

# Accelerating Online Multiple-Choice Scoring: A SIMD Optimization on x86\_64 Architectures

Hoàng Minh Thiên

# Table of Contents

1. Scope of this research
2. The problem
3. The solutions
4. GCC (sometimes) doesn't auto-vectorize
5. Giving the compiler some hints
6. Optimizing with handwritten SIMD intrinsics
7. Going further: optimizing the structure itself
8. Updated implementations
9. Conclusions
10. Notes
11. Acknowledgements

# Scope of this research

- SIMD on Intel x86\_64 CPUs with AVX2 and AVX512BW, AVX512VL, AVX512F, AVX512DQ
- GNU Compiler Collection (GCC)  $\geq 10.0$
- C++20
- Single-threaded

# The problem

- Online tests are becoming more and more popular
- Servers usually bottleneck at peak exam hours (due to an enormous number of incoming requests)
- One element in the critical path is the scoring logic

# The solutions

- Modern CPUs provide single-instruction, multiple-data (SIMD) for speeding up repetitive workload

→ There is room for improvements here!

- Using automatic parallelization libraries (OpenMP, OpenCL, Intel's oneMKL, AMD's AOCL)
- Relying on compiler's auto-vectorization functionality (with `-O2` , `-O3` , `-mavx2` , etc. flags)
- DIY: handwritten SIMD intrinsics (easier) or inline assembly (more complex)

In this research, we focus on SIMD intrinsics

# GCC (sometimes) doesn't auto-vectorize

## The naive code

```
1  #include <immintrin.h>
2  #include <stdint.h>
3
4  #include <array>
5  #include <iostream>
6  #include <vector>
7
8  using ByteArray = std::vector<char>;
9
10 std::vector<int32_t> score(const std::vector<ByteArray> &exams,
11                          const ByteArray &correct_answers,
12                          const ByteArray &points) {
13     std::vector<int32_t> scored_exams_points(exams.size());
14     for (size_t i = 0; i < exams.size(); ++i) {
15         for (size_t j = 0; j < exams[i].size(); ++j) {
16             if (exams[i][j] == correct_answers[j]) {
17                 scored_exams_points[i] += static_cast<int32_t>(points[j]);
18             }
19         }
20     }
21     return scored_exams_points;
22 }
```

```

10 std::vector<int32_t> score(const std::vector<ByteArray> &exams,
11                           const ByteArray &correct_answers,
12                           const ByteArray &points) {
13     std::vector<int32_t> scored_exams_points(exams.size(), 0);
14     for (size_t i = 0; i < exams.size(); ++i) {
15         for (size_t j = 0; j < exams[i].size(); ++j) {
16             if (exams[i][j] == correct_answers[j]) {
17                 scored_exams_points[i] += static_cast<int32_t>(points[j]);
18             }
19         }
20     }
21     return scored_exams_points;
22 }

```

```
x86-64 gcc 15.1  -O3 -ffast-math -mavx512f 7/36
A  [Icons]
1  score(std::vector<std::vector<char, std::allocator<
2      movabs    rax, -6148914691236517205
3      push     r15
4      push     r14
5      push     r13
6      push     r12
7      mov      r12, rdi
8      push     rbp
9      mov      rbp, rdx
10     push     rbx
11     mov      rbx, rcx
12     sub      rsp, 24
13     mov      r14, QWORD PTR [rsi]
14     mov      r15, QWORD PTR [rsi+8]
15     sub      r15, r14
16     mov      r13, r15
17     sar      r13, 3
18     imul     r13, rax
19     test     r13, r13
20     je       .L21
21     lea      rdx, [0+r13*4]
22     mov      rdi, rdx
23     mov      QWORD PTR [rsp], rdx
    Edit on Compiler Explorer
```

# Giving the compiler some hints

## The boolean multiplication code

```
1  #include <stdint.h>
2
3  #include <array>
4  #include <vector>
5
6  using ByteArray = std::vector<char>;
7
8  std::vector<int32_t> score(const std::vector<ByteArray> &exams,
9                          const ByteArray &correct_answers,
10                         const ByteArray &points) {
11     std::vector<int32_t> scored_exams_points(exams.size());
12     for (size_t i = 0; i < exams.size(); ++i) {
13         for (size_t j = 0; j < exams[i].size(); ++j) {
14             if (exams[i][j] == correct_answers[j]) {
15                 scored_exams_points[i] += static_cast<int32_t>(points[j]);
16             }
17         }
18     }
19     return scored_exams_points;
20 }
```

```

A ▾ 🗑 + ▾ V 🔍 🐞 C++ ▾
10 std::vector<int32_t> score(const std::vector<ByteArray
11                          const ByteArray &correct
12                          const ByteArray &points)
13     std::vector<int32_t> scored_exams_points(exams..
14     for (size_t i = 0; i < exams.size(); ++i) {
15         for (size_t j = 0; j < exams[i].size(); ++j
16             // Idea: to reduce false branch predict
17             scored_exams_points[i] += (exams[i][j] -
18             static_cast<i
19     }
20 }
21 return scored_exams_points;
22 }

```

```

x86-64 gcc 15.1 ▾ 🗑 🟢 -O3 -ffast-math -march=core2
9/36
A ▾ ⚙ ▾ 🔍 🗑 🛠 + ▾ 🖌
1 score(std::vector<std::vector<char, std::allocator<
2     movabs    rax, -6148914691236517205
3     push     r15
4     push     r14
5     push     r13
6     mov      r13, rcx
7     push     r12
8     push     rbp
9     mov      rbp, rdx
10    push     rbx
11    sub      rsp, 40
12    mov      r14, QWORD PTR [rsi]
13    mov      r15, QWORD PTR [rsi+8]
14    mov      QWORD PTR [rsp+8], rdi
15    sub      r15, r14
16    mov      rbx, r15
17    sar      rbx, 3
18    imul     rbx, rax
19    test     rbx, rbx
20    je       .L20
21    lea      rdx, [0+rbx*4]
22    mov      rdi, rdx
23    mov      QWORD PTR [rsp+16], rdx

```

# Optimizing with handwritten SIMD intrinsics

# SIMD in x86 CPUs

- Normal assembly instructions are **scalar** ones: takes one `A` and `B` and produce a result `C` . For example:  
`1 + 2 = 3`
- Single-instruction, multiple-data (SIMD) means that the instructions take multiple `A` and `B` and produce multiple result `C` . For example: `[1, 2, 3] + [4, 5, 6] = [5, 7, 9]`

# SIMD **intrinsics** introduction

- Intrinsics are compiler-generated functions for writing code closer to assembly.
- They provide us `__m128i` (SSE family), `__m256i` (AVX/AVX2 family), and `__m512i` (AVX512 family) as integer data types.
- We can pack many smaller integer types in those data types, as demonstrated later → This means that we can process multiple data at once.
- We will focus on AVX2 and AVX512 as they're widely available on modern CPUs.
- For more details, please see Intel® Intrinsics Guide at <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>.

## `_mm256_sad_epu8` intrinsic

As defined by Intel® Intrinsics Guide:

Compute the absolute differences of packed **unsigned** 8-bit integers in `a` and `b`, then horizontally sum each consecutive 8 differences to produce four **unsigned** 16-bit integers, and pack these **unsigned** 16-bit integers in the low 16 bits of 64-bit elements in `dst`.

Symbol:

```
1  __m256i _mm256_sad_epu8 (__m256i a, __m256i b)
```

→ This highlights the complex nature of x86\_64 architecture: one instruction does multiple operations

## `_mm256_sad_epu8` visualized

Vector/INT8	0	1	2	3	4	5	6	7	...	31
A	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08		0x20
B	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00		0x00
TMP	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08		0x20
Result (DST)	0x24	0x00	0x00	0x00	0x00	0x00	0x00	0x00		0x00

→ By `_mm256_sad_epu8(A, _mm256_setzero_si256())`, we've calculated sums of four groups of eight 8-bit integers.

Note:

- `_mm256_setzero_si256` is an intrinsic to create a new, zeroed `__m256i`.
- `_mm512_sad_epu8` and `_mm512_setzero_si512` has the same functionality to this 256-bit variant.

# AVX2

```
#include <immintrin.h>
#include <stdint.h>

#include <array>
#include <vector>

using ByteArray = std::vector<char>;

std::vector<int32_t> score(const std::vector<ByteArray> &exams,
                        const ByteArray &correct_answers,
                        const std::vector<int8_t> &points) {
    std::vector<int32_t> scored_exams_points(exams.size(), 0);
    // We are targeting AVX2, so we have at most 256-bit registers,
    // which means that we can pack 32 MCQs to score at a time.
    constexpr int32_t BATCH_SIZE = 32;

    // Process each exam
    for (size_t i = 0; i < exams.size(); ++i) {
        auto &exam = exams[i];

        // Prefetch the next exam
        // https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html
        if (i + 1 < exams.size()) {
            __builtin_prefetch(&exams[i + 1]);
        }
    }
}
```

# AVX512

```
#include <immintrin.h>
#include <stdint.h>

#include <array>
#include <vector>

using ByteArray = std::vector<char>;

std::vector<int32_t> score(const std::vector<ByteArray> &exams,
                        const ByteArray &correct_answers,
                        const std::vector<int8_t> &points) {
    // The idea is the same as the AVX2 functions, the only difference is that
    // we have double the registers' size (512 bits instead of 256 bits)
    std::vector<int32_t> scored_exams_points(exams.size());
    constexpr int32_t BATCH_SIZE = 64;

    // Process each exam
    for (size_t i = 0; i < exams.size(); ++i) {
        auto &exam = exams[i];

        // Prefetch the next exam
        if (i + 1 < exams.size()) {
            __builtin_prefetch(&exams[i + 1]);
        }

        size_t j = 0;
```

# Benchmarking system

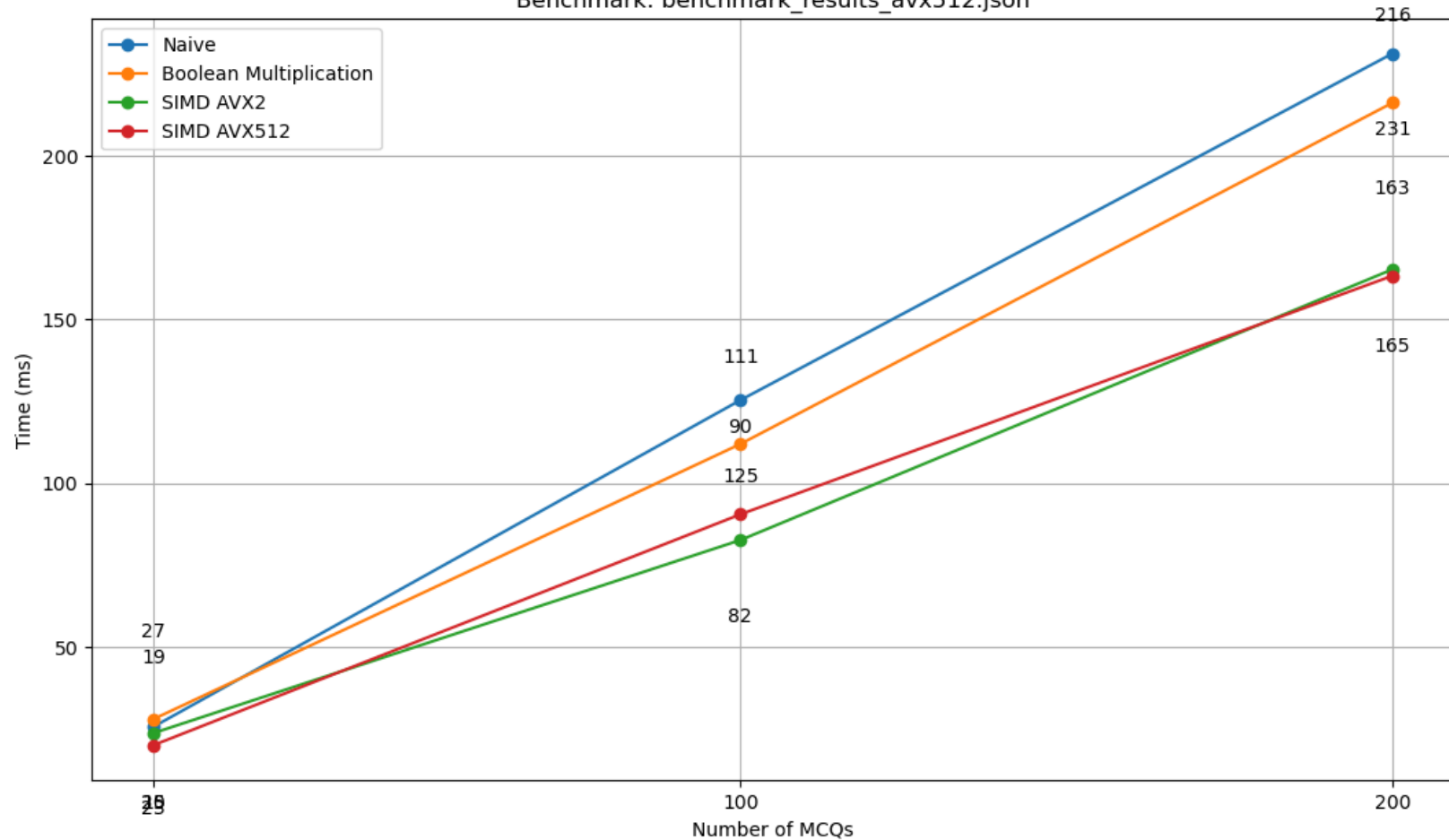
- Ubuntu 24.04.2 LTS x86\_64 6.8.0-60-generic
- 11th Gen Intel i5-1135G7 @ 2.419 GHz
- 4 GB RAM
- Enabled compiler flags:

```
-O3: release build  
-Wall: show all warnings  
-mavx2 -mavx512bw -mavx512vl -mavx512f -mavx512dq: enable AVX2, AVX512{BW, VL, F, DQ}  
-fno-omit-frame-pointer -fsanitize=address -fno-sanitize-recover=all: AddressSanitizer (to detect memory issues)
```

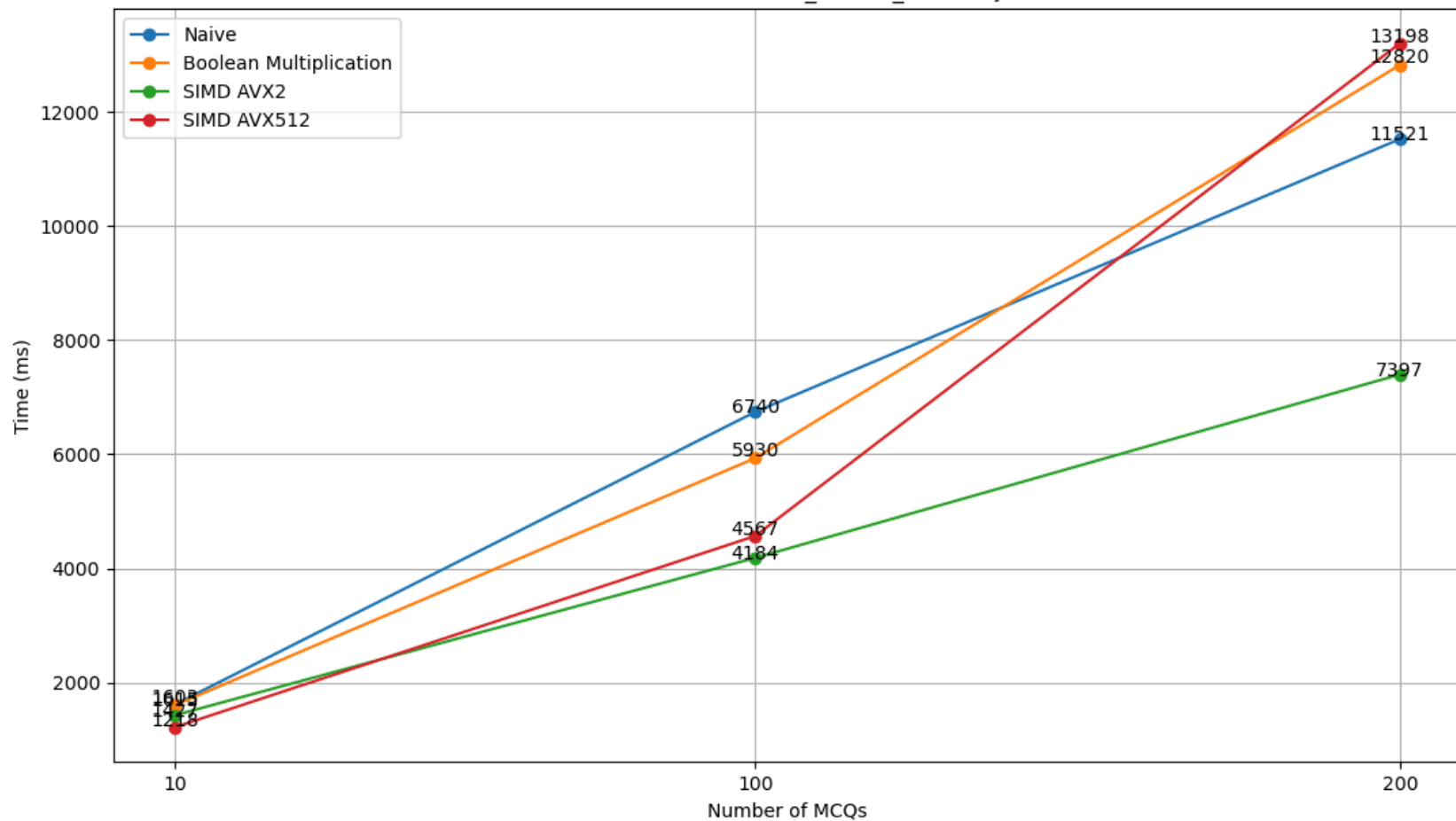
- Benchmarked using Google's benchmark library, with `BENCHMARK_LTO=ON` .
- We will use **real time** (wall time), not CPU time.

# Benchmarks

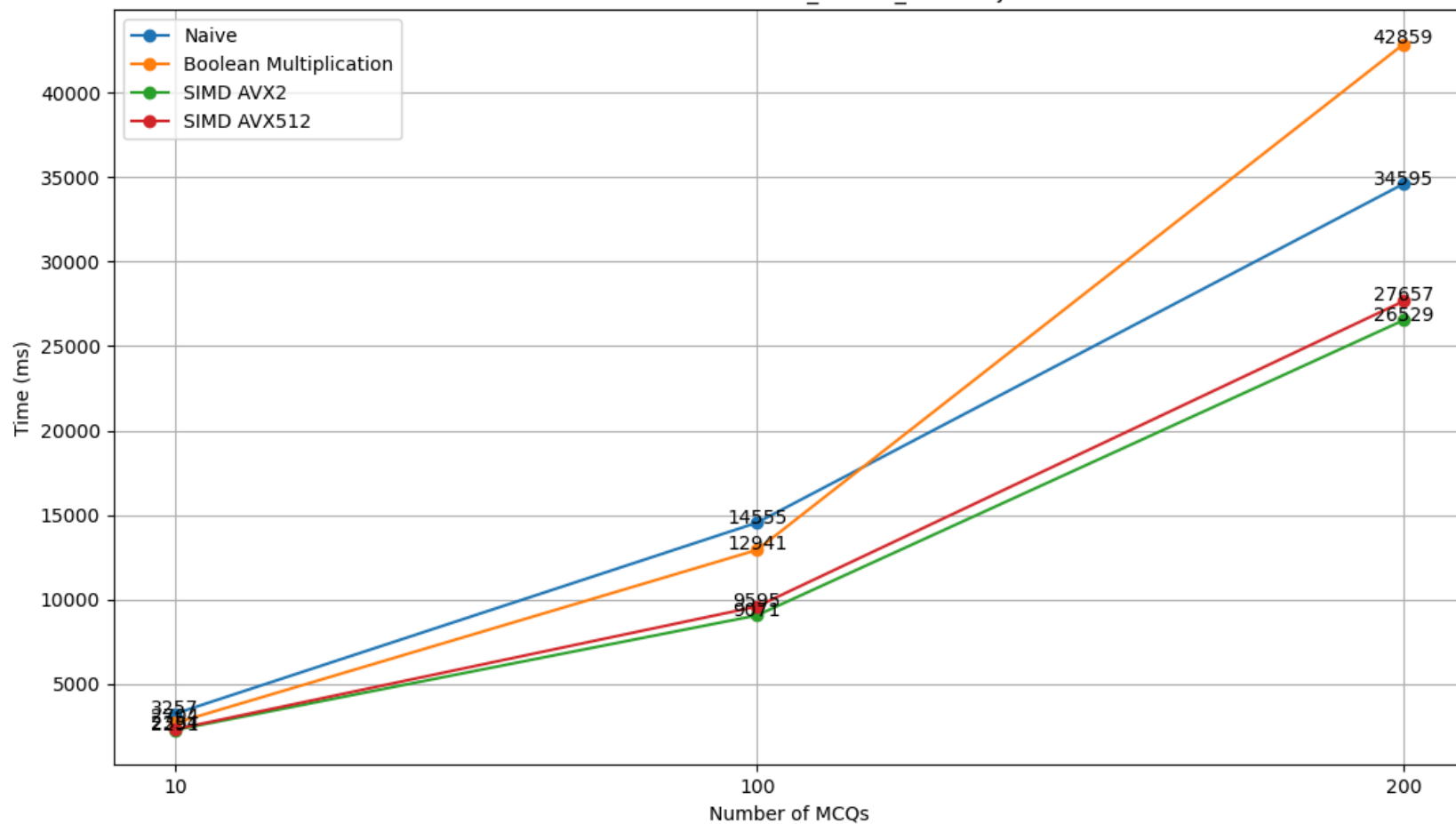
Scoring time for 100000 exams  
Benchmark: benchmark\_results\_avx512.json



Scoring time for 5000000 exams  
Benchmark: benchmark\_results\_avx512.json



Scoring time for 10000000 exams  
Benchmark: benchmark\_results\_avx512.json



Going further: optimizing the structure itself

# Data structure changes

```
#include <vector>

using ByteArray = std::vector<char>;
```

Problem: `std::vector` access time is slower than C arrays. How can we achieve:

- The ease of use of `std::vector` ?
- The performance of C arrays?

→ Create a new data structure ourselves!



# Updated implementations

# AVX2

```
#include <immintrin.h>
#include <stdint.h>

#include <vector>

std::vector<int32_t> score(const std::vector<ByteArray> &exams,
                        const ByteArray &correct_answers,
                        const ByteArray &points) {
    std::vector<int32_t> scored_exams_points(exams.size(), 0);

    // Process each exam
    for (size_t i = 0; i < exams.size(); ++i) {
        auto &exam = exams[i];

        // Prefetch the next exam
        // https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html
        if (i + 1 < exams.size()) {
            __builtin_prefetch(&exams[i + 1]);
        }

        for (size_t j = 0, _j = 0; j < correct_answers.block_count_avx2();
            ++j, _j = j << 5) {
            // Vectorize exam's MCQs
            __m256i v1 = _mm256_loadu_si256(
                reinterpret_cast<const __m256i *>(exam.data() + _j));
            // Vectorize correct MCQs
```

# AVX512

```
#include <immintrin.h>
#include <stdint.h>

#include <vector>

std::vector<int32_t> score(const std::vector<ByteArray> &exams,
                        const ByteArray &correct_answers,
                        const ByteArray &points) {
    std::vector<int32_t> scored_exams_points(exams.size(), 0);

    // Process each exam
    for (size_t i = 0; i < exams.size(); ++i) {
        auto &exam = exams[i];

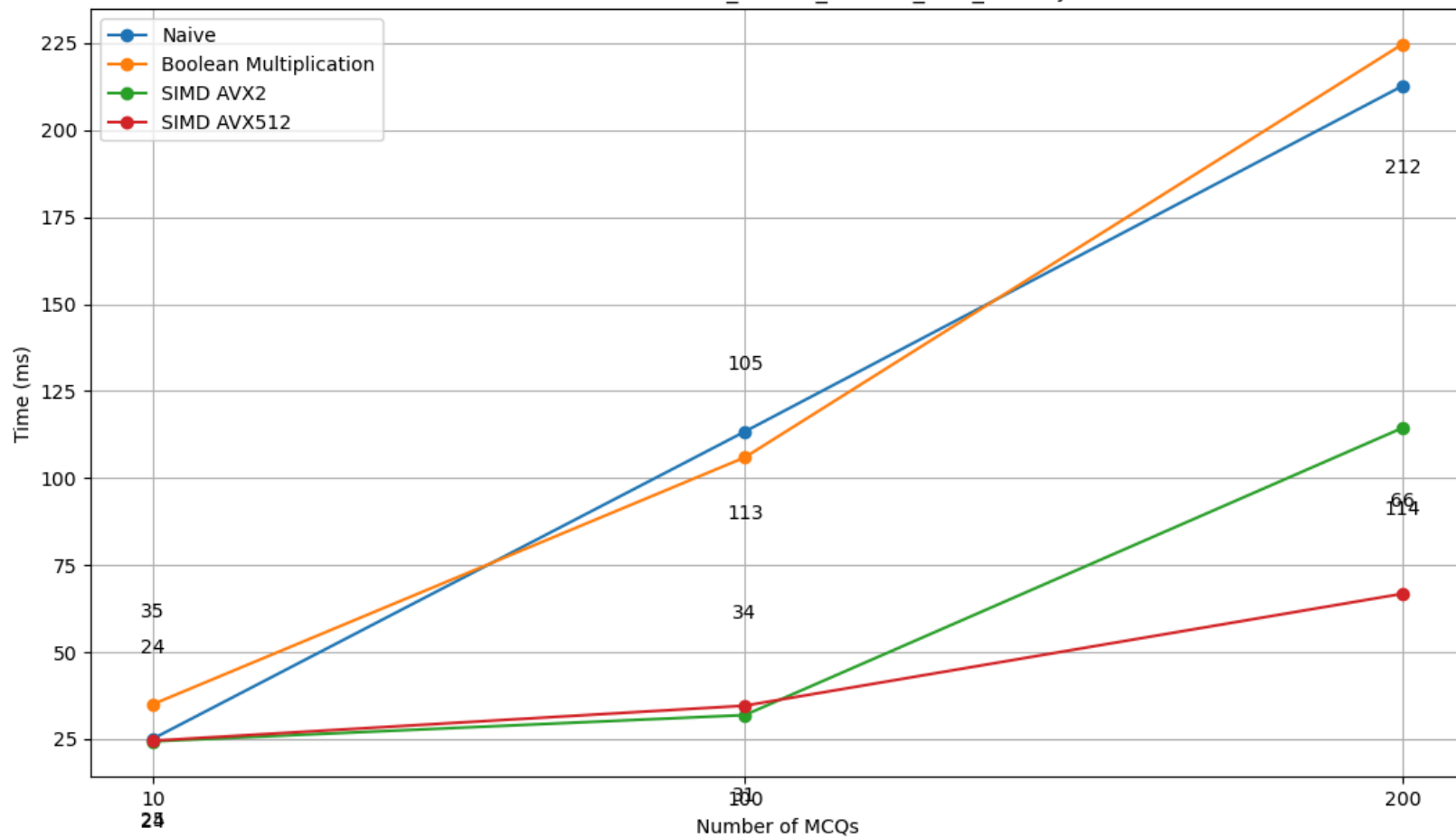
        // Prefetch the next exam
        if (i + 1 < exams.size()) {
            __builtin_prefetch(&exams[i + 1]);
        }

        for (size_t j = 0, _j = 0; j < correct_answers.block_count_avx512();
            ++j, _j = j << 6) {
            // Load the exam and the correct answers
            __m512i v1 = _mm512_loadu_si512(exam.data() + _j);
            const __m512i v2 = _mm512_loadu_si512(correct_answers.data() + _j);

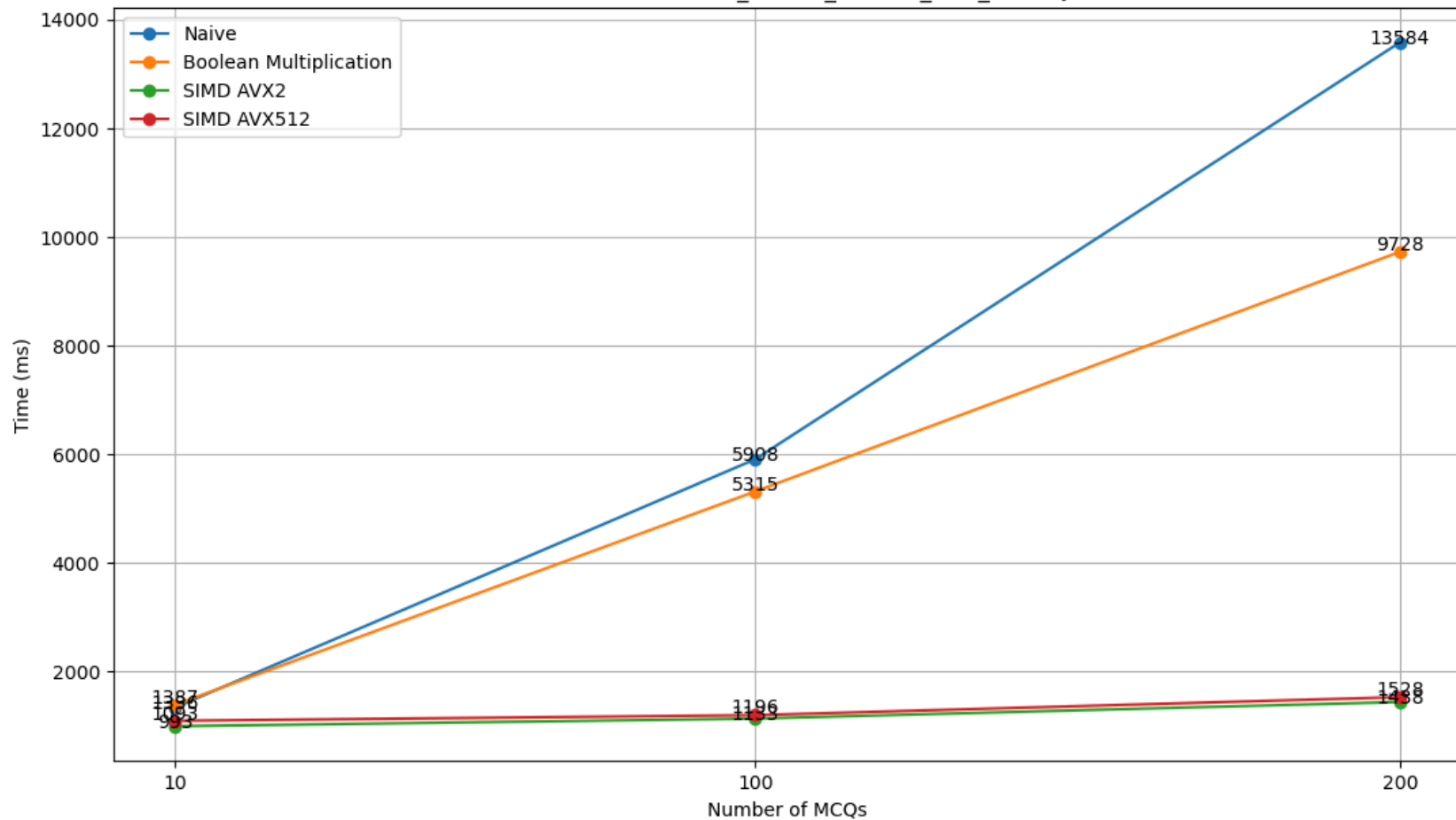
            // Compute the mask
```

# Benchmarks

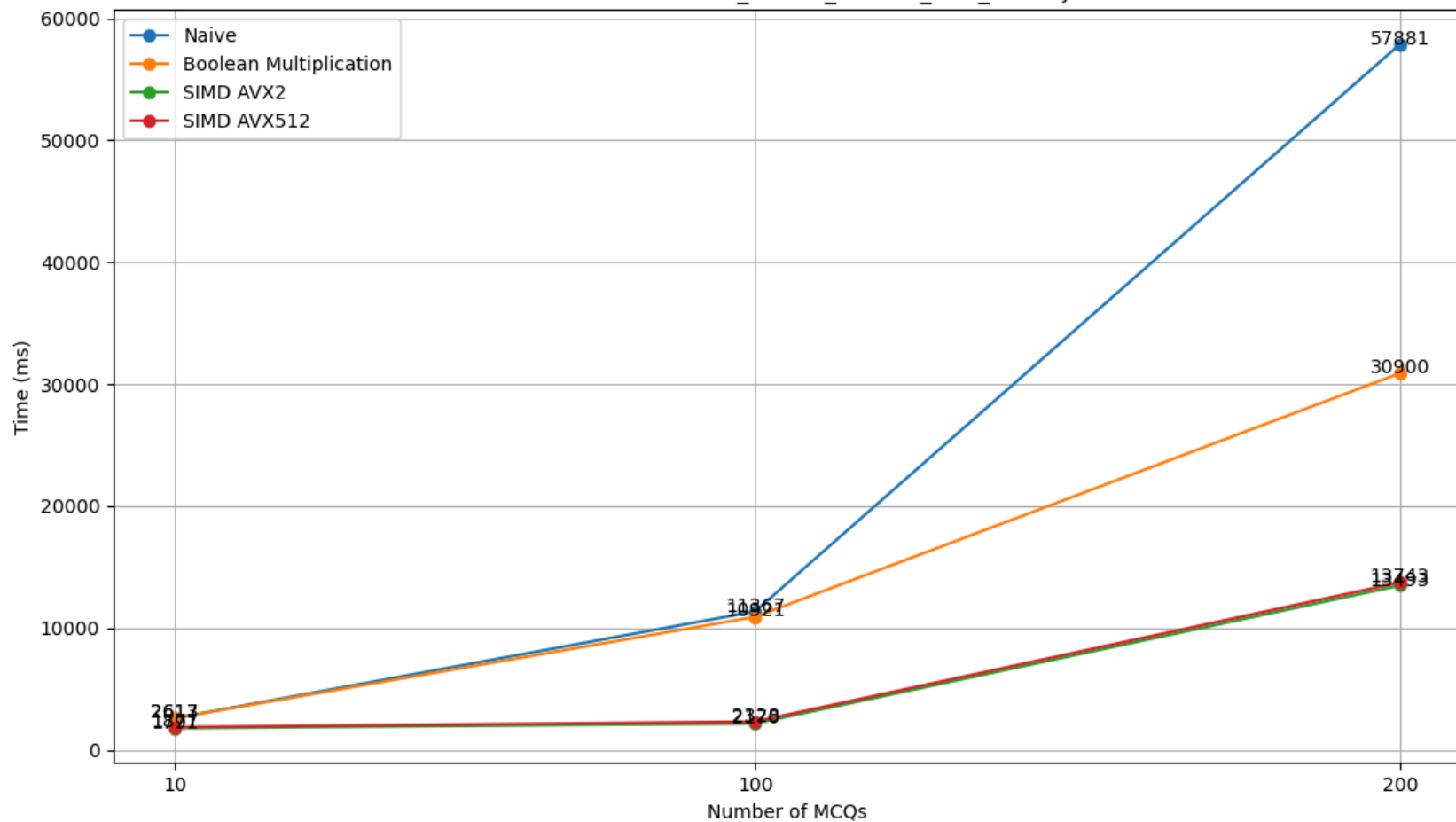
Scoring time for 100000 exams  
Benchmark: benchmark\_results\_avx512\_new\_struct.json



Scoring time for 5000000 exams  
Benchmark: benchmark\_results\_avx512\_new\_struct.json



Scoring time for 10000000 exams  
Benchmark: benchmark\_results\_avx512\_new\_struct.json



# Drawbacks

- More complex to set up.
- We're using more memory than required (if the number of MCQs is not a multiple of 32). E.g. when the number of questions is 200, the capacity will be raised to 224 (AVX2) or 256 (AVX512).
- Need to reimplement many operator overloading (unlike prebuilt ones from `std` )

# Conclusions

- SIMD utilization, if done correctly, can bring up to 3-7x performance improvement.
- Higher-memory, higher-performance (with custom, correctly aligned data structures).
- We can't entirely depend on GCC's auto-vectorization, yet.
- AVX512 doesn't provide a big boost of performance like advertised (because of the loading overhead).

# Notes

- The code repository is at <https://github.com/hmthien050209/simd-research> (with the older `std::vector<char>`-based version located on branch `avx512_vec` )
- For our operation, we don't need to care about number signedness, because our points are positive
- There are also `__m128d` , `__m256d` , and `__m512d` , which are double-precision floating point data types. For now, we focus on working with integer data types.
- x86\_64 CPUs from AMD also provide 3DNow!, 3DNow! Professional, but they're deprecated in 2010. [1]

- 
1. M. Yam, "AMD drops 3DNow! support from future CPUs," Tom's Hardware, Aug. 24, 2010. Online. Available: <https://www.tomshardware.com/news/3dnow-simd-extensions-phenom-sse,11128.html> ↩

# Acknowledgements

- Đoàn Ngọc Bình Minh for providing me a VM on his laptop for the above benchmarks.
- Intel® Intrinsic Guide.
- Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture.

Thank you!