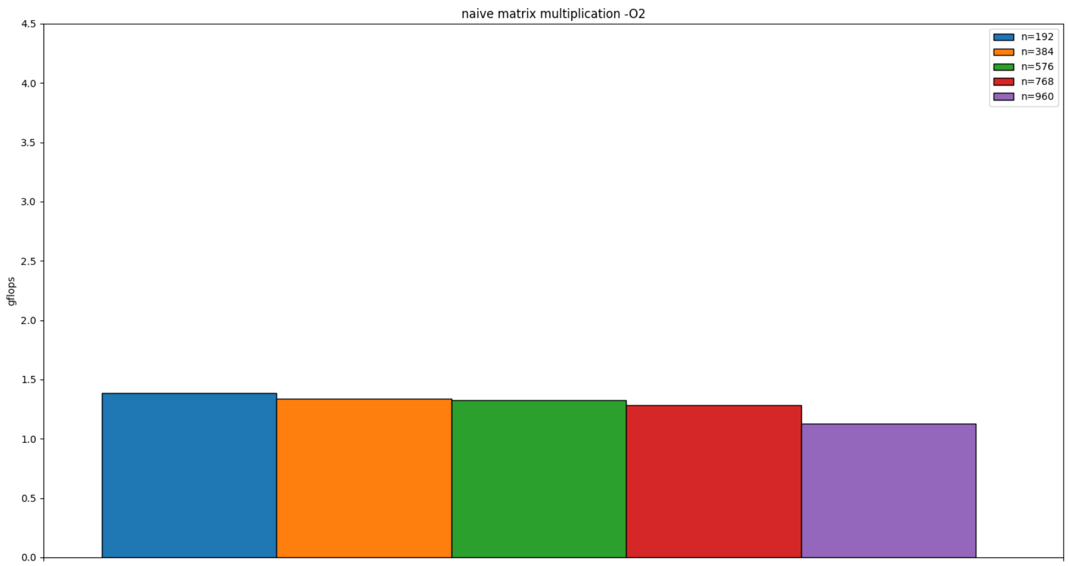


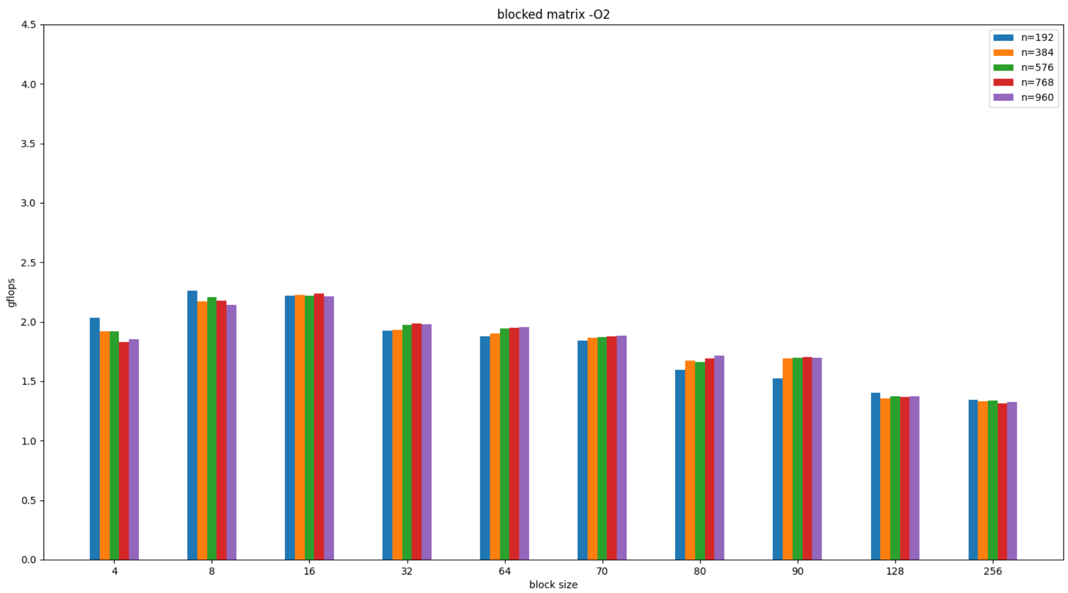
GFLOPS of standard version and blocked version in -O1:



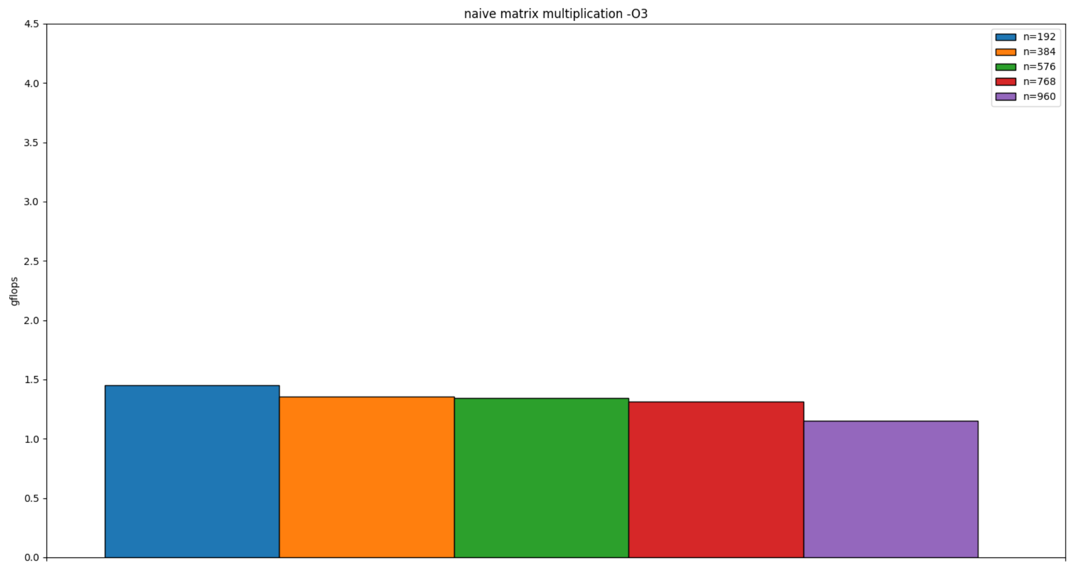


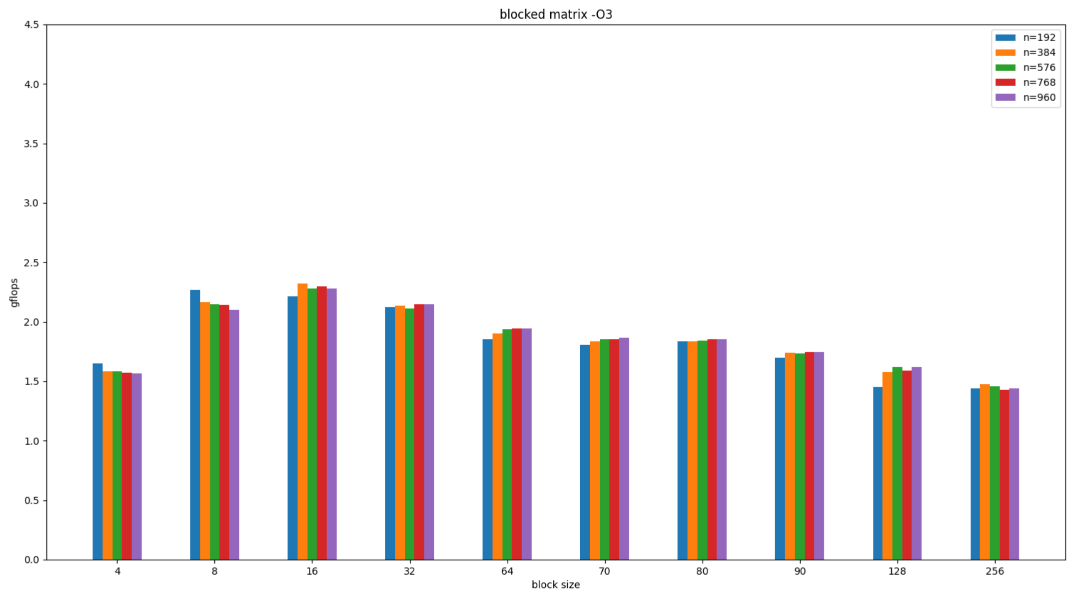
GFLOPS of standard version and blocked version in -O2:



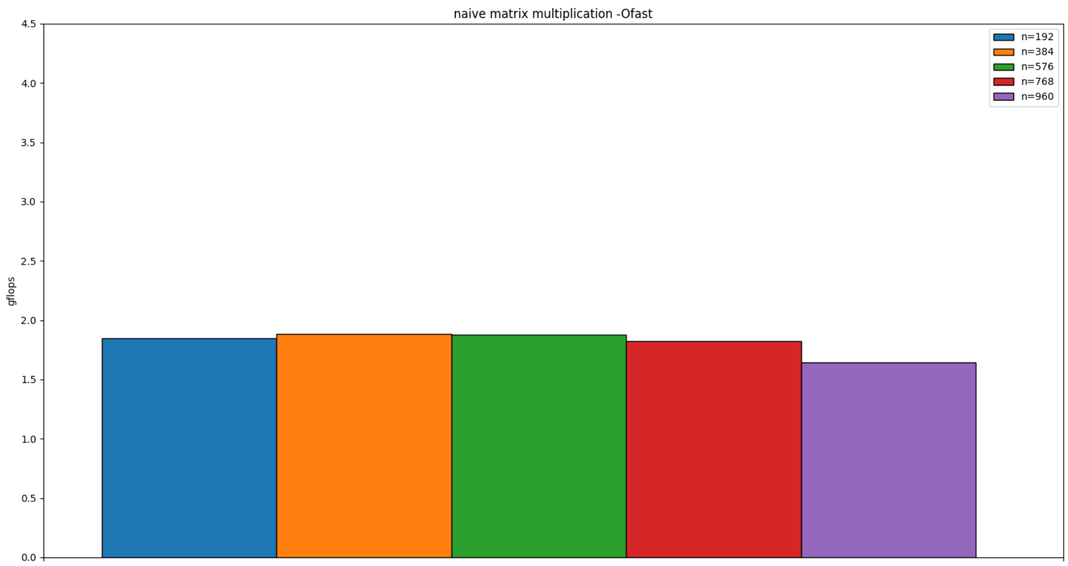


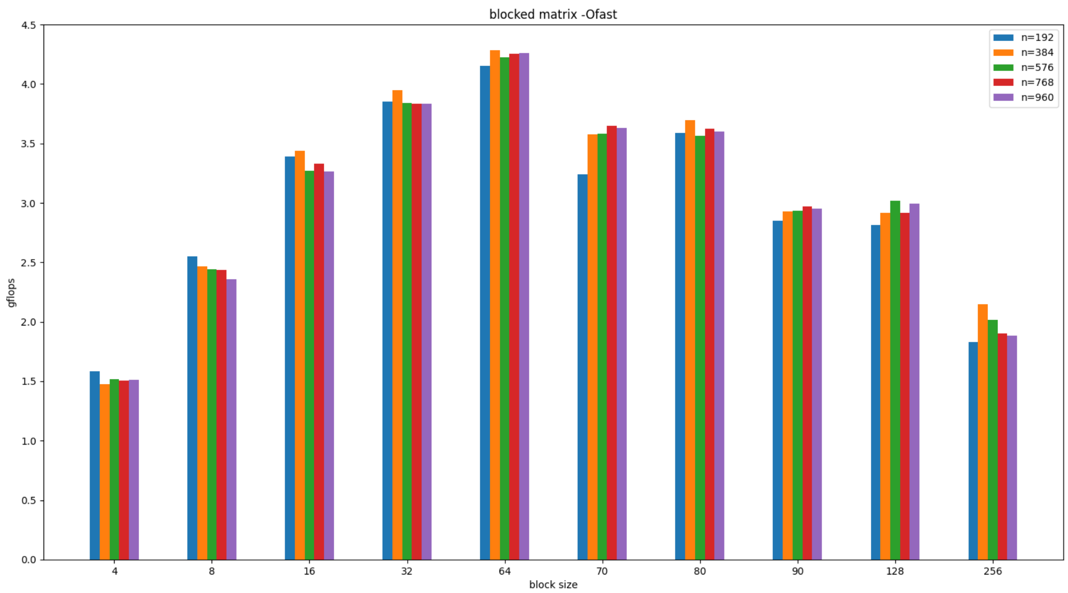
GFLOPS of standard version and blocked version in -O3:





GFLOPS of standard version and blocked version in -Ofast:





the number of float-point operations on a matrix multiplication of $AB = C$ is $n \times k \times m$, where A is a $n \times k$ matrix and B is a $k \times m$ matrix.

so FLOPS for a matrix multiplication is $\frac{n \cdot k \cdot m}{time(s)} = \frac{n \cdot k \cdot m}{time \times 10^9 (ns)} \times 10^9 = \frac{n \cdot k \cdot m}{time \times 10^9 (ns)}$ GFLOPS

d.

i use `<chrono>` in std c++ 11 to measure the executing time of matrix multiplication function.

```
auto begin = std::chrono::high_resolution_clock::now();
MMUL();
auto end = std::chrono::high_resolution_clock::now();
```

```
auto T = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin);
```

i think it's an appropriate way to measure "a function" executing time.
but if i want to measure "a program" execution time, the command, `time`, provided by linux is the better way.
moreover, the linux kernel tool, `perf`, is an even better way to measure a process executing time and collect some CPU info in executing program, e.g. cache miss, cache reference, instruction counts.

e.
the difference between standard and blocked version are, the counts of **cache references** and **cache misses**.
since the blocked version divides the big matrix into pieces, the cache misses counts is lower than the standard one.

for example, this is the CPU info of standard version in $n = 960$:

```
 Performance counter stats for './matrix00' (10 runs):

        48,461,646      cache-misses              #    4.540 % of all cache
refs         ( +- 11.90% )
     1,067,320,326      cache-references
( +-  2.53% )
    52,413,831,567      instructions              #    3.41  insn per cycle
( +-  0.00% )
    15,360,089,940      cycles
( +-  0.88% )

             4.055 +- 0.235 seconds time elapsed  ( +-  5.80% )
```

and the CPU info of blocked version in $n = 960$ and block size is 128:

```
 Performance counter stats for './blocked-matrixB12800' (10 runs):

         1,818,247      cache-misses              #    0.367 % of all cache
refs         ( +-  1.88% )
       494,989,842      cache-references
( +-  0.87% )
    74,008,535,839      instructions              #    4.08  insn per cycle
( +-  0.00% )
    18,121,499,016      cycles
( +-  0.04% )
```

```
            4.732 +- 0.284 seconds time elapsed  ( +-  6.01% )
```

same version but block size is 64:

```
 Performance counter stats for './blocked-matrixB6400' (10 runs):

         2,398,344      cache-misses              #    7.242 % of all cache
refs       ( +-  1.32% )
        33,118,625      cache-references
( +-  6.44% )
    74,391,504,589      instructions              #    4.02  insn per cycle
( +-  0.00% )
    18,526,118,447      cycles
( +-  0.08% )

             4.835 +- 0.299 seconds time elapsed  ( +-  6.18% )
```

can be seen that the **cache misses in blocked version is less than standard version_**.

let's look into the impact of different data size:

- $n = 192$

standard version:

```
 Performance counter stats for './matrix00' (10 runs):

           106,895      cache-misses              #    1.346 % of all cache
refs       ( +-  3.98% )
         7,940,962      cache-references
( +-  0.46% )
       544,471,973      instructions              #    3.25  insn per cycle
( +-  0.00% )
       167,631,313      cycles
( +-  0.26% )

           0.07048 +- 0.00107 seconds time elapsed  ( +-  1.52% )
```

blocked version, block size = 128:

```
 Performance counter stats for './blocked-matrixB12800' (10 runs):

            98,791      cache-misses              #    3.204 % of all cache
```

```
refs       ( +-  2.15% )
         3,083,612      cache-references
( +-  0.83% )
       715,911,619      instructions              #    3.54  insn per cycle
( +-  0.00% )
       202,427,783      cycles
( +-  0.12% )

          0.084390 +- 0.000992 seconds time elapsed  ( +-  1.18% )
```

- $n = 384$

standard version:

```
 Performance counter stats for './matrixO0' (10 runs):

         2,112,244      cache-misses              #    3.404 % of all cache
refs       ( +-  6.72% )
        62,043,825      cache-references
( +-  0.47% )
     3,720,123,833      instructions              #    3.54  insn per cycle
( +-  0.01% )
     1,052,366,945      cycles
( +-  0.23% )

           0.42687 +- 0.00193 seconds time elapsed  ( +-  0.45% )
```

blocked version, block size = 128:

```
 Performance counter stats for './blocked-matrixB128O0' (10 runs):

           220,852      cache-misses              #    0.799 % of all cache
refs       ( +-  0.78% )
        27,640,480      cache-references
( +-  0.40% )
     5,084,310,460      instructions              #    3.87  insn per cycle
( +-  0.00% )
     1,312,470,789      cycles
( +-  0.08% )

           0.52843 +- 0.00116 seconds time elapsed  ( +-  0.22% )
```

- $n = 576$

standard version:

```
 Performance counter stats for './matrix00' (10 runs):

        10,765,510      cache-misses              #    5.176 % of all cache
refs        ( +-  7.13% )
       207,970,299      cache-references
( +-  0.89% )
    11,866,131,508      instructions              #    3.64  insn per cycle
( +-  0.00% )
     3,261,742,974      cycles
( +-  0.31% )

            1.1011 +- 0.0822 seconds time elapsed  ( +-  7.47% )
```

blocked version, block size = 128:

```
 Performance counter stats for './blocked-matrixB12800' (10 runs):

           623,016      cache-misses              #    0.639 % of all cache
refs        ( +-  5.10% )
        97,500,272      cache-references
( +-  1.36% )
    16,524,654,323      instructions              #    3.97  insn per cycle
( +-  0.00% )
     4,160,590,668      cycles
( +-  0.09% )

             1.344 +- 0.107 seconds time elapsed  ( +-  7.95% )
```

- $n = 768$

standard version:

```
 Performance counter stats for './matrix00' (10 runs):

        22,595,185      cache-misses              #    4.379 % of all cache
refs        ( +-  5.66% )
       516,047,305      cache-references
( +-  0.85% )
    27,318,174,007      instructions              #    3.68  insn per cycle
( +-  0.00% )
     7,420,095,030      cycles
( +-  0.15% )
```

```
          2.139 +- 0.175 seconds time elapsed  ( +-  8.20% )
```

blocked version, block size = 128:

```
 Performance counter stats for './blocked-matrixB12800' (10 runs):

         1,121,814      cache-misses              #    0.475 % of all cache
refs       ( +-  3.22% )
       236,109,274      cache-references
( +-  1.18% )
    38,314,518,495      instructions              #    4.04  insn per cycle
( +-  0.00% )
     9,483,123,332      cycles
( +-  0.07% )

          2.660 +- 0.194 seconds time elapsed  ( +-  7.29% )
```

- $n = 960$

standard version:

```
 Performance counter stats for './matrix00' (10 runs):

        48,461,646      cache-misses              #    4.540 % of all cache
refs       ( +- 11.90% )
     1,067,320,326      cache-references
( +-  2.53% )
    52,413,831,567      instructions              #    3.41  insn per cycle
( +-  0.00% )
    15,360,089,940      cycles
( +-  0.88% )

          4.055 +- 0.235 seconds time elapsed  ( +-  5.80% )
```

blocked version, block size = 128:

```
 Performance counter stats for './blocked-matrixB12800' (10 runs):

         1,987,038      cache-misses              #    0.408 % of all cache
refs       ( +-  8.74% )
       486,950,362      cache-references
( +-  1.00% )
    74,010,443,050      instructions              #    4.08  insn per cycle
```

```
( +-  0.00% )
    18,156,541,410      cycles
( +-  0.11% )

             4.796 +- 0.289 seconds time elapsed  ( +-  6.02% )
```

can be seen the **cache misses/references in blocked version is much less than standard one**, and, **though these two counts increase when test data size increases, the amount of increasing in blocked version is much less than standard one**.

obviously, the **blocked version exploit more on the cache space**.

so i expect the execution time of blocked version is shorter than standard one.

however, the results are not as expected.
in the O0 flag, the performance of blocked version is worse than the standard one, because the instruction counts is much more than standard one.

the turth is, without the compiler optimization, blocked version takes no advantage.

a for-loop generally need three steps:

1. check the stop condition
2. do something in the for-loop section
3. change the counting variable

the additional three for-loop has increase the count of checking the stop condition, and not to mention the burden added by the `std::min()` function.
hence, the count of instructions is much more than standard one, consequently, the GFLOPS performace is worse than the standard one.

```
inline void MMUL() noexcept
{
    for (int x = 0; x < n; x += BLOCK)
        for (int y = 0; y < m; y += BLOCK)
            for (int z = 0; z < k; z += BLOCK)
            {
                // the std::min() function is called in every iteration !
                for (int xx = x; xx < std::min(n, x + BLOCK); ++xx)
                    for (int yy = y; yy < std::min(m, y + BLOCK); ++yy)
                        for (int zz = z; zz < std::min(k, z + BLOCK);
++zz)
                            C[xx][yy] += A[xx][zz] * B[zz][yy];
            }
}
```

there's a simple way to imporve the performance by modifying the code:

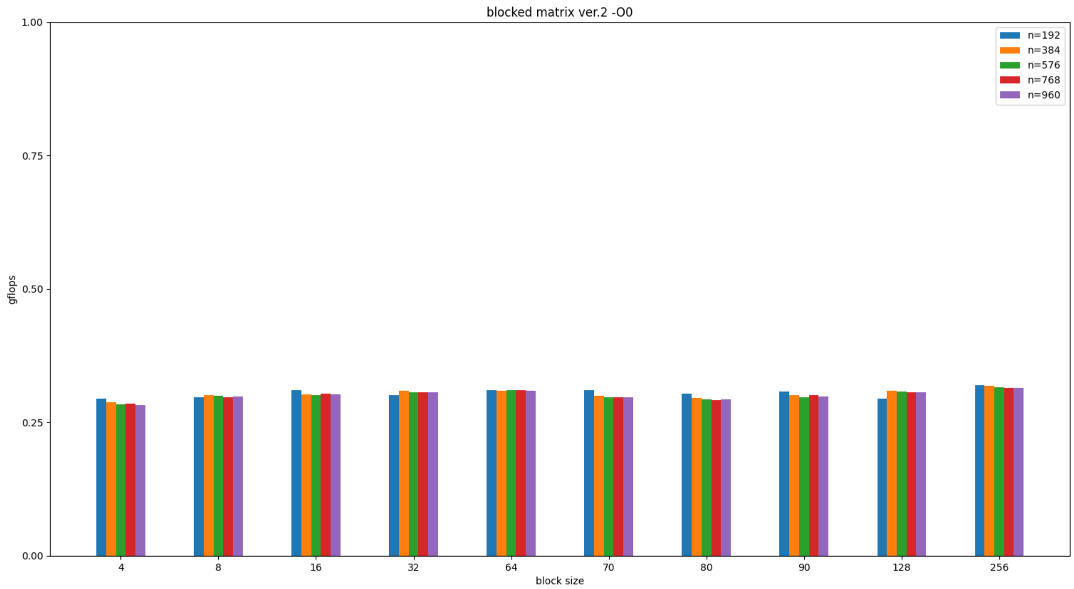
```
#define min(a,b) ((a)>(b)?(b):(a))
inline void MMUL() noexcept
{
    int x, y, z, xx, yy, zz, maxx, maxy, maxz;
    for (x = 0; x < n; x += BLOCK)
        for (y = 0; y < m; y += BLOCK)
            for (z = 0; z < k; z += BLOCK)
            {
                maxx = min(n, x + BLOCK);
                maxy = min(m, y + BLOCK);
                maxz = min(k, z + BLOCK);
                for (xx = x; xx < maxx; ++xx)
                    for (yy = y; yy < maxy; ++yy)
                        for (zz = z; zz < maxz; ++zz)
                            C[xx][yy] += A[xx][zz] * B[zz][yy];
            }
}
```

a brief example in block size = 128 :

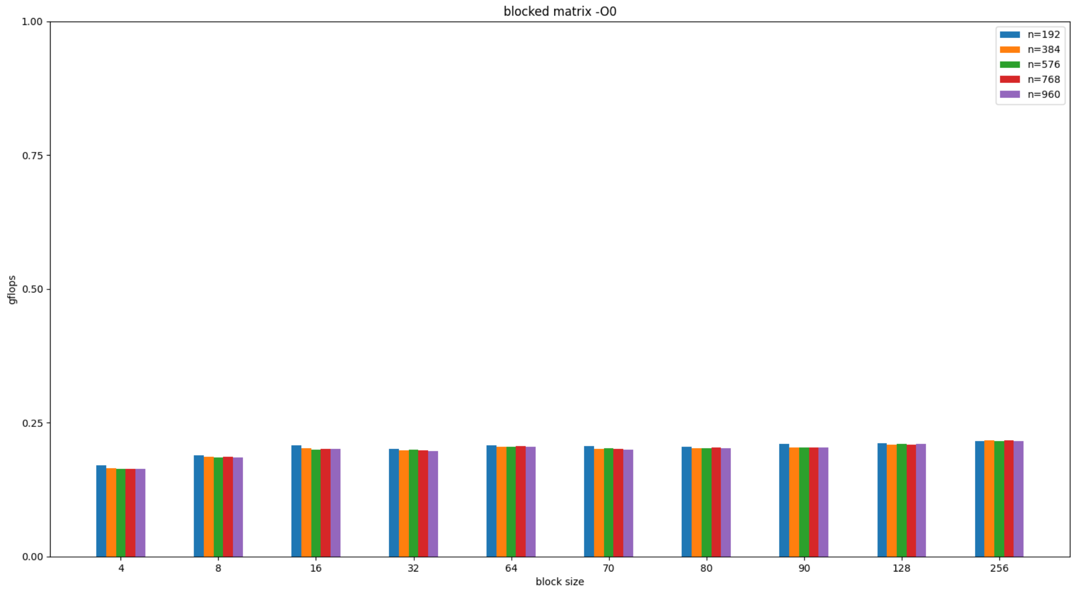
```
 Performance counter stats for './blocked-matrixB128O0V2' (10 runs):

         1,951,914      cache-misses              #    0.423 % of all cache
refs        ( +-  1.90% )
       461,827,246      cache-references
( +-  0.77% )
    52,476,574,760      instructions              #    4.03  insn per cycle
( +-  0.00% )
    13,013,724,908      cycles
( +-  0.12% )

             3.450 +- 0.212 seconds time elapsed  ( +-  6.13% )
```

can be seen the instructions is reduced.

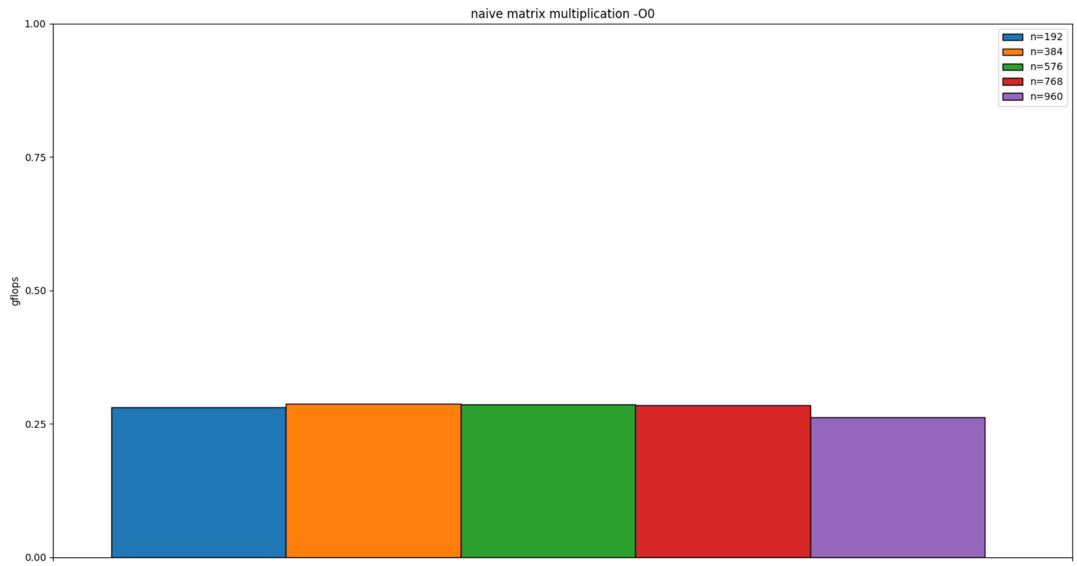
here's the GFLOPS performance in -O0 of modified version:



comparing to the unmodifying one:



and standard one:



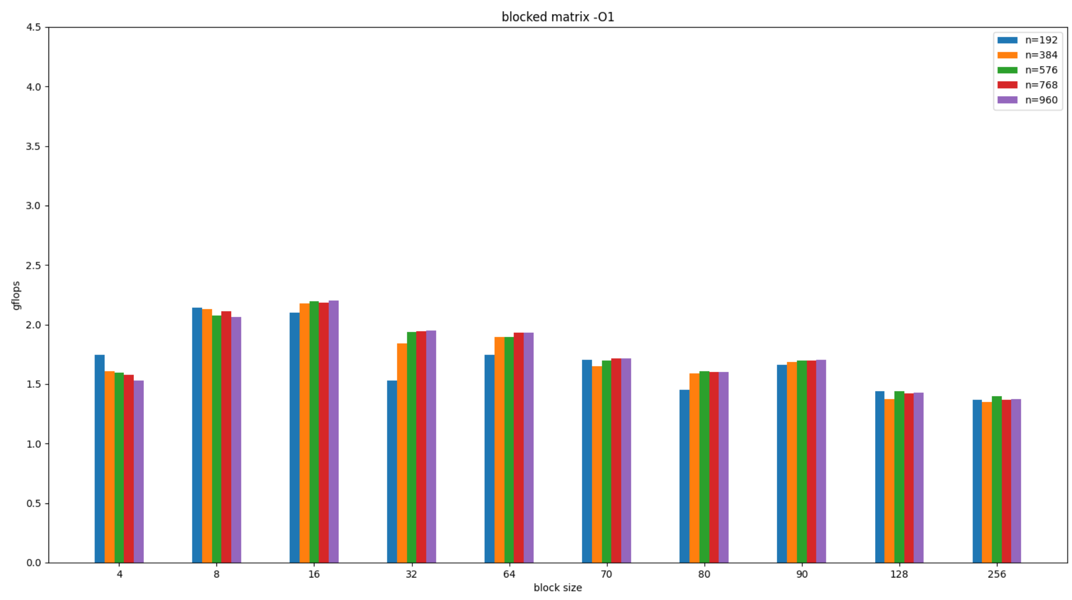
can see the performace is better than both.

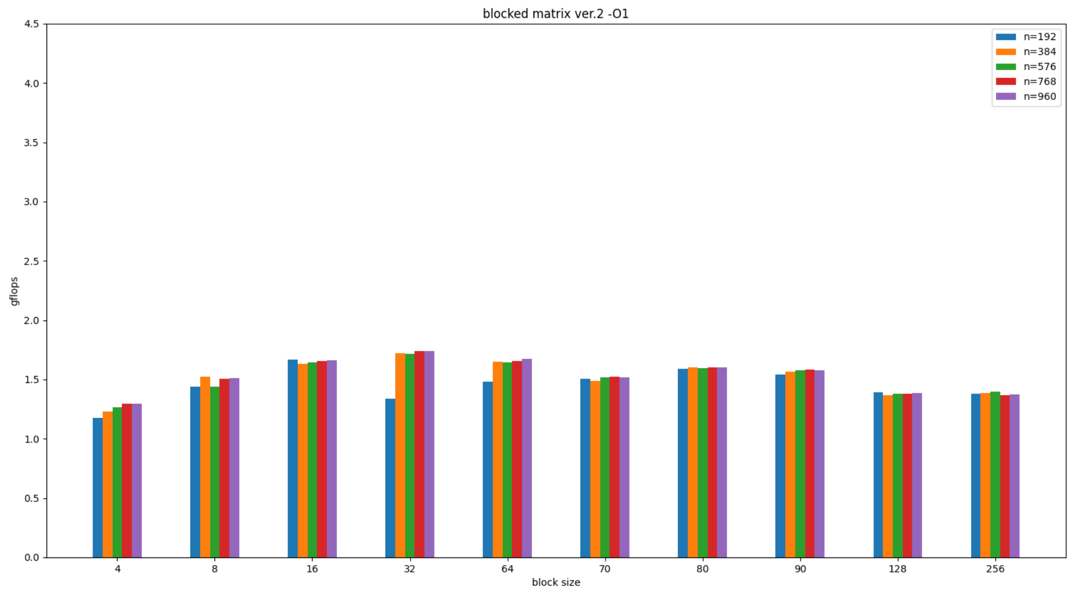
there's a more convenient way to optimize without modifying the program, by using gcc optimization flag, O1, O2, O3, Ofast, the performance can be imporved a lot.

one more thing to mention, there's a weird phenomenon that when using O1 optimization on the modified blocked version, the performance is worse than the unmodified one. but in the later optimization (O2,O3,Ofast) the performance is as good as the other one.

this phenomenon is shown below:

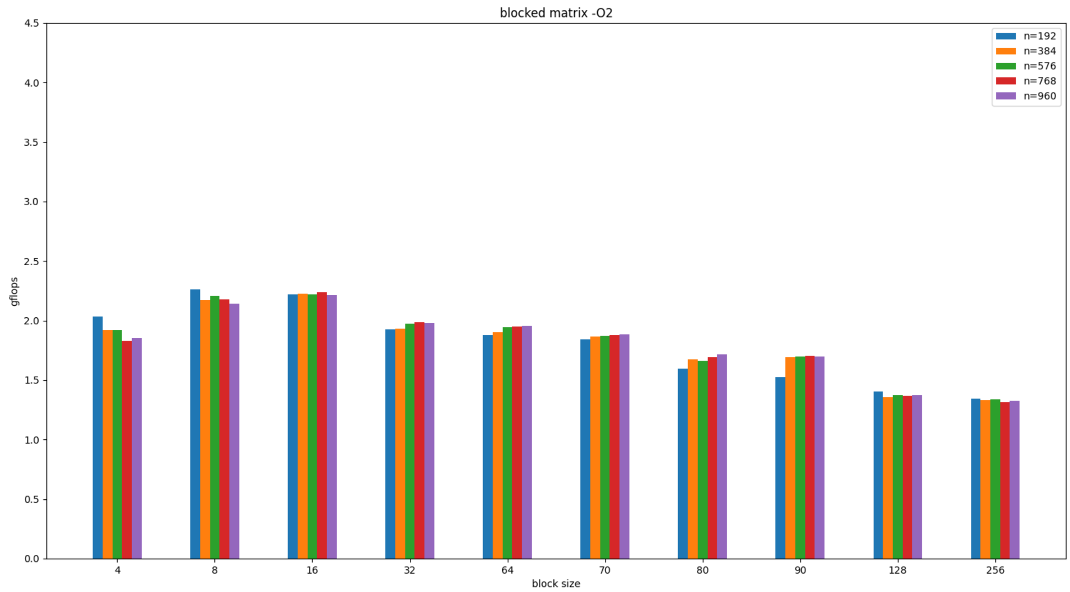
GFLOPS performace of blocked version and modified blocked version in O1:

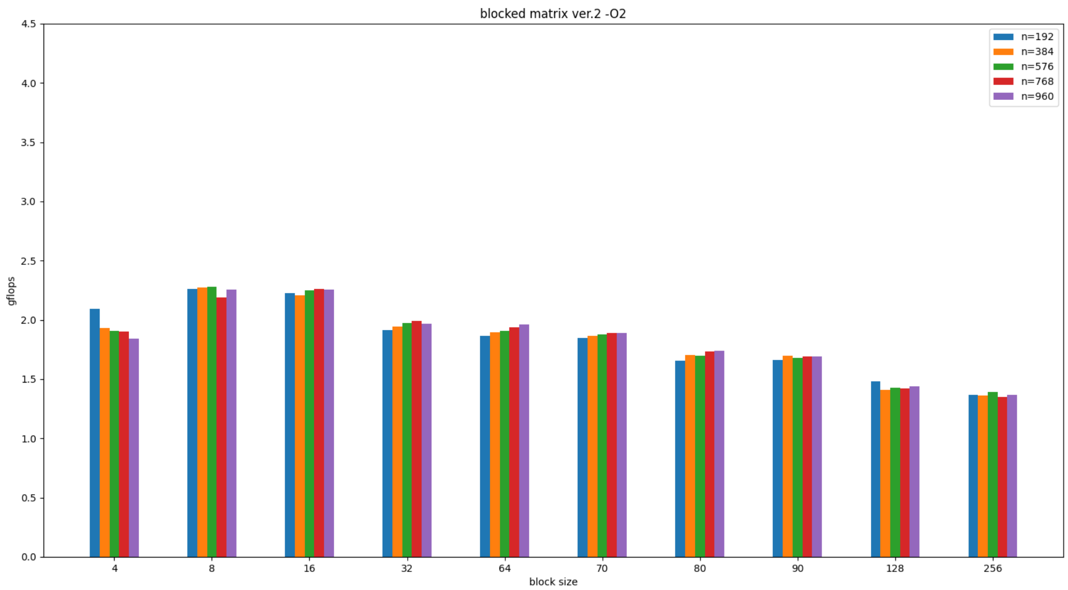




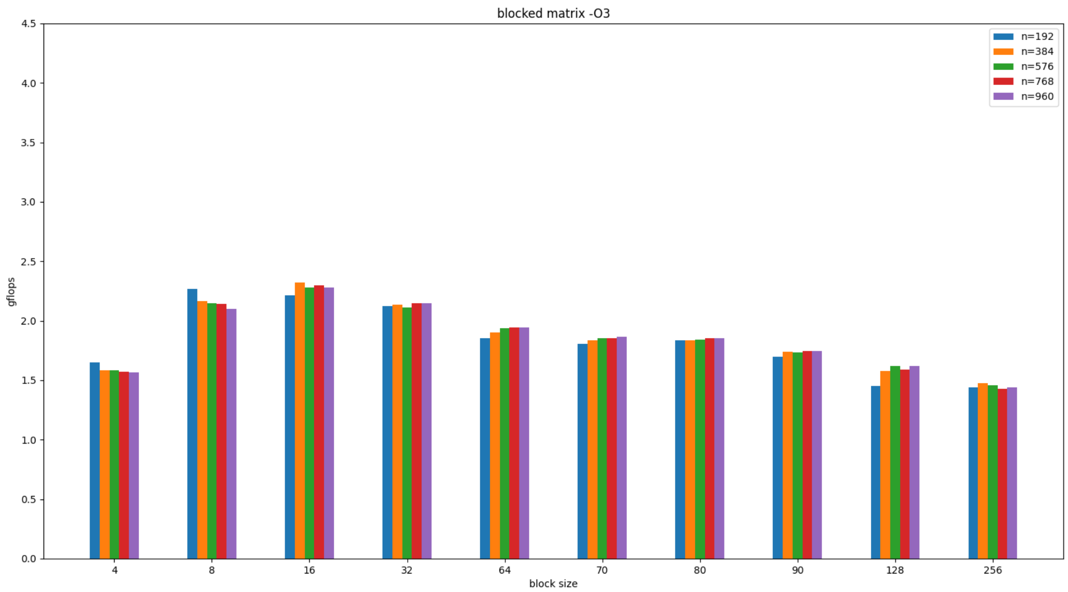
can be seen the performace is worse.

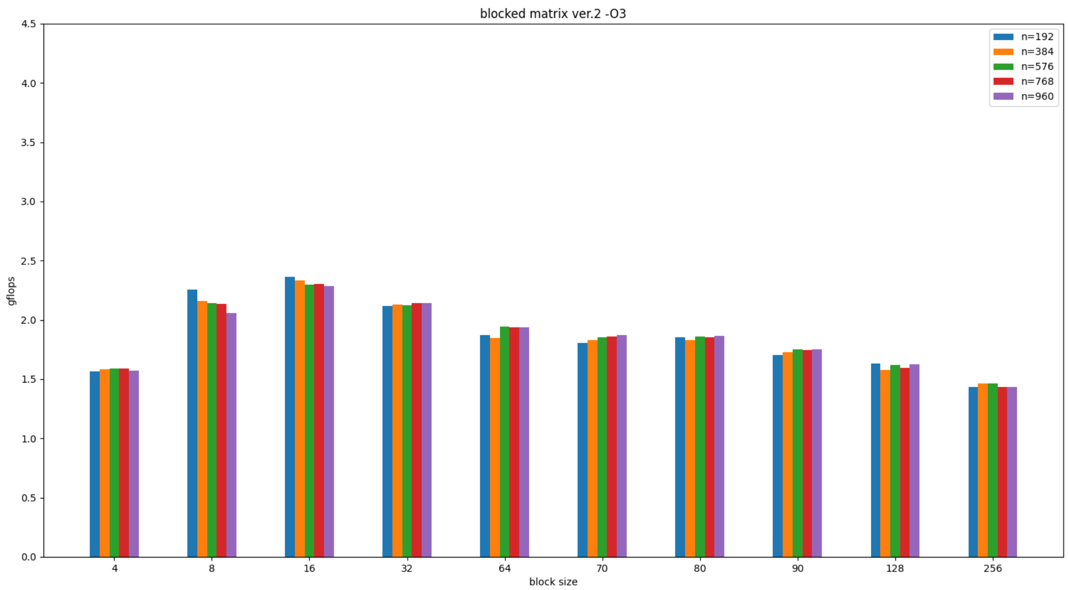
GFLOPS performace of blocked version and modified blocked version in O2:



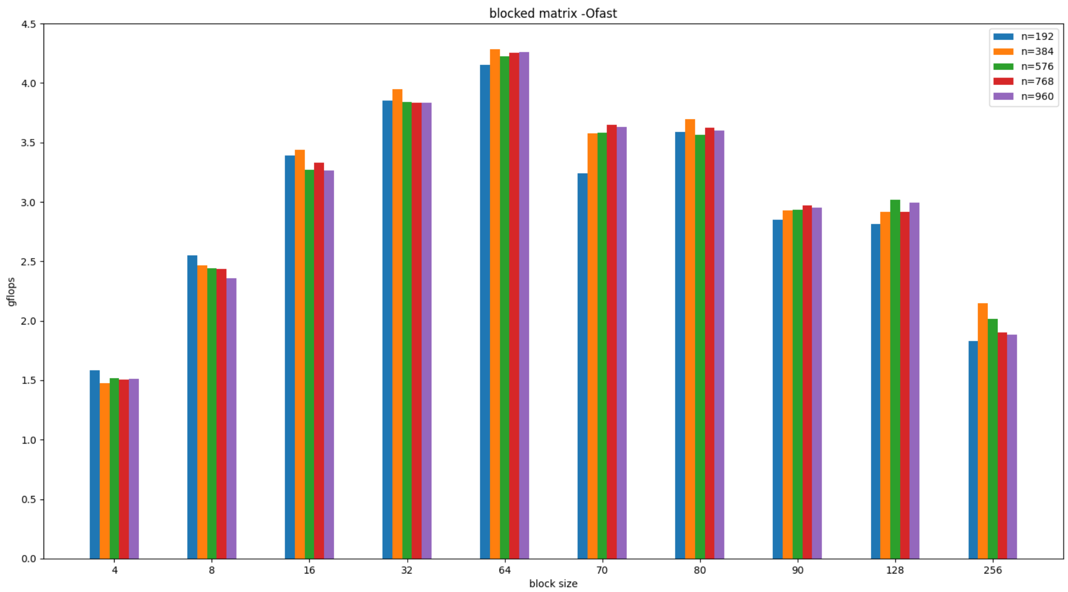


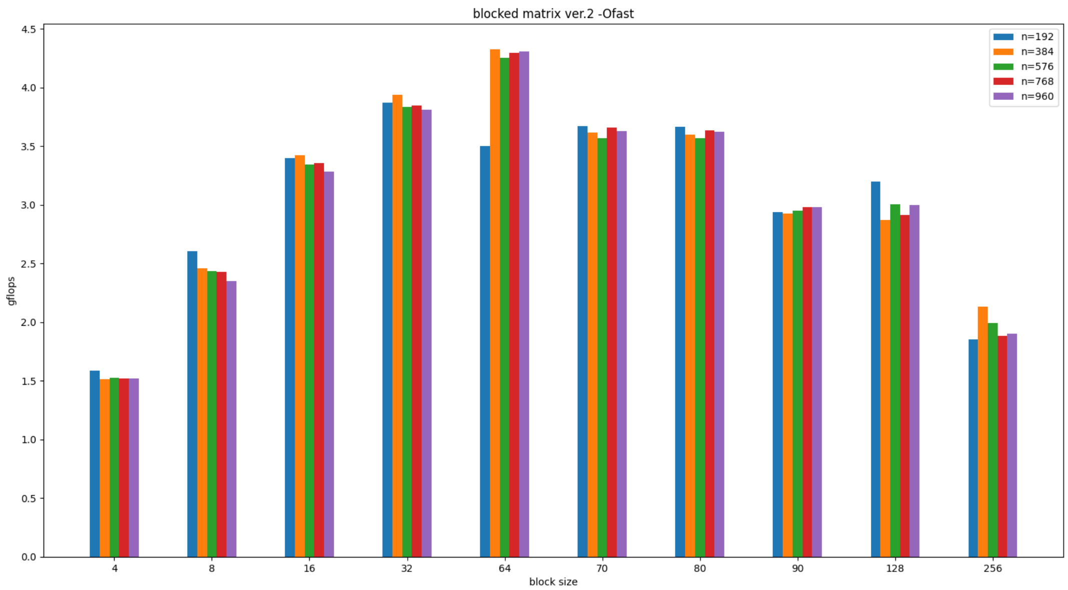
GFLOPS performace of blocked version and modified blocked version in O3:





GFLOPS performace of blocked version and modified blocked version in Ofast:





i think the reason why is that the compiler optimization in O1 has some conflicts with the manual optimization, so that the compiler can't 100% optimize the program.
but in O2 and others, the optimization is aggressive enough to optimize the whole program.

so the conclusion is that, since the additional instructions in three-extra for-loop, blocked version takes advantage **only when we do some extra optimization**.

f.
`O0`: no optimization. it's default flag if no optimization level is specified.

`O1`: optimize minimally. the compiler tries to reduce code size and execution time, but performing no optimizations that take a great deal of compilation time.

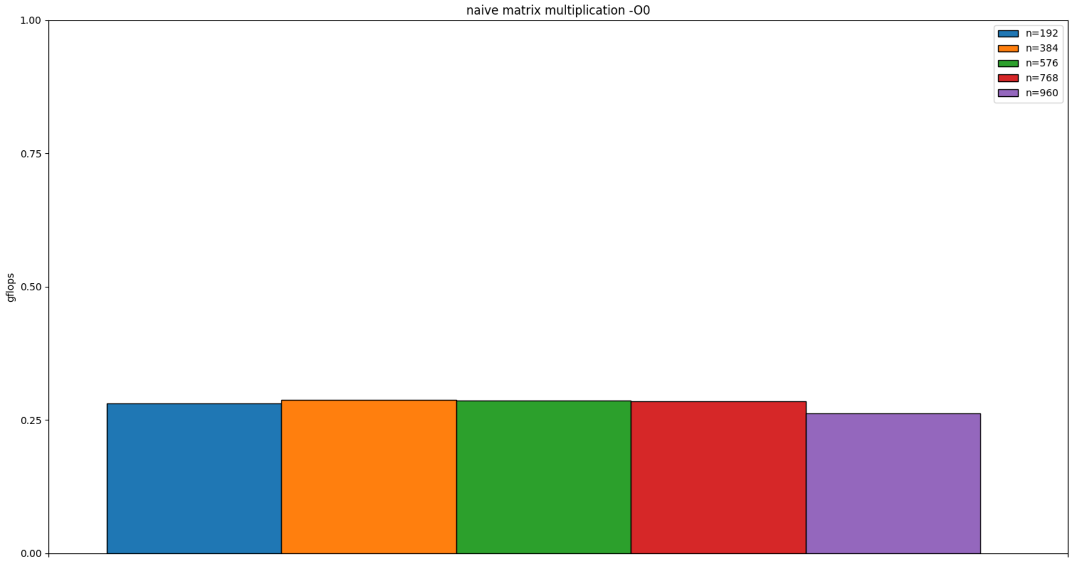
`O2`: optimize more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O1, this option increases both compilation time and the performance of the generated code.

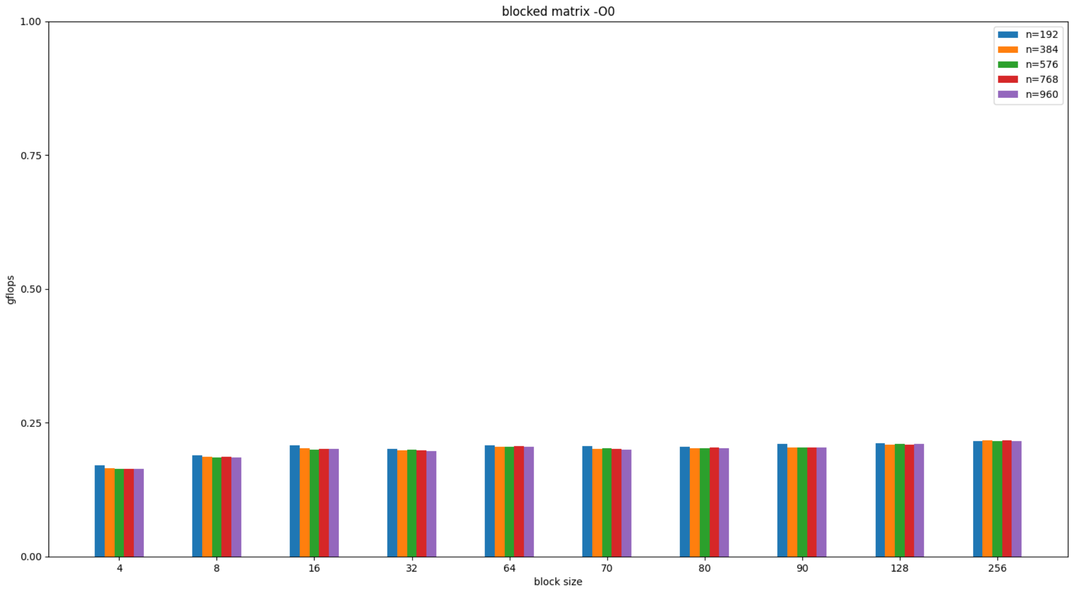
`O3`: optimize even more.

`Ofast`: optimize very aggressively that disregard strict standards compliance.

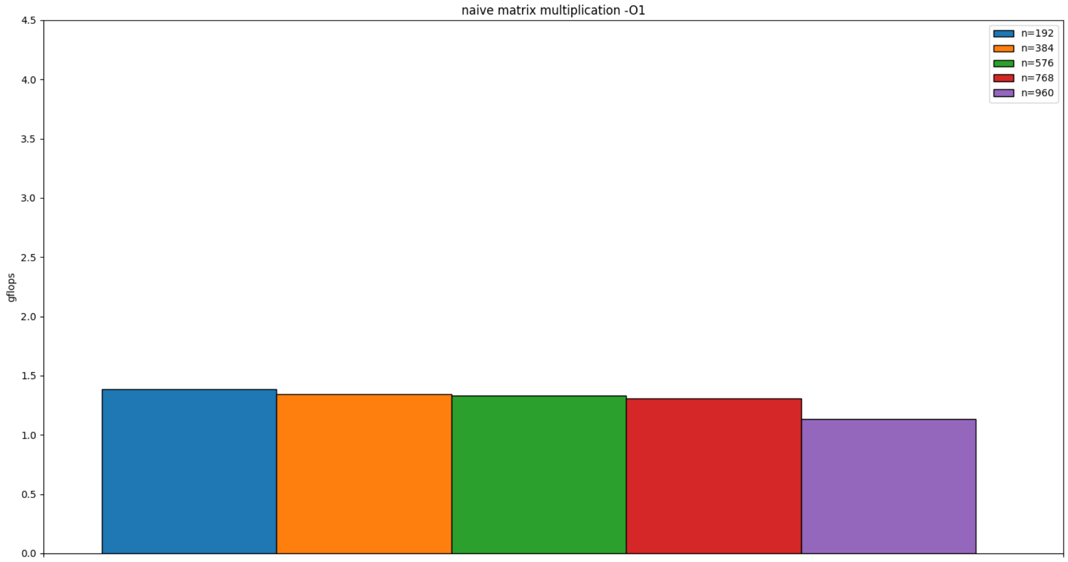
basicly, optimize more means faster execution time.
but sometimes optimization can cause unexpected result. e.g. `Ofast` will change the floating number operation order, make the result different.

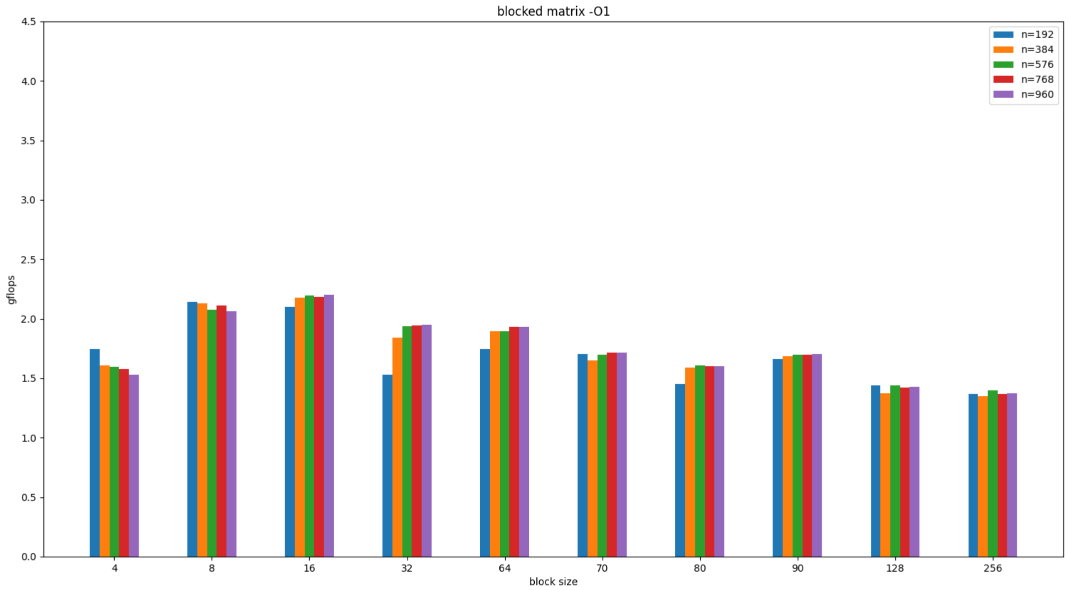
GFLOPS of standard version and blocked version in -O0:



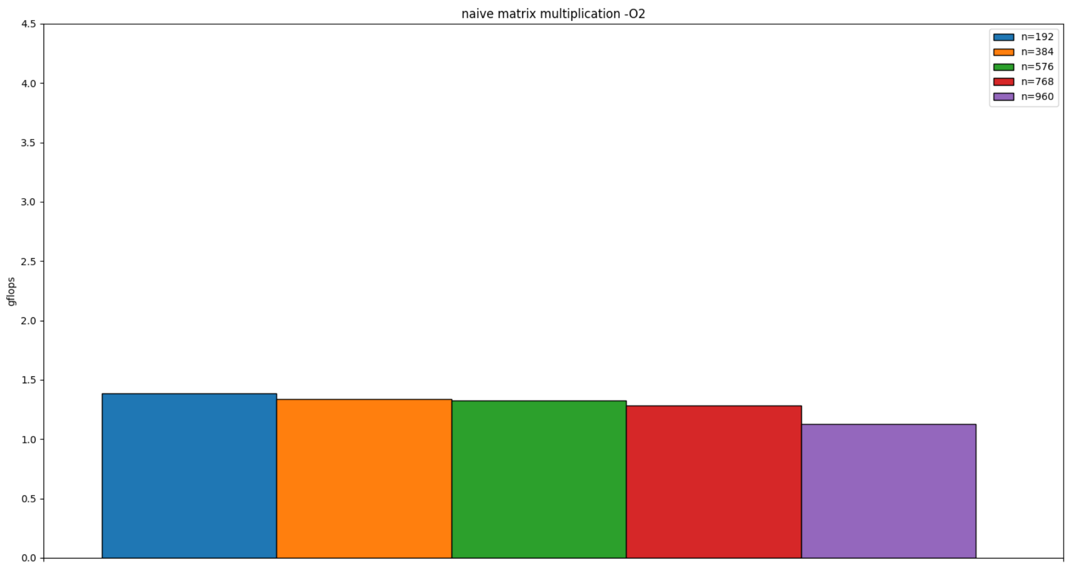


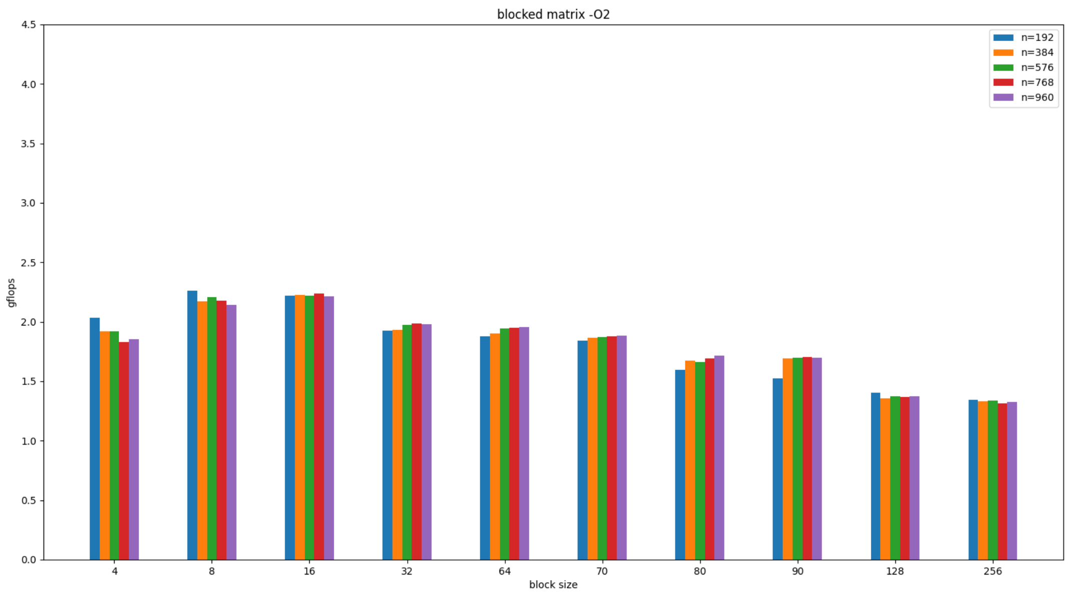
GFLOPS of standard version and blocked version in -O1:



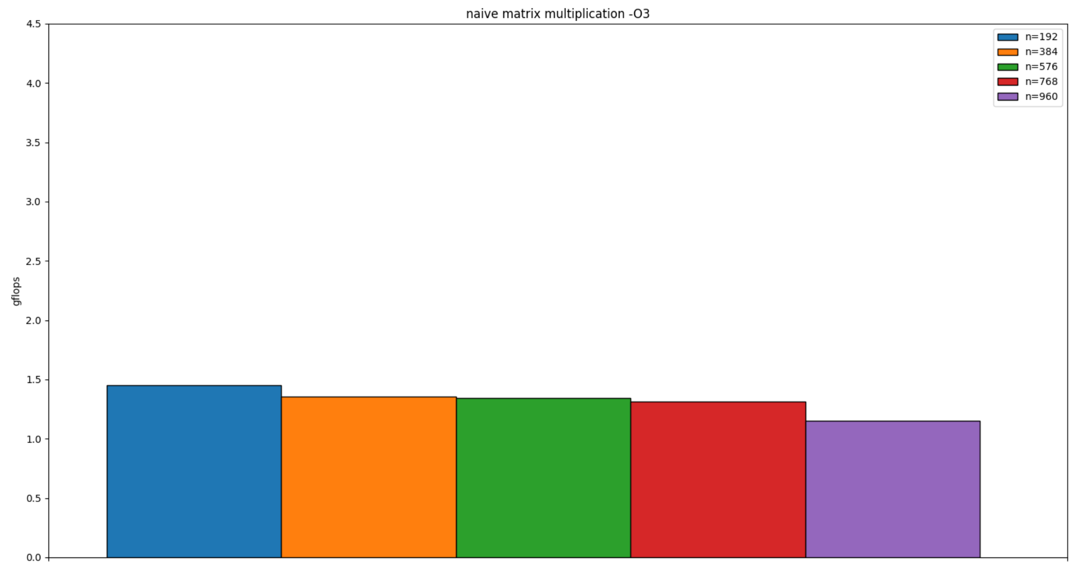


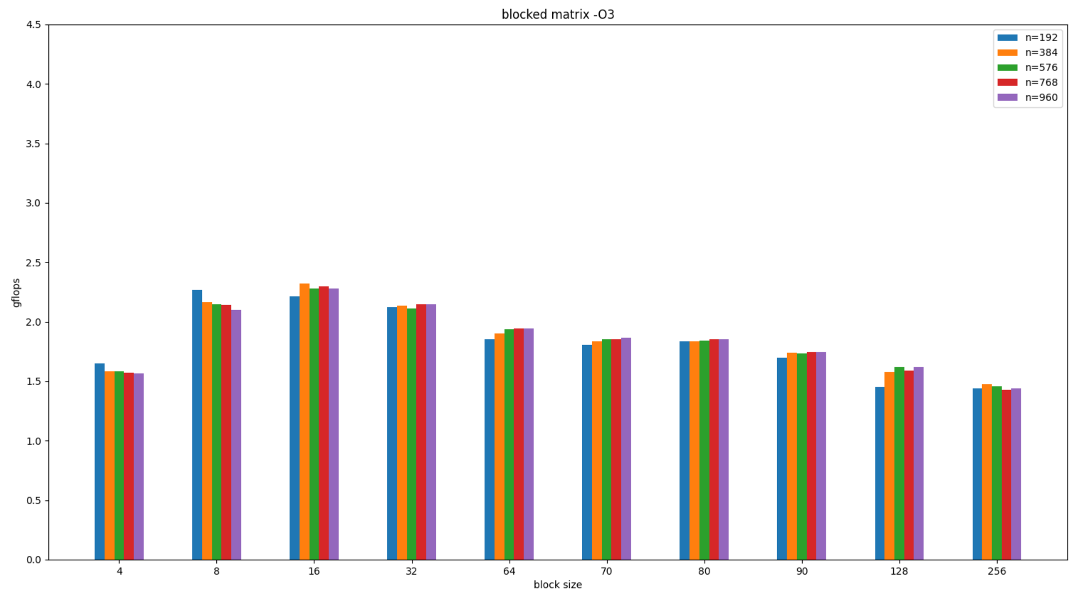
GFLOPS of standard version and blocked version in -O2:



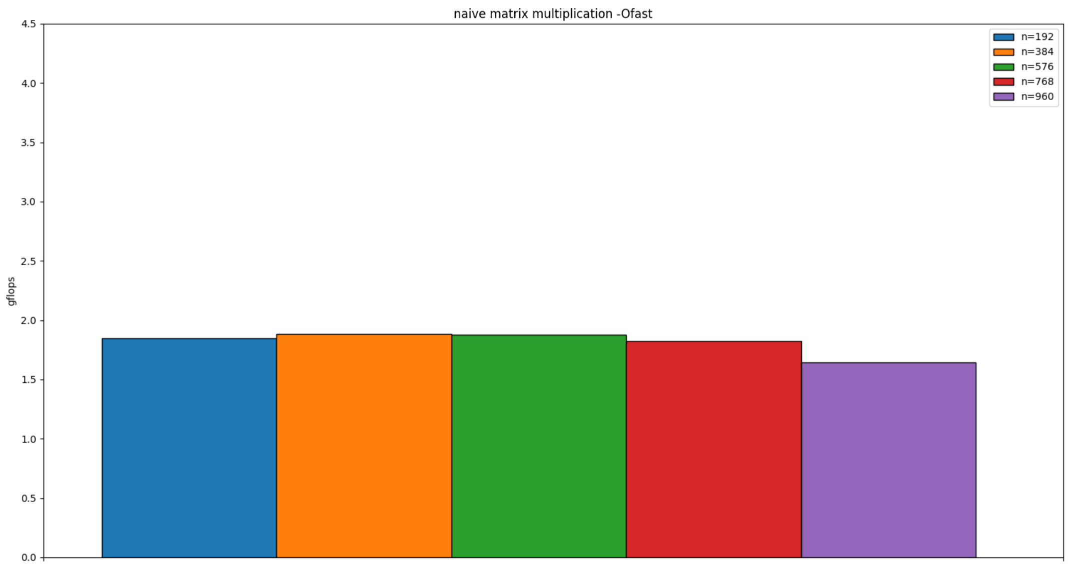


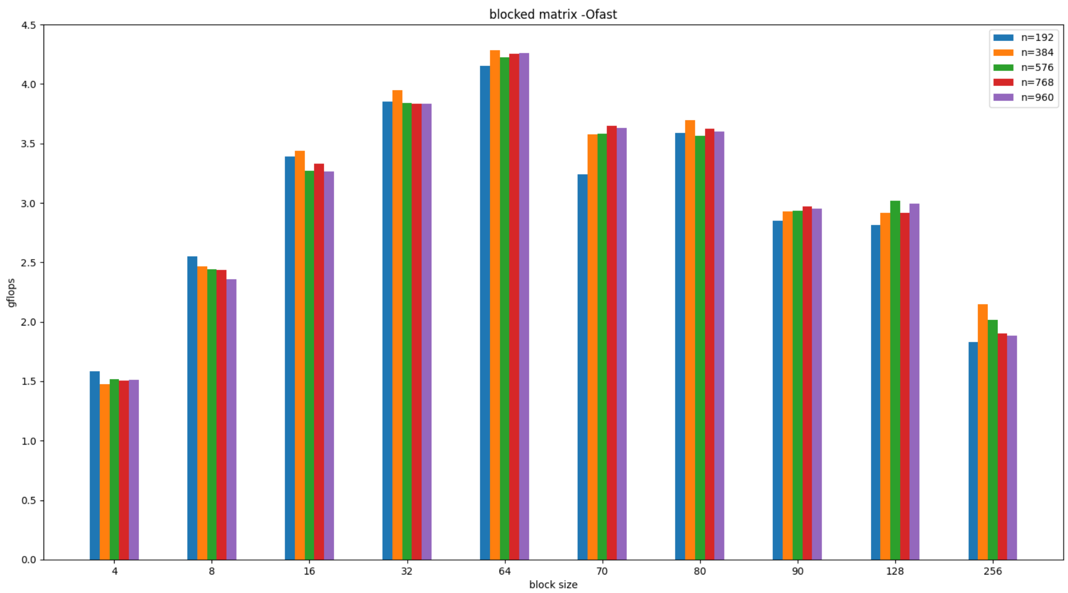
GFLOPS of standard version and blocked version in -O3:





GFLOPS of standard version and blocked version in -Ofast:





advanced questions

a.

this problem has two solutions:

1. change for-loop order:
   the second and third, the fifth and sixth loop have been exchanged.
   so in every iteration, the array `B` reads the next column instead of next row.

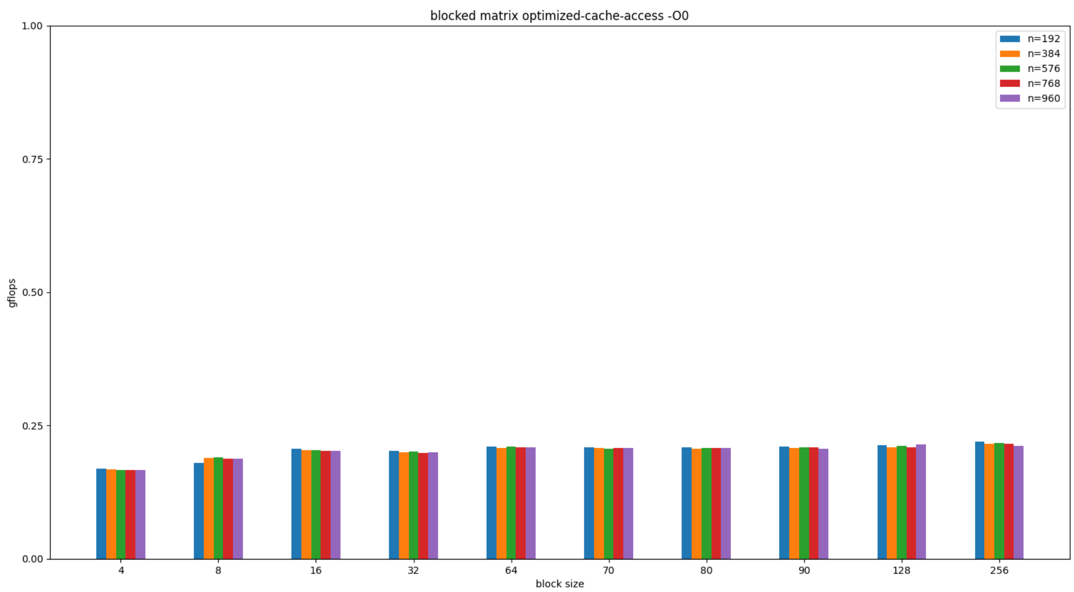
```
inline void MMUL() noexcept
{
    for (int x = 0; x < n; x += BLOCK)
        for (int z = 0; z < k; z += BLOCK)
```

```
            for (int y = 0; y < m; y += BLOCK)
            {
                for (int xx = x; xx < std::min(n, x + BLOCK); ++xx)
                    for (int zz = z; zz < std::min(k, z + BLOCK); ++zz)
                        for (int yy = y; yy < std::min(m, y + BLOCK);
++yy)
                            C[xx][yy] += A[xx][zz] * B[zz][yy];
            }
}
```

2. transpose matrix B:
   when read the data into array `B`, transposing it.
   when performing matrix multiplication, transposing it again. so that in every iteration the array `B` reads the next column instead of next row.

```
inline void MMUL() noexcept
{
    for (int x = 0; x < n; x += BLOCK)
        for (int y = 0; y < m; y += BLOCK)
            for (int z = 0; z < k; z += BLOCK)
            {
                for (int xx = x; xx < std::min(n, x + BLOCK); ++xx)
                    for (int yy = y; yy < std::min(m, y + BLOCK); ++yy)
                        for (int zz = z; zz < std::min(k, z + BLOCK);
++zz)
                            C[xx][yy] += A[xx][zz] * B[yy][zz];
            }
}
```

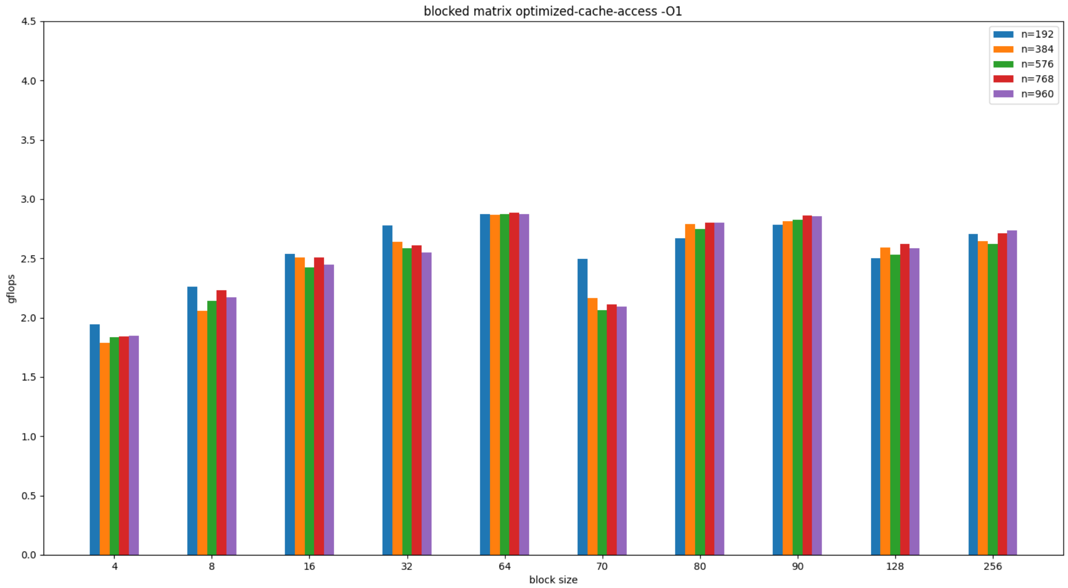
GFLOPS perfomance of optimized-cache blocked matrix in O0:



```
 Performance counter stats for './blocked-cache-matrixB12800' (10 runs):

         1,443,658      cache-misses              #    1.516 % of all cache
refs        ( +-  1.87% )
        95,256,474      cache-references
( +-  0.10% )
    74,008,679,699      instructions              #    4.09  insn per cycle
( +-  0.00% )
    18,095,313,580      cycles
( +-  0.24% )

             4.765 +- 0.286 seconds time elapsed  ( +-  5.99% )
```

it doesn't go beyond the standard version but same as blocked one, since the instruction counts is much more than standard one.

GFLOPS perfomance of optimized-cache blocked matrix in O1:

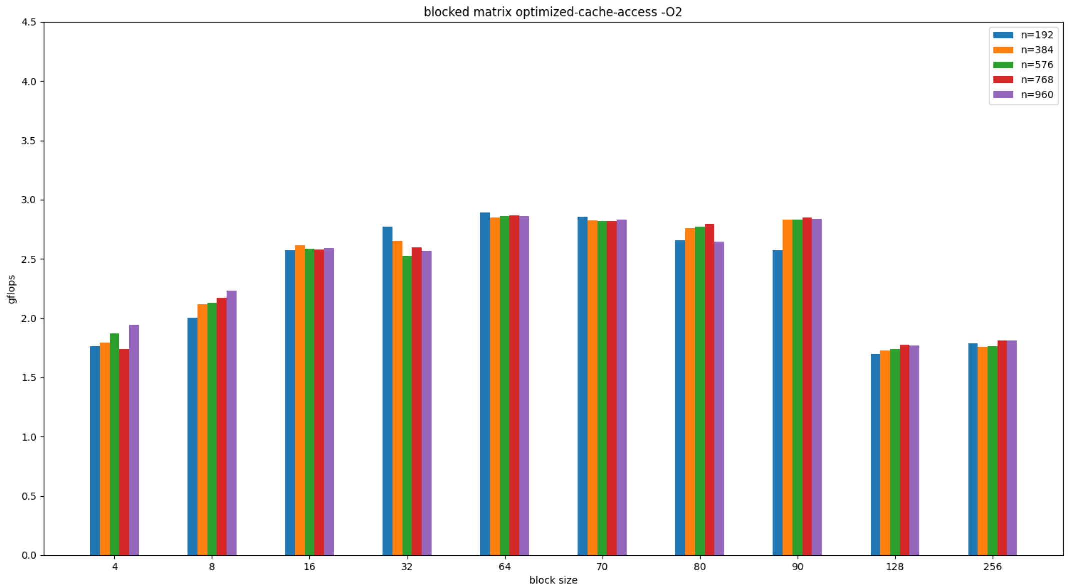



```
 Performance counter stats for './blocked-cache-matrixB12801' (10 runs):

         1,056,336      cache-misses              #    1.120 % of all cache
refs        ( +-  6.15% )
        94,308,408      cache-references
( +-  0.62% )
     9,988,970,682      instructions              #    2.85  insn per cycle
( +-  0.00% )
     3,501,391,361      cycles
( +-  0.25% )
```

```
           1.1624 +- 0.0892 seconds time elapsed  ( +-  7.68% )
```

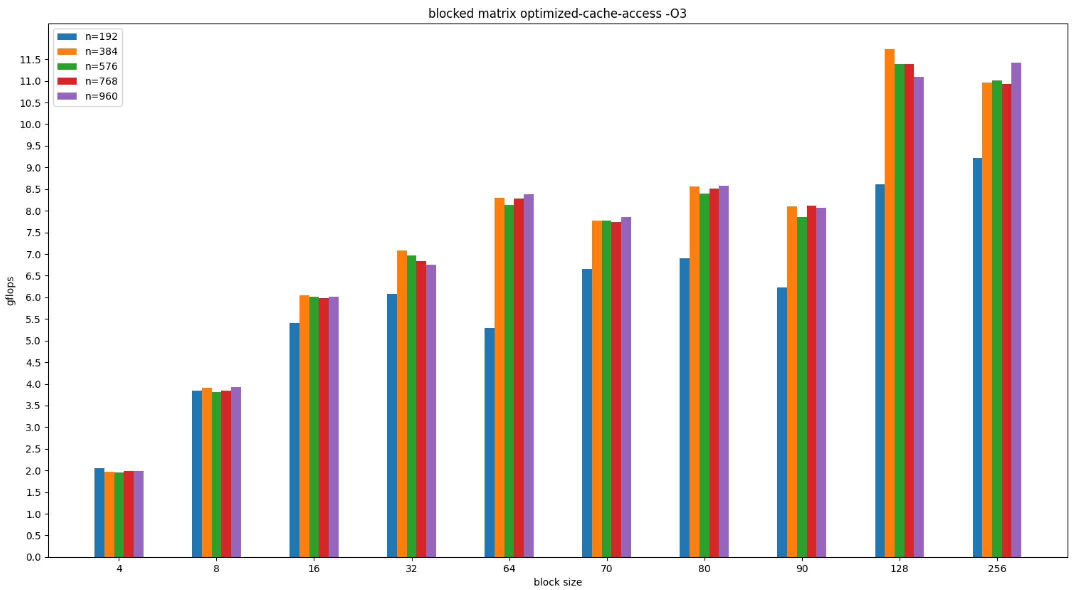
can see the performance is far beyond the blocked one and standard one.

GFLOPS perfomance of optimized-cache blocked matrix in O2:



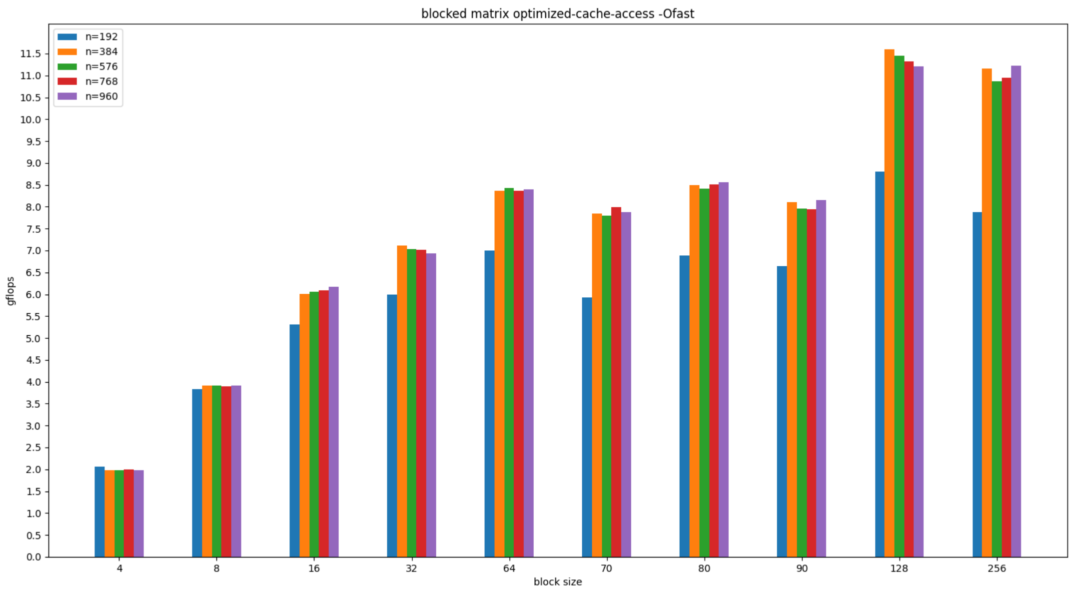

```
 Performance counter stats for './blocked-cache-matrixB128O2' (10 runs):

         1,149,807      cache-misses              #    1.229 % of all cache
refs       ( +-  6.05% )
        93,522,492      cache-references
( +-  0.55% )
     9,965,972,021      instructions              #    2.83  insn per cycle
( +-  0.00% )
     3,520,036,555      cycles
( +-  0.37% )

           1.1585 +- 0.0893 seconds time elapsed  ( +-  7.71% )
```

GFLOPS perfomance of optimized-cache blocked matrix in O3:




```
 Performance counter stats for './blocked-cache-matrixB128O3' (10 runs):

        1,173,975      cache-misses              #    0.996 % of all cache
refs       ( +-  2.30% )
      117,900,900      cache-references
( +-  3.34% )
    5,016,204,948      instructions              #    2.76  insn per cycle
( +-  0.00% )
    1,816,051,619      cycles
( +-  0.50% )

          0.73223 +- 0.00354 seconds time elapsed  ( +-  0.48% )
```

GFLOPS perfomance of optimized-cache blocked matrix in Ofast:




```
Performance counter stats for './blocked-cache-matrixB128Ofast' (10 runs):

        1,090,282      cache-misses              #    1.163 % of all cache
refs        ( +-   3.76% )
       93,785,136      cache-references
( +-   0.77% )
    5,016,269,439      instructions              #    2.64  insn per cycle
( +-   0.00% )
    1,899,711,429      cycles
( +-   0.36% )

          0.76323 +- 0.00245 seconds time elapsed  ( +-   0.32% )
```

b.

1. ver. 1:

same as blocked version, whose block size is 4, but implement in SSE.

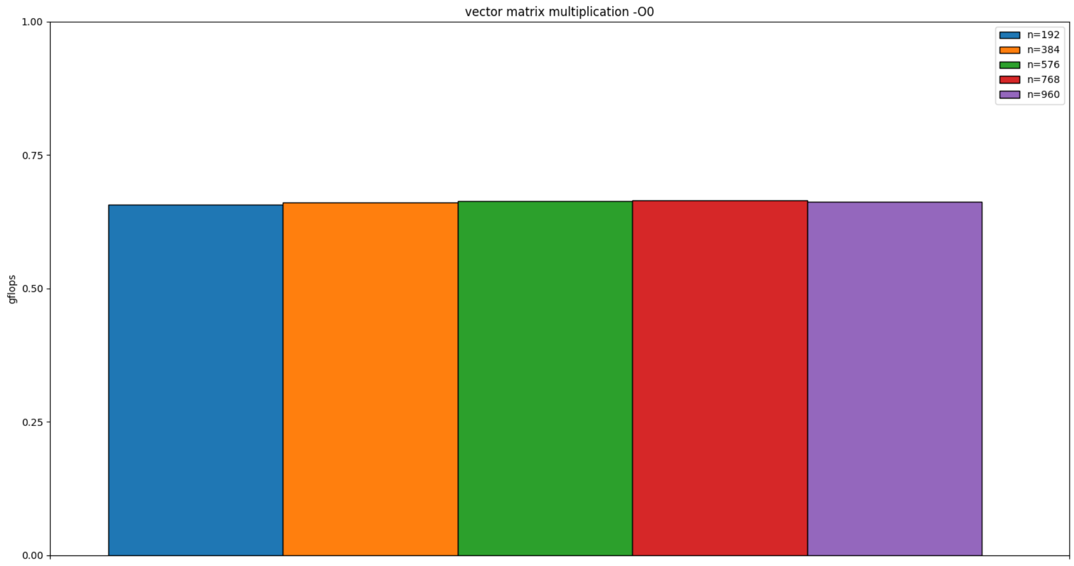
```
#include <xmmintrin.h>
#include <pmmintrin.h>
#include <tmmintrin.h>
#include <smmintrin.h>
#include <nmmintrin.h>
#include <immintrin.h>

constexpr int MAXSIZE = 960;

// SSE requires the data shall be aligned with 16
float A[MAXSIZE][MAXSIZE] __attribute__((aligned(16)));
```

```
float B[MAXSIZE][MAXSIZE] __attribute__((aligned(16)));
float C[MAXSIZE][MAXSIZE] __attribute__((aligned(16)));

int n, k, m;

inline void MMUL() noexcept
{
        for (int x = 0; x < n; x += 4)
                for (int y = 0; y < m; y += 4)
                {
                        __m128 r0 = _mm_load_ps(&C[x+0][y]);
                        __m128 r1 = _mm_load_ps(&C[x+1][y]);
                        __m128 r2 = _mm_load_ps(&C[x+2][y]);
                        __m128 r3 = _mm_load_ps(&C[x+3][y]);

                        for (int z = 0; z < k; z += 4)
                        {
                                // load A[x:x+3][k:k+3] and B[k:k+3]
[y:y+3]
                                __m128 I0 = _mm_load_ps(&A[x+0][z]);
                                __m128 I1 = _mm_load_ps(&A[x+1][z]);
                                __m128 I2 = _mm_load_ps(&A[x+2][z]);
                                __m128 I3 = _mm_load_ps(&A[x+3][z]);

                                __m128 I4 = _mm_set_ps(B[z+3][y+0], B[z+2]
[y+0], B[z+1][y+0], B[z+0][y+0]);
                                __m128 I5 = _mm_set_ps(B[z+3][y+1], B[z+2]
[y+1], B[z+1][y+1], B[z+0][y+1]);
                                __m128 I6 = _mm_set_ps(B[z+3][y+2], B[z+2]
[y+2], B[z+1][y+2], B[z+0][y+2]);
                                __m128 I7 = _mm_set_ps(B[z+3][y+3], B[z+2]
[y+3], B[z+1][y+3], B[z+0][y+3]);

                                // calculate
                                __m128 T0 = _mm_mul_ps(I0, I4);
                                __m128 T1 = _mm_mul_ps(I0, I5);
                                __m128 T2 = _mm_mul_ps(I0, I6);
                                __m128 T3 = _mm_mul_ps(I0, I7);

                                __m128 T4 = _mm_mul_ps(I1, I4);
                                __m128 T5 = _mm_mul_ps(I1, I5);
                                __m128 T6 = _mm_mul_ps(I1, I6);
                                __m128 T7 = _mm_mul_ps(I1, I7);

                                __m128 T8 = _mm_mul_ps(I2, I4);
                                __m128 T9 = _mm_mul_ps(I2, I5);
                                __m128 T10 = _mm_mul_ps(I2, I6);
                                __m128 T11 = _mm_mul_ps(I2, I7);

                                __m128 T12 = _mm_mul_ps(I3, I4);
                                __m128 T13 = _mm_mul_ps(I3, I5);
                                __m128 T14 = _mm_mul_ps(I3, I6);
```

```
__m128 T15 = _mm_mul_ps(I3, I7);

// transpose
__m128 T16 = _mm_unpacklo_ps(T0, T1);
__m128 T17 = _mm_unpacklo_ps(T2, T3);
__m128 T18 = _mm_unpackhi_ps(T0, T1);
__m128 T19 = _mm_unpackhi_ps(T2, T3);

__m128 T20 = _mm_unpacklo_ps(T16, T17);
__m128 T21 = _mm_unpackhi_ps(T16, T17);
__m128 T22 = _mm_unpacklo_ps(T18, T19);
__m128 T23 = _mm_unpackhi_ps(T18, T19);

T20 = _mm_add_ps(T20, T21);
T20 = _mm_add_ps(T20, T22);
T20 = _mm_add_ps(T20, T23);

T20 = _mm_shuffle_ps(T20, T20, 0xD8);
r0 = _mm_add_ps(T20, r0);

T16 = _mm_unpacklo_ps(T4, T5);
T17 = _mm_unpacklo_ps(T6, T7);
T18 = _mm_unpackhi_ps(T4, T5);
T19 = _mm_unpackhi_ps(T6, T7);

T20 = _mm_unpacklo_ps(T16, T17);
T21 = _mm_unpackhi_ps(T16, T17);
T22 = _mm_unpacklo_ps(T18, T19);
T23 = _mm_unpackhi_ps(T18, T19);

T20 = _mm_add_ps(T20, T21);
T20 = _mm_add_ps(T20, T22);
T20 = _mm_add_ps(T20, T23);

T20 = _mm_shuffle_ps(T20, T20, 0xD8);
r1 = _mm_add_ps(T20, r1);

T16 = _mm_unpacklo_ps(T8, T9);
T17 = _mm_unpacklo_ps(T10, T11);
T18 = _mm_unpackhi_ps(T8, T9);
T19 = _mm_unpackhi_ps(T10, T11);

T20 = _mm_unpacklo_ps(T16, T17);
T21 = _mm_unpackhi_ps(T16, T17);
T22 = _mm_unpacklo_ps(T18, T19);
T23 = _mm_unpackhi_ps(T18, T19);

T20 = _mm_add_ps(T20, T21);
T20 = _mm_add_ps(T20, T22);
T20 = _mm_add_ps(T20, T23);

T20 = _mm_shuffle_ps(T20, T20, 0xD8);
```

```
                                    r2 = _mm_add_ps(T20, r2);

                                    T16 = _mm_unpacklo_ps(T12, T13);
                                    T17 = _mm_unpacklo_ps(T14, T15);
                                    T18 = _mm_unpackhi_ps(T12, T13);
                                    T19 = _mm_unpackhi_ps(T14, T15);

                                    T20 = _mm_unpacklo_ps(T16, T17);
                                    T21 = _mm_unpackhi_ps(T16, T17);
                                    T22 = _mm_unpacklo_ps(T18, T19);
                                    T23 = _mm_unpackhi_ps(T18, T19);

                                    T20 = _mm_add_ps(T20, T21);
                                    T20 = _mm_add_ps(T20, T22);
                                    T20 = _mm_add_ps(T20, T23);

                                    T20 = _mm_shuffle_ps(T20, T20, 0xD8);
                                    r3 = _mm_add_ps(T20, r3);
                            }
                            _mm_store_ps(&C[x+0][y], r0);
                            _mm_store_ps(&C[x+1][y], r1);
                            _mm_store_ps(&C[x+2][y], r2);
                            _mm_store_ps(&C[x+3][y], r3);
                    }
    }
```

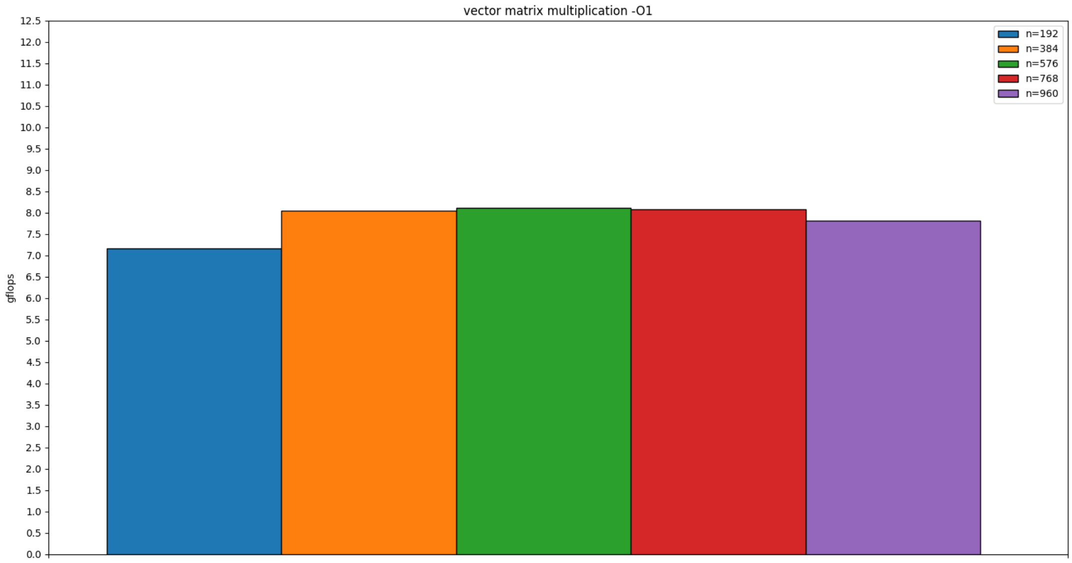
GFLOPS of vector matrix in O0:




```
Performance counter stats for './vector-matrixO0' (10 runs):

        5,136,434      cache-misses              #    4.227 % of all cache
refs      ( +- 24.77% )
```

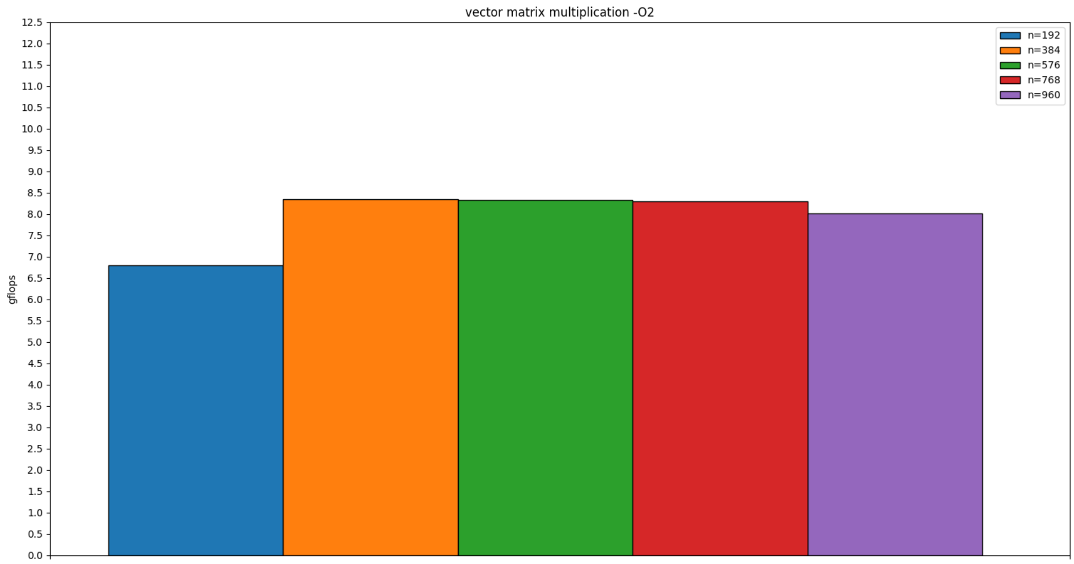
```
       121,505,626      cache-references
( +-  3.66% )
    15,415,859,793      instructions              #    2.07  insn per cycle
( +-  0.00% )
     7,445,628,148      cycles
( +-  1.21% )

            2.140 +- 0.197 seconds time elapsed  ( +-  9.19% )
```

GFLOPS of vector matrix in O1:



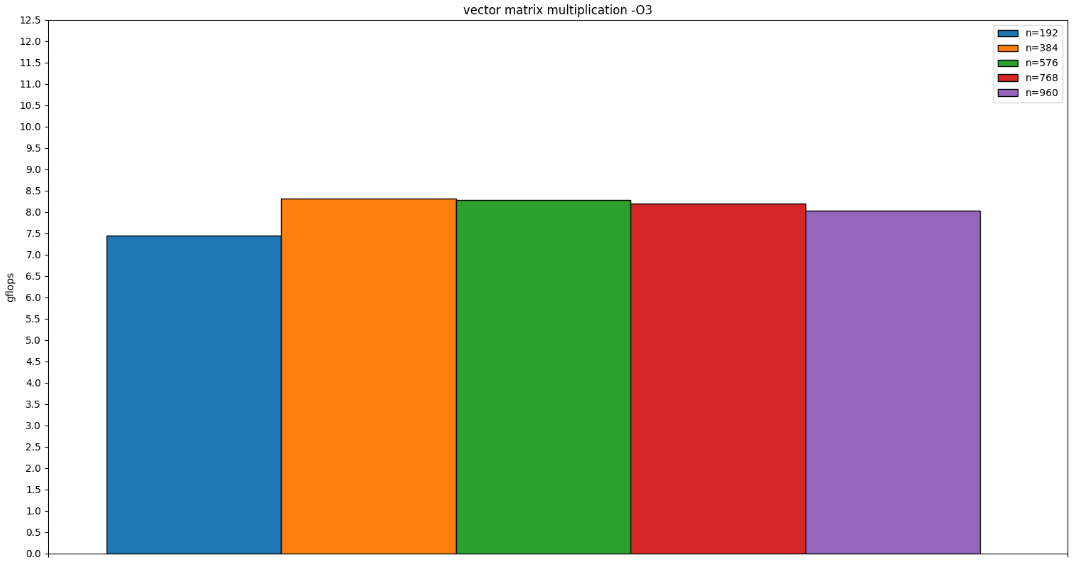

```
 Performance counter stats for './vector-matrix01' (10 runs):

         5,495,734      cache-misses              #    4.884 % of all cache
refs       ( +- 36.82% )
       112,515,831      cache-references
( +-  6.68% )
     5,739,731,344      instructions              #    2.55  insn per cycle
( +-  0.00% )
     2,255,235,243      cycles
( +-  2.76% )

           0.8472 +- 0.0507 seconds time elapsed  ( +-  5.98% )
```

GFLOPS of vector matrix in O2:



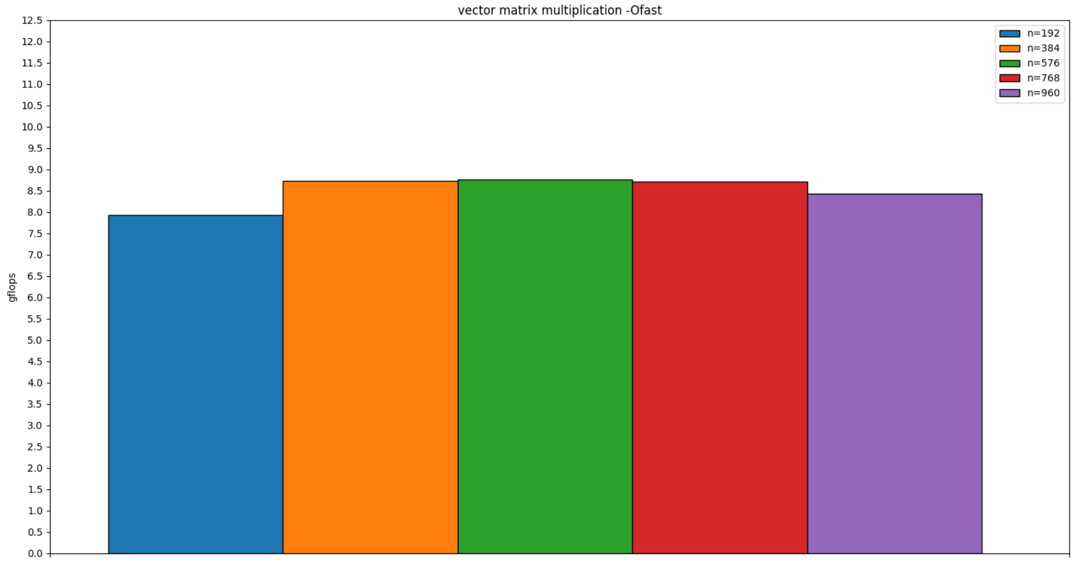

```
 Performance counter stats for './vector-matrixO2' (10 runs):

         5,745,843      cache-misses              #    5.167 % of all cache refs      ( +- 35.16% )
       111,206,271      cache-references                                              ( +-  6.36% )
     5,780,430,038      instructions              #    2.59  insn per cycle           ( +-  0.00% )
     2,229,083,887      cycles                                                        ( +-  3.09% )

            0.8188 +- 0.0571 seconds time elapsed  ( +-  6.98% )
```

GFLOPS of vector matrix in O3:




```
 Performance counter stats for './vector-matrixO3' (10 runs):

         5,911,455      cache-misses              #    5.311 % of all cache
refs       ( +- 33.20% )
       111,309,399      cache-references
( +-  6.33% )
     5,781,305,434      instructions              #    2.55  insn per cycle
( +-  0.00% )
     2,271,456,706      cycles
( +-  3.65% )

            0.8275 +- 0.0635 seconds time elapsed  ( +-  7.67% )
```

GFLOPS of vector matrix in Ofast:




```
 Performance counter stats for './vector-matrixOfast' (10 runs):

         5,878,528      cache-misses              #    5.279 % of all cache
refs       ( +- 31.02% )
       111,364,292      cache-references
( +-  5.88% )
     5,739,807,647      instructions              #    2.52  insn per cycle
( +-  0.00% )
     2,274,499,690      cycles
( +-  3.92% )

            0.8171 +- 0.0688 seconds time elapsed  ( +-  8.41% )
```

there're lots of code for reordering the data. hence the performance is as well as expected.

2. ver. 2:

perform like this.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots \\ a_{2,1} & a_{2,2} & \\ \dots & & \end{bmatrix}, B = \begin{bmatrix} b_{1,1} & b_{1,2} & \dots \\ b_{2,1} & b_{2,2} & \\ \dots & & \end{bmatrix}$$

$$AB = \begin{bmatrix} a_{1,1} \begin{bmatrix} b_{1,1} & b_{1,2} & \dots \end{bmatrix} + a_{1,2} \begin{bmatrix} b_{2,1} & b_{2,2} & \dots \end{bmatrix} + \dots \\ a_{2,1} \begin{bmatrix} b_{1,1} & b_{1,2} & \dots \end{bmatrix} + a_{2,2} \begin{bmatrix} b_{2,1} & b_{2,2} & \dots \end{bmatrix} + \dots \\ \dots \end{bmatrix}$$

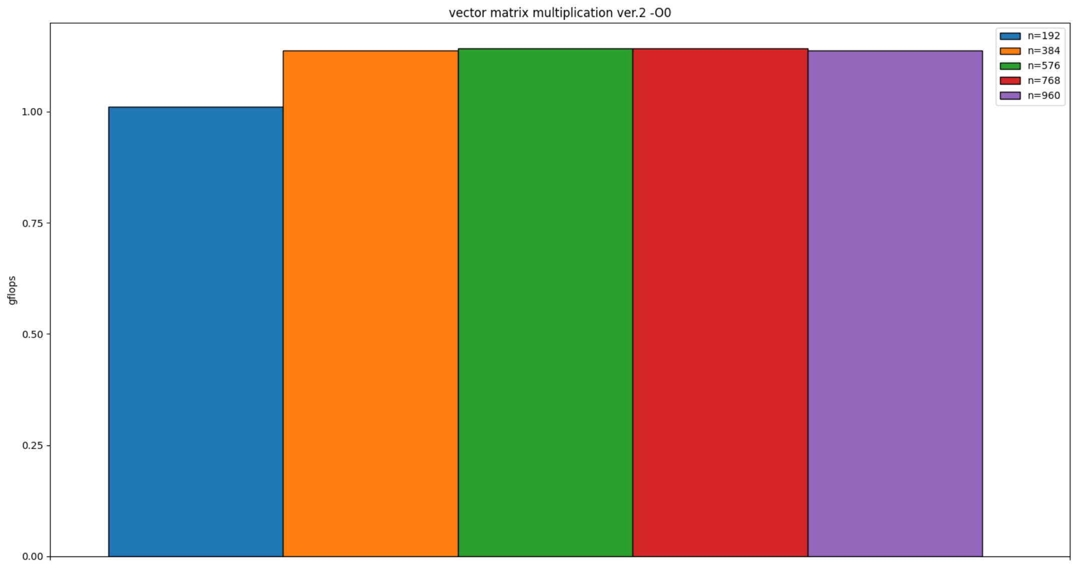
```
#include <xmmintrin.h>
#include <pmmintrin.h>
```

```
#include <tmmintrin.h>
#include <smmintrin.h>
#include <nmmintrin.h>
#include <immintrin.h>

constexpr int MAXSIZE = 960;

float A[MAXSIZE][MAXSIZE] __attribute__((aligned(16)));
float B[MAXSIZE][MAXSIZE] __attribute__((aligned(16)));
float C[MAXSIZE][MAXSIZE] __attribute__((aligned(16)));

int n, k, m;

inline void MMUL() noexcept
{
    for (int x = 0; x < n; x += 4)
                for (int y = 0; y < m; y += 4)
                {
                        __m128 xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6,
xmm7, xmm8;

                        __m128 xmm9  = _mm_setzero_ps();
                        __m128 xmm10 = _mm_setzero_ps();
                        __m128 xmm11 = _mm_setzero_ps();
                        __m128 xmm12 = _mm_setzero_ps();

                        for (int z = 0; z < k; z += 4)
                        {
                                xmm0 = _mm_load_ps(&B[z+0][y]);
                                xmm1 = _mm_load_ps(&B[z+1][y]);
                                xmm2 = _mm_load_ps(&B[z+2][y]);
                                xmm3 = _mm_load_ps(&B[z+3][y]);

                                // load A[x][z:z+3]
                                xmm4 = _mm_load_ps(&A[x+0][z]);
                                xmm5 = _mm_shuffle_ps(xmm4, xmm4, 0x00);
                                xmm6 = _mm_shuffle_ps(xmm4, xmm4, 0x55);
                                xmm7 = _mm_shuffle_ps(xmm4, xmm4, 0xAA);
                                xmm8 = _mm_shuffle_ps(xmm4, xmm4, 0xFF);

                                xmm5 = _mm_mul_ps(xmm5, xmm0);
                                xmm6 = _mm_mul_ps(xmm6, xmm1);
                                xmm5 = _mm_add_ps(xmm5, xmm6);

                                xmm7 = _mm_mul_ps(xmm7, xmm2);
                                xmm8 = _mm_mul_ps(xmm8, xmm3);
                                xmm7 = _mm_add_ps(xmm7, xmm8);

                                xmm5 = _mm_add_ps(xmm5, xmm7);
                                xmm9 = _mm_add_ps(xmm9, xmm5);

                                // load A[x+1][z:z+3]
```

```
xmm4 = _mm_load_ps(&A[x+1][z]);
xmm5 = _mm_shuffle_ps(xmm4, xmm4, 0x00);
xmm6 = _mm_shuffle_ps(xmm4, xmm4, 0x55);
xmm7 = _mm_shuffle_ps(xmm4, xmm4, 0xAA);
xmm8 = _mm_shuffle_ps(xmm4, xmm4, 0xFF);

xmm5 = _mm_mul_ps(xmm5, xmm0);
xmm6 = _mm_mul_ps(xmm6, xmm1);
xmm5 = _mm_add_ps(xmm5, xmm6);

xmm7 = _mm_mul_ps(xmm7, xmm2);
xmm8 = _mm_mul_ps(xmm8, xmm3);
xmm7 = _mm_add_ps(xmm7, xmm8);

xmm5 = _mm_add_ps(xmm5, xmm7);
xmm10 = _mm_add_ps(xmm10, xmm5);

// load A[x+2][z:z+3]
xmm4 = _mm_load_ps(&A[x+2][z]);
xmm5 = _mm_shuffle_ps(xmm4, xmm4, 0x00);
xmm6 = _mm_shuffle_ps(xmm4, xmm4, 0x55);
xmm7 = _mm_shuffle_ps(xmm4, xmm4, 0xAA);
xmm8 = _mm_shuffle_ps(xmm4, xmm4, 0xFF);

xmm5 = _mm_mul_ps(xmm5, xmm0);
xmm6 = _mm_mul_ps(xmm6, xmm1);
xmm5 = _mm_add_ps(xmm5, xmm6);

xmm7 = _mm_mul_ps(xmm7, xmm2);
xmm8 = _mm_mul_ps(xmm8, xmm3);
xmm7 = _mm_add_ps(xmm7, xmm8);

xmm5 = _mm_add_ps(xmm5, xmm7);
xmm11 = _mm_add_ps(xmm11, xmm5);

// load A[x+3][z:z+3]
xmm4 = _mm_load_ps(&A[x+3][z]);
xmm5 = _mm_shuffle_ps(xmm4, xmm4, 0x00);
xmm6 = _mm_shuffle_ps(xmm4, xmm4, 0x55);
xmm7 = _mm_shuffle_ps(xmm4, xmm4, 0xAA);
xmm8 = _mm_shuffle_ps(xmm4, xmm4, 0xFF);

xmm5 = _mm_mul_ps(xmm5, xmm0);
xmm6 = _mm_mul_ps(xmm6, xmm1);
xmm5 = _mm_add_ps(xmm5, xmm6);

xmm7 = _mm_mul_ps(xmm7, xmm2);
xmm8 = _mm_mul_ps(xmm8, xmm3);
xmm7 = _mm_add_ps(xmm7, xmm8);

xmm5 = _mm_add_ps(xmm5, xmm7);
xmm12 = _mm_add_ps(xmm12, xmm5);
```

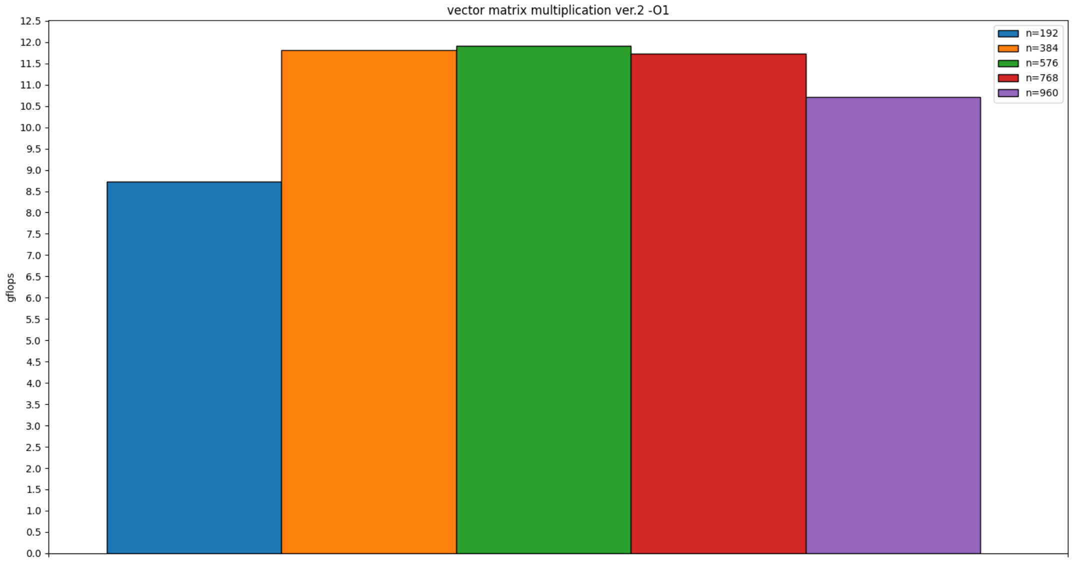
```
                }
                _mm_store_ps(&C[x+0][y], xmm9);
                _mm_store_ps(&C[x+1][y], xmm10);
                _mm_store_ps(&C[x+2][y], xmm11);
                _mm_store_ps(&C[x+3][y], xmm12);
            }
}
```

GFLOPS of vector matrix ver.2 in O0:



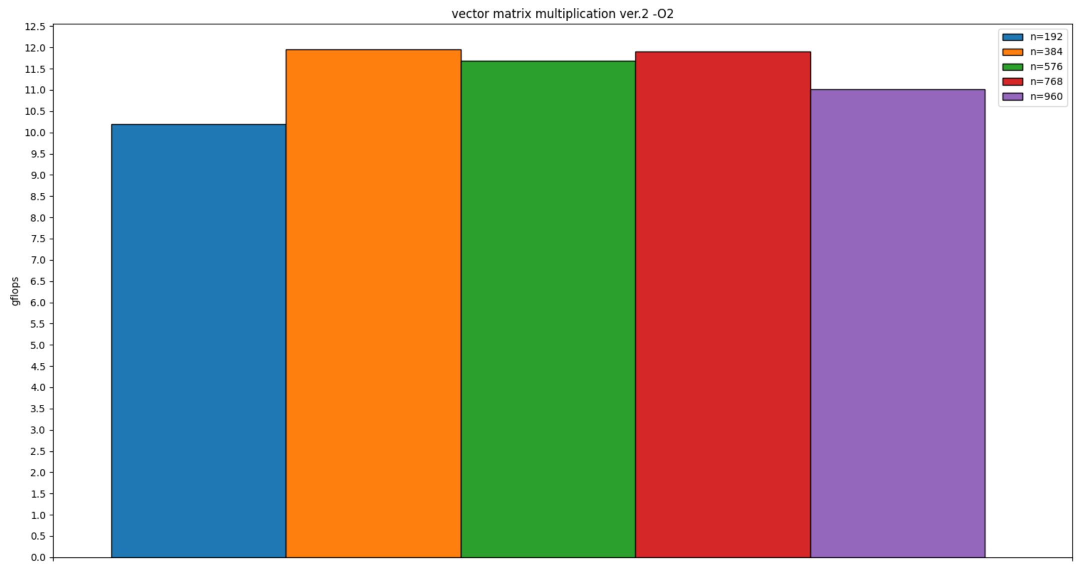

```
 Performance counter stats for './vector-matrixO0V2' (10 runs):

         4,113,644      cache-misses              #    3.435 % of all cache
refs        ( +- 39.15% )
       119,768,329      cache-references
( +-  4.72% )
     9,647,414,398      instructions              #    1.96  insn per cycle
( +-  0.02% )
     4,929,116,914      cycles
( +-  2.89% )

             1.557 +- 0.153 seconds time elapsed  ( +-  9.80% )
```

GFLOPS of vector matrix ver.2 in O1:



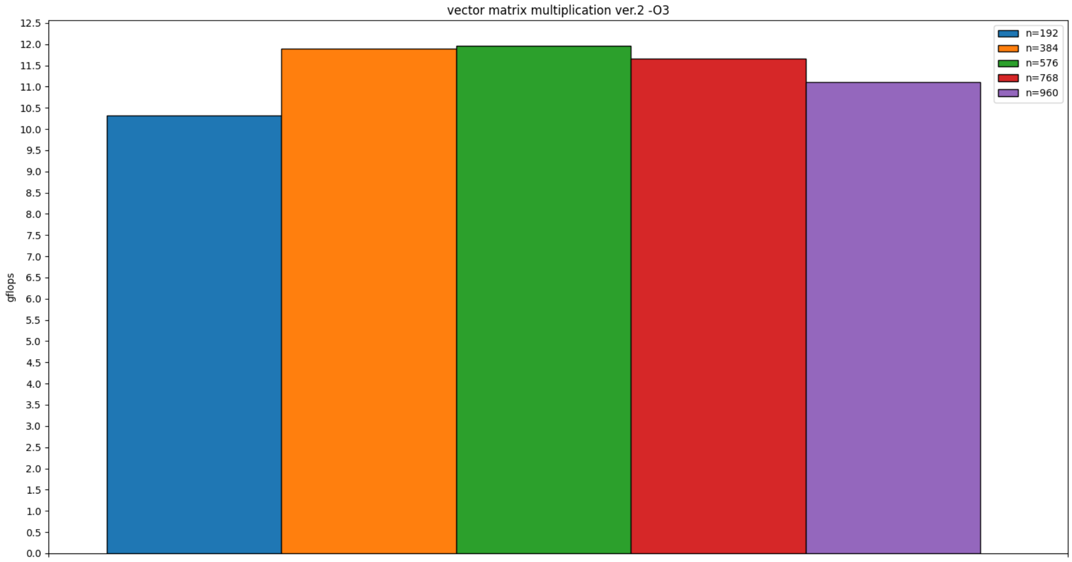

```
 Performance counter stats for './vector-matrixO1V2' (10 runs):

         6,865,125      cache-misses              #    6.530 % of all cache
refs      ( +- 32.20% )
       105,130,003      cache-references
( +-  7.32% )
     4,702,736,429      instructions              #    2.44  insn per cycle
( +-  0.03% )
     1,927,423,913      cycles
( +-  2.49% )

            0.7806 +- 0.0190 seconds time elapsed  ( +-  2.44% )
```

GFLOPS of vector matrix ver.2 in O2:



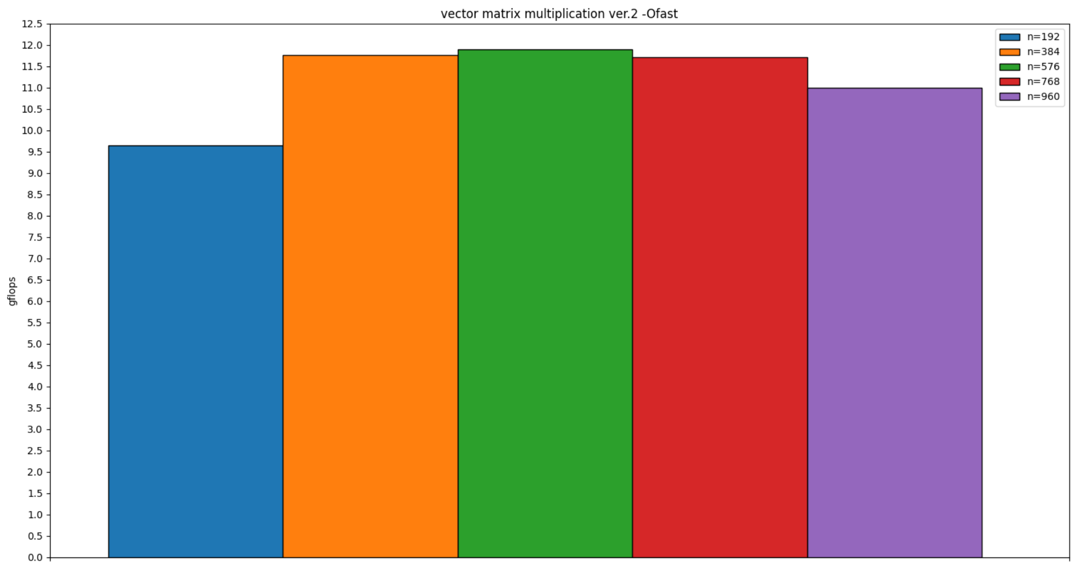

```
 Performance counter stats for './vector-matrixO2V2' (10 runs):

         5,772,104      cache-misses              #    5.213 % of all cache
refs      ( +- 29.60% )
       110,725,637      cache-references
( +-  5.54% )
     4,700,010,103      instructions              #    2.45  insn per cycle
( +-  0.00% )
     1,918,294,414      cycles
( +-  2.25% )

            0.7468 +- 0.0312 seconds time elapsed  ( +-  4.18% )
```

GFLOPS of vector matrix ver.2 in O3:




```
 Performance counter stats for './vector-matrixO3V2' (10 runs):

         7,736,970      cache-misses              #    7.212 % of all cache
refs      ( +- 27.18% )
       107,284,428      cache-references
( +-  7.46% )
     4,701,056,279      instructions              #    2.35  insn per cycle
( +-  0.00% )
     2,003,483,495      cycles
( +-  3.52% )

            0.7878 +- 0.0369 seconds time elapsed  ( +-  4.68% )
```

GFLOPS of vector matrix ver.2 in Ofast:




```
 Performance counter stats for './vector-matrixOfastV2' (10 runs):

         7,426,730      cache-misses              #    6.947 % of all cache
refs       ( +- 30.82% )
       106,911,190      cache-references
( +-  7.81% )
     4,728,534,699      instructions              #    2.38  insn per cycle
( +-  0.00% )
     1,987,159,173      cycles
( +-  3.30% )

            0.7834 +- 0.0340 seconds time elapsed  ( +-  4.34% )
```

comparing with version 1, this version doesn't need the data reordering (less instructions), more the cache miss counts but less the cache reference counts, so the performance is much better.

---

source code

https://github.com/OEmiliatanO/CSE_computer_organzation_final_project

---

reference

2017q1 matrix
Matrix Multiplication using SIMD
How to vectorize with gcc?
How many GCC optimization levels are there?
Options That Control Optimization
Introduction to SSE Programming
Intel® Intrinsics Guide