Eurecom

CompMeth

Assignment 2: Simple SIMD programming example

Author:
Martina Fogliato

Professor:
R. Knopp

April 11, 2018

Contents

1 Introduction

The aim of this laboratory session was to compare performances of vector operations against a scalar implementation with those obtained with SIMD instructions. In particular, we benchmarked instructions for SSE4 and AVX2 Intel processors, which are respectively 128-bit and 256-bit SIMD. I wrote the code for AVX512 as well (512-bit), which can be found in Section 4.
SIMD (Single-Instruction Multiple-Data) exploit data-level parallelism by performing the same operation on different data points at the same time, which makes them particularly suitable for vectorial and DSP operations. Therefore, the expected result is to have a speed-up in the execution of the instructions.
Routines have been timed with the provided *time_meas.h* functions as well as with *Callgrind* from *Valgrind*'s tool suite, in order to compute the speed-up with respect to the scalar case. Eventually further optimizations relative to the compiler have been added.

All source files are available on github.

2 Benchmarking without optimizations

2.1 Routines

The routine to be benchmarked performs a simple product between two vectors:

$$Z[n] = X[n] * Y[n], n = 0, ..., N-1$$

where X, Y and Z are Q15 numbers (on 16 bits).
Each implementation has been tested 1 million times, on vector lengths that go from `Nmin` to `N`, both user-supplied.
All tests have been run on my laptop with 8Gb of RAM and whose processor is a 7th generation Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz.

2.1.1 Scalar

For the scalar implementation I reused the `FIX_MPY` routine from the previous laboratory to multiply the elements of the arrays in a simple loop:

```
1  void componentwise_multiply_real_scalar(int16_t *x,int16_t *y,int16_t
   ↪  volatile *z,uint16_t N) {
2          int i;
3
4          for(i=0; i<N; i++){
```

```
                z[i] = FIX_MPY(x[i], y[i]);
        }
}
```

The modifier `volatile` on `z` is particularly useful for benchmarking with compiler optimizations, since it prevents the optimizations from being applied on the volatile object as it can change at any time. In this way the results will be more consistent.

2.1.2 SSE4

The SSE4 is a SIMD CPU instruction set by Intel which can operate on 16 bytes at a time, that is why the operands have been cast into an intrinsic Intel type `__m128i`.
Since I used `FIX_MPY` for the scalar case, I selected `_mm_mulhrs_epi16` for the SSE4 implementation. According to the Intel Intrinsics Guide *"__m128i _mm_mulhrs_epi16 (__m128i a, __m128i b), Multiply packed 16-bit integers in a and b, producing intermediate signed 32-bit integers. Truncate each intermediate integer to the 18 most significant bits, round by adding 1, and store bits [16:1] to dst."*. This means that it basically performs the same operations as `FIX_MPY`, except that it rounds the result instead of truncating it.

```
void componentwise_multiply_real_sse4(int16_t *x,int16_t *y,int16_t
↪  volatile *z,uint16_t 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]);
        }
}
```

2.1.3 AVX2

AVX2 (Advanced Vector eXtensions) extends operations on 256 bits, meaning that it can operate on 32 bytes at a time. Operands are cast to `__m256i` intrinsic type and multiplied through the function `_mm256_mulhrs_epi16`.

```
void componentwise_multiply_real_avx2(int16_t *x,int16_t *y,int16_t
↪  volatile *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++){
                z256[i] = _mm256_mulhrs_epi16(x256[i], y256[i]);
        }
}
```

2.2 Timing

2.2.1 Rdtscp

At first, I used the functions and the time structure provided in *time_meas.h* file in order to profile the critical portions of the above mentioned routines. They make use of TSC, the Time Stamp Counter, a 64-bit register present on x86 processors which counts the number of cycles since reset.
In the original file `rdtsc` instruction was used, but I replaced it with `rdtscp` instruction, as suggested by the Intel manual. In fact, the manual states: *"The RDTSCP instruction waits until all previous instructions have been executed before reading the counter."*, thus it partially solves the problem of instructions being executed out of order.
However, nothing prevents the Operating System from preempting the process when it needs to execute other routines. This is why, in Figure 1, we can see small drifts from a perfect straight-line behavior in the number of clock cycles, even if running a large number of tests does mitigate this effect.
I tried to declare the benchmarking routine as a thread, giving it the highest priority, but the performance did not change a lot.

The timer is reset at the beginning of each test (scalar, SSE4 and AVX2), it is started before calling a multiplication routine and then stopped when it ends, accumulating in this way the number of clock cycles.
Using MatLab, I plotted both the number of cycles vs. array length (`N`) for all three routines, and the speed-up with respect to the scalar case.
In Figure 1, due to the large amount of time required to run tests without optimization, I used 1000 as maximum value of N. As expected, the increase in the number of cycles is linear with respect to `N`.
Since SEE4 is working on 16 bytes at a time, the expected speed-up is 8x, but the actual value is around 7 and reaches the maximum at around `N`=4096, as from Figure 2. Instead,

AVX2 works with 32 bytes, thus should produce a 16x speed-up, but the actual value is around 13. The speed-up has been tested only for values of `N` which are power of 2, from 16 to 32768.

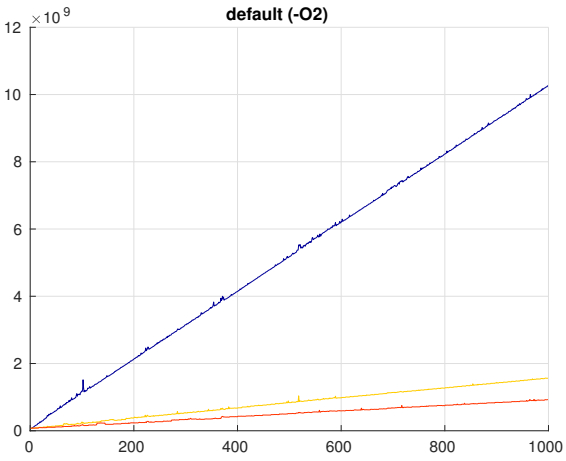


Figure 1: Clock cycles vs. N

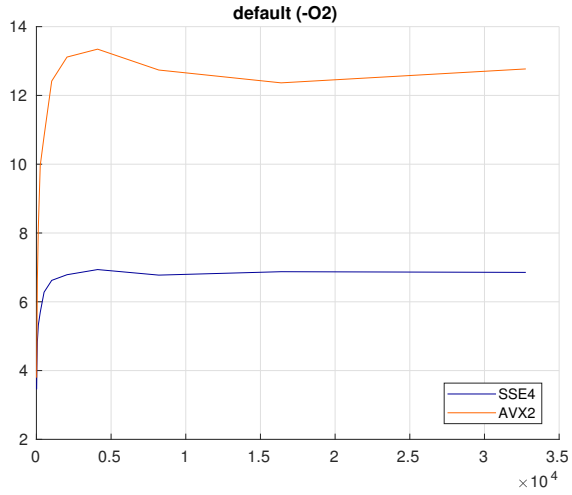


Figure 2: Speed-up -O2

2.2.2 Callgrind

Callgrind belongs to the *Valgrind* profiling tool suite and *"records the call history among functions in a program's run as a call-graph"*. It adds a huge time overhead to each test, thus I reduced the number of loop cycles from 1 million to 10000. As a graphical support to display the results, I used *kcachegrind*.
Figure 3 shows the three function calls and the time the program spends in each of them.

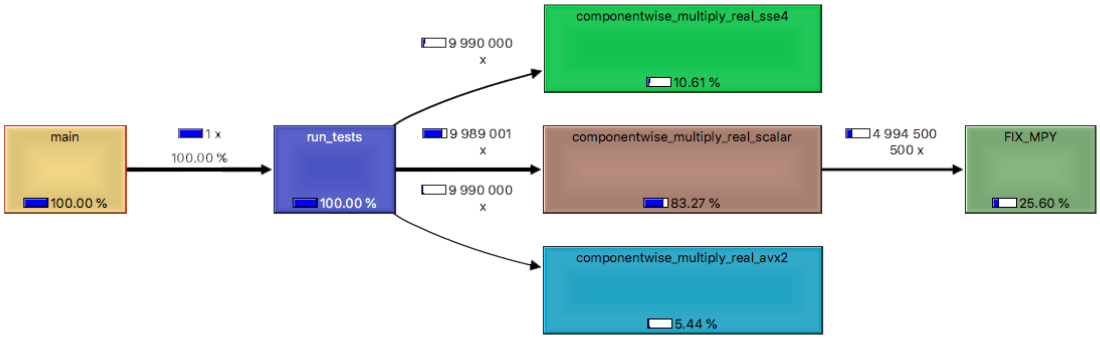


Figure 3: Callgrind results

2.3 Assembly analysis

The *gdb* command *disassembly* provides an efficient tool to display the assembly code relative to each function. I noticed that, if the flag `-mavx2` is used during compilation, the compiler automatically compiles SSE4 instructions with assembly instructions which are proper of the AVX set. Thus, in order to see the correct assembly code related to the SSE instruction set, I removed the flag when compiling the SSE4 routine.
A snippet from the SSE4 implementation is reported here:

```
Dump of assembler code for function componentwise_multiply_real_sse4:
 [...]
 0x0000000000000fce <+124>:        movaps %xmm1,-0x20(%rbp)
   0x0000000000000fd2 <+128>:        movaps %xmm0,-0x10(%rbp)
   0x0000000000000fd6 <+132>:        movdqa -0x10(%rbp),%xmm0
   0x0000000000000fdb <+137>:        movdqa -0x20(%rbp),%xmm1
   0x0000000000000fe0 <+142>:        pmulhrsw %xmm1,%xmm0
   0x0000000000000fe5 <+147>:        movaps %xmm0,(%rax)
   0x0000000000000fe8 <+150>:        addl   $0x1,-0x3c(%rbp)
   0x0000000000000fec <+154>:        cvtsi2sdl -0x3c(%rbp),%xmm2
   0x0000000000000ff1 <+159>:        movsd  %xmm2,-0x68(%rbp)
   0x0000000000000ff6 <+164>:        movzwl -0x5c(%rbp),%eax
   0x0000000000000ffa <+168>:        cvtsi2sd %eax,%xmm0
   0x0000000000000ffe <+172>:        movsd  0xe2(%rip),%xmm1
   0x0000000000001006 <+180>:        divsd  %xmm1,%xmm0
   0x000000000000100a <+184>:        callq  0x8d0 <ceil@plt>
   0x000000000000100f <+189>:        ucomisd -0x68(%rbp),%xmm0
   0x0000000000001014 <+194>:        ja     0xf8d
    ↪  <componentwise_multiply_real_sse4+59>
   0x000000000000101a <+200>:        nop
   0x000000000000101b <+201>:        leaveq
   0x000000000000101c <+202>:        retq
```

It is worth noticing the use of some additional 128-bit registers (xmm0, xmm1, xmm2, ...) and of SIMD instructions which are proper of the SSE instruction set, like `movdqa` and `pmulhrsw`.
The assembly code of the AVX2 implementation is exactly the same but it makes use of special 256-bit registers (ymm0, ymm1, ymm2, ...) and instructions like `vmovdqa` and `vpmulhrsw`, which are the 256-bit version of the previously mentioned ones.

3 Benchmarking with optimizations

3.1 O3

The default *gcc* optimization flag is `-O0`. Thus, I ran again all tests using `-O3`, which adds the following flags:

```
-finline-functions
-funswitch-loops
-fpredictive-commoning
-fgcse-after-reload
-ftree-loop-vectorize
-ftree-loop-distribution
-ftree-loop-distribute-patterns
-floop-interchange
-fsplit-paths
-ftree-slp-vectorize
-fvect-cost-model
-ftree-partial-pre
-fpeel-loops
-fipa-cp-clone
```

Among the other effects of `-O3`: it considers all functions for inlining, even if they are not declared inline, moves branches with loop invariant conditions out of the loop, and so on.

3.1.1 Timing

In this case, it is useless to perform an analysis with *Callgrind*, as function calls have been removed due to the fact it `-O3` forces inline functions.
The profiling of the different implementations has been executed with `rdtscp` and the speed up is e evident from Figure 5: the scalar case is more than 6 times faster, the SSE4 and the AVX2 implementation are almost twice as fast with respect to the `-O2` case.
The speed-up guaranteed by the SSE4 and AVX2 with respect to the scalar case under `-O3` optimization is shown in Figure 4.

3.2 Other optimization flags

Eventually, I ran the same tests adding some optimization flags which, by default, are not included in `-O3`:

- `-fselective-scheduling`: schedule instructions according to the selective scheduling, as an alternative to *gcc* default one

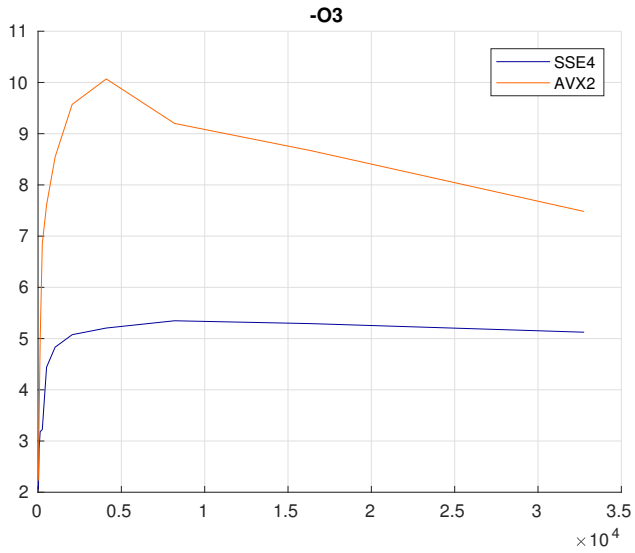


Figure 4: Speed-up -O3

- `-fsel-sched-pipelining`: enable software pipelining of innermost loops during selective scheduling
- `-ffast-math`: allows optimizations especially on floating point arithmetics, which do not preserve strict IEEE compliance
- `-fwhole-program`: Assume that the current compilation unit represents the whole program being compiled. All public functions and variables with the exception of main and those merged by attribute `externally_visible` become static functions and in effect are optimized more aggressively by interprocedural optimizers.
- `-floop-parallelize-all`: parallelize all the loops that can be analyzed to not contain loop carried dependences
- `-funroll-loops`: unroll loops whose number of iterations can be determined at compile time or upon entry to the loop
- `-faggressive-loop-optimizations`: performs aggressive analysis to derive an upper bound for the number of iterations of loops

3.2.1 Timing

Figure 5 shows a comparison between the default case compiled with `-O0`, the one compiled with `-O3` only and the one which includes additional flags together with `-O3`.
The additional flags cause a performance increase of about 20% for the scalar case, but nearly

no change for the SSE4 and the AVX2 implementation. This is visible from the speed-up plot as well (Figure 6).

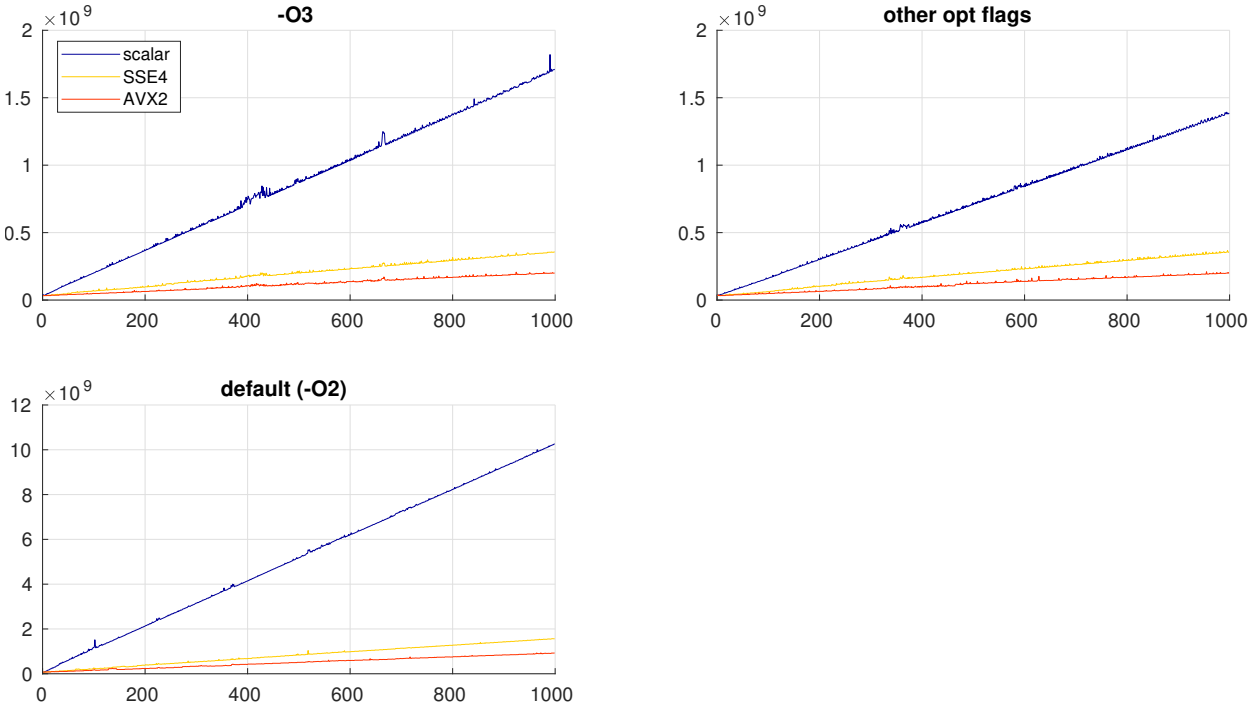


Figure 5: Comparison between optimization flags

3.2.2 Assembly analysis

The more "aggressive" optimization is visible from the assembly dump. As an example, the loop which initializes the vectors with random numbers has been unrolled:

```
0x0000000000000953 <+259>:      call   0x830 <rand@plt>
0x0000000000000958 <+264>:      mov    WORD PTR [r12],ax
0x000000000000095d <+269>:      mov    r15d,0x2
0x0000000000000963 <+275>:      call   0x830 <rand@plt>
0x0000000000000968 <+280>:      mov    WORD PTR [r13+0x0],ax
0x000000000000096d <+285>:      call   0x830 <rand@plt>
0x0000000000000972 <+290>:      mov    WORD PTR [r12+r15*1],ax
0x0000000000000977 <+295>:      call   0x830 <rand@plt>
0x000000000000097c <+300>:      mov    WORD PTR [r13+r15*1+0x0],ax
0x0000000000000982 <+306>:      add    r15,0x2
0x0000000000000986 <+310>:      call   0x830 <rand@plt>
```

However, the loops containing SIMD instructions are still not unrolled.

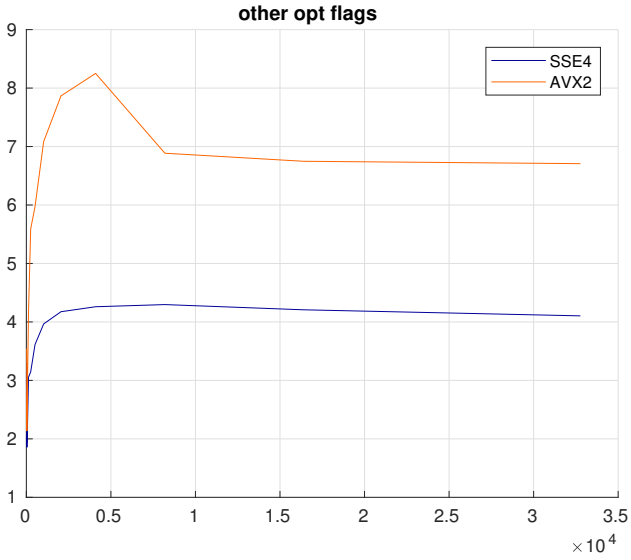


Figure 6: Speed-up with optimization flags

3.3 Loop unrolling

As previously said, what is worth noticing in the assembly dump is that, even with `-O3` options and all additional optimization flags set, the loops containing SIMD instructions are not unrolled. This is probably the reason why we do not see any dramatic change in the number of instructions per cycle for the SSE4 and AVX2 implementation between the default case and the others.
Thus, I modified the original code in order to partially unroll by hand the SIMD loops as follows:

```
1   for(i=0; i<ceil(N/8.0); i+=8){
2           z128[i] = _mm_mulhrs_epi16(x128[i], y128[i]);
3           z128[i+1] = _mm_mulhrs_epi16(x128[i+1], y128[i+1]);
4           z128[i+2] = _mm_mulhrs_epi16(x128[i+2], y128[i+2]);
5           z128[i+3] = _mm_mulhrs_epi16(x128[i+3], y128[i+3]);
6           z128[i+4] = _mm_mulhrs_epi16(x128[i+4], y128[i+4]);
7           z128[i+5] = _mm_mulhrs_epi16(x128[i+5], y128[i+5]);
8           z128[i+6] = _mm_mulhrs_epi16(x128[i+6], y128[i+6]);
9           z128[i+7] = _mm_mulhrs_epi16(x128[i+7], y128[i+7]);
10  }
```

The same has been done for the AVX2 implementation.

3.3.1 Timing

Plotting again the number of clock cycles for each value of N between 1 and 1000 (Figure 7), it is possible to see that all tests are overall faster with the manual unrolling.

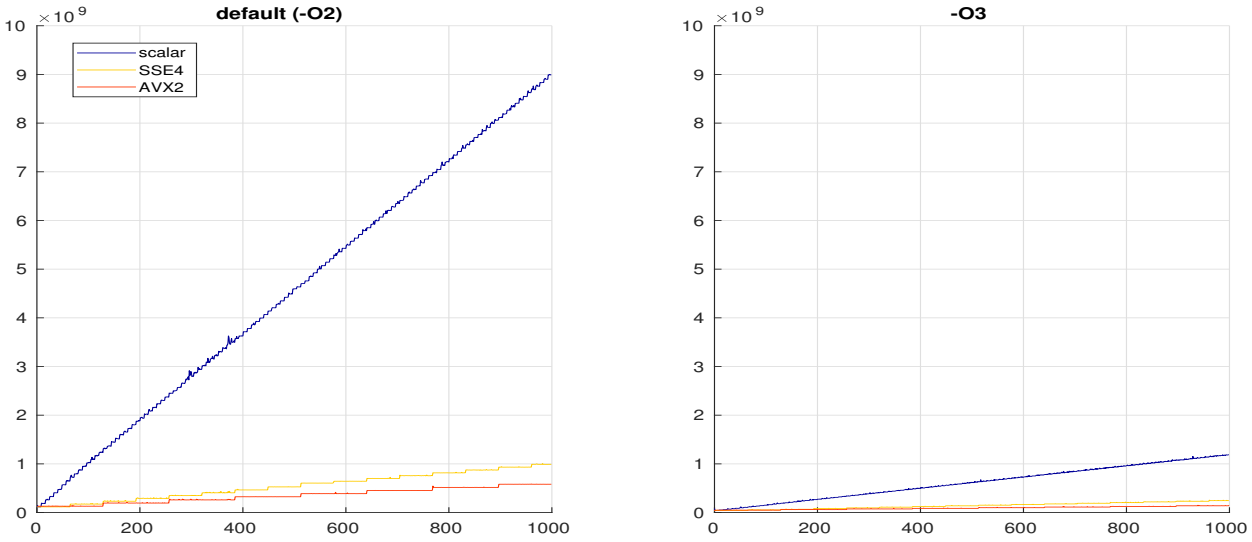


Figure 7: -O2 and -O3 tests with manual unrolling

4 AVX512

The AVX512 routine is similar to the previously described ones, but operands are cast to `__m512i` and `_mm512_mulhrs_epi16()` functions is used:

```
void componentwise_multiply_real_avx512(int16_t *x,int16_t *y,int16_t *z,
 ↪  uint16_t N) {

        __m512i *x512 = (__m512i *)x;
        __m512i *y512 = (__m512i *)y;
        __m512i *z512 = (__m512i *)z;
        int i;


        for(i=0; i<ceil(N/32.0); i++){
                z512[i] = _mm512_mulhrs_epi16(x512[i], y512[i]);
        }
}
```

My processor does not have any support for AVX512 instruction extension, so in order to perform benchmarking it should be tested on the machines available at Eurecom.

5 Complex numbers

5.1 Routines

I wrote my own routines to perform the same tests described above on complex numbers, for the scalar, the SSE4 and the AVX2 cases.
As before, I assumed to have two arrays of numbers to be multiplied, but, this time, each array is composed of numbers which have both real and imaginary part as Q15. Therefore, each array has the real part of the first number as first element, then its imaginary part as second element, then the real part of the second number followed by its imaginary part and so on.

X_{re}	Y_{re}
X_{im}	Y_{im}
X'_{re}	Y'_{re}
X'_{im}	Y'_{im}
...	...

5.1.1 Scalar

Considering two complex numbers $X = X_{re} + iX_{im}$ and $Y = Y_{re} + iY_{im}$, the mathematical expression which defines their multiplication is the following:

$$Z = (X_{re} * Y_{re} - X_{im} * Y_{im}) + i(X_{re} * Y_{im} + X_{im} * Y_{re})$$

Therefore, I simply implemented this expression using `FIX_MPY` function, filling the `z` array with the real part of the result first and the imaginary part after, both on 16 bits.

```
1 void componentwise_multiply_complex_scalar(int16_t *x,int16_t *y,int16_t
  ↪  volatile *z,uint16_t N) {
2  int i;
3
4  for(i=0; i<N; i+=2){
5
6    z[i] = FIX_MPY(x[i], y[i]);
7    z[i] = SAT_ADD16(FIX_MPY(x[i+1], -y[i+1]), z[i]);
```

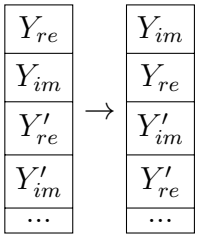
```
    z[i+1] = FIX_MPY(x[i+1], y[i]);
    z[i+1] SAT_ADD16(FIX_MPY(x[i], y[i+1]), z[i]);

  }
}
```

5.1.2 SSE4 and AVX2

The implementation for SSE4 and AVX2 is conceptually the same even if in the first case functions dealing with 128 bits are used while in the second case 256 bits are processed at a time. I will describe the SSE4 implementatio only, since the differences are minimal.
For the real part, I first changed the sign of all imaginary parts of `y` vector, using `_mm_sign_epi16` function and the following `reflip`:

```
const static int16_t reflip[32]  __attribute__((aligned(32))) = {1,-1,1,-1,...};
```

Then, I multiplied and added the result with the `x` vector.
The imaginary part, instead, in more complex. First, the `y` vector elements need to be shuffled in the following way:



and multiplied with the elements of `x`. At this point, the first 8 bytes of `z` contain all the real parts, while the following 8 bytes contain the imaginary parts. However, I want to interleave them, which can be accomplished with `_mm_unpacklo_epi32` and `_mm_unpackhi_epi32` for the lowest and the highest 128 bits respectively.
Then, to reproduce the same result obtained with `FIX_MPY` the numbers should be right-shifted by 15 positions and finally packed together in the `z` array.
The result may actually differ by one from that of the scalar case since `FIX_MPY` does not perform rounding.

```
void componentwise_multiply_complex_sse4(int16_t *x,int16_t *y,int16_t
 ↪  volatile *z,uint16_t N) {

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

```
6   int i;
7   __m128i z128_im, z128_tmp_lo, z128_tmp_hi;
8
9   for(i=0; i<ceil(N/8.0); i++){
10      //real part
11    z128[i] = _mm_madd_epi16(x128[i], _mm_sign_epi16(y128[i], *(__m128i
      ↪  *)reflip));
12
13      //imaginary part
14    z128_im = _mm_madd_epi16(x128[i],
      ↪  _mm_shufflelo_epi16(y128[i],_MM_SHUFFLE(2, 3, 0, 1)));
15    z128_tmp_lo = _mm_unpacklo_epi32(z128[i], z128_im);
16    z128_tmp_hi = _mm_unpackhi_epi32(z128[i], z128_im);
17    z128[i] = _mm_srai_epi32(z128_tmp_lo, 15);
18    z128_im = _mm_srai_epi32(z128_tmp_hi, 15);
19    z128[i] = _mm_packs_epi32(z128[i], z128_im);
20  }
21 }
```

5.2 Timing

Figure 8 and Figure 9 show the graphical results for the number of clock cycles and the speed-up obtained with complex number multiplication in all the cases previously explained: default, -O3 and additional gcc flags.
In this case, the number of tests performed for each N in not 1000000 anymore but 10000, due to the huge amount of time it takes to execute the complex multiplication.
N ranges from 0 to 2000 instead of 1000 since each number is composed by real and imaginary part, thus we need to double the size of the arrays to multiply the same number of elements as before.

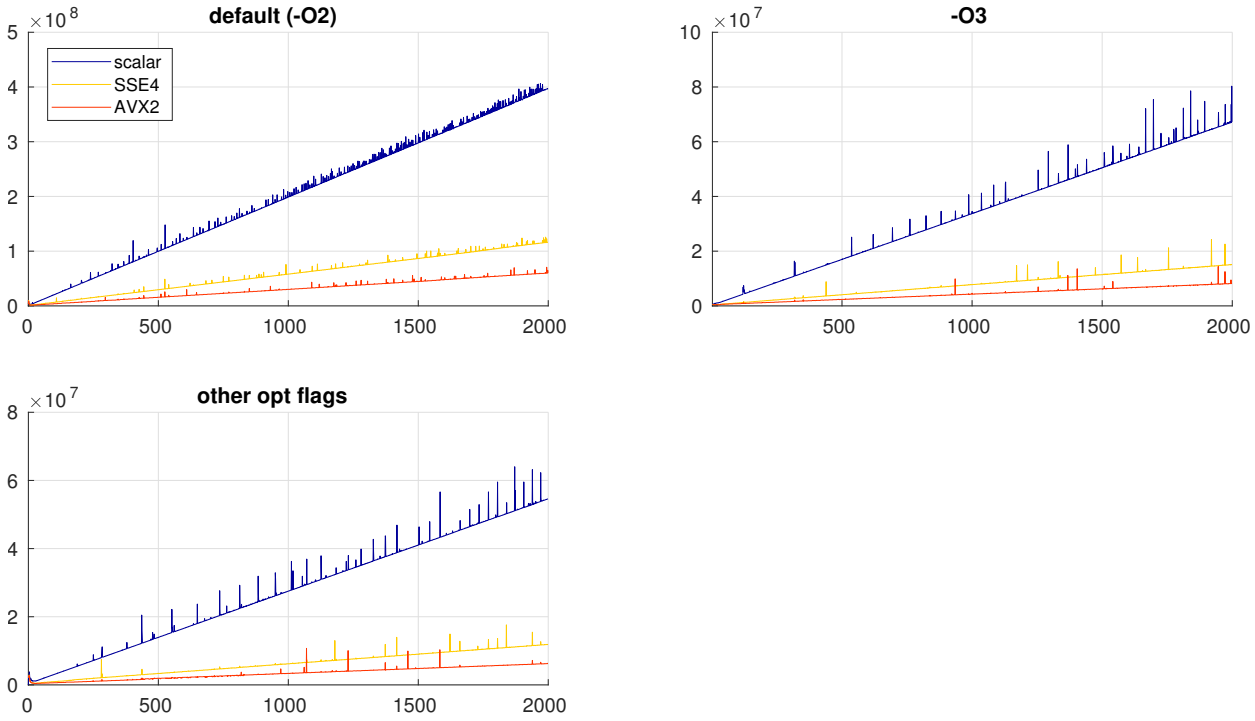


Figure 8: clock cycles vs N

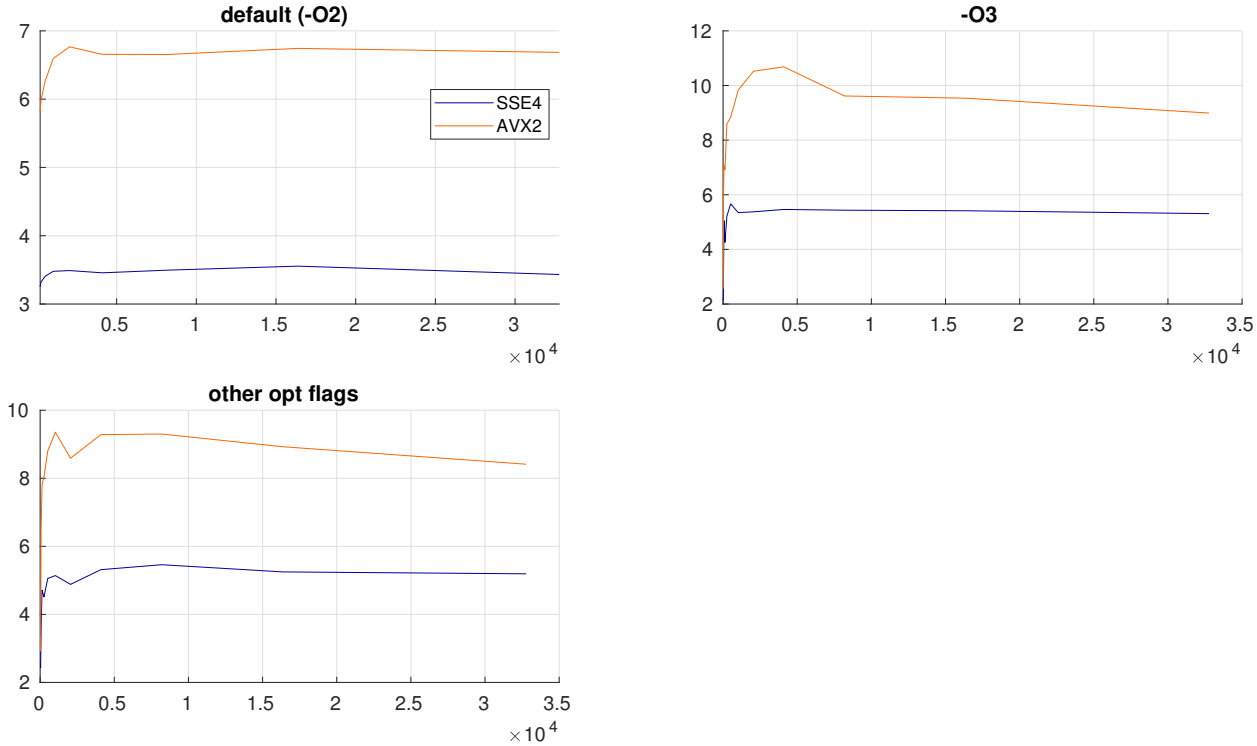


Figure 9: Speedup complex numbers