Sarcomp - A High-Performance Library for SAR Processing

Maron Schlemon[a,b], Martin Schulz[b], Rolf Scheiber[a], Bengisu Elis[b], and Marc Jäger[a]
[a]German Aerospace Center (DLR), Münchener Str. 20, 82334 Oberpfaffenhofen, Germany
[b]Technical University of Munich, Boltzmannstraße 3, 85748 Garching, Germany

Abstract

Synthetic Aperture Radar (SAR) captures vast amounts of raw data from satellites and specialized aircraft, necessitating advanced processing for image synthesis. This paper delves into optimization techniques for SAR processing, spotlighting the Fast Fourier Transforms (FFT) and Sinc interpolations. The primary goal is enhancing SAR processing efficiency. We introduce Sarcomp, a SAR processing library in C with a Python API, leveraging SIMD, OpenMP, and cache optimizations. Compatible across x86 (Intel and AMD) and ARM architectures, Sarcomp offers a platform for accelerating SAR processing workflows. Based on tests across multiple architectures, our results validate the effectiveness of the proposed methods.

1 Introduction

SAR enables the generation of high-resolution 2D and 3D representations of landscapes with the ability to remain unaffected by light or weather conditions. Predominantly positioned on satellites or specialized aircraft, these systems capture large amounts of raw data, requiring sophisticated processing for image creation. Their significance spans a range of applications, such as climate and forest studies, environmental and Earth monitoring, change detection, and security-related operations.

Like conventional radar, SAR emits frequency-modulated pulses that are backscattered and subsequently captured. Over a period, these cumulatively collected echo signals form a virtual aperture that exceeds the antenna's physical length, allowing the coverage of larger areas. The raw data, often captured across multiple receiving antennas and channels, undergoes an AD conversion process before being transferred to storage drives. After data acquisition, the image formation process generates SAR images [1].

SAR algorithms can be classified into variants based on time and frequency domains. For a comprehensive examination of this subject matter, the reader is encouraged to consult the specialized textbook on SAR signal processing by Cumming et al. [2]. SAR image formation is inherently complex, necessitating multiple sequential processing operations. Given the extensive data sets involved, it becomes imperative to partition this data into manageable blocks for efficient processing. The sub-sequential image formation necessitates substantial resource allocation regarding data handling, computational power, and storage capabilities. To enhance efficiency, both in terms of runtime and throughput, for frequency domain-based algorithms, our approach can be characterized in the following manner:

- **Function Identification:** The initial step entails discerning which computationally intensive functions require the most processing time.
- **Function Optimization:** These identified functions are outsourced to C and enhanced with SIMD instructions. This granular optimization level ensures the CPU's potential is fully exploited, ensuring optimal utilization rates. These functions are implemented in our specialized SAR library, Sarcomp. It is written in C, enhanced with SIMD and OpenMP, and provides a Python API, allowing developers to integrate and deploy it within their workflows.
- **Generic Design Philosophy:** A pivotal factor of our development strategy was versatility. We expect library users to leverage the function pool within Sarcomp to construct the necessary SAR algorithm kernel. We offer a modular and integrative design approach by conceptualizing the SAR processing pipeline as an aggregation of SIMD kernels, each unit ranging from core functions to kernels and kernels to the overarching processing pipeline.

In essence, Sarcomp can be analogized to `numpy`, tailored explicitly for SAR processing. Illustrating our methodology, we detail the utilization of Sarcomp in developing and enhancing the Extended Omega-K (EOK) algorithm [3], a representative of frequency-domain-based algorithms. It is structured around five processing phases: range compression, velocity interpolation, motion compensation, the EOK kernel (which integrates 2D FFTs and stolt mapping), and azimuth compression. As depicted in Figure 1, FFTs and interpolations form the essential kernels of this SAR imaging pipeline due to their recurrent application on many data blocks. The FFT, in particular, is utilized for its efficiency in computing the Discrete Fourier Transform (DFT), an instrumental tool in spectrum analysis.

The following sections present a detailed exploration of our optimization techniques for SAR image processing. We use FFTs and Sinc interpolations as specific examples; however, the main objective is to refine general SAR processing routines for higher computational efficiency. To-

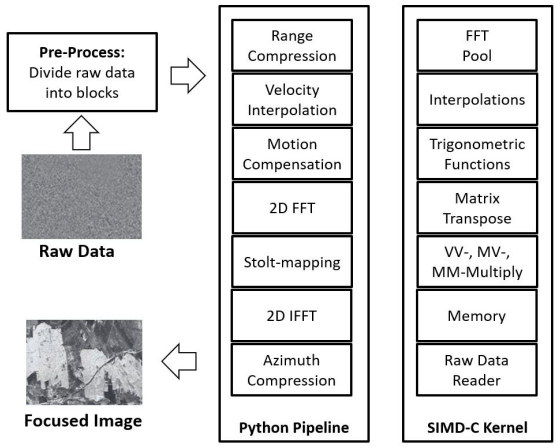

Figure 1 SAR imaging pipeline (EOK) in Python and its core functions implemented in C (VV, MV, MM being vector-vector, matrix-vector and matrix-matrix multiplies)

gether with these findings, we introduce Sarcomp and its Python API interface. Sarcomp embodies low-level optimization techniques, harnessing the capabilities of SIMD instructions and cache optimization strategies. With compatibility spanning x86 and ARM architectures, Sarcomp provides a comprehensive platform for researchers and developers to accelerate their SAR data processing routines. Additionally, the manuscript elaborates on an OpenMP-integrated methodology for SAR image processing, offering a systematic approach to extensive dataset management. Empirical results from tests on Intel, AMD, and ARM systems validate the efficacy and adaptability of the proposed methodologies. This research contributes to the scientific discourse on high-efficiency SAR processing systems. In particular, we make the following contributions:

- We developed a fully vectorized SAR imaging pipeline and aligned it with the system's register sizes, caches, and memory space. This precise alignment results in high Single Instruction, Multiple Data (SIMD) levels, and high hardware utilization rates.
- We efficiently distribute SIMD kernels across many compute cores to further parallelize data processing.

While open-source libraries can be a convenient solution in some cases, we decided to develop solutions for our particular situation. We intend to offer our library under a licensing agreement, bringing along a series of obligations and responsibilities. We must maintain complete control over our code base to ensure its functionality. Moreover, removing open-source libraries helps us avoid licensing issues and retain full legal power. Additionally, due to the complex structures inherent in our processing pipelines, adapting to various libraries would only add complexity and overhead. Instead, a customized solution that integrates with our existing pipeline is more suitable. This approach provides an efficient, custom, and legally secure solution for our SAR processing requirements. The insights gained from this study apply to resource-limited and server-based scenarios requiring efficient performance.

2 Compute Kernel Optimizations

SIMD or vector instructions constitute a parallel computing architecture within the Central Processing Unit (CPU) that allows the execution of the same operation on multiple data points simultaneously. Special hardware within the CPU facilitates this parallelism, designed to process vectors or arrays of data efficiently with a single instruction. The advantage of SIMD instructions is their ability to significantly increase data throughput, especially in applications that involve large datasets requiring the repetitive performance of the same operation, such as multimedia processing, scientific computations, and machine learning. SIMD achieves this enhanced throughput through the following:

- **Data Alignment:** The efficiency of SIMD instructions often depends on the memory alignment of the data. Properly aligned data enables the CPU to load and process data in larger chunks, optimizing memory bandwidth usage and reducing the number of instructions needed to process a given data volume.
- **Sequential Data Loading:** The effectiveness of SIMD instructions increases when the CPU can load data sequentially from memory. This approach minimizes the latency associated with memory access and optimizes cache and bandwidth.
- **Parallel Arithmetic Operations:** SIMD units within the CPU can simultaneously execute arithmetic operations such as addition, subtraction, multiplication, and division on multiple data points. This capability significantly speeds up computational tasks.

The Fused Multiply-Add (FMA) units enhance the efficiency of SIMD instructions. The FMA unit merges multiplication and addition operations into a single instruction, which reduces rounding errors that can accumulate when performing these operations separately. This feature is valuable in high-precision applications like SAR image formation. It effectively doubles the computational throughput for operations that fit its model, marking it as an efficient component of modern CPUs.
Figure 2 shows the CPU core pipeline functionality specific to the Skylake microarchitecture, a product line from Intel. In the context of SIMD instructions and their execution on the CPU, the Skylake microarchitecture integrates several vector processing units across different ports. The diagram identifies vector units associated with Ports 0, 1, and 5. Ports 0 and 1 support Vector Fused Multiply-Add (Vec FMA), Vector Multiply (Vec MUL), Vector Addition (Vec Add), Vector Arithmetic Logic Unit operations (Vec ALU), and Vector Shift (Vec Shft). Including FMA units in these ports is critical because it enables the execution of multiply-add operations on vectors concurrently, thus doubling the computational throughput for these operations. Port 5 has capabilities for Vector Shuffle (Vec

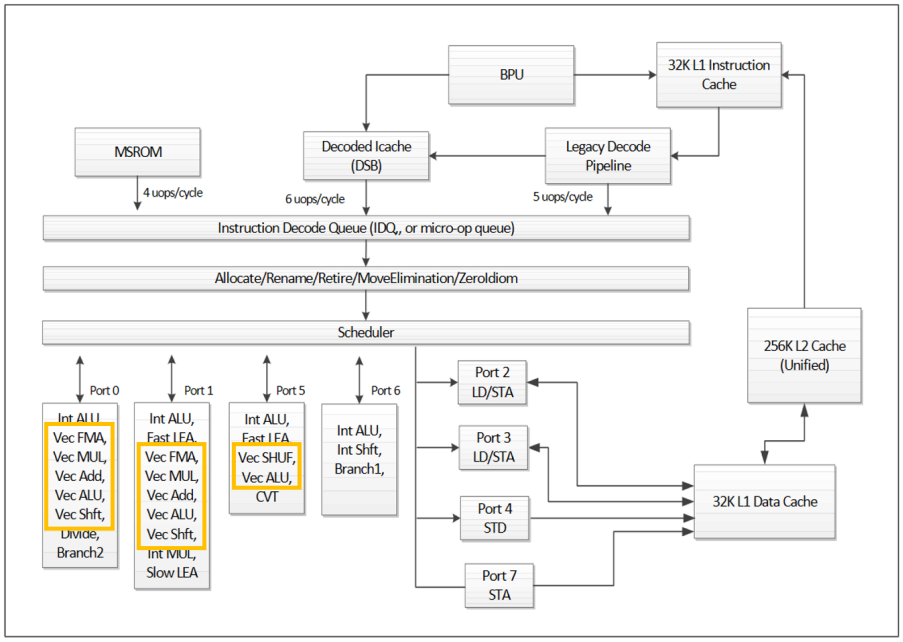

Figure 2 CPU core functionality of Intel's Skylake micro-architecture with its vector units marked

SHUF), Vector ALU operations (Vec ALU), and Vector Convert (CVT). These operations are crucial for data manipulation and performance optimization.

The Skylake architecture also details other components essential for SIMD operation efficiency. The L1 and L2 caches are vital in data storage and retrieval, feeding the execution units with the necessary data. The L1 Instruction Cache, Legacy Decode Pipeline, and Decoded ICache (DSB) all contribute to the efficient decoding of instructions. Following decoding, the Instruction Decode Queue (IDQ) holds the micro-operations before they are scheduled for execution. The scheduler is responsible for dispatching these operations to the appropriate execution ports based on availability and the type of operation to be performed. The Skylake microarchitecture, with its advanced SIMD capabilities, embodies the sophisticated design necessary to handle the computational demands of modern applications, particularly those benefitting from parallel processing.

3 Baseline Model for SAR Image Formation

We use DLR's existing Python pipeline of the EOK as our base model and investigate its performance towards real-time capability. We measured the processing time of each function of the EOK blocks (Figure 1) and profiled them using Intel's VTune profiling software. This approach provided an in-depth insight into the given code. Although the Python prototype uses a set of libraries, the most significant ones are Numpy and Scipy, which we label as *Python*. This pipeline reference provides good performance as it already implicitly benefits from SIMD optimizations (as Figure 3 shows) such as the pocketFFT C implementation and, therefore, offers a comparable baseline.

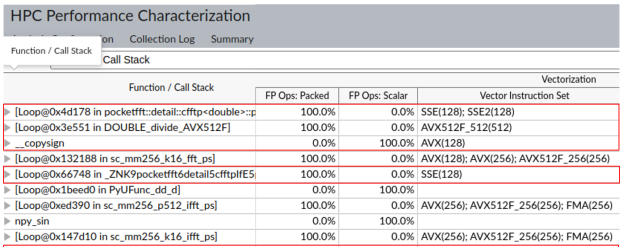

Figure 3 Performance profile for the *Numpy+* prototype in caparison to Sarcomp using Intel VTune.

Our methodology involves timing each function within the EOK blocks and employing Intel's VTune for profiling, which delivered comprehensive performance insights. While the Python prototype depends on various libraries, notably Numpy and Scipy—collectively referred to as Python in this context—it inherently leverages SIMD optimizations. This is highlighted in Figure 3, where our custom-implemented functions, prefixed with 'sc', achieve 100% vectorization, harnessing the full potential of SIMD units. Moreover, our implementations utilize wider SIMD registers than the pocketFFT from Numpy, offering superior performance. Following identifying pipeline bottlenecks, we re-engineered key functions in C, optimizing them with SIMD instructions tailored to specific vendors.

Our previous publications explored the efficient deployment of FFT and sinc interpolation, detailing their importance in SAR processing [4, 5]. While these works have dived into implementing SIMD instructions, the present study seeks to shift the focus toward practical application. Specifically, we illustrate employing these SIMD-optimized routines to assemble a pipeline component through Sarcomp's Python API. We aim to bridge the gap between low-level performance enhancements and high-level application development. We offer developers a

streamlined approach to leverage optimized SAR processing routines within more complex workflows.

4 Python API

This section examines the steps in executing the range compression, an integral SAR image processing pipeline component. The range compression covers the following systematic steps:

- **Raw Reader Functionality:** The first step involves the raw reader function, which efficiently reads byte-aligned data from a file and translates it to an aligned float (32 bit) memory space. Operating on aligned memory in the context of SIMD instructions enhances the operation's performance, but it is not strictly required. The raw reader uses vectorized 'load' and 'convert' instructions for efficient data transition. Furthermore, the raw reader strategically divides data into blocks, ensuring compatibility with the CPU's cache specifications.
- **Real-to-Complex FFT:** After data reading, the real-to-complex FFT is executed. The resultant output is in split-complex format with separate memory space allocation for the real and imaginary components. This design choice is not arbitrary; we can use SIMD instructions more effectively by maintaining this separation, thus enhancing computational performance.
- **Matrix-Vector Multiplication:** The next operation involves an element-wise matrix-vector (MV) multiplication with the reference function (or replica). This process represents the compression in the frequency domain.
- **Transition to Time Domain:** The final step of the range compression process is reverting the data to the time domain. This transition sets the stage for the following pipeline operations. Conceptually, this procedure mirrors a convolution operation in the time domain.

Listing 1 shows a practical representation of the process described above. A complete guide and the Python implementation of Sarcomp are accessible at `www.sarcomp.de`.

5 Generic Code Design

SAR processing has resulted in two significant developments: a) server and ground-based SAR processing and b) resource-constrained airborne and satellite processing. We accommodate this with an adaptable, efficient generic code design that allows our software to perform proficiently across various system architectures. Currently, we are deploying our library on Intel and AMD architectures. However, we are actively working towards integrating the ARM architectures, incorporating its Scalable Vector Extension (SVE) intrinsics into our software design. The strategic

```
import sarcomp as sc
# Read data & replica
rawData = sc.readraw(FILE_RAW_DATA)
replica = sc.readraw(FILE_REPLICA)
# Create fft plans
fft_plan = sc.fft.fft(rawData.blk.shape)
ifft_plan = sc.fft.ifft(rawData.blk.out.
    shape)
# Execute the range compression
sc.rg_comp(rawData, replica, fft_plan,
    ifft_plan)
```

Listing 1 Sarcomp implementation of the range compression

methodology we followed to develop a generic code design unfolds as follows:

- **CPU configuration:** As stated earlier, the installation starts with generating a CPU configuration file. This file is required to determine the CPU vendor, the number of cores for threading setup, and the array of supported SIMD instructions. Our library uses AVX, AVX2, FMA, and ARM Neon instructions for improved processing efficiency.
- **Inclusion of intrinsics and macros:** Upon CPU identification, the configuration file integrates Intel, AMD, or ARM intrinsics header files based on the system's CPU vendor. Our custom-designed generic macros are also incorporated into this file to facilitate system-specific code optimization and execution.
- **Library compilation:** Using the system-specific information and optimized macros, the final step involves the compilation of the library explicitly tailored for the target system.

6 Processing Results

As illustrated in Figure 4, we implemented the matched filter following a well-structured procedure: Using OpenMP, we divide the data into several blocks and assign each to a specific thread for parallel processing. The processing starts with executing an FFT on each line within each block. Following this, we execute an elementwise multiplication using the supplied reference function as the other operand. We then feed the resulting output into an inverse FFT operation. These steps are essential in data transformation between time and frequency domains, thus facilitating the identification of radar targets.
Using OpenMP's capabilities for data partitioning and parallel processing, coupled with our SIMD processing kernels, our implementation enables efficient processing of SAR data. This results in a highly optimized processing workflow.

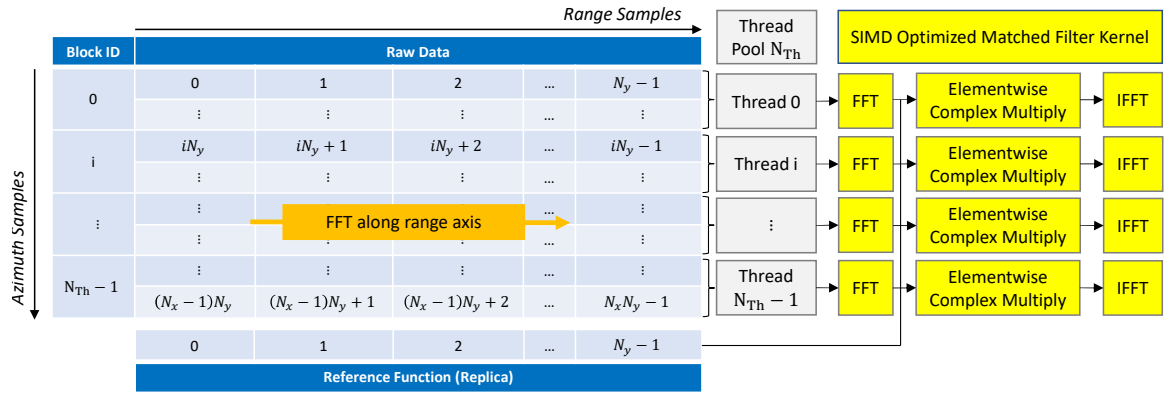

Figure 4 Multi-threading workflow of the matched filter

6.1 Test Environment

We employed two hardware configurations comprising ground-based servers and our airborne onboard processing architecture to evaluate our implementation. Our ground-based testing involved two server systems, an Intel(R) Xeon(R) Gold 6154 and an AMD EPYC™ 7662, each featuring two sockets. As for our airborne tests, we performed on the architecture composed of three Real-Time-Processing Units (RTPs), each running on an Intel® Core™ i7-3610QE processor. The results and discussion about the onboard system testing are extensively covered in [5]; therefore, our focus will be server-based evaluations.

GCC and Intel's ICX compilers were employed for the tests, with both -O2 and -O3 optimization flags activated for peak code performance and the `-march=native` flag to ensure utilization of all instruction subsets supported by the local machine. After extensive tests are performed in these varied settings, only the best results, representing the maximum achievable performance, are presented in our analysis.

6.2 Performance Measuring

We benchmarked our FFT implementation against the FFTW version 3.10 [6], explicitly focusing on one-dimensional complex-to-complex in-place transforms of single- and double-precision data. The input data for this test consisted of a series of power-of-two sizes ranging from 32 to 32,768 samples. We determined performance by adhering to the benchFFT methodology [7], calculating the GFLOPS using Equation 1 from [7] for an FFT of length N and execution time T (in nanoseconds):

$$GFLOPS = \frac{5Nlog_2(N)}{T_{FFT}} \quad (1)$$

We compiled the FFTW library using the configuration flags `-enable-float -enable-avx2 -enable-fma` for a thorough and equitable comparison. These flags enable the same instruction sets we utilize in our FFT implementation. Furthermore, to obtain the best performance, we use the `FFTW_EXHAUSTIVE` planner flag in FFTW. Our evaluation focuses primarily on the most recent version of FFTW, version 3.10. This choice aligns with our emphasis on FFTW's extensive documentation and established reputation, positioning it as a robust point of comparison. We examine the performance of our FFT implementation, denoted as SIMDFFT, against the FFTW on two different hardware architectures, Intel and AMD. Performance is measured using GFLOPS, a standard metric for assessing computational speed, particularly in data-heavy tasks such as FFT computations.

6.3 SIMD FFT Performance Evaluation

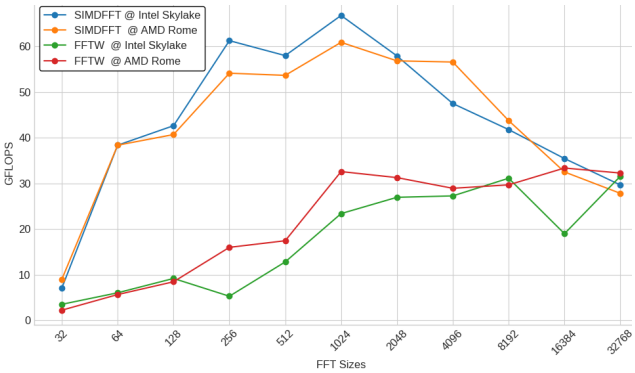

Figure 5 Single precision 1D-FFTs performance

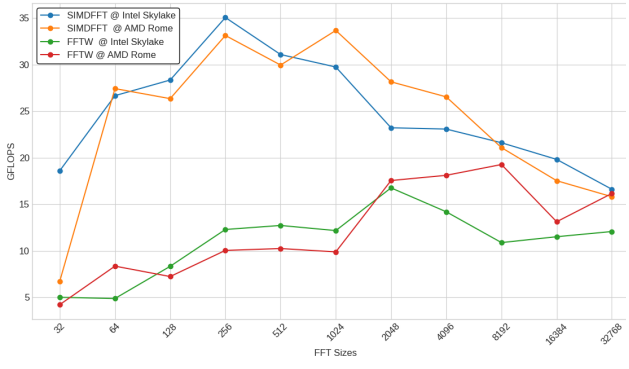

Figure 6 Double precision 1D-FFTs performance

Figures 5 and 6 compare the Gigaflops (GFLOPS) achieved by two different implementations of our SIMDFFT and the FFTW library - over a range of FFT sizes from 32 to 32,768 for one-dimensional (1D), in-

place complex-to-complex FFTs. The x-axis represents the FFT sizes, while the y-axis measures the performance in GFLOPS.

In the case of single-precision 1D-FFTs, as shown in Figure 5, SIMDFFT outperforms FFTW on Intel Skylake and AMD Rome architectures across almost all FFT sizes. The performance advantage is particularly pronounced at smaller FFT sizes. However, as the FFT size increases, the performance gap between SIMDFFT and FFTW narrows. This trend is mirrored in the double precision 1D-FFTs performance chart, depicted in Figure 6.

The significant performance difference at smaller FFT sizes can be attributed to the last stage of our FFT algorithm, which involves memory manipulation operations. These operations become increasingly impactful on performance as the FFT kernel size increases. For larger FFT sizes, such as 16,384 and 32,768, the performance of SIMDFFT converges closer to that of FFTW, indicating that memory manipulation becomes a bottleneck. Future work will explore alternative methods to address this issue and maintain a consistent performance advantage at larger FFT sizes.

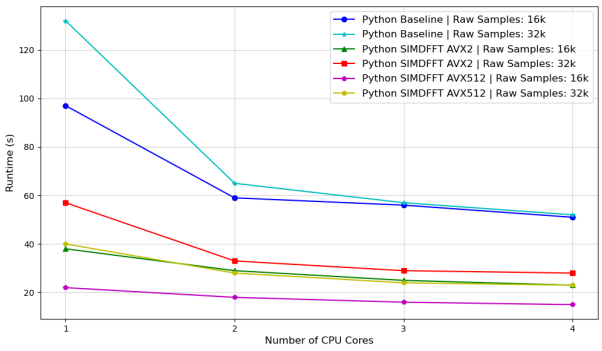

Figure 7 Range compression results for different range and azimuth lengths

In a comparative performance analysis for range compression, using two datasets—Dataset 1 with 16,384 raw samples and Dataset 2 with 32,768 raw samples—the SIMDFFT versions with AVX2 and AVX512 instruction sets exhibited a substantial outperformance over the Python baseline solution, as Figure 7 shows.

With dataset 1, featuring a block size of 512x16384 and 240,384 azimuth samples, SIMDFFT AVX2 and AVX512 showed reductions in runtime as the CPU core count increased. This trend was pronounced with Dataset 2, which had a block size of 512x32768 and 105,167 azimuth samples, indicating that the SIMDFFT optimizations scale effectively with data size. The AVX512 SIMDFFT variant, in particular, demonstrated a stark performance improvement over the baseline and AVX2, with the most significant runtime decreases observed in the 32k raw samples configuration when utilizing all four processing cores. This data suggests that the SIMDFFT with AVX512 optimizations is highly efficient in using CPU resources, leading to significantly accelerated processing times and highlighting the potential of SIMD optimizations in high-performance computing tasks involving large datasets.

7 Conclusion

This study has presented a detailed account of a highly optimized matched filter implementation in SAR image processing. Our work focused on creating SIMD-based SAR kernels and using OpenMP efficiently for data partitioning and parallel execution.

We demonstrated the benefits of our approach through an extensive evaluation process, which included a thorough FFT analysis and a performance investigation of the matched filter for the range compression as an example. Our results suggest that our approach successfully achieves computational speed, providing robust performance across different hardware architectures.

In summary, our work underlines the importance of meticulous algorithmic choices and strategic use of hardware capabilities in dealing with the complexities of SAR image processing. Future work can explore further potential enhancements in this direction, such as fine-tuning for different hardware architectures and further optimizing the imaging pipeline to handle larger and more complex data sets.

8 Literature

[1] 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 .

[2] 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).

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

[4] 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), Porto, Portugal, 2020, pp. 313-320, doi: 10.1109/SBAC-PAD49847.2020.00050

[5] M. Schlemon, M. Schulz and R. Scheiber, "Resource-Constrained Optimizations For Synthetic Aperture Radar On-Board Image Processing," 2022 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2022, pp. 1-8, doi: 10.1109/HPEC55821.2022.9926327

[6] 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.

[7] Matteo Frigo and Steven G. Johnson, "FFT Benchmark Methodology," 2019, https://www.fftw.org/speed/method.htm