Experiments on Speeding Up the Recursive Fast Fourier Transform by using AVX-512 SIMD instructions

Giacomo Sansone and Marco Cococcioni

University of Pisa, Department of Information Engineering
Largo Lucio Lazzarino, 1 – 56122 Pisa (ITALY)
giacomo.sansone@yahoo.com
marco.cococcioni@unipi.it

Abstract. The *Fast Fourier Transform* is probably one of the most studied algorithms of all time. New techniques regarding hardware and software are often applied and tested on it, but the interest in FFT is still large because of its applications - signal and image processing, numerical computations, etc. In this paper, we start from a trivial recursive version of the algorithm and we speed it up using AVX-512 Single Instruction Multiple Data (SIMD) instructions on an Intel i7 CPU with native support to AVX-512. In particular, we study the impact of two different storage choices of vector of complex numbers: *block interleaving* and *complex interleaving*. Experimental results show that automatic vectorization provides a 10.65% ($\sim 1.12\times$) speedup, while with vectorization done by hand the speedup reaches 33.78% ($\sim 1.51\times$). We have made our code publicly available, which could be helpful for SIMD instructions teaching purposes.

Keywords: Recursive Fast Fourier Transform, SIMD instructions, AVX-512, complex number arithmetic, complex interleaving/block interleaving, memoization, automatic vectorization.

1 Introduction

FFT has been studied far and wide. Every year, new results about its implementation appear, boosting the speed of some of the most famous versions, such as FFTW [1]. Recently, NEC-SX Aurora Vector Engine has been used to test the behaviour of some FFT's implementations on large vector registers (256 double, 16384 bit per register) [2]. That is an important result for our work, since it pushes the usage of SIMD/vectorized architectures. Nevertheless, outside the world of High Performance Computing (HPC), the most available SIMD technology is the AVX-512 extension (see Section 3) which is spreading among x86 CPUs, both Intel and AMD. Hence in this work we focus on the latter, with the following contributions:

- we experiment a different and uncommon way to memorize complex numbers;
- we work on a manual vectorization of the FFT, keeping it simple and readable so that it can be used for teaching purposes;
- we measure the performances of different versions, pointing out the advantages of using AVX-512.

2 Different data layouts: an overview

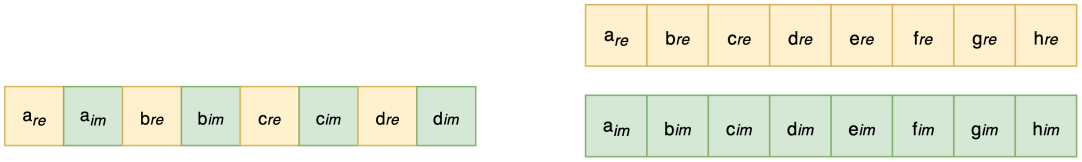
We can memorize complex numbers as pairs of floating point numbers, choosing the dimension (single/double precision) best suited for the needs of the computation at hand. Once we have a `complex` structure, by instantiating an array we obtain a sequence of numbers where the real and imaginary parts are staggered. We can call this kind of memorization *complex interleaved* (Figure 1a).

An alternative is to memorize separately the two components in two arrays of floating point numbers. This is called *block interleaved* memorization (Figure 1b): that is not common at all since existing software and standards for C/C++ only support the interleaved data format [3], but it could be useful dealing with vector registers and data gathering from memory.

Exploiting a mixture of these memorizations to boost the performance of algorithms on SIMD architectures has already been studied [3], achieving up to 2x performance improvements over state of the art library implementations. Our work will study and compare both types of memorization.

3 The AVX-512 instruction set

It has been a while now since computers have had some kind of SIMD extensions. SSE and AVX2 are fundamental in the history of this process, though their usage was limited by the length of their registers, respectively 128 and 256 bits. AVX-512 came out in 2013 as an improvement of the latter, introducing new instructions and providing registers 512 bits long.



(a) Complex Interleaved (b) Block Interleaved

Fig. 1: Different memorizations for array of complex numbers

Despite the speedup you can get from these instructions, AVX-512 is not always appropriate: it does not make IO-bound programs faster, as well as programs with complex conditional behaviours, since there are no parallel operations to execute; the tasks which can be boosted because of their parallelism regard AI, cryptography, mathematical computations... Programmers should understand where and when this extension could be useful, in order to gain a speedup which is independent from the algorithm itself.

Browsing the Intel's intrinsics guide[1] is a great starting point: this way one may familiarize with nomenclature and the different available instructions.

3.1 Loop unrolling

One of the main usage of SIMD instructions is through *loop unrolling*, which allows to avoid loops or diminish the number of iterations. For instance, suppose one need to compute an *Hadamard porduct* of two arrays of 8 `double`s each. Exploiting AVX-512, one can proceed in this way:

```
    // Declaration of the arrays
    double vector1[8];
    double vector2[8];
    double result [8];
    // Load the two arrays
    __m512d _vec1 = _mm512_load_pd((void*)vector1);
    __m512d _vec2 = _mm512_load_pd((void*)vector2);
    // Compute the sum using an AVX-512 intrinsic
    __m512d _res  = _mm512_add_pd(_vec1, _vec2);
    // Store the result
                    _mm512_store_pd((void*)result, _res);
```

In case the length of the arrays was greater than 8 (but, for simplicity, still multiple of 8) one could iterate this snippet $N/8$ times.

3.2 Superword Level Parallelism

SLP is another technique widely adopted by programmers and compilers to perform vectorization. It consists of gathering instructions which are similar but not directly linked, and computing them using SIMD registers. An example of this technique is shown in Section 5.2.

4 The recursive version of the FFT algorithm

Radix-2 algorithm is the simplest way to decrease the complexity of the DFT (Discrete Fourier Transform), from $O(N^2)$ to $O(NlogN)$, though nowadays FFT algorithms are thousands of lines of code long (they perform different operations based on different kinds of input and of the available hardware). Furthermore, an iterative version of the algorithm can be way more optimized than a recursive one, which is forced to use the stack an exponential amount of times.

Despite this, our work just aims to experiment with the operation of manual vectorization of the code; for this reason, we looked for an algorithm which is both interesting and useful in real-world applications: the recursive FFT is simple and follows the mathematical expression provided by Cooley and Tukey in 1965 [4]. It was easier to get into, but feasible enough to test AVX-512 capabilities and the two types of memorization as shown above. As most of the real-world use cases of the FFT, we will just consider input whose length is a power of 2.

[1] `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`

4.1 Twiddle factor: how memoization can help

The first thing we observed about the trivial version of FFT we used was the enormous usage of trigonometric functions to calculate the roots of the unity. For each call of the function on an input of length N, one have to use all the roots of order N, expressed as

$$e^{-j\frac{2\pi k}{N}} = \cos(\frac{2\pi k}{N}) - j\sin(\frac{2\pi k}{N}) \quad k \in \{0, 1, ..., N-1\}$$

The trigonometric functions are tremendously time-consuming for CPUs. We made a simple benchmark[2] of the FFT with an input of 2^{13} both using and avoiding the computation of these roots: the latter was two times faster than the former.

An immediate observation is that once one computed the roots of order N, the following calls of the function with an input of the same length can use them again. This technique is well known in literature as *twiddle factor* [5].

The needed roots can be both calculated before executing the algorithm or computed them on the way and saved for further use. That is what we did: we introduced a *look-up table* where we saved the results of the computation for N, so that we could access them later. The idea recalls the *memoization* of dynamic programming.

5 The vectorized version of the recursive FFT

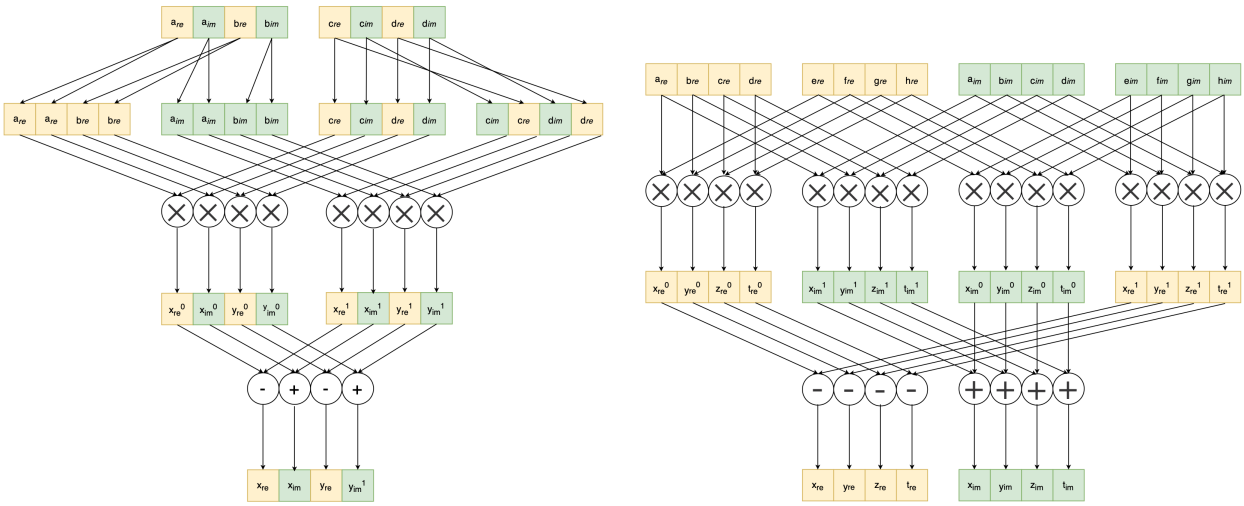
We made two different versions of the AVX-512 FFT, one with a complex-interleaved memorization (called CI_AVX), another with a block-interleaved memorization (called BI_AVX). The C++ source code of both the versions has been made public available and can be downloaded from this link: `https://github.com/pcineverdies/FFT-AVX-512`.

5.1 Link to Hadamard Product

The main element to vectorize the function is to exploit the radix-2 expression of the FFT: given an input X of size $N = 2^M$, its DFT is equal to

$$DFT(X)_k = DFT(X_e)_k + e^{-j\frac{2\pi k}{N}} \cdot DFT(X_o)_k \quad k \in \{0, 1, ..., N-1\}$$

where $DFT(X_e)$ is the DFT of the even terms of X, $DFT(X_o)$ is the DFT of its odd terms. As we compute these two elements using recursion, the result is made by the element-wise product between the vector of the roots and $DFT(X_o)$, added to $DFT(X_e)$. That is an interesting result, since element-wise product can be easily vectorized with both the memorizations of complex numbers. An intuitive idea of the process is shown in Figure 2 (that figure is inspired by one found in [3]).



(a) Complex Interleaved's case

(b) Block Interleaved's case

Fig. 2: Element-wise product using two memorizations.

[2] The machine we used has an Intel® Xeon®, 15 GiB of RAM and Ubuntu 4.15.0-171 as OS.

5.2 Base cases of recursion

As a consequence of the DFT's expression, when the input is a sequence of 4 or 2 elements everything can be brought back to additions and subtractions between real and imaginary parts of the input. That is why we can avoid a recursion up to a sequence of length 1 (the DFT of a number is the number itself), and stop at a length of 4. We also added the cases of input with a length of 2 and 1, which are stand-alone situations.

In this base cases, we tried to apply SLP, as shown in the snippet below (DFT of an input of length 4 in the array of complex `wave`, with CI memorization): the first block of instructions gathers data in a correct way, while the second one computes the additions/subtractions which give us the final result.

```
    // ...
    // _vecX are __m512d data
    if(n == 4){
        // Load of data
        _vec0 = _mm512_broadcast_f64x2(_mm_load_pd((double*)&(wave[0])));
        _vec1 = _mm512_broadcast_f64x2(_mm_load_pd((double*)&(wave[1])));
        _vec1 = _mm512_permute_pd(_vec1, 0b01100110);
        _vec2 = _mm512_broadcast_f64x2(_mm_load_pd((double*)&(wave[2])));
        _vec3 = _mm512_broadcast_f64x2(_mm_load_pd((double*)&(wave[3])));
        _vec3 = _mm512_permute_pd(_vec3, 0b01100110);

        // Compute DFT
        _vec0 = _mm512_mask_sub_pd(_vec0, 0b01111000, _vec0, _vec1);
        _vec0 = _mm512_mask_add_pd(_vec0, 0b10000111, _vec0, _vec1);
        _vec0 = _mm512_mask_sub_pd(_vec0, 0b11001100, _vec0, _vec2);
        _vec0 = _mm512_mask_add_pd(_vec0, 0b00110011, _vec0, _vec2);
        _vec0 = _mm512_mask_sub_pd(_vec0, 0b10110100, _vec0, _vec3);
        _vec0 = _mm512_mask_add_pd(_vec0, 0b01001011, _vec0, _vec3);

        // Store result
        _mm512_store_pd((void*)&wave[0], _vec0);

        return;
    }
    // ...
```

6 Experimental results

In the next subsection we provide the obtained numerical results, while in the following we discuss why, in our implementation, block interleaved does not give any additional speedup.

6.1 Numerical Results

We measured the performance of six versions of the FFT:

- NO_AVX, which is a standard version of the FFT, optimized with the twiddle factor, compiled with `O3` flag but without auto-vectorization;
- VE_AVX, which is the same as above, though auto-vectorization is enabled;
- CI_AVX, which is the version vectorized by hand with *complex interleaved* memorization, compiled with `O3` flag.
- BI_AVX, which is the version vectorized by hand and *block interleaved* memorization, compiled with `O3` flag.

In order to do that, we calculated how much time passed between the call of the function and its termination: after 2^{13} measures, we extracted the median of the data, which is more statistically stable than the arithmetic average.

Some of the results are shown in Table 1, while a complete overview for N between 2^3 and 2^{17} can be found in Figure 3. As it is evident from the numbers, the vectorized versions are more efficient the the standard one, by the **33.78%**($\sim 1.51\times$).

6.2 Why is *Block Interleaved* not good enough in our setting?

As we immediately notice from the result, the *block interleaved* version is quite similar to the *complex interleaved* one; in some cases it is even slightly slower. That is not what we expected: since this memorization method is not common, we would need a major speedup to use it.

Table 1: Execution time (μs) of FFT for some values of N

N	**NO_AVX**	**VE_AVX**	**CI_AVX**	**BI_AVX**
4096	976.0	952.0	717.0	730.0
8192	1955.0	1887.0	1415.0	1444.0
16384	2918.0	2784.0	2393.0	2312.0
32768	4506.0	4244.0	3168.0	3368.0
65536	8078.5	7778.5	5892.0	5489.5
131072	15409.0	13767.0	10203.5	10379.5

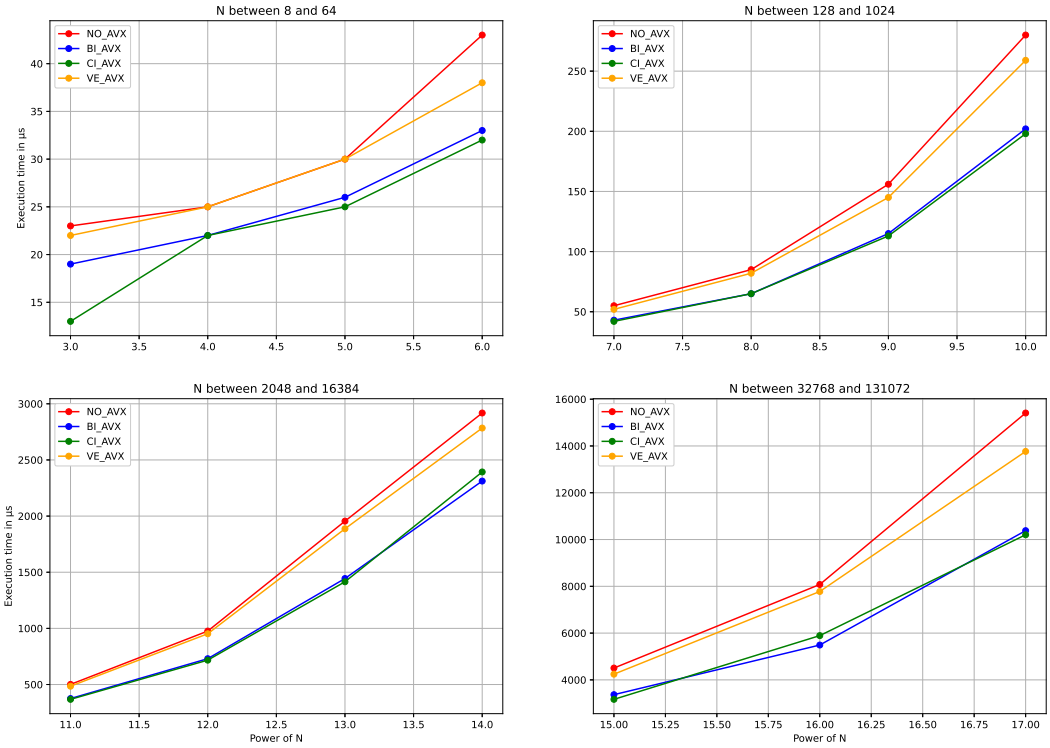


Fig. 3: Charts of the measures

In [3] the authors use a *mixed* version of the two methods: they start from a CI input, then they use the computations of the algorithm itself to get a BI memorization (which makes some operations faster, such as the element-wise product, since it reduces the usages of slow instructions as permutations) and they finally end up with a CI result. Instead, since we start directly from a block interleaved version, we are note able to replicate their speedups when using BI.

7 Conclusions

The final result of our experiments is a speedup of a **33.78%**($\sim 1.51\times$) between the first trivial version and the vectorized one. The automatic vectorization reaches a much lower speedup, which amounts to **10.66%**($\sim 1.12\times$). We have shown that BI memorization, while being not common and not compatible with standards like POSIX, does not provide any advantage over CI.

We would like to point out the importance of the AVX extension for programs which aim to achieve efficiency and speed. Clearly, writing our own vectorized code is not the best way to exploit this functionality, since we could make mistakes and it becomes difficult to maintain: the right approach should be to ask the compilers to introduce the functionality mentioned in the previous sections, suggesting some choices using `pragmas`.

In the end, the result could be not satisfying enough: in that case the programmer can disassemble the compiled code and try changing some instructions to speedup the program. And that is why it is important to be familiar with this extension. This is what we have learnt in this study. As a future work, we will:

- extend the code to exploit multi-threading, using the recently introduced C++ standard class for multi-threading;
- realize a port on CPU clusters [6];
- investigate how to optimize the impact on cache hierarchies [7].

Acknowledgments Work partially supported by H2020 project TEXTAROSSA (grant no. 956831), `https://textarossa.eu/`) and partially by the Italian Ministry of Education and Research (MUR), CrossLab project (Departments of Excellence). We also want to thank Prof. Carlo Vallati for providing the machine used to run the experiments and Emanuele Ruffaldi for interesting discussions on the topic.

References

1. M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
2. P. Vizcaino, F. Mantovani, and J. Labarta, "Accelerating fft using nec sx-aurora vector engine," in *Euro-Par 2021: Parallel Processing Workshops*, R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, and L. Ricci, Eds. Cham: Springer International Publishing, 2022, pp. 179–190.
3. D. T. Popovici, F. Franchetti, and T. M. Low, "Mixed data layout kernels for vectorized complex arithmetic," in *2017 IEEE High Performance Extreme Computing Conf. (HPEC)*, 2017, pp. 1–7.
4. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
5. W. M. Gentleman and G. Sande, "Fast Fourier Transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, p. 563–578. [Online]. Available: https://doi.org/10.1145/1464291.1464352
6. D. Sharp, M. Stoyanov, S. Tomov, and J. Dongarra, "A more portable HeFFTe: Implementing a fallback algorithm for Scalable Fourier Transforms," in *2021 IEEE High Performance Extreme Computing Conf. (HPEC)*, 2021, pp. 1–5.
7. D. Takahashi, *High-Performance FFT Algorithms*. Springer Singapore, 2020, pp. 41–68. [Online]. Available: https://doi.org/10.1007/978-981-13-9965-7_6