Resource-Constrained Optimizations For Synthetic Aperture Radar On-Board Image Processing

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

2nd Martin Schulz
Technical University of Munich
Garching, Germany
martin.w.j.schulz@tum.de

3rd Rolf Scheiber
German Aerospace Center (DLR)
Oberpfaffenhofen, Germany
rolf.scheiber@dlr.de

***Abstract*—Synthetic Aperture Radar (SAR) can be used to create realistic and high-resolution 2D or 3D reconstructions of landscapes. The data capture is typically deployed using radar instruments in specially equipped, low flying planes, resulting in a large amount of raw data, which needs to be processed for image reconstruction. However, due to limited on-board processing capacities on the plane (power, size, weight, cooling, communication bandwidth to ground stations, etc.) and the need to capture many images during a single flight, the raw data must be processed on-board and then sent to the ground station efficiently as image products. In this paper we describe the processing architecture of the digital beamforming SAR (DBFSAR) of the German Areaospace Center (DLR) and the special steps that had to be taken to enable the on-board processing. We explain the required software optimizations and under which conditions their integration in the SAR imaging process leads to (near) real-time capability. We further describe the lessons learned in our work and discuss how they can be applied to other processing scenarios with limited resource availability.**

***Index Terms*—Synthetic Aperture Radar, High Performance Algorithms, Resource Constrained Processing, On-Board Radar Processing;**

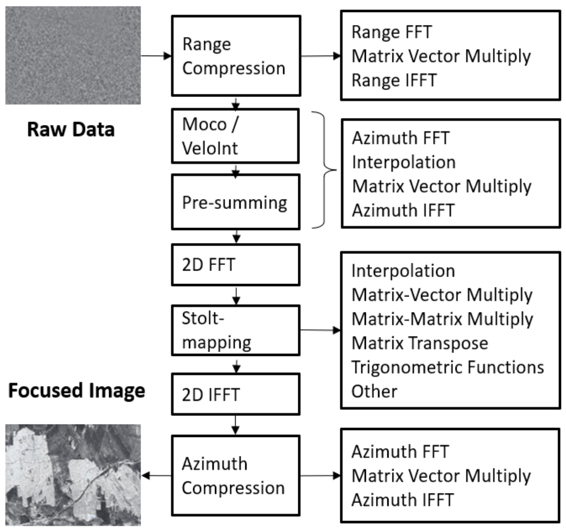


Fig. 1: Extended Omega-K algorithm

I. Introduction

Over the last decades, SAR systems have been extensively used for earth remote sensing. They provide high-resolution, light and weather-independent images for many applications, including climate change research, environmental and earth system monitoring, change detection or security-related applications like the Maritime Moving Target Indication (MMTI) [1] employed for monitoring the marine environment. Similar to a conventional radar, electromagnetic waves in the form of a series of short pulses are transmitted from a low-flying plane, back-scattered and finally collected by the receive antennas on the moving plane. The combination of the echo signals received over a period of time allows for the construction of a virtual aperture that is much longer than the physical antenna length. SAR images result from processing the raw data and are a measure of the scene's reflectivity [2]. The generation of these images involves a significant amount of handling, processing and data transfer. For example, a typical DBFSAR image has 32 k x 285 k complex-valued samples, which translate to around 70 GB of single precision float raw data.

SAR imaging routines are divided into time and frequency domain based algorithms. For an in-depth analysis of the subject, the interested reader is referred to the textbook on SAR signal processing by Cumming et al. [3]. In this paper, we investigate the acceleration of the Extended Omega-K (EOK) algorithm shown in Figure 1. It is a representative of the class of frequency-based algorithms and can be divided into five major processing steps: range compression, velocity interpolation, motion compensation, EOK kernel (2D FFTs and stolt mapping), and finally azimuth compression. According to Figure 1, the Fast Fourier Transforms (FFTs) and interpolations are the fundamental functions of this SAR imaging process, because they are repeatedly used on large amounts of data. The FFT is well known as an efficient way of calculating the Discrete Fourier Transform (DFT), an essential tool for spectrum analysis. It is widely employed, especially in filtering algorithms, and thus appears in almost every processing step of the frequency domain based SAR algorithms and hence dominates the overall processing performance.

In order to assess our development regarding real-time capability, we first derive an appropriate definition of real-time for SAR image processing. During a flight, the aircraft with the radar equipment needs to be re-positioned, realigned and readjusted after each image sequence in order to map the same, or potentially different, areas with another set of parameters. In this context, we then speak of a real-time capability, when SAR images can be produced within this time window of repositioning and realignment before the next image sequence starts. However, this only serves as a minimal requirement of the current state-of-the-art system. We expect that new operational airborne SAR sensors require significantly faster image processing. Achieving such real-time capabilities is far from trivial, especially as the equipment on board of planes is a) severely resource limited, restricting the computational capabilities and b) very static and typically limited to hardware available at the initial commissioning of the SAR radar itself, which has much longer life cycles than the IT equipment. The latter stems from the fact that airborne hardware cannot be effortlessly replaced or extended due to tedious and complex certification steps. Consequently, any application targeted for such environments must be optimized to fully exploit the underlying hardware, and embedded in an environment and workflow that can be efficiently executed on it.

In this paper, we introduce DLR's DBFSAR platform, as one of the state-of-the-art systems to capture airborne based SAR images and discuss its hardware setup, limitations, and the requirements that stem from it. We present a series of novel optimizations targeting both the computational kernels and the overall processing workflow. In particular, we make the following contributions:

- We optimize the computational kernels to make full use of the underlying hardware. We not only vectorize the core functions of the EOK algorithm but also align them with the register sizes, caches, and memory space of the given system. This precise alignment results in very high SIMD levels and, thus, we achieve an ideal hardware utilization rate.
- We efficiently distribute the fast kernels over many compute cores and processing boards to further parallelize the data processing.
- We further optimize the execution environment, data management and a multi-processor setup and show how this enables real-time SAR processing.

The results shown here are typical for edge-based applications, which not only require high-performance, but also have to work under the constraints typically found in remote or isolated platforms. Finally, we show the lessons learned and how our work can also benefit other application areas with limited resources.

II. Related Work

Several real-time processing approaches have addressed the issue of accelerating SAR algorithms using field programmable gate arrays (FPGAs) [9], digital signal processors (DSPs) [10], central processing units (CPUs) [11] and graphics processing units (GPUs) [12]. Le et al. [9] introduced an FPGA design that processes the SAR image using a Range-Doppler Algorithm (RDA) [4]–[6], which due to its use of approximations, is less accurate and less robust towards higher squint angles if compared to the proposed EOK algorithm [8]. In fact, a crucial problem in most airborne SAR sensors is the compensation of motion errors, induced by atmospheric turbulence and/or air pressure. If not compensated, the image resolution will considerably deteriorate. The motion compensation requires an enormous increase in computational effort.

Wang et al. [10] proposed a multi-core and memory optimized implementation of the RDA using a DSP. SAR raw data is usually stored in a row-major-order, which results in an efficient range processing. On the other hand, azimuth operations are slow because of non-sequential data access. Therefore, it is convenient to transpose the data after range processing in order to overcome this issue. For the same reason, Wang et al. [10] use a block-wise ping-pong DMA based corner turn that enables fast azimuth processing. Similarly, we implement an efficient vectorized transpose function that is executed prior to an azimuth operation.

Fatica et al. [12] accelerated the SAR processing by employing a GPU on a mobile platform. The use of a GPU in the context of SAR represents a reasonable solution because various functions in the SAR processing flow (e.g., matrix-vector multiply) can be highly parallelized in a SIMD fashion.

SAR processing works with complex data. Usually, libraries use the interleaved-complex data format which stores real and imaginary values as pairs sequentially in memory. As Popovici et al. [20] indicated by showing a vectorized complex multiply, this format comes with an avoidable overhead, which is why we split the complex format that stores real and imaginary data into separate memory spaces. Furthermore, it facilitates the implementation of SIMD instructions.

III. The DBFSAR System and Processing Units

As described in detail by Reigber et al. [21], the German Aerospace Center (DLR) developed the state-of-the-art advanced high-resolution airborne SAR system with digital beam-forming capabilities, known as DBFSAR, and uses it routinely to capture high-quality SAR-based images. It operates at X-band and features twelve simultaneous receive and four sequential transmit channels each with 1.8 GHz bandwidth, flexible DBF antenna setups. Furthermore, it is equipped with a high-precision navigation and positioning unit.

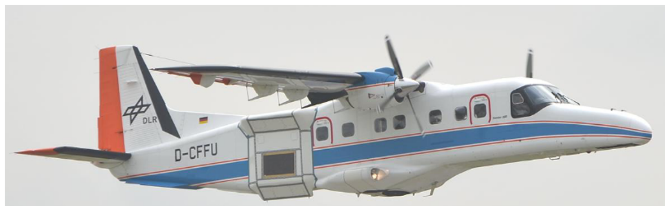


Fig. 2: DLR's aircraft with the SAR equipment (rectangle at the back of the plane).

The system is certified and mounted on an aircraft, as shown in an example in Figure 2. In addition, an LTE mobile network is installed below the aircraft's fuselage in order to transmit the on-board processed data to a ground station.

Figure 3 outlines the on-board processing system architecture used for SAR image reconstruction. It consists of three low-power, real-time processor boards (RTPs), each with an Intel central processing unit (CPU) of four cores and 16 GB of random access memory (RAM). The system was state-of-the-art at the time the aircraft was commissioned, and cannot be updated anymore due to regulatory and re-certification restrictions. We note that, if built today, other hardware may be installed (including the potential use of accelerators, as far as the low power constraints would allow), but the problems would ultimately be the same. Similar requirements exist in many other processing scenarios in which high-performance processing has to be executed on hardware with severely limited resources.

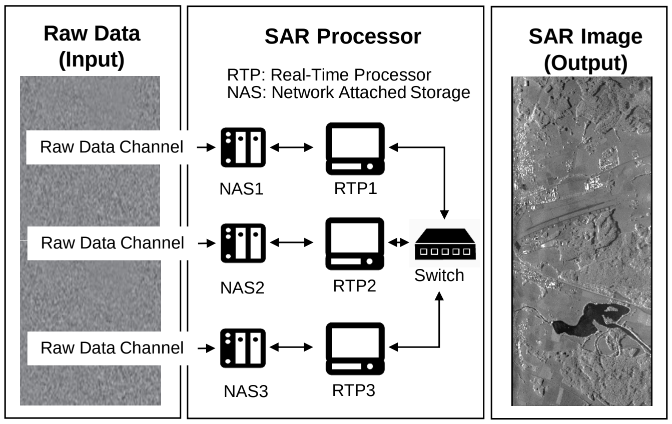


Fig. 3: System setup and simplified SAR imaging flow.

The system operates a network attached storage (NAS) that is connected to each RTP via Ethernet and using the Network File System (NFS) protocol. All RTPs communicate through a 10 Ethernet Gbit/s switch allowing fast data access with an average throughput of 750 MB/s. During the data acquisition step, the two dimensional raw data acquired from the receive antennas is written through the receive channels to the NAS drives. Optimally, all twelve CPU cores should be used for the SAR processing. However, one core per RTP is reserved for system and OS tasks. Table I displays the details about the processing hardware and specifies one of the development servers on the ground for DLR's SAR software. The general approach is to develop and test the SAR routines on the server first and then to port them to the on-board system. The supported vector instructions for the on-board system are based on the Streaming SIMD Extensions (SSE4.2) and the Advanced Vector Extensions (AVX). The development server additionally supports AVX2, FMA and AVX512 instructions.

To enable common but architecture independent development, we followed a generic code design that allows us to switch between different architectures efficiently. An example gives Listing 1 for the *_mm256_fmadd_ps* FMA instruction which is only supported by the development server. It multiplies and adds two single precision *ymm* vectors and saves the result in the register labeled as *dest*. This instruction can be translated into AVX instructions by executing *_mm256_add_ps* and then *_mm256_mul_ps*. We defined a new macro *fma_add()* that performs this operation depending on the supported instruction sets. In order for this to work, we generated a CPU configuration file that appends the appropriate instructions to the context.

```
#include "cpu_config.h"

#if FMA == TRUE
#define fma_add(dest, v1, v2, v3) {  \
 dest = _mm256_fmadd_ps(v1, v2, v3);}
#else
#define fma_add(dest, v1, v2, v3) {  \
 v1   = _mm256_mul_ps(v1, v2);       \
 dest = _mm256_add_ps(v1, v3);}
#endif
```

Listing 1: Generic Code Design

In our context, we defined real-time t_{rt_0} for the previously described DBFSAR system as a 5 minute delay using historical data. However, recent developments have led to a more stringent real-time definition of $t_{rt_{0,1}}$ equivalent to the data acquisition interval. Section IV discusses under which conditions real-time image processing regarding $t_{rt_{0,1}}$ is possible. For these investigations, we assume:

$$t_{rt_0} = 300s \tag{1}$$
$$t_{rt_1} = 90s \tag{2}$$

TABLE I: Hardware line for development and on-board system

	On-Board Hardware	Development Server
Processor		
Name	Intel® Core™ i7-3610QE	Intel(R) Xeon(R) Gold 6154
Code Name	Ivy Bridge	Skylake
Cores	3x4	2x18
Hyper threading	Off	Off
Base frequency	2.3 GHz	3.0 GHz
Supported instructions	SSE4.2, AVX	SSE4.2, AVX, AVX2, AVX512, FMA
Memory		
Type	DDR3	DDR4
Capacity	16384 (2x8192) GB	394 GB
Operation	Dual Channel	-
Frequency	1333 MHz	2400 MHz
Operating System		
Distribution	Linux / openSUSE	Linux / Ubuntu

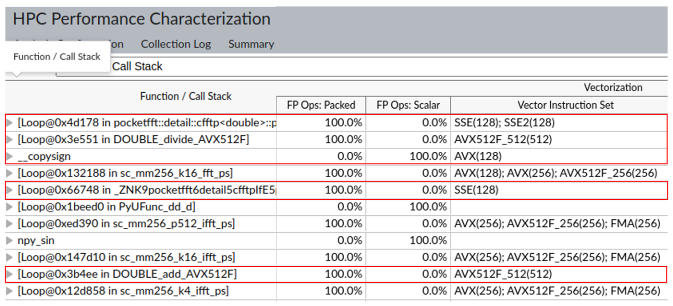
We note that the software was originally designed for the initial workloads during the aircraft's commissioning time. In the meantime, though, the application requirements and complexity have changed, increasing the workload, while the underlying system has to stay constant. In order to successfully deploy the SAR application on the given hardware, we therefore must increase the efficiency in processing.

IV. Compute Kernel Optimizations

In this section, we focus on the optimization of the computational kernels and their usage of the installed system hardware. In addition, we discuss optimizations regarding system and workflow usage, as well as parametric optimizations. The following tests were executed on the development server. We will provide an estimation for the on-board computer based on the findings.

A. Evaluation of the Base Model

Since SAR image processing is a complex task, we use DLR's already existing Python pipeline of the EOK for on-ground processing as our base model and investigate its performance towards real-time capability. For this, we measure the processing time of each function of the EOK blocks and additionally profile them by using Intel's VTune profiling software. This provided us with an in-depth insight into the given code. While the Python prototype uses many libraries, the most significant one is Numpy, which we label as *Numpy+*. This reference shows good performance as it already implicitly benefits from SIMD optimizations (as Figure 4 shows) and therefore offers a realistic baseline.


HPC Performance Characterization

Function / Call Stack | Collection Log | Summary

Call Stack

Function / Call Stack	Vectorization		
	FP Ops: Packed	FP Ops: Scalar	Vector Instruction Set
[Loop@0x4d178 in pocketfft::detail::cfftp<double>::p	100.0%	0.0%	SSE(128); SSE2(128)
[Loop@0x3e551 in DOUBLE_divide_AVX512F]	100.0%	0.0%	AVX512F_512(512)
__copysign	0.0%	100.0%	AVX(128)
[Loop@0x132188 in sc_mm256_k16_fft_ps]	100.0%	0.0%	AVX(128); AVX(256); AVX512F_256(256)
[Loop@0x66748 in _ZNK9pocketfft6detail5cfftpIfE5	100.0%	0.0%	SSE(128)
[Loop@0x1beed0 in PyUFunc_dd_d]	0.0%	100.0%	
[Loop@0xed390 in sc_mm256_p512_ifft_ps]	100.0%	0.0%	AVX(256); AVX512F_256(256); FMA(256)
npy_sin	0.0%	100.0%	
[Loop@0x147d10 in sc_mm256_k16_ifft_ps]	100.0%	0.0%	AVX(256); AVX512F_256(256); FMA(256)
[Loop@0x3b4ee in DOUBLE_add_AVX512F]	100.0%	0.0%	AVX512F_512(512)
[Loop@0x12d858 in sc_mm256_k4_ifft_ps]	100.0%	0.0%	AVX(256); AVX512F_256(256); FMA(256)

Fig. 4: Performance Profile for the *Numpy+* prototype using Intel VTune.

B. Vectorization Techniques to Enhance Processing Time

For each bottleneck that we identified, the corresponding function was outsourced into C code and manually SIMD optimized using Intel's Intrinsic Instruction Set Library (ISL). It has the potential to enable a high degree of parallelism by executing a single instruction on a set of data (vectors). The ISL is implemented on Intel's general purpose processors in order to enhance the processing performance. Especially, when the data is stored sequentially in memory, their use can be applied efficiently to create fast single core kernels (components). Thus, it provides a convenient solution to implement SIMD instructions [22]–[24].

While we describe the enhanced Radix-4 FFT as an efficient FFT algorithm in Schlemon et al. [19], in this paper we focus on our algorithmic and optimization contributions of the *sinc* interpolation as it is the dominant function in the velocity interpolation and stolt mapping. We show how we have aligned the algorithm to the hardware with a particular view on vectorization. For the sake of brevity, we only consider the single precision instructions labeled as *_ps*.

Equation 3 represents our version of the Whittaker–Shannon (*sinc*) interpolation. The intention here is to give an easy understanding of the subject and to avoid detailed descriptions and derivations for which interested readers are referred to Schanze et al. [25]. Equation 5 defines the number of input samples $x(n)$ that are loaded to calculate the interpolated value $y(x)$. In general, the larger N_H is, the higher the interpolation accuracy, but this also increases the computational effort which leads to a trade-off. Hence, the operator is able to set N_H depending on the pre-defined objective.

$$y(t) = \sum_{n=-N_H}^{+N_H} x(n)\, sinc(t-n) \tag{3}$$

$$sinc(n) = \begin{cases} \frac{\sin(\pi n)}{\pi n}, & \text{if } n \neq 0 \\ 1, & \text{if } n = 0 \end{cases} \tag{4}$$

$$N_H = 8, 16 \tag{5}$$

However, following the idea of the base algorithm is not sufficient when deploying it on modern hardware. We therefore introduce a series of novel steps that enable such optimizations. It is important to align N_H to the register size of the given CPU. The Intel i7-3610QE processor of the onboard system has a total of 16 *ymm* registers that can hold 8 single precision floating point values each. We set N_H as a multiple of 8 (e.g., 8, 16,...) in order to align with the *ymm* registers and thus reduce the scalar overhead. Figure 5 shows the SIMD scheme of the *sinc* interpolation for $N_H = 16$.

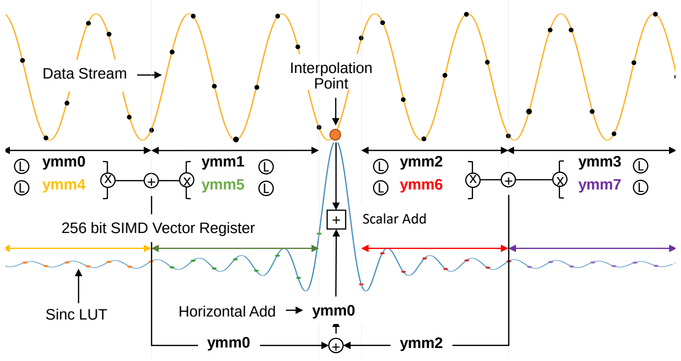


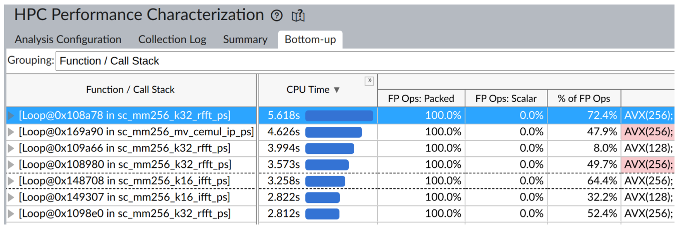
Fig. 5: SIMD scheme of the sinc interpolation with $N_H = 16$

The corresponding *sinc* vectors need to be generated by performing vectorized sine calculations. However, this is inefficient considering that we first have to calculate the argument of the *sinc* function according to Equation 3. The argument vector is then passed to the *_mm256_sin_ps* instruction in order to build the sine vectors. Intel points out that this instruction generates a sequence of instructions, which may perform worse than a native scalar operation. We solved this issue by creating a look-up-table that sequentially stores the relevant $2N_H$ sinc values to avoid the described overhead. This allows us to use the relatively fast load instruction *_mm256_load_ps* and, thus, we eliminate the argument and sine calculation overhead. Once the vectors ymm0-ymm7 are loaded into the registers, we perform the instructions shown in Listing 2. Since the dot product is not an elementary instruction in Intel's ISL, it needs to be manually developed, which results in a sequence of instructions and thus in a high computational effort. Therefore, we avoided the use of dot products and implemented a horizontal add function *hadd* that calculates the sum over a given vector efficiently. Our implementation of *hadd* is shown Listing 2. The program was compiled and tested with *-march=native -O2 -mfma -std=c99*. The FMA flag was automatically included for the server development. Apart from standard libraries, no external software was utilized.

```
float hadd(__m256 vec) {
 xmm0 = _mm256_castps256_ps128(vec);
 xmm1 = _mm256_extractf128_ps(vec, 1);
 xmm0 = _mm_add_ps(xmm0, xmm1);
 xmm1 = _mm_movehl_ps(xmm0, xmm0);
 xmm0 = _mm_add_ps(xmm1, xmm1);
 xmm1 = _mm_movehdup_ps(xmm0);
 xmm0 = _mm_add_ps(xmm0, xmm1);
  return _mm256_cvtss_f32(xmm0);}

ymm0 = _mm256_mul_ps(ymm0, ymm4);
ymm1 = _mm256_mul_ps(ymm1, ymm5);
ymm2 = _mm256_mul_ps(ymm2, ymm6);
ymm3 = _mm256_mul_ps(ymm3, ymm7);

ymm0 = _mm256_add_ps(ymm0, ymm1);
ymm2 = _mm256_add_ps(ymm2, ymm3);

ymm0 = _mm256_add_ps(ymm0, ymm2);

float intp = x[x_pos] + hadd(ymm0);
```

Listing 2: SIMD sinc interpolation

We used the Intel VTune Profiler to evaluate our developments in performance-critical contexts and compared it with the existing solution for the range compression. Figure 6 displays a vectorization snapshot of our implementations. By aligning the functions to the CPU registers, we achieve ideal vectorization levels and utilization rates. Figure 7 shows the results. Our library, labeled as Sarcomp, reduces the processing time by 50%. Furthermore, it reaches a significant higher single-precision throughput by about 60% as well. Sarcomp also shows improved performance in cycles per instruction. Although both implementations have a comparable level of vectorization, the reason for the performance gain of Sarcomp lies in the distribution of the SIMD instructions. In our FFT software we use the larger *ymm* registers, which are able to handle more data simultaneously. However, we still have a major scalar overhead in other parts of the EOK processing pipeline which prevents us from reaching overall higher vectorization levels. Therefore, a part of our future work is to rearrange the algorithm with the goal of avoiding all scalar operations, as well as utilizing longer vector lengths, as this matches today's architectural trends.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	FP Ops: Packed	FP Ops: Scalar	% of FP Ops	
[Loop@0x108a78 in sc_mm256_k32_rfft_ps]	5.618s	100.0%	0.0%	72.4%	AVX(256);
[Loop@0x169a90 in sc_mm256_mv_cemul_ip_ps]	4.626s	100.0%	0.0%	47.9%	AVX(256);
[Loop@0x109a66 in sc_mm256_k32_rfft_ps]	3.994s	100.0%	0.0%	8.0%	AVX(128);
[Loop@0x108980 in sc_mm256_k32_rfft_ps]	3.573s	100.0%	0.0%	49.7%	AVX(256);
[Loop@0x148708 in sc_mm256_k16_ifft_ps]	3.258s	100.0%	0.0%	64.4%	AVX(256);
[Loop@0x149307 in sc_mm256_k16_ifft_ps]	2.822s	100.0%	0.0%	32.2%	AVX(128);
[Loop@0x1098e0 in sc_mm256_k32_rfft_ps]	2.812s	100.0%	0.0%	52.4%	AVX(256);

Fig. 6: Ideal vectorization levels of Sarcomp's core functions.

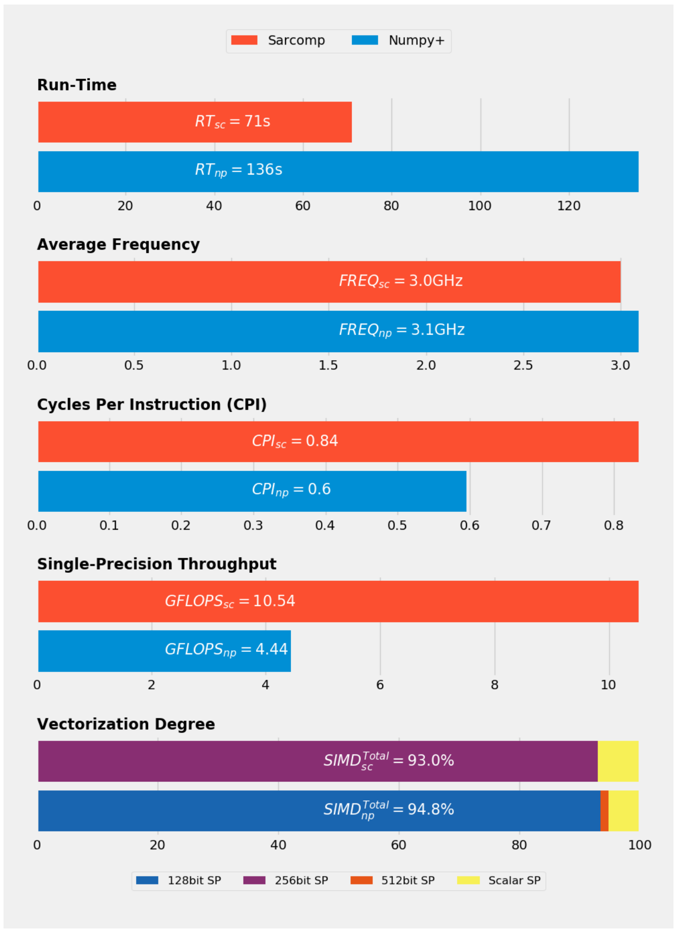


Fig. 7: Performance metrics of our vectorization of the range compression.

C. Improving I/O Access

As shown in Figure 1, the range compression consists of three processing steps: (1) an FFT of the raw data in the range direction, (2) an element-wise complex matrix vector multiply, and (3) an inverse range FFT in order to transfer the data back to the time domain. These three steps represent a matched filter, which is commonly used for data compression. However, the data has to be provided efficiently. After the data acquisition step, the real-valued raw data is saved as bytes on a NAS drive. Since the kernels are based on single precision SIMD instructions, the data first needs to be read from a file and loaded to the memory, then converted to floating point values and finally stored in an aligned memory space. For this purpose, we implemented a memory mapped and vectorized raw reader that additionally also handles the block processing by shifting every block to the next stage of the processing pipeline. Tests show that, in addition to vectorized reading, memory mapping is a significant contributor to improvements, as data is projected directly into memory providing faster access. However, the block size has a major impact on the overall performance. The reason behind that is that the radix FFT requires the data shape to be power of two. When this is not the case, a zero padding has to be performed that can lead to a significant increase in data size and thus to processing and memory consumption overhead. Figure 8 shows this effect by comparing *Sarcomp* against *Sarcomp Opt.* or *Numpy+* against *Numpy+ Opt*, respectively. *Opt.* represents the case when optimized block sizes are used. In addition, the improvement of the run time by integrating the enhanced kernels into the EOK pipeline is shown. Sarcomp speeds up the overall block processing performance by a factor of 3.

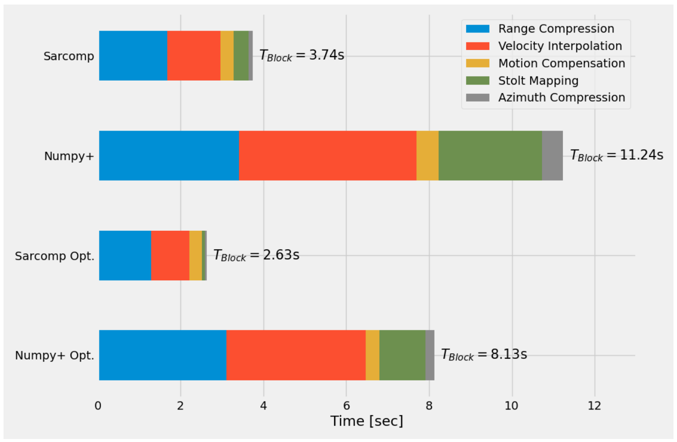


Fig. 8: Block performance comparison between *Numpy+* and *Sarcomp*.

D. Impact of the Image Resolution

The resolution of a SAR image has a major impact on the processing time as well. Figure 9a shows the reference SAR image with a resolution of (0.5 m, 0.5 m). In order to see the effect of reducing the resolution, we generated the same image but with lower resolutions as shown exemplary in Figure 9b. The deterioration of the image quality is apparently visible. This trade-off between processing time and image resolution is clarified in the flight campaign and depends finally on its goal. However, often it is necessary to quickly evaluate or verify the current parameter settings during a flight. Therefore, it is

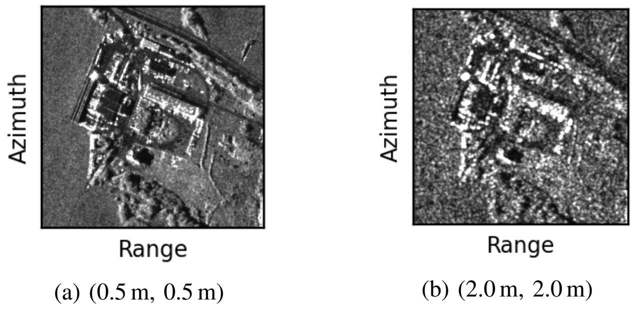


Fig. 9: Resolution comparison

common to generate a SAR image with reduced resolution, making quick adjustment possible. Figure 10 presents the single core run-time of processing different image resolutions. Regarding the real-time limit of $t_{rt_{0,1}}$, we conclude that the real-time condition t_{rt_0} is satisfied for all resolutions when Sarcomp was used. However, real-time capability is regarding t_{rt_1} not given for any resolution in the single core case. One might expect a decrease in computation time is proportional to the selected resolution, e.g., if the resolution is scaled down by a factor of 2, the computation time is reduced to the same extent. However, Figure 10 shows that this is not the case. This is a consequence of the SAR image formation process, which needs to read in and pre-process all data first by means of range compression and velocity interpolation, independent of the selected resolution. The data sub-sampling is implemented as part of these operations, as a simple sub-sampling at raw data level is undesirable and would lead to image artifacts. Consequently, only the subsequent EOK operations (2D-FFTs and stolt mapping) are executed on a smaller amount of data. Yet, a reduction in run-time is possible, when a decrease in resolution is acceptable.

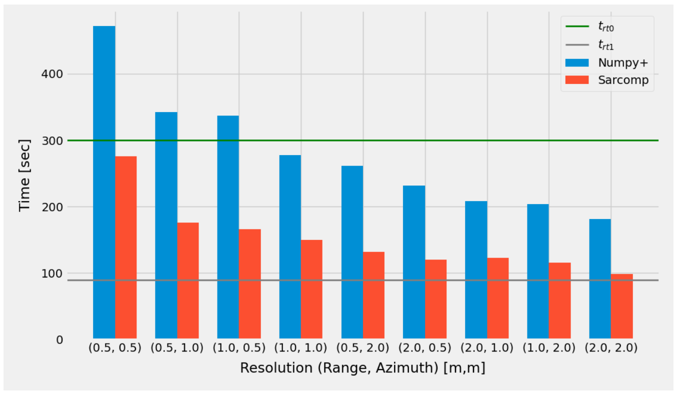
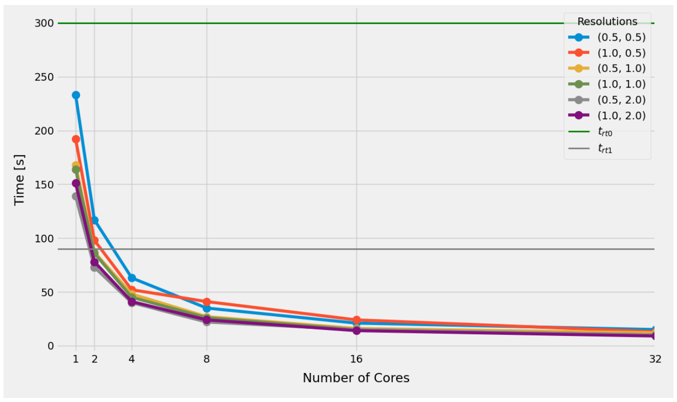


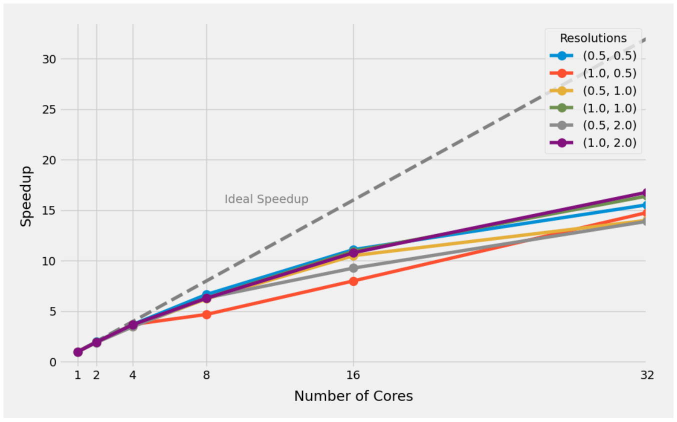
Fig. 10: Single core performance comparison between Numpy+ and Sarcomp for different resolutions.

E. Multi Core Implementation

Another aspect of optimal system usage is the parallel exploitation of all components. Based on the optimized algorithms tailored to the architecture, as described above, we distribute the kernels over the cores of the 3 RTPs and thus exploit all available parallelism on the target platform. Figure 11a shows our tests of the multi-core server implementation of Sarcomp for each given resolution. As pointed out, our single core implementation does not provide real-time capability as the processing times for all resolutions are higher than t_{rt_1}. A threshold case is given when two processing cores are used. t_{rt_1} is met when the resolution is bigger than (0.5, 1.0), as illustrated in Figure 11a. In Figure 11b, the corresponding speedup is displayed. In the case of two cores, the speed-up is close to ideal, improving the overall run-time by a factor of two. This is also valid when 4 cores are employed, which enables the real-time capability for all resolutions as the run times are smaller than t_{rt_1}. The other cases show the impact of using 8, 16 and 32 cores. For all of them, we achieve real-time capability per our definition. However, the performance gain is smaller when more processing cores are used as the thread synchronization effort increases.



(a) Multi-core run-time



(b) Speed Up

Fig. 11: SAR image processing performance

Currently, Sarcomp uses the split-complex data format as it simplifies vectorization using Intel's ISL. As explained, we used DLR's already existing EOK Python pipeline that works with the standard-complex format where the real and imaginary parts are stored as adjacent pairs in main memory. This leads to a major performance barrier because after each processing stage an evaluation of the intermediate results is necessary, which results in many copying procedures between the two formats. As a result, the run-time increases rapidly. Therefore, our future work consists of updating the pipeline to enable the split-complex functionality. We assume that we are able to improve the processing performance further.

V. Lessons Learned

If we put our findings into the context of the on-board processing, where a total of nine cores (three per CPU) are available, we can state that the DBFSAR system is, per our definition, beyond real-time capable for the specified image resolutions. We achieved this by understanding the problem entirely and thus being able to align performance critical parameters such as block-size, resolution, accuracy etc. with the architecture of the given system. Regarding algorithmic performance, a gain can be realized by hardware-oriented development. It turned out that SIMD instructions provide an effective way of optimization with respect to computational effort. This requires an increased programming effort and does not ensure a performance improvement in terms of SAR image quality. However, there is a clear benefit with respect to the achievable resolution within the limited time-frame imposed by the real-time requirement. We transfer the findings to a project that started this year, in which we take the application characterization gained here to a new platform to be deployed, e.g. ensure that the CPU supports the needed instruction sets and that a sufficient number of cores is available. In addition, we select the dimensions in a way that block processing can run without any loss of performance.

VI. Conclusion

We have shown that hardware-oriented optimizations together with the alignment of SAR parameters, e.g., block-size, resolution, etc., accelerate SAR routines and thus the overall SAR application run time. We have demonstrated significant advances in optimization of key algorithms and have explained how these algorithms can be deployed in the context of several resource constraints.

In our future work, we will further improve the algorithms based on the optimal utilization of all hardware resources and increased vectorization levels. According to the performed tests, it can be concluded that, by definition, DLR's DBFSAR system is capable of processing SAR data in real-time. In particular, the chosen test data covering an area on ground of 3 km by 10 km is shown to be processed efficiently on-board with a resolution of 0.5 m by 0.5 m within the stringent real-time requirements of 90 seconds using at least 3 cores. We will verify this in the next flight campaign that is scheduled in 2023. The developed hardware accelerations will also be of benefit for the on-board processing of future drone-based SAR systems and will be equally employed within the ground segment servers for off-line processing.

References

[1] Baumgartner, Stefan V. und Joshi, Sushil Kumar (2021) Onboard Processing Concept for Maritime Surveillance Demonstrated with DLR's Airborne Radar Sensors F-SAR and DBFSAR. In: Proceedings of the European Conference on Synthetic Aperture Radar, EUSAR. European Conference on Synthetic Aperture Radar (EUSAR), 2021-03-29 - 2021-04-01, Leipzig, Germany.

[2] A. Moreira, P. Prats-Iraola, M. Younis, G. Krieger, I. Hajnsek and K. P. Papathanassiou, "A tutorial on synthetic aperture radar," in IEEE Geoscience and Remote Sensing Magazine, vol. 1, no. 1, pp. 6-43, March 2013, doi: 10.1109/MGRS.2013.2248301.

[3] Cumming, Ian G.; Wong, Frank Hay-chee (2005): Digital processing of synthetic aperture radar data. Algorithms and implementation. Boston, London: Artech House (Artech House remote sensing library).

[4] C. Wu, "A digital system to produce imagery from SAR data," in Proc. AIAA Conf.: Syst. Des. Driven Sens., Oct. 1976.

[5] J. R. Bennett and I. G. Cumming, "A digital processor for the production of SEASAT synthetic aperture radar imagery," in Proc. SURGE Workshop, ESA Pub. SP-154, Frascati, Italy, Jul. 16–18, 1979.

[6] C. Wu, K. Y. Liu, and M. J. Jin, "Modeling and a correlation algorithm for spaceborne SAR signals," IEEE Trans. Aerosp. Electron. Syst., vol. AES-18, no. 5, pp. 563–575, Sep. 1982.

[7] 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

[8] Alivizatos, Emmanouil Reigber, Andreas Moreira, Alberto Uzunoglu, Nikolaos. (2004). SAR Processing with Motion Compensation using the Extended Wavenumber Algorithm.

[9] C. Le et al., "Onboard FPGA-based SAR processing for future spaceborne systems," Proceedings of the 2004 IEEE Radar Conference (IEEE Cat. No.04CH37509), 2004, pp. 15-20, doi: 10.1109/NRC.2004.1316388.

[10] D. Wang and M. Ali, "Synthetic Aperture Radar on low power multi-core Digital Signal Processor," 2012 IEEE Conference on High Performance Extreme Computing, 2012, pp. 1-6, doi: 10.1109/HPEC.2012.6408665.

[11] G. Li, F. Zhang, L. Ma, W. Hu and W. Li, "Accelerating SAR imaging using vector extension on multi-core SIMD CPU," 2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS), 2015, pp. 537-540, doi: 10.1109/IGARSS.2015.7325819

[12] M. Fatica and E. Phillips, "Synthetic aperture radar imaging on a CUDA-enabled mobile platform," 2014 IEEE High Performance Extreme Computing Conference (HPEC), 2014, pp. 1-5, doi: 10.1109/HPEC.2014.7040960.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] M. Schlemon and J. Naghmouchi, "FFT Optimizations and Performance Assessment Targeted towards Satellite and Airborne Radar Processing," 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 313-320, doi: 10.1109/SBAC-PAD49847.2020.00050.

[20] D. T. Popovici, F. Franchetti and T. M. Low, "Mixed data layout kernels for vectorized complex arithmetic," 2017 IEEE High Performance Extreme Computing Conference (HPEC), 2017, pp. 1-7, doi: 10.1109/HPEC.2017.8091024.

[21] Reigber, A. et al: The High-Resolution Digital-Beamforming Airborne SAR System DBFSAR. MDPI Remote Sensing. vol. 12, no. 11, pg. 1710, 2020

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

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

[24] Intel Corporation, "Intrinsics Guide," https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[25] T. Schanze, "Sinc interpolation of discrete periodic signals," in IEEE Transactions on Signal Processing, vol. 43, no. 6, pp. 1502-1503, June 1995, doi: 10.1109/78.388863.