FFT optimizations and performance assessment targeted towards satellite and airborne radar processing

1st Maron Schlemon
Microwaves and Radar Institute
German Aerospace Center (DLR)
Oberpfaffenhofen, Germany
maron.schlemon@dlr.de

2nd Jamin Naghmouchi
Next Generation Computing
Innovationsgesellschaft TU Braunschweig mbH
Braunschweig, Germany
j.naghmouchi@itubs.de

***Abstract*—Following the re-invention of the FFT algorithm by Cooley and Tukey in 1965, a lot of effort has been invested into optimization of this algorithm and all its variations.**

In this paper, we discuss its use and optimization for current and future radar applications, and give a brief survey on implementations that have claimed relatively high advantages in terms of performance over existing solutions.

Correspondingly, we present an in-depth analysis of state-of-the-art solutions and our own implementation that will allow the reader to evaluate the performance improvements on a fair basis. Therefore, we discuss the development of a high-performance Fast Fourier Transform (FFT) using an enhanced Radix-4 decimation in frequency (DIF) algorithm, compare it against the Fastest Fourier Transform in the West (FFTW) auto-tuned library as well as other solutions and frameworks.

***Index Terms*—FFT, Radix-4 DIF, High Performance Algorithms, Parallel Computing, Embedded Computing, SAR, Chirp Compression**

I. Introduction

Over the last decades the number of earth-observation satellites in orbit has been constantly growing, as well as the spacecraft payload complexity and computational demand continuously have increased. Today, satellites provide close to full earth coverage and produce a significant amount of data that needs to be downlinked to Earth for processing. Modern synthetic aperture radar (SAR) systems are continuously developing towards higher spatial resolution [1] . Hence, downlink constraints combined with the constantly growing data throughput of missions require faster data handling, processing, and transfer. Therefore, the demand for onboard generation of final image products and/or compression techniques keeps increasing. Similar facts are valid for airborne systems, whereas those additionally might have to store large data takes onboard. Present on-board processing solutions show constraints regarding computational performance, size, and transfer speeds.

Thus, we have investigated the acceleration of processing routines for onboard processing systems for different algorithms such as the (extended-) omega-k algorithm (EOK), for which the FFT turned out to be one of the limiting factors, since it is used multiple times on large amounts of data. The extended-omega-k algorithm is used for radar focusing with/without integrated motion compensation(moco). More generally, the FFT is used in radar chirp compression algorithms, where the compression kernel consists of three subroutines: range FFT, multiplication by a reference function (range focusing) and finally an inverse FFT.

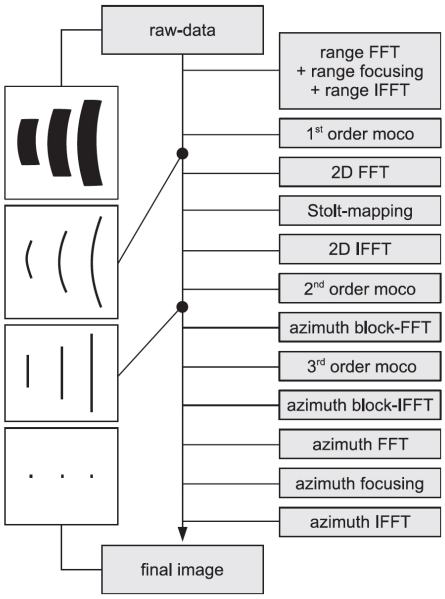

Fig. 1. EOK Algorithm [1]

The FFT is well known as the fast/efficient way of calculating the Discrete Fourier Transform (DFT), an essential tool for spectrum analysis and for facilitating the computation of discrete convolution and correlation. It is widely employed e.g. in filtering algorithms, polynomial multiplication, fast algorithms for discrete sine and cosine transforms (used e.g. in JPEG or MPEG encoding), solving difference equations, and other numeric applications.

Due to its widespread use, many derivative algorithms of the FFT such as the Radix-N, Split-Radix, Mixed-Radix, and prime factor FFTs have been developed ever since [2]–[7], [15]. However, this paper focuses on the improvement of the Radix-4 DIF algorithm for the reason of computational efficiency in the presents of vector instructions, as it will be described in Sections II and III. Existing frameworks are described in Section IV and the performance of this implementation is then compared in depth against existing frameworks in Section V.

II. The enhanced Radix-4 Algorithm

Like all fast FFT algorithms, the Radix-4 reduces the computational complexity of an N-point DFT by decomposing the original data. Many papers have shown that depending on the data size, the Radix-4 compared to other Radix-N or Mixed-Radix implementations is more efficient. [9]- [11] [15]. The derivation of the Radix-4 algorithm can be found in [2]. Our optimizations include

- the separation of the input data into its real and imaginary parts to make vectorization easier and to reduce unnecessary overhead,
- loop resolution in order to split the Radix-4 algorithm into its stages and individual optimization of each stage,
- restructuring the butterfly in order to reduce the computational effort by reusing already calculated data,
- pre-calculation and pre-vectorization of the twiddle factors and providing them to the Radix kernel and finally,
- pre-identification of elements that must be swapped in the reordering stage in order to eliminate unnecessary swaps.

We explore these optimizations in the following sections. The matrix representation in (1a) illustrates the calculations of a Radix-4 kernel.

$$\begin{bmatrix} X^F(k) \\ X^F\left(k+\frac{N}{4}\right) \\ X^F\left(k+\frac{N}{2}\right) \\ X^F\left(k+\frac{3N}{4}\right) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \begin{bmatrix} A^F(k) \\ B^F(k) \\ C^F(k) \\ D^F(k) \end{bmatrix}, \quad (1)$$

$$k = 0, 1, ..., N-1, \forall N = 4^n, n \in \mathbb{N}$$

$$A^F(k) = \sum_{n=0}^{\frac{N}{4}-1} x(n) W_N^{kn}, \quad (1a)$$

$$B^F(k) = W_N^k \sum_{n=0}^{\frac{N}{4}-1} x(n+1) W_N^{kn}, \quad (1b)$$

$$C^F(k) = W_N^{2k} \sum_{n=0}^{\frac{N}{4}-1} x(n+2) W_N^{kn}, \quad (1c)$$

$$D^F(k) = W_N^{3k} \sum_{n=0}^{\frac{N}{4}-1} x(n+3) W_N^{kn}, \quad (1d)$$

$$W_N^{kn} = \exp(-j\frac{2\pi}{N}kn). \quad (2)$$

$A^F(k), B^F(k), C^F(k)$ and $D^F(k)$ are the $N/4$ point DFTs, where the twiddle factors are given by W_N^{kn}. $x(n)$ is a sampled sequence of the data (signal), and X^F is the k-th DFT coefficient.

A. Stages of the Radix-4

In an iterative process the initial data size N is quartered. The number of stages is implicitly given by n, each with its own range R where

$$R_{Stage} = N_{Stage}/4. \quad (3)$$

The range defines the iteration limit of each stage as a result of the decomposition of the data, which is shown in Table I for a 4096-point Radix-4 FFT. The vector sizes are discussed in detail in Section III. Since the Radix-4 output is in bit-reversed order [2], a reordering step is required. The VEC-FFT uses a scalar function that iterates through the data array and calculates the bit-reversal in order to swap the corresponding values. Section II-C describes optimization techniques for increasing the reordering efficiency.

TABLE I
Size breakdown for $N = 4096$-point FFT

Stage	Elements N	Range R	Vector Size (Single Precision)
1	4096	1024	8 ymm (AVX)
2	1024	256	8 ymm (AVX)
3	256	64	8 ymm (AVX)
4	64	16	8 ymm (AVX)
5	16	4	4 xmm (SSE)
6	4	1 (4)	4 xmm (special)
Reordering	-	2016	scalar reorder

B. Optimizations of the Radix-4 Kernel

Setting $k = 0$ in 1a allows for a simple view of the optimizations implemented in the proposed VEC-FFT. The first optimization technique is to split the complex data into its real and imaginary parts, which results into two butterflies. The real- and imaginary-valued butterflies have (respectively) the following forms

$$\begin{bmatrix} X_{re}^F(0) \\ X_{re}^F\left(\frac{N}{4}\right) \\ X_{re}^F\left(\frac{N}{2}\right) \\ X_{re}^F\left(\frac{3N}{4}\right) \end{bmatrix} = \begin{bmatrix} A_{re}^F & B_{re}^F & C_{re}^F & D_{re}^F \\ A_{re}^F & B_{im}^F & -C_{re}^F & -D_{im}^F \\ A_{re}^F & -B_{re}^F & C_{re}^F & -D_{re}^F \\ A_{re}^F & -B_{im}^F & -C_{re}^F & D_{im}^F \end{bmatrix}$$

$$\begin{bmatrix} X_{im}^F(0) \\ X_{im}^F\left(\frac{N}{4}\right) \\ X_{im}^F\left(\frac{N}{2}\right) \\ X_{im}^F\left(\frac{3N}{4}\right) \end{bmatrix} = \begin{bmatrix} A_{im}^F & B_{im}^F & C_{im}^F & D_{im}^F \\ A_{im}^F & -B_{re}^F & -C_{im}^F & D_{re}^F \\ A_{im}^F & -B_{im}^F & C_{im}^F & -D_{im}^F \\ A_{im}^F & B_{re}^F & -C_{im}^F & -D_{re}^F \end{bmatrix}.$$

Having the data in split-complex format simplifies vectorization using SIMD instructions because the data can be directly loaded and processed. Using the standard format given in (1a) requires an extraction of the real- and imaginary-valued vectors, causing an avoidable but relatively

small instruction overhead.

The second technique consists of rearranging the butterflies and reusing pre-calculated values from look-up-tables. For example, the sum $A^F_{re} + C^F_{re}$ can be reused instead of calculating it twice, as shown in (4).

$$\begin{bmatrix} A^F_{re} & B^F_{re} & C^F_{re} & D^F_{re} \\ A^F_{re} & B^F_{im} & -C^F_{re} & -D^F_{im} \\ A^F_{re} & -B^F_{re} & C^F_{re} & -D^F_{re} \\ A^F_{re} & -B^F_{im} & -C^F_{re} & D^F_{im} \end{bmatrix}$$

$$\begin{aligned} \text{sum}^F_{\text{re}}(AC) &= A^F_{\text{re}} + C^F_{\text{re}} \\ \text{sum}^F_{\text{re}}(BD) &= B^F_{\text{im}} + D^F_{\text{im}} \\ \text{sub}^F_{\text{re}}(AC) &= A^F_{\text{re}} - C^F_{\text{re}} \\ \text{sub}^F_{\text{re}}(BD) &= B^F_{\text{im}} - D^F_{\text{im}} \end{aligned}$$

The result is an efficient structure of the Radix-4 kernel in (1a).

$$\begin{bmatrix} X^F_{\text{re}}(0) \\ X^F_{\text{re}}\left(\frac{N}{4}\right) \\ X^F_{\text{re}}\left(\frac{N}{2}\right) \\ X^F_{\text{re}}\left(\frac{3N}{4}\right) \end{bmatrix} = \begin{bmatrix} \text{sum}^F_{\text{re}}(AC) + \text{sum}^F_{\text{re}}(BD) \\ \text{sub}^F_{\text{re}}(AC) + \text{sub}^F_{\text{re}}(BD) \\ \text{sum}^F_{\text{re}}(AC) - \text{sum}^F_{\text{re}}(BD) \\ \text{sub}^F_{\text{re}}(AC) - \text{sub}^F_{\text{re}}(BD) \end{bmatrix} \tag{4}$$

This analogously applies to the imaginary-valued butterfly:

$$\begin{aligned} \text{sum}^F_{\text{im}}(AC) &= A^F_{\text{im}} + C^F_{\text{im}} \\ \text{sub}^F_{\text{im}}(BD) &= B^F_{\text{re}} - D^F_{\text{re}} \\ \text{sub}^F_{\text{im}}(AC) &= A^F_{\text{im}} - C^F_{\text{im}} \\ \text{sum}^F_{\text{im}}(BD) &= B^F_{\text{re}} + D^F_{\text{re}} \end{aligned}$$

$$\begin{bmatrix} X^F_{\text{im}}(0) \\ X^F_{\text{im}}\left(\frac{N}{4}\right) \\ X^F_{\text{im}}\left(\frac{N}{2}\right) \\ X^F_{\text{im}}\left(\frac{3N}{4}\right) \end{bmatrix} = \begin{bmatrix} \text{sum}^F_{\text{im}}(AC) + \text{sum}^F_{\text{im}}(BD) \\ \text{sub}^F_{\text{im}}(AC) - \text{sub}^F_{\text{im}}(BD) \\ \text{sum}^F_{\text{im}}(AC) - \text{sum}^F_{\text{im}}(BD) \\ \text{sub}^F_{\text{im}}(AC) + \text{sub}^F_{\text{im}}(BD) \end{bmatrix} \tag{5}$$

C. Reorder Stage Optimizations

The output of a Radix-4 DIF FFT is stored in bit-reversed order. For the sake of brevity, Table II illustrates the bit-reversal of an 8-point (Radix-2) FFT [2]. To restore the original order, a reordering stage is required that swaps elements (1,4) and (3,6). The proposed VEC-FFT uses a pre-calculated and FFT size specific reordering look-up-table in order to identify the necessary swaps. Following this approach, the number of swaps for the 4096-point example given in Table I is 2016. In order to increase the performance, we unrolled the reordering loop. For our test system (Table III), an unrolling factor of four showed the best performance.

TABLE II
REORDER PROCEDURE FOR AN 8-POINT FFT

Element	Original order	Bit-reversed order	Swap
0	000	000	No
1	001	100	Yes
2	010	010	No
3	011	110	Yes
4	100	001	Yes
5	101	101	No
6	110	011	Yes
7	111	111	No

D. Pre-Calculation of Twiddle Factors

In a similar manner, the twiddle factors are calculated in advance in order to make them available for execution. Due to their symmetry properties, one can consider reusing previously calculated twiddle factors as suggested in [8]. If $N = 8$ is assumed, then

$$\begin{aligned} W_8^{(1)} = W_8^{(1+N/2)} &= \exp\left(j\frac{2\pi}{8}\cdot 1\right) = \exp\left(j\frac{2\pi}{8}\cdot 5\right) \\ &= 0.707106 - j0.707106 \end{aligned}$$

Equivalent to (4) and (5), it is more efficient to split the twiddle factors into their real and imaginary valued parts. In the next section, we discuss the vectorization of the enhanced Radix-4 algorithm.

III. VECTORIZATION USING INTEL'S VECTOR INTRINSICS

Single instruction multiple data (SIMD) instructions are implemented on Intel's general purpose processors in order to enhance performance. Only a single instruction is fetched and applied to a set of data (vectors). This approach is suitable for the split-complex Radix-4 Butterflies in (4) and (5). The following sections give an overview about the instructions used by the proposed VEC-FFT.

A. Streaming SIMD Extensions (SSE)

The Streaming SIMD Extensions use 128 bit (xmm) registers and process, e.g., four packed single precision (sp) or two double precision (dp) floating point values. Listing 1 shows the SSE instructions used by VEC-FFT. The first operation loads four contiguous sp floating point values into an xmm vector, while the second stores the register data back to the memory. The last three instructions are arithmetic vector addition, vector subtraction and element-wise vector multiplication of the vectors a and b [12], [13].

```
1 __m128 _mm_load_ps  (float const * mem_addr)
2 void   _mm_store_ps (float * mem_addr, __m128 a)
3
4 __m128 _mm_add_ps   (__m128 a, __m128 b)
5 __m128 _mm_sub_ps   (__m128 a, __m128 b)
6 __m128 _mm_mul_ps   (__m128 a, __m128 b)
```

Listing 1. Streaming SIMD Extensions (SSE): Loading 4 single precision values into the xmm (128 bit) registers

B. Advanced Vector Extensions (AVX)

Advanced Vector Extensions use larger registers or vectors of 256 bit (ymm). They can process, e.g., eight packed sp or four packed dp floating point values. ymm vectors are preferred because of their ability to process more data simultaneously. Listing 2 shows the AVX instructions used by VEC-FFT.

```
__m256 _mm256_load_ps (float const * mem_addr)
void _mm256_store_ps (float * mem_addr, __m256 a)

__m256 _mm256_add_ps (__m256 a, __m256 b)
__m256 _mm256_sub_ps (__m256 a, __m256 b)
__m256 _mm256_mul_ps (__m256 a, __m256 b)
```

Listing 2. Advaced Vector Extensions: Loading 8 single precision values into the ymm (256 bit) registers

C. Fused Multiply Add (FMA)

Fused Multiply Add (FMA) instructions are particularly efficient since they execute two operations simultaneously. For example, the _mm256_fmadd_ps() instruction multiplies packed sp floating-point elements of vectors a and b and adds the intermediate result to the packed elements in vector c [13], [14]. Listing 3 shows the FMA instructions used by VEC-FFT.

```
__m256 _mm256_fmadd_ps (__m256 a,__m256 b,__m256 c)
__m256 _mm256_fmsub_ps (__m256 a,__m256 b,__m256 c)
```

Listing 3. Fused Multiply Add (FMA) instructions used by VEC-FFT

The right column of Table I shows the vectorization structure along the stages of the VEC-FFT. The first four stages are efficient regarding the usage of the larger ymm vectors. At stage five a breakdown to the smaller xmm registers is required, since the range size is four. Particularly grave are stage six and the reordering stage, as they cannot be easily vectorized. The reason is that the introduced load and store operations are only applicable on contiguous data in memory. However, stage six executes arithmetic operations on adjacent values. A solution to this problem is described in [15], [16], which suggests generating the correct vectors through a vectorized (4,4)-xmm transpose. The VEC-FFT utilizes a comparable technique to build the vectors, however, instead of transposing, Intel's gather and set operations are implemented. As shown in Listing 4, they allow, by specifying the data elements, a non-contiguous or stride data access at the cost of being less efficient than the regular load/store operations introduced in Listing 1 and 2 [13], [14]. Therefore stage six has a special xmm structure while the reordering stage remains scalar.

```
__m128 _mm_set_ps (float e3, ..., float e0)
```

Listing 4. Set instruction for loading non-contiguous data

IV. Processing Strategy

An essential step for optimized algorithms is to determine the architecture of the given system in order to load the CPU specific code and set the compiler flags. For example, if AVX is supported and FMA is not, the FMA instructions have to be replaced by AVX. Furthermore, the compilation flag "-mfma" has to be removed. All relevant parameters are saved to an automatically generated configuration file which ensures that the appropriate FFT kernel (function) is loaded. The next step is to create a data size and direction (forward or backward) based FFT plan which loads the twiddle factors and other look-up-tables containing FFT size specific information like the ranges illustrated in Tab. I. Finally, the FFT can be calculated by calling the execution function that takes as input the generated plan and data in split complex format. This process is shown in List. 5.

```
#include 'cpuconfig.h'

vecfftplan p;
p = __make_vecfft_plan__(size, direction);

__execute_vecfft__(p, real_array, imag_array);
```

Listing 5. VEC-FFT Process

V. Test

We divided our tests into two sections. First, we compared our implementations with other solutions. Subsequently, we focused on an extensive test against FFTW, since it is widely used and well known to the science community [17]. Initially, we tested the VEC-FFT against FFTW's version 3.3.4 for a 4096- and 16384-point FFT. However, since most studies in the field of high performance FFTs have mainly focused on testing against the scalar version of FFTW [18]–[20], this paper compares both, FFTW's scalar as well as the SIMD implementation of version 3.3.8 with VEC-FFT. For this purpose, various tests that cover 64- to 16384-point **single thread** 1D FFTs were conducted. For the FFTW comparison, the performance was measured using the Read Time-Stamp Counter (RDTSC) instruction in Listing 6 which returns the number of CPU clock cycles since last reset (start). For each size, 10000 measurements were taken and statistically evaluated in order to eliminate the impact of the operating system and its processes. We have ascertained that 1000 or 10000 repetition are enough for a good statistical evaluation.

```
unsigned long long cycles()
{
unsigned int lo, hi;

__asm__ __volatile__ (
"rdtsc" : "=a" (lo), "=d" (hi));
return ((unsigned long long)hi << 32) | lo;
}

unsigned long long start, stop;

start = cycles();
fftwf_execute(p1);
stop = cycles();
```

Listing 6. Performance assessment

The tests were executed on an Intel processing units that offer AVX, AVX2 and FMA as shown in Table III. The program was compiled and tested several times using different compiler and optimization flags:

- -march=native -O2 -mfma -std=c99
- -march=native -O3 -mfma -std=c99

- -march=native -ftree-vectorize -O2 -mfma -std=c99
- -march=native -ftree-vectorize -O3 -mfma -std=c99

We discuss the best solutions in the following sections.

TABLE III
TEST SYSTEM

Processor	Name	Intel Core i7 8700
	Code Name	Coffee Lake
	Max TDP	65 W
	Cores	6
	Hyper Threading	Off
	L1 D cache:	32 KB
	L1 I cache:	32 KB
	L2 cache:	256KB
	L3 cache:	12288 KB
	Base frequency	3.2 GHz
Memory	Type	DDR4
	Capacity	64 (2x32) GB
	Operation	Dual Channel
	Frequenzy	2333 Mhz
	CL	15 Clocks
	tRCD	15 Clocks
	tRP	15 Clocks
	tRAS	36 Clocks
Operating System	Linux	
	Distribution	Ubuntu 18.04 LTS
	Version	18.04 LTS (64 bit)

A. Test Against Other Solutions

We have performed a total of three different tests. If the reference mentioned measured the performance in execution time, we also carried out a time measurement in order to be able to compare the results. The execution times of the proposed VEC-FFT represent the median of a measurement of 1000 repetitions. Since these tests depend on the test system, the results are only a superficial assessment.

1) A 65536-point FFT test: In our first test, we compared a 65536-point FFT with [21].

TABLE IV
65536-POINT FFT - MEASURED IN SECONDS

Reference	VEC-FFT
0.1181	0.0002

2) Test of multiple FFT sizes: In this test, we tested our VEC-FFT against [22].

TABLE V
MULTIPLE FFT SIZES - MEASURED IN SECONDS

Size	Reference	VEC-FFT
256	0.0025	10ˆ7
1024	0.0031	10ˆ6
4096	0.0077	5*10ˆ6
16384	0.0197	3*10ˆ5

TABLE VI
256-POINT FFT - MEASURED IN CLOCK CYCLES

Reference	VEC-FFT
2763	1248

3) A 256-point FFT test: In this test, we tested our VEC-FFT against [15] by measuring the clock cycles for 256-point FFT. The result is shown in Table VI Other comparisons can be considered with [18], [19].

B. Initial Test Against FFTW's Version 3.3.4

The configuration for the FFTW library version 3.3.4 is as follows:

- **Precision:**
  –enable-float
- **Performance mode:**
  FFTW offers four different performance modes depending on the planning effort. In this benchmark the FFTW_EXHAUSTIVE mode has been chosen to measure FFTW's peak performance. Since the planning time increases dramatically with the data size, FFTW offers the opportunity to save and import the generated plan to eliminate the planning effort. This tool is called "wisdom" and can be built as shown in Ls. 7.

```
/* Generate and export FFTW plan */
fftwf_plan p1;
p1 = fftwf_plan_dft_1d(num, inputarray,
  inputarray, FFTW_FORWARD, FFTW_EXHAUSTIVE);
fftwf_export_wisdom_to_filename(fftwplan);

fftwf_execute(p1);

/* For a new calculation the plan has to be
  imported only */
fftwf_plan p1;
fftwf_import_wisdom_from_filename(fftwplan);
p1 = fftwf_plan_dft_1d(num, inputarray,
  inputarray, FFTW_FORWARD, FFTW_EXHAUSTIVE);

fftwf_execute(p1);

```

Listing 7. FFTW's WISDOM mechanism

Next, the runtime in clock cycles of the VEC-FFT is compared to that of the FFTW for 4096-point and 16384-point FFTs, in Figs. 2 and 4, respectively. The plots show the cumulative distribution of the runtime. It can be seen that the VEC-FFT outperforms the FFTW by a factor of up to 3.42.

In order to guarantee similar outputs, the discrepancy in the FFT output between the two libraries in terms of magnitude and phase is addressed in Figs. 3 and 5 (again for 4096-point and 16384-point FFTs, respectively). It can be seen that the difference between the two outputs is negligible (in the order of $10e-5$), meaning that the two operations are interchangeable in the processing chain.

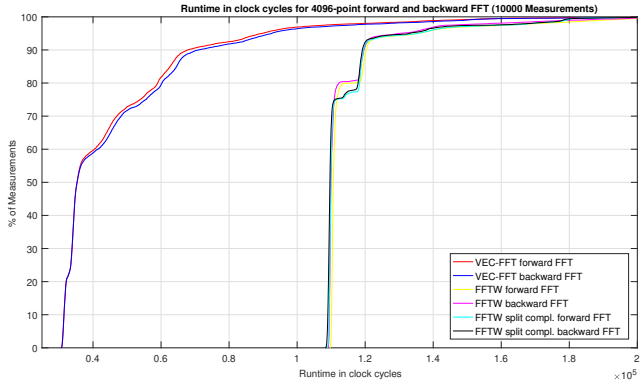

Fig. 2. Statistical performance evaluation for a 4096-point FFT

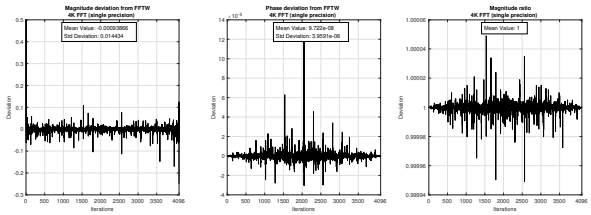

Fig. 3. Magnitude and phase deviation for a 4096-point FFT

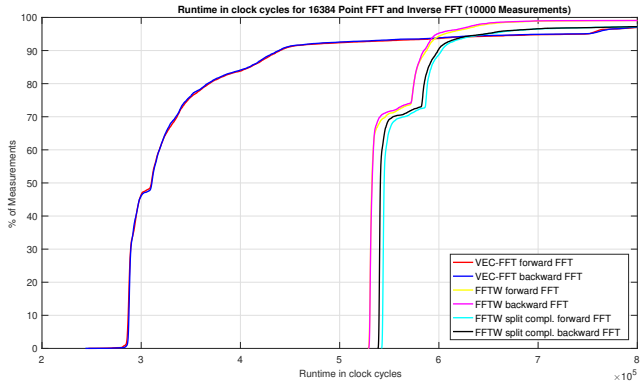

Fig. 4. Statistical performance evaluation for a 16384-point FFT

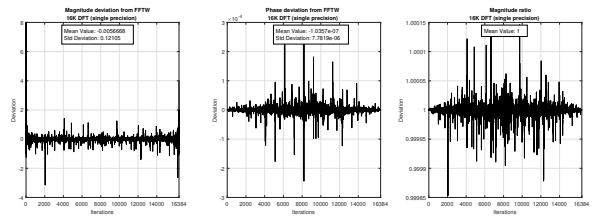

Fig. 5. Magnitude and phase deviation for a 16384-point FFT

C. Test Against FFTW's Latest Version 3.3.8

For FFTW version 3.3.8 we additionally used the following compilation flags in order to perform a SIMD test:

–enable-sse, –enable-sse2, –enable-avx, –enable-avx2, –enable-avx-128-fma, –enable-fma

A total of five tests were performed: our first compares the VEC-FFT with FFTW's scalar implementation of the FFT, whereas the second examines FFTW's SIMD version. The third explores VEC-FFT without reordering stage in order to measure its impact. The fourth evaluates Intel's *set* instructions which were used in the last stage of the Radix-4 algorithm in order to generate the vectors for the FFT kernel, as it is described in section III and Table I. Finally, the effect of the breakdown from the larger 256 bit ymm to the 128 bit xmm registers of stage 5 are analyzed.

Tables VII–IX show the performance results for each of the FFT (VEC-FFT, FFTW-scalar and FFTW-SIMD) the tests where MIN, MEDIAN, MEAN and SD represent the minimum, median, mean value and the standard deviation of the runtime in number of clock cycles, respectively.

1) First test: comparison to FFTW-scalar: Comparing Table VII with VIII, the first test shows that the proposed VEC-FFT calculates the FFT up to four times faster than FFTW's scalar version for the given FFT sizes. For a 4096-point FFT, the difference in performance decreases significantly by a factor of two. The reason behind this variation is the increase in the number of swaps which makes the reordering function more important. To measure its impact, a comparison between Table VII and Table X is considered. The following equations calculate the reordering effort E for the 4096- and 16384-point FFT:

$$\begin{aligned} E^{Median}_{reordering}(4096) &= E^{Median}_{FFT} - E^{Median}_{FFTNR} \\ &= 25442 - 17343 \\ &= 8099 \text{ clock cycles} \end{aligned} \tag{6}$$

$$\begin{aligned} E^{Median}_{reordering}(16384) &= E^{Median}_{FFT} - E^{Median}_{FFTNR} \\ &= 229619 - 90200 \\ &= 139419 \text{ clock cycles} \end{aligned} \tag{7}$$

While for a 4096-point FFT the reordering effort with 8000 clock cycles is about $1/3$, it takes with approx. 140000 clock cycles $2/3$ for a 16384-point FFT of the overall performance. This effect can also be seen in Fig. 6, where the FFT sizes are shown on the x-axis and the corresponding effort in clock cycles on the logarithmically scaled y-axis.

2) Second test: comparison to SIMD-FFTW: In turn, the second test compares the performance of VEC-FFT (Table VII) and SIMD-FFTW (Table IX). The result is that, up to a 4096-point FFT, the SIMD version of FFTW is about 30% more efficient than VEC-FFT. Especially for a 16384-point FFT, the difference in performance is a factor of two. As

TABLE VII
VEC-FFT PERFORMANCE IN CLOCK CYCLES

Size	MIN	MEDIAN	MEAN	SD
64	253	262	265	10
256	1207	1238	1248	87
1024	5428	5508	5625	308
4096	24974	25442	25722	891
16384	227400	229619	234061	11282

TABLE VIII
FFTW SCALAR PERFORMANCE IN CLOCK CYCLES

Size	MIN	MEDIAN	MEAN	SD
64	712	834	833	6
256	4128	4736	4741	211
1024	17388	21524	21028	1140
4096	84410	90437	93206	10283
16384	415294	445100	452253	33262

already mentioned, from this size on, the slow reordering function of the VEC-FFT has a strong impact explaining the gap to SIMD-FFTW.

TABLE IX
FFTW SIMD PERFORMANCE IN CLOCK CYCLES

Size	MIN	MEDIAN	MEAN	SD
64	302	313	318	62
256	965	987	1010	195
1024	3501	3758	3840	306
4096	18866	19428	19784	980
16384	101444	102974	106601	9411

3) Third test: VEC-FFT without reordering stage: Hence, the third test was conducted in order to compare the VEC-FFT without the reordering stage (VEC-FFT NR) with SIMD-FFTW. Considering Tables IX and X, it can be seen that VEC-FFT NR improved being now about 10% faster than SIMD-FFTW. Accordingly, Fig. 6 shows that the curve of VEC-FFT NR lies below SIMD-FFTW's curve.

TABLE X
VEC-FFT PERFORMANCE IN CLOCK CYCLES WITHOUT REORDERING

Size	MIN	MEDIAN	MEAN	SD
64	187	199	199	9
256	766	802	804	31
1024	3649	3712	3746	162
4096	17079	17343	17464	437
16384	88576	90200	91601	6222

4) Fourth test: impact of the breakdown to the smaller xmm registers: Table XI shows the result of the fourth test for a 4096-point FFT. The median performance for the first four ymm stages is around 9000 clock cycles, which translates into an average of 2250 clock cycles. The breakdown to the xmm register in stage five has a negative impact on the performance, causing an additional 1250 clock cycles.

5) Fifth test: impact of using Intel's set instructions: Furthermore, Table XI shows that the *set* instructions linked to the non-contiguous data access in stage six have a stronger impact on the performance, raising the total number of

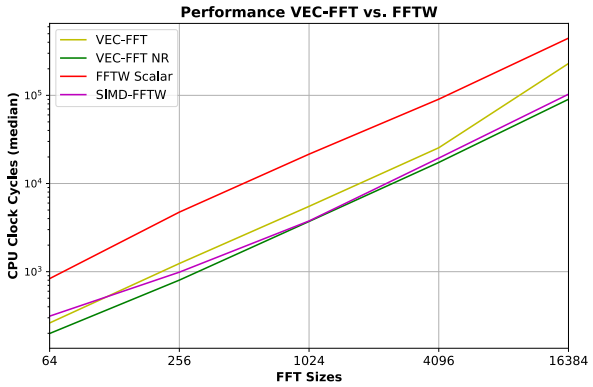

Fig. 6. Results for scalar and SIMD test

TABLE XI
VEC-FFT STAGE PERFORMANCES FOR A 4096-POINT FFT

Stage	Register	Clock cycles
1 - 4	8 ymm (AVX)	9004
5	4 xmm (SSE)	3585
6	4 xmm (SSE)	4754
Reordering	scalar	8099
Total	-	25442

required clock cycles to 4754.

In order to estimate the stability, the standard deviation can be considered. Stability can be understood as the ability of reproducing the median runtime performance. In this respect, the VEC-FFTW and FFTW show equivalent figures.

VI. CONCLUSION AND FUTURE WORK

This paper shows that performance optimization of FFT algorithms has not yet come to an end, although it is getting more advanced and competitive. Still, for embedded/onboard software computing routines almost every cycle matters due to very limited resources. Especially, for spaceborne or airborne platforms with complex payloads the demands for fast processing routines in software is constantly growing. These demands do not only concern the FFT but also other signal processing routines that depend on the scientific use case and are subject to changes during missions.

We have accelerated FFT processing using an optimized Radix-4 Algorithm that can performance-wise be compared against automatically tuned libraries such as FFTW. In order to achieve such high-performance goals, we have fully parallelized the FFT algorithm and computed the most efficient memory mapping for look-up tables for each iteration stage separately. Compared to FFTW version 3.3.4 the execution time could be improved by a factor of 1.83 to 3.42 through VEC-FFT, when neglecting side-effects of the operating system. Using the example of a 4096-point FFT, it can be observed that within the first stages, which allow for more efficient vectorization, the utilization of vector instructions is

at 96.7%, which proves that for these no memory bottleneck is present. However, the last two stages as well as the reordering of the Radix-4 algorithm show less efficiency. Therefore, our future work will concern the optimization of these less efficient stages by employing a mix of Radix algorithms similar to FFTW version 3.3.8.

REFERENCES

[1] A. Reigber, E. Alivizatos, A. Potsis and A. Moreira, "Extended wavenumber-domain synthetic aperture radar focusing with integrated motion compensation," in IEE Proceedings - Radar, Sonar and Navigation, vol. 153, no. 3, pp. 301-310, June 2006

[2] Rao, K. R., D. N. Kim, and J. J. Hwang. "Applications." Fast Fourier Transform-Algorithms and Applications. Springer, Dordrecht, 2010. 235-316.

[3] D. Kolba and T. Parks, "A prime factor FFT algorithm using high-speed convolution," in IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 25, no. 4, pp. 281-294, August 1977, doi: 10.1109/TASSP.1977.1162973.

[4] Z. Qian and M. Margala, "Low-Power Split-Radix FFT Processors Using Radix-2 Butterfly Units," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 24, no. 9, pp. 3008-3012, Sept. 2016, doi: 10.1109/TVLSI.2016.2544838.

[5] P. Rodriguez V, M. S. Pattichis and R. Jordan, "Computational SIMD framework: split-radix SIMD-FFT algorithm, derivation, implementation and performance," 2002 14th International Conference on Digital Signal Processing Proceedings. DSP 2002 (Cat. No.02TH8628), Santorini, Greece, 2002, pp. 861-864 vol.2, doi: 10.1109/ICDSP.2002.1028226.

[6] S. Ocovaj and Z. Lukac, "Optimization of conjugate-pair split-radix FFT algorithm for SIMD platforms," 2014 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, 2014, pp. 373-374, doi: 10.1109/ICCE.2014.6776047.

[7] B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang and N. Sun, "Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU," 2011 International Conference on Field-Programmable Technology, New Delhi, 2011, pp. 1-6, doi: 10.1109/FPT.2011.6132672.

[8] Lihong Jia, Yonghong Gao, Jouni Isoaho and Hannu Tenhunen, "A new VLSI-oriented FFT algorithm and implementation," Proceedings Eleventh Annual IEEE International ASIC Conference (Cat. No.98TH8372), Rochester, NY, USA, 1998, pp. 337-341, doi: 10.1109/ASIC.1998.723029

[9] R. Meyer and K. Schwarz, "FFT implementation on DSP-chips-theory and practice," International Conference on Acoustics, Speech, and Signal Processing, Albuquerque, NM, USA, 1990, pp. 1503-1506 vol.3, doi: 10.1109/ICASSP.1990.115692.

[10] R. Neuenfeld, M. Fonseca and E. Costa, "Design of optimized radix-2 and radix-4 butterflies from FFT with decimation in time," 2016 IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS), Florianopolis, 2016, pp. 171-174, doi: 10.1109/LASCAS.2016.7451037.

[11] Z. A. Abbas, N. B. Sulaiman, N. A. M. Yunus, W. Z. Wan Hasan and M. K. Ahmed, "An FPGA implementation and performance analysis between Radix-2 and Radix-4 of 4096 point FFT," 2018 IEEE 5th International Conference on Smart Instrumentation, Measurement and Application (ICSIMA), Songkla, Thailand, 2018, pp. 1-4, doi: 10.1109/ICSIMA.2018.8688777.

[12] Agner Fog, "Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs," Technical University of Denmark

[13] Intel Corporation, "Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual,"

[14] Intel Corporation, "Intrinsics Guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/#=undefined &techs=FMA&expand=2297,3924,4202,2607,2755,2553&text=256

[15] K. Nadehara, T. Miyazaki and I. Kuroda, "Radix-4 FFT implementation using SIMD multimedia instructions," 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No.99CH36258), Phoenix, AZ, USA, 1999, pp. 2131-2134 vol.4

[16] R. Al Na'mneh, W. D. Pan and R. Adhami, "Communication efficient adaptive matrix transpose algorithm for FFT on symmetric multiprocessors," Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST '05., Tuskegee, AL, USA, 2005, pp. 312-315, doi: 10.1109/SSST.2005.1460928.

[17] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," in Proceedings of the IEEE, vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.840301.

[18] P. Rodríguez V., "A radix-2 FFT algorithm for Modern Single Instruction Multiple Data (SIMD) architectures," 2002 IEEE International Conference on Acoustics, Speech, and Signal Processing, Orlando, FL, 2002, pp. III-3220-III-3223

[19] W. Xu, Z. Yan and D. Shunying, "A high performance FFT library with single instruction multiple data (SIMD) architecture," 2011 International Conference on Electronics, Communications and Control (ICECC), Ningbo, 2011, pp. 630-633

[20] D. Takahashi, "Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors," Innovative architecture for future generation high-performance processors and systems (iwia 2007), Maui, HI, 2007, pp. 53-59

[21] Z. Fang, J. Xiao and Y. Guo, "A High Performance Implementation of Ultra-long In-place FFT," 2019 6th Asia-Pacific Conference on Synthetic Aperture Radar (APSAR), Xiamen, China, 2019, pp. 1-4, doi: 10.1109/APSAR46974.2019.9048478.

[22] R. Al Na'mneh, W. D. Pan and R. Adhami, "Communication efficient adaptive matrix transpose algorithm for FFT on symmetric multiprocessors," Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST '05., Tuskegee, AL, USA, 2005, pp. 312-315, doi: 10.1109/SSST.2005.1460928.