EURECOM
COMPUTATIONAL METHODS
2018

Laboratory Report

# Intel SIMD instructions performances

Professor: Raymond Knopp

Authors:

**Simili** Michele

# Contents

# 1 Introduction

## 1.1 Objective of the Laboratory

All the work discussed in this report is available at my Github page at `https://github.com/mikilauda95/comp_meth`

The objective of this laboratory is to test the impact on performances using the SIMD instructions compared to scalar implementations while doing the same operation multiple times. Single instructions multiple data are particular kind of assembly instructions which compute the same operation for multiple data at a time. In this case, each data is 16 bits long and the multiplication operation is considered. The Intel processor of the machine on which the experiments are run is an Intel core i7-665U and provides two kinds of SIMD instructions working with vector of 128 bits and 256 bits which correspond respectively to 4 and 16 data. Finally, the multiplication between complex number will be considered.

# 2
# The Code

## 2.1 Overview of the simulation

The performance will be evaluated measuring the speed of multiplications over increasing range of numbers. In order to measure the time of execution of a certain operation, it has been used a particular structure working as a timer able to measure tick cycles. This timer is started before each operation and stopped and reset immediately after.

## 2.2 Data structures and operation implementations

### 2.2.1 The real case

In this lab, operands are integers represented in 16 bits and they are store in a int_16 array. Then the array is initialized to random numbers.

**Listing 2.1:** *int16_t array initialization*

```
x = (int16_t *) aligned_alloc(32,sizeof(int16_t) * VECTOR_SIZE);
y = (int16_t *) aligned_alloc(32,sizeof(int16_t) * VECTOR_SIZE);
z = (int16_t *) aligned_alloc(32,sizeof(int16_t) * VECTOR_SIZE);
// FILL THE ARRAY WITH RANDOM NUMBERS
int i;
srand(time(NULL));
for (i = 0; i < VECTOR_SIZE; i++) {
x[i] = (int16_t)rand();
y[i] = (int16_t)rand();
/*printf("x: %d y: %d\n", x[i], y[i]);*/
}
```

However, the SIMD instructions work on particular data types that are __m128i and __m256i. These data types pack multiple int16_t in one element. The packing procedure is very trivial. It is just necessary to cast the array pointer from int16_t to and array pointer of the data structure it has to be used.

**Listing 2.2:** *Cast from int_16 to __m128_i array*

```
__m128i *x128 = (__m128i *)x;
__m128i *y128 = (__m128i *)y;
__m128i *z128 = (__m128i *)z;
```

This is possible because the *int16_ t* array are stored sequentially in memory. However segmentation faults have been encountered when using operations with *__ m256i* type. Probably the cause come from a non alignment of the data. So, in order to enforce this, instead of using a *malloc* for reserving memory, the *aligned_ alloc* is used instead for forcing the alignment.

For implementing the multiplication the right SIMD instruction must be used depending on the case: _mm256_mulhrs_epi16 for the 128 case and _mm_mulhrs_epi16 for the 256 bits case. These multiply two arrays of 16 bits and return an array of 16 bits, meaning that it will truncate the the multiplication result if it overflows (as the result should be represented in at least 32 bits).

**Listing 2.3:** *Scalar multiplication*

```
static inline void componentwise_multiply_real_scalar(int16_t *x
   ,int16_t *y,int16_t *z, int N) {
        int i;
        for (i = 1; i <= N; i++) {
                z[i] = x[i] * y[i];
        }
}
```

**Listing 2.4:** *SSE4 vectorized multiplication*

```
static inline void componentwise_multiply_real_128(int16_t *x,
   int16_t *y,int16_t *z, int N) {

        __m128i *x128 = (__m128i *)x;
        __m128i *y128 = (__m128i *)y;
        __m128i *z128 = (__m128i *)z;

        int i;
        for (i = 0; i < ceil(N/8.0); i++) {
                z128[i] = _mm_mulhrs_epi16(x128[i], y128[i]);
        }
}
```

**Listing 2.5:** *AVX2 vectorized multiplication*

```
static inline void componentwise_multiply_real_256(int16_t *x,
   int16_t *y,int16_t *z, uint16_t N) {

        __m256i *x256 = (__m256i *)x;
        __m256i *y256 = (__m256i *)y;
        __m256i *z256 = (__m256i *)z;

        int i;
        for (i = 0; i  ceil(N/16.0); i+=1) {
                z256[i] = _mm256_mulhrs_epi16(x256[i], y256[i]);
        }
}
```

The for loop for the SIMD, will iterate on less elements compared to the scalar case, as every time we are doing more multiplications at once.

The *static inline function* is used in order to reduce the overhead from the call of the function. The timing is measured starting the timer before the "call" of the function and stop it just after. After every measurement, the timer is reset. As the timing is affected by an error due to cache misses or preemption, the same operation is run multiple times and only the minimum is considered as we assume that the error is only additive. However, also the average has been implemented, returning similar result. probably due to the fact that after few iterations the cache contents are set correctly.

### 2.2.2 The Complex Case

As a reminder, the complex multiplication is performed in the following way:

$$Z_c = X_c * Y_c$$

$$Z_r e + i * Z_i m = (X_r e + i * X_i m) * (Y_r e + i * Y_i m)$$

$$Z_r e = X_r e * Y_r e - X_i m * Y_i m$$

$$Z_i m = X_r e * Y_i m - X_i m * Y_r e$$

In the complex case, two different situations are analyzed depending on how the inputs numbers are provided:

- Case A : Every complex numbers are stored in a particular structure with Real and Imaginary part.
- Case B : The real part and the imaginary part are found in two different arrays.

**Case A** Here is the data structure used for each complex number: ADD structure. So the operands will be stored in arrays of *cstructs* instead of *int16_ t*.

Taking a look at this array in memory, we can see that real and imaginary parts are alternated:

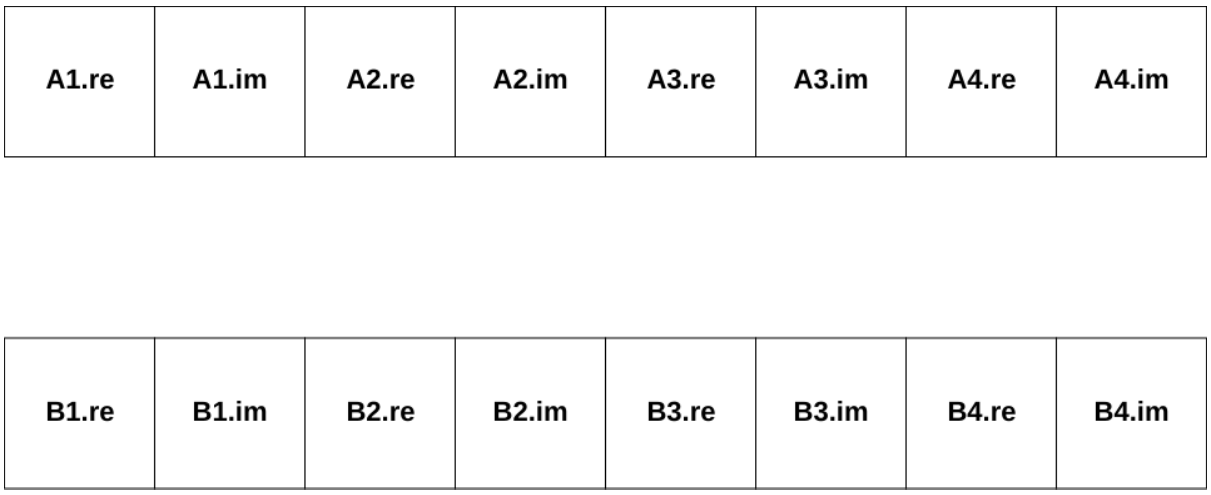


**Figure 2.1:** *Complex structure in memory*

So, casting this array to the an array of SIMD data type, will keep this configuration. However, a particular SIMD can be used to exploit this configuration, the *_mm_madd_epi16*. Which performs the following operation:

| R0 | R1 | R2 | R3 |
|---|---|---|---|
| (a0 * b0) + (a1 * b1) | (a2 * b2) + (a3 * b3) | (a4 * b4) + (a5 * b5) | (a6 * b6) + (a7 * b7) |

**Figure 2.2:** *_mm_madd_epi16*

Now using the function textit_mm256_sign_epi16 for inverting the sign and *_mm256_shufflelo_epi16*for shuffling the elements, it is possible to arrange the SIMD array in such a way to compute the multiplication.

**Listing 2.6:** *Complex number multiplication case A*

```
static inline void cmult(__m128i a,__m128i b, __m128i *re32,
    __m128i *im32) {
            __m128i mmtmpb;
            /*printf("cmult\n");*/
            mmtmpb      = _mm_sign_epi16(b,*(__m128i*)reflip);
            *re32       = _mm_madd_epi16(a,mmtmpb);
            mmtmpb      = _mm_shufflelo_epi16(b,_MM_SHUFFLE(2,3,0,1));
            mmtmpb      = _mm_shufflehi_epi16(mmtmpb,_MM_SHUFFLE
                (2,3,0,1));
            *im32    = _mm_madd_epi16(a,mmtmpb);
}
```

However the result of the *_mm_madd_epi16* will be in 32 bits and so it need to be packed again afterwards:

**Listing 2.7:** *Pack complex number result*

```
static inline __m128i cpack(__m128i xre, __m128i xim) {
            __m128i cpack_tmp1, cpack_tmp2;
            cpack_tmp1 = _mm_unpacklo_epi32(xre,xim);
            cpack_tmp1 = _mm_srai_epi32(cpack_tmp1,15);
            cpack_tmp2 = _mm_unpackhi_epi32(xre,xim);
            cpack_tmp2 = _mm_srai_epi32(cpack_tmp2,15);
            return(_mm_packs_epi32(cpack_tmp1,cpack_tmp2));
}
```

**Case B** In this case the real part and imaginary parts are kept separated. The complex multiplication can be performed straightforward using the SIMD multiplications, addition and subtraction:

**Listing 2.8:** *Case B complex number multiplication*

```
static inline void componentwise_multiply_complex_128B(int16_t *
    xre, int16_t *yre, int16_t *xim, int16_t *yim, int16_t *zre,
    int16_t *zim, int N) {
        __m128i *xre128 = (__m128i *)xre;
        __m128i *yre128 = (__m128i *)yre;
        __m128i *xim128 = (__m128i *)xim;
        __m128i *yim128 = (__m128i *)yim;
        __m128i *zre128 = (__m128i *)zre;
        __m128i *zim128 = (__m128i *)zim;
        __m128i tmp;
        int i;
        for (i = 0; i < ceil(N/8.0)*2; i++) {
                zre128[i] = _mm_mulhrs_epi16(xre128[i], yre128[i
                   ]);
                zre128[i] = _mm_sub_epi16(zre128[i],
                   _mm_mulhrs_epi16(xim128[i], yim128[i]));
                zim128[i] = _mm_mulhrs_epi16(xre128[i], yim128[i
                   ]);
                zim128[i] = _mm_add_epi16(zre128[i],
                   _mm_mulhrs_epi16(xim128[i], yre128[i]));
        }
}
```

## 2.3 Compiling and Optimizations

The compilation of the code required the inclusion and the linkage of several libraries. A *Makefile* has been written to facilitate this. Moreover, *CFLAGS* variable has been declared overridden in order to be able to modify it while launching the command make. This has been done to try different optimizations without touching the makefile. Finally, including the -S option, it is possible to generate the assembly.

### 2.3.1 Assembly code

Here is an extract of the assembly, the implementation of the *static inline void componentwise_multiply_real_128*. In this case optimization -O2 is used as it is the default for gcc:

**Listing 2.9:** *componentwise_multiply_real_128 assembly code*

```
componentwise_multiply_real_128:
.LFB3683:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $112, %rsp
        movq    %rdi, -72(%rbp)
        movq    %rsi, -80(%rbp)
        movq    %rdx, -88(%rbp)
        movl     %ecx, -92(%rbp)
        movq     -72(%rbp), %rax
        movq     %rax, -56(%rbp)
        movq     -80(%rbp), %rax
        movq     %rax, -48(%rbp)
        movq     -88(%rbp), %rax
        movq     %rax, -40(%rbp)
        movl     $0, -60(%rbp)
        jmp      .L167
.L169:
        movl     -60(%rbp), %eax
        cltq
        salq     $4, %rax
        movq     %rax, %rdx
        movq     -48(%rbp), %rax
        addq     %rdx, %rax
        vmovdqa (%rax), %xmm0
        movl     -60(%rbp), %eax
        cltq
        salq     $4, %rax
        movq     %rax, %rdx
        movq     -56(%rbp), %rax
        addq     %rdx, %rax
        vmovdqa (%rax), %xmm1
        movl     -60(%rbp), %eax
        cltq
        salq     $4, %rax
        movq     %rax, %rdx
        movq     -40(%rbp), %rax
        addq     %rdx, %rax
        vmovaps %xmm1, -32(%rbp)
        vmovaps %xmm0, -16(%rbp)
        vmovdqa -16(%rbp), %xmm0
        vmovdqa -32(%rbp), %xmm1
        vpmulhrsw        %xmm0, %xmm1, %xmm0
        vmovaps %xmm0, (%rax)
        addl     $1, -60(%rbp)
.L167:
        vcvtsi2sd        -60(%rbp), %xmm2, %xmm2
        vmovsd  %xmm2, -104(%rbp)
        vcvtsi2sd        -92(%rbp), %xmm0, %xmm0
        vmovsd  .LC16(%rip), %xmm1
        vdivsd  %xmm1, %xmm0, %xmm0
        call     ceil@PLT
        vucomisd         -104(%rbp), %xmm0
        ja       .L169
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE3683:
        .size   componentwise_multiply_real_128, .-
           componentwise_multiply_real_128
        .type   componentwise_multiply_real_256, @function
```

As we can see the of the function is represented by the sequence:

**Listing 2.10:** *SIMD assembly instructions*

```
vmovaps %xmm1, -32(%rbp)
vmovaps %xmm0, -16(%rbp)
vmovdqa -16(%rbp), %xmm0
vmovdqa -32(%rbp), %xmm1
vpmulhrsw       %xmm0, %xmm1, %xmm0
vmovaps %xmm0, (%rax)
```

These operations simply run special load instruction for loading the operands in special registers *xmm0* and *xmm1.* And then multiply these using *vpmulhrsw* special instruction putting the result in *xmm0* which will be then stored.

### 2.3.2 Optimizations

The program has been compiled using several kinds of optimization configurations provided by *gcc*:

- -O1: The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- -O2 Optimize even 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 Here gcc tries to optimize the program very aggressively. In particular it performs optimizations on loops with the loop unrolling. We will see that this results in a big advantage for the scalar case.

# 3 Simulation results

## 3.1 Real numbers

In picture  3.1, it can be found the comparisons between the scalar and SIMD implementation on increasing range of numbers.

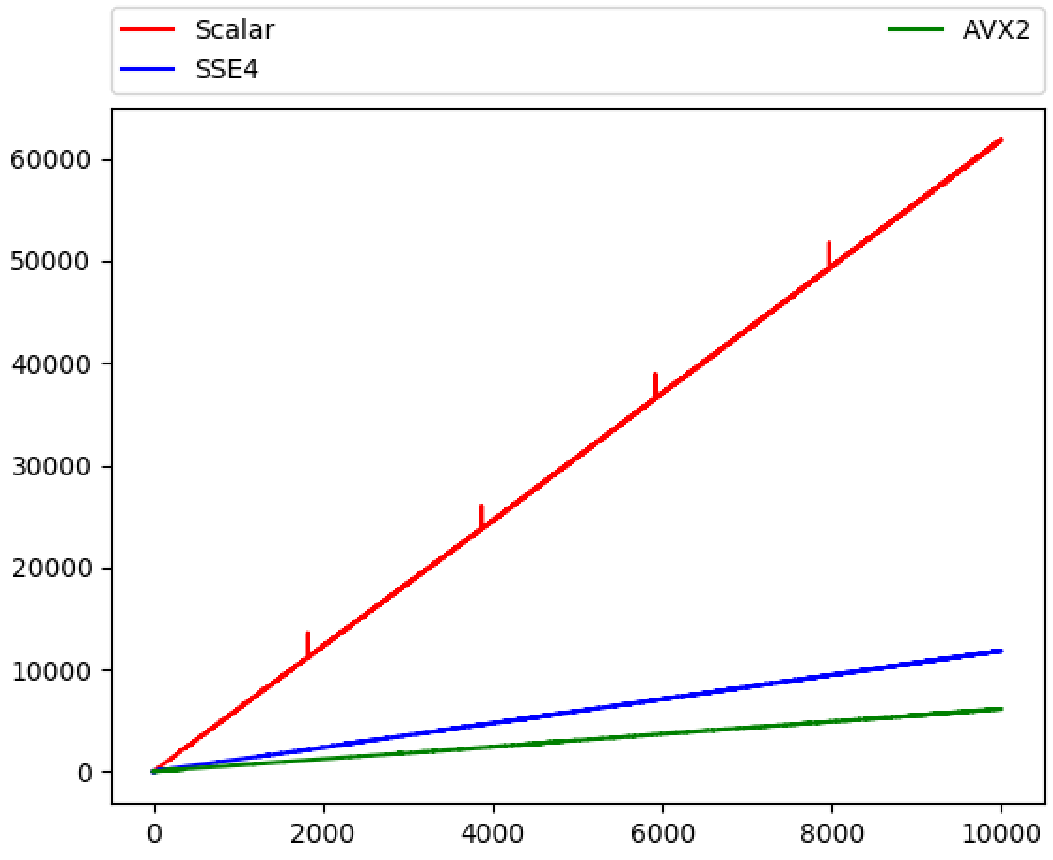


**Figure 3.1:** *Complex structure in memory*

As expected the more is the parallelism, the better performances are reached. The time of execution in fact increases linearly with the number of numbers to be multiplied.

However a strong irregularity is found. This can be due to two factors that highly affect the simulation: the cache misses and the preemption by the Operating System.

While the first factor can be managed somehow, for example running the same instruction that work on the same data multiple times so that data is loaded into the cache and hopefully kept there, the second factor is the most annoying.

One way to approach the problem has been to run the program several time and keeping only the smallest times of execution. However, even following this approach the problem is not solved completely. t happened to have several and heavy processes running at the same time and getting very irregular plots.

Another proof that the irregularity comes from the preemption are the very irregular plots obtained when running the program together with other heavy applications.

Another interesting plot to consider is the speedup of the SIMD performance with respect to scalar case. This translates to the ratio of the scalar delay with the vector one.

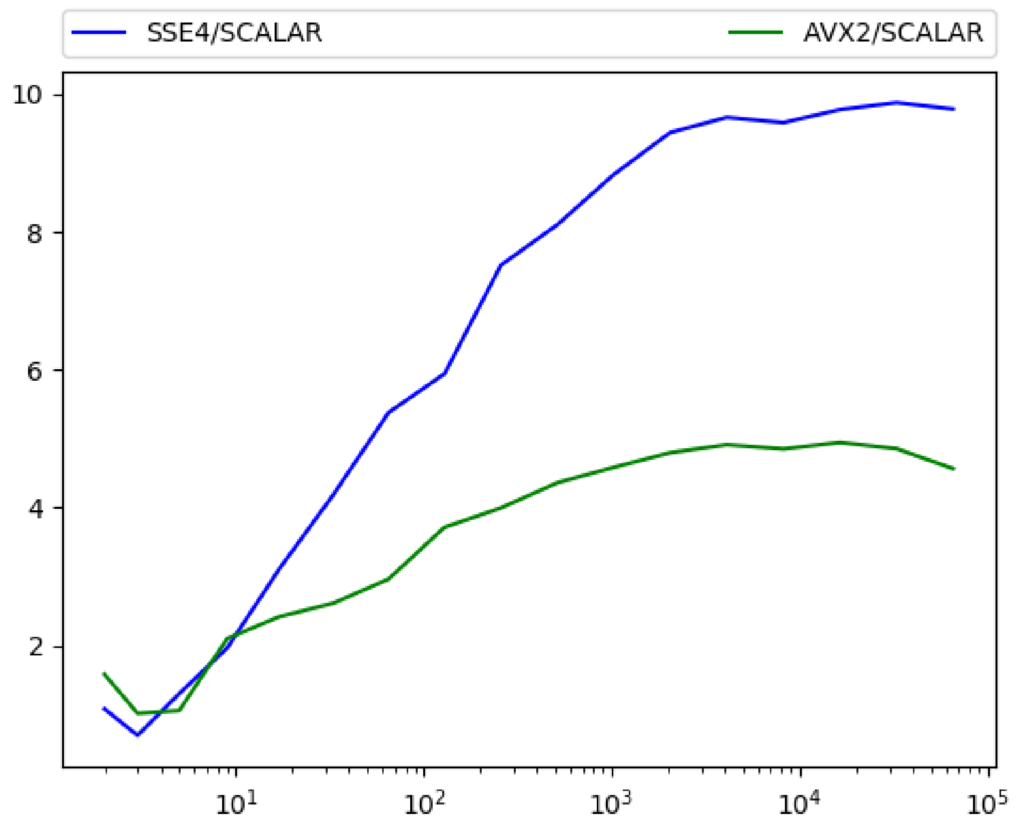


**Figure 3.2:** *SIMD Speedup*

This result is very interesting because we would expect a constant graph being the ratio between to linear behaviors. So, we conclude that the characteristics found are not perfectly linear.

The speedup in fact grows while processing few operands, until it saturates (approximately at 4096 operands), probably because the L1 cache is full at this point and the speedup is not evident because cache misses dominate.

### 3.1.1 Optimizations

The results of the impact of optimization flags on the performances of the code can be appreciated in the following images:

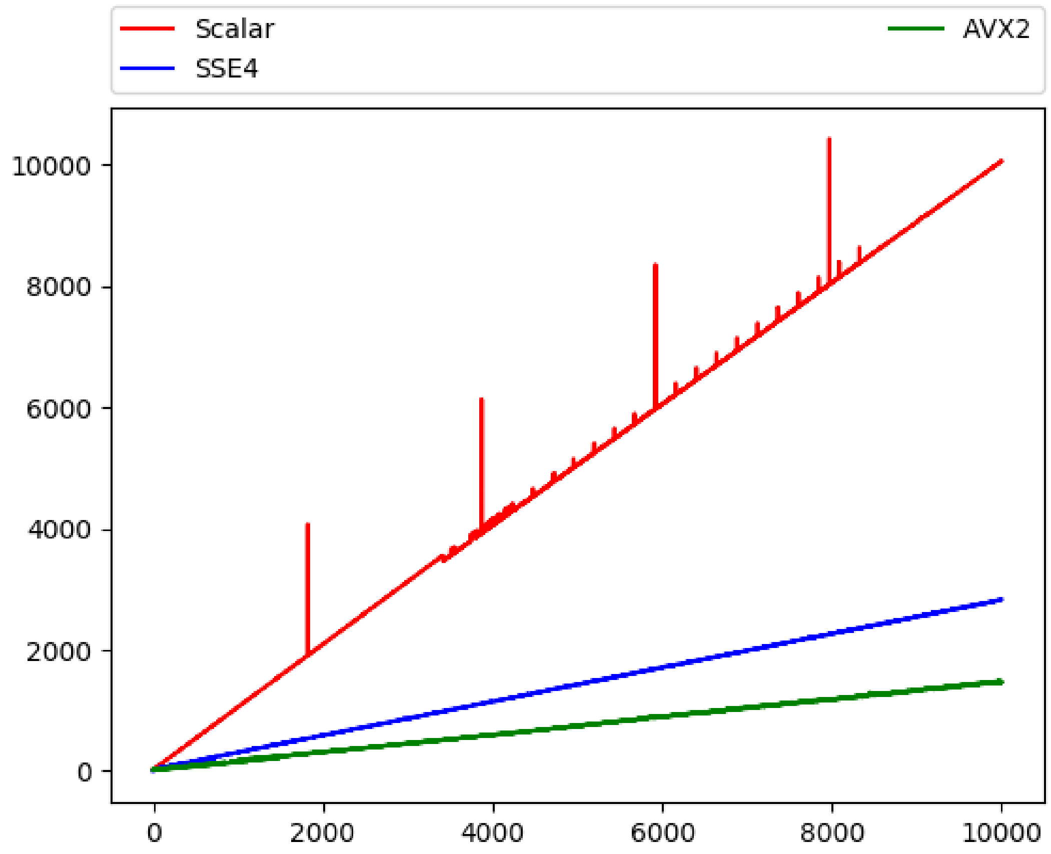


**Figure 3.3:** *-O1*

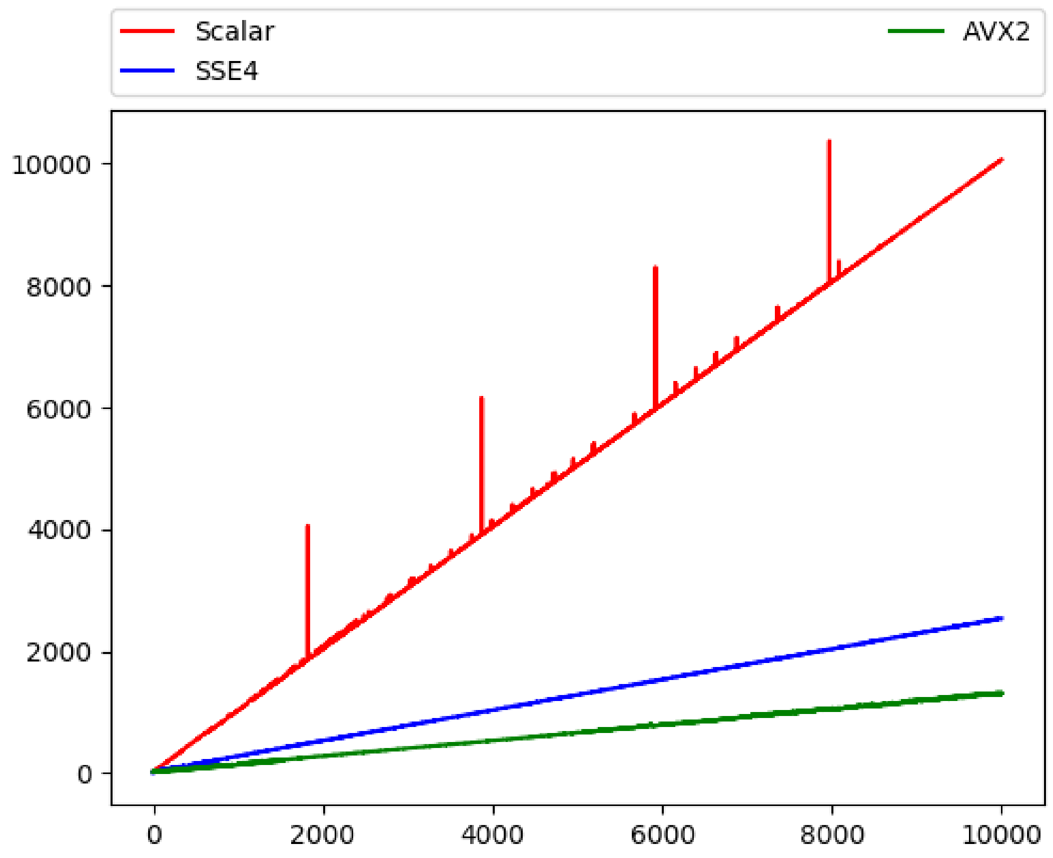

Scalar
SSE4
AVX2
10000
8000
6000
4000
2000
0
0
2000
4000
6000
8000
10000

**Figure 3.4:** *-O2*

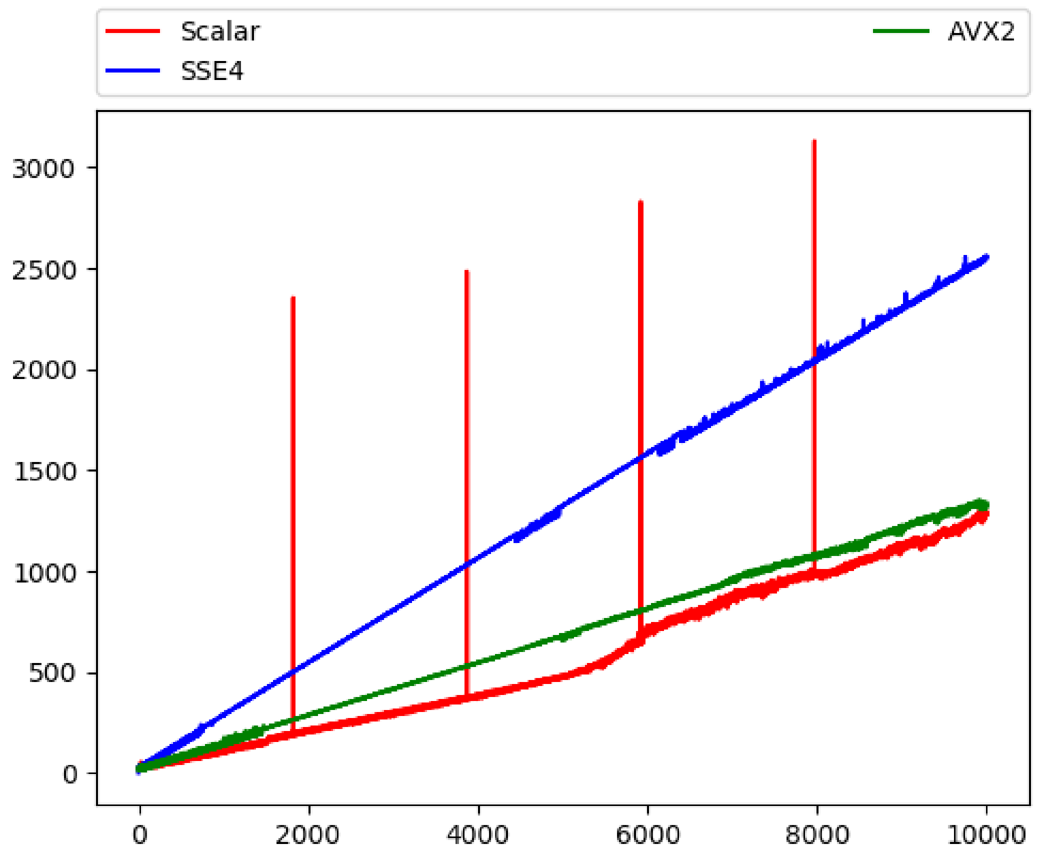
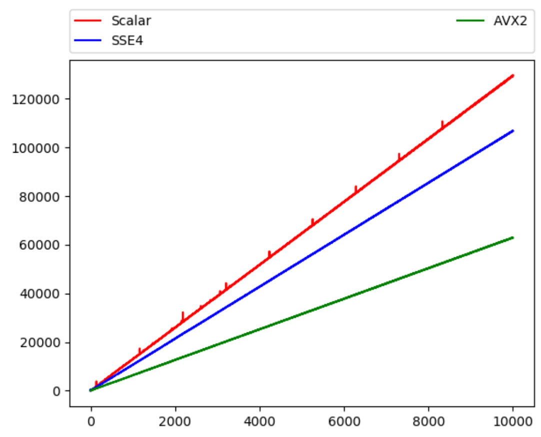


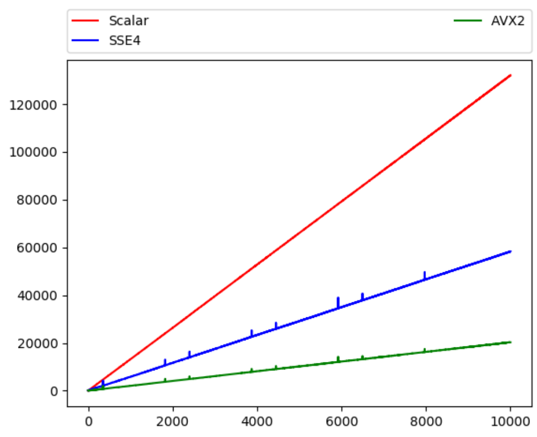
**Figure 3.5:** *-O3*

We see that optimizing the code always results in better performance (with an improving factor of around from 6 to 10).

However, the most interesting results comes with -O3. In fact we can see that performing loop unrolling makes the compiler also recognize that it is possible to use the SIMD instructions. This results in even better performances with respect to manual use of SIMD. The very high spikes at regular intervals in the scalar case suggests that cache misses that given the very high performances affects considerably the performances.

### 3.1.2 Complex numbers





In the second implementation of the complex multiplication, the algorithm suffers a lot in the scalar case. So the advantage of using SIMD is much more evident. However this implementation should not taken into consideration because it is very unlikely to have array of complex numbers as two array with real and imaginary part.

As a general comment, the complex number multiplication can take advantage of the SIMD, however this advantage is less evident then with real number. This is probably due to the overhead given from ordering and packing the data before and after computing the multiplication.