# Optimization of a Synthetic-Aperture Radar Image Processing Algorithm for SoC-FPGAs

***Abstract*—SAR image generation is done by very computationally intensive algorithms. Hence, it is a challenge to create systems capable of real-time on-board operation. This work looks into the optimization of one SAR image generation algorithm, the Backprojection algorithm, which optimizes each signal individually. The improvement of this methodology reveals that we are able to optimize the circuit design to consume less resources and minimize the impact on the quality of the resulting image.**

***Index Terms*—Backprojection algorithm, SAR, wordlength optimization, approximate computing**

## I. Introduction

The Backprojection (BP) algorithm is widely used for Synthetic-Aperture Radar (SAR) image generation. With the growing adoption of SAR for several applications, such as monitoring traffic, crops, deforestation, natural disasters such as earthquakes, volcanos and hurricanes, for topographic imaging of planets and ocean floor, among many others [1], a wide variety of algorithms have been developed and modified to meet the requirements of each individual application [2]. The BP algorithm is considered the standard for high quality images, however, it is also known for its high computational complexity, in the order of $N^3$, making it an inadequate choice for real-time applications [3].

Since image processing algorithms can tolerate some precision loss or error in some of the pixels, as long as most of the values are correct, or close enough, it is possible to make a custom implementation of an algorithm in hardware with smaller, and faster, arithmetic units.

In this work, the BP algorithm was studied to understand the physical meaning of its variables, and a novel optimization methodology to determine which variables need more precision and the ones that could be represented with less bits, while still producing very high-quality images.

The paper is organized as follows: section VI presents related work done on hardware accelerators for the BP algorithm and section II introduces the BP algorithm. The proposed methodology starts with the analysis on the algorithm, presented in section III, section IV details the developed architecture and section V discusses the results of the developed work, which are compared to related work in section VI. Section VII discusses future work and provides final remarks.

## II. Backprojection Algorithm

The BP algorithm is based on the projection of the echoes received by the radar which is performed for each of the image pixels [4]. For every pulse, the contribution is given by Eq. 1.

$$s(m,\tau_n) = N_{fft} \cdot \texttt{fftshift}\{\texttt{ifft}(S(f_k,\tau_n))\} \cdot \\ \cdot \exp\left(\frac{j2\pi f_1(m-1)}{N_{fft}\Delta f}\right) \tag{1}$$

where $N_{fft}$ is the Fast Fourier Transform (FFT) length, $S(f_k,\tau_n)$ is the phase history, $f_k$ is the frequency sample per pulse, $\tau_n$ is the transmission time of each pulse, $f_1$ is the minimum frequency for every pulse, $m$ is the range bin and $\Delta f$ is the frequency step size. The value of each pixel, at location $r$, is the sum of the interpolated value of $s(m,\tau_n)$, represented in the following equation as $s_{int}(r,\tau_n)$, is given by [3]

$$I(r) = \sum_{n=1}^{Np} s_{int}(r,\tau_n) \tag{2}$$

Since the calculation of each pixel is independent, the algorithm is extremely parallelizable.

## III. BP Algorithm Optimization Methodology

The BP algorithm is an algorithm with high computational complexity, $O(N^3)$, or $X \times Y \times P$, where $X$ is the number of pixels in the x-axis, $Y$ is the number of pixels in the y-axis and $P$ is the number of pulses. To understand how the algorithm can be optimized, it is important to analyse it considering different aspects. The following analysis were performed:

- precision analysis, to understand the impact of each variable in the quality of the resulting image;
- data dependency, to reduce the number of memory transfers necessary during the execution of the algorithm.

The BP algorithm implementation used in this work was done in C programming language, and it was based on the MATLAB implementation [3]. The C implementation of the algorithm is presented in Listing 1. The algorithm iterates over every pulse and every pixel, calculating the contribution of each pulse and accumulating it in the `image` variable, as seen in Equation 1.

The algorithm was implemented using double-precision floating-point variables, because it is the best numerical representation normally used. However, the question lies on whether that precision is really necessary to achieve high-quality images. The precision analysis aims to answer this question.

Sections III-A and III-B detail the analysis performed on the BP algorithm.

```c
void backprojection()
{
    // Calculate the range to every bin in the range profile (m)
    linspace(- (float) half_nfft, ((float) half_nfft) - 1, input_data.nfft, r_vec);

    for (p = 0; p < input_data.np; p++) {
        memset(rc, 0, NFFT_DEFAULT * sizeof(struct complex_d));
        for (i = 0; i < input_data.k; i++) {
            // rc is filled with values, rest is 0
            rc[i].re = input_data.phdata[p][i].re;
            rc[i].im = input_data.phdata[p][i].im;
        }

        ifft_transform(rc, input_data.nfft, nfft_inv);
        fftshift(rc, input_data.nfft);

        for (x = 0; x < NX; x++) {
            for (y = 0; y < NY; y++) {

                x_value = (input_data.ant_x[p] - input_data.x_mat[x][y]);
                x_dist = x_value * x_value;

                y_value = (input_data.ant_y[p] - input_data.y_mat[x][y]);
                y_dist = y_value * y_value;

                z_value = (input_data.ant_z[p] - input_data.z_mat[x][y]);
                z_dist = z_value * z_value;

                dR = sqrt(x_dist + y_dist + z_dist) - input_data.r0[p];

                value = pi4C * input_data.min_f[p] * dR;
                phCorr.re = cos(value);
                phCorr.im = sin(value);

                interp_index = (int) ((dR + 2048.f * maxWr_nfft) / maxWr_nfft);

                if (interp_index >= 0 && (interp_index < input_data.nfft - 1)) {

                    t = (dR - r_vec[interp_index]) / (r_vec[interp_index + 1] - r_vec[interp_index]);
                    interp_result.re = (1.0 - t) * rc[interp_index].re + t * rc[interp_index + 1].re;
                    interp_result.im = (1.0 - t) * rc[interp_index].im + t * rc[interp_index + 1].im;

                    image[x][y].re += (interp_result.re * phCorr.re - interp_result.im * phCorr.im);
                    image[x][y].im += (interp_result.re * phCorr.im + interp_result.im * phCorr.re);

                }
            }
        }
    }
}
```

Listing 1: BP algorithm implementation in C.

## A. *Precision Analysis*

A precision study was performed, with the objective of understanding what variables influence the most the quality of the image and what variables can be represented with inferior wordlengths.

The precision study performed on the BP algorithm consisted of two steps:

- Range evaluation: provides the minimum integer bits necessary to represent each variable.
- Fixed-point testing: each variable was represented with the minimum integer bits necessary, as described in the first step, and the fractional bits varied between 0 and $64 - integer\_bits$. That is, the algorithm was executed and an image was generated for every possible fixed-point configuration for each variable. This was done using a Python script that changed the configuration for each of the variables, executed the algorithm, generated an image and, finally, assessed the quality of the generated image.

This study provides important information regarding the algorithm: which variables have a greater impact on the image quality and which variables can be represented with few bits and still render a high quality image. To evaluate the quality of the images, Structural Similarity (SSIM) is used and values above 0.99 are considered acceptable. Figure 1 shows the data dependencies of the algorithm, where each node represents a variable. Each of these variables was subjected to the precision study. It is important to note that the precision study has more variables than the dependency graph, since these additional variables are auxiliary.

Figure 2 shows the image generated using the same dataset, with one difference: the number of fractional bits in the `complex_image` variable, the variable that stores the contributions of each pulse. As can be observed, to obtain full precision only 45 fractional bits are necessary, however, it is possible to obtain an SSIM of 0.99 with 26 fractional bits, 19 bits less. Similarly to this variable, all variables in

TABLE I: Minimum number of fractional bits necessary to represent each variable in order to generate an image with a SSIM of 1.0 and a SSIM of 0.99.

| **Variable** | **Integer bits** | **Minimum fractional bits for** $SSIM = 1.0$ | **Minimum fractional bits for** $SSIM \geq 0.99$ |
|---|---|---|---|
| **ant_x** | 14 | 11 | 10 |
| **ant_y** | 14 | 11 | 10 |
| **ant_z** | 14 | 11 | 11 |
| **dR** | 7 | 32 | 12 |
| **image** | 1 | 45 | 26 |
| **interp_res** | 1 | 41 | 23 |
| **ph_corr** | 2 | 25 | 4 |
| **ph_data** | 1 | 34 | 14 |
| **r0** | 15 | 10 | 10 |
| **rc** | 1 | 41 | 23 |
| **r_vec** | 7 | 26 | 7 |
| **t** | 2 | 19 | 2 |
| **value** | 16 | 23 | 3 |
| **x_dist** | 26 | 14 | 13 |
| **y_dist** | 26 | 19 | 0 |
| **z_dist** | 27 | 16 | 0 |
| **x_mat** | 7 | 25 | 5 |
| **y_mat** | 7 | 23 | 5 |
| **z_mat** | 1 | 1 | 0 |
| **x_value** | 14 | 23 | 13 |
| **y_value** | 14 | 21 | 10 |
| **z_value** | 14 | 11 | 11 |

the algorithm share this feature. The SSIM values for the `complex_image` variable are presented in Table II.

Table I shows the number of fractional bits each variable needs to achieve a SSIM of 1.0 and 0.99. From the twenty-two variables in the table, only three need over 40 fractional bits, only two needed over 30 fractional bits, seven needed over 20 fractional bits and only nine needed 10 fractional bits to achieve 1.0 of SSIM. If the objective is an SSIM of over 0.99, only three variables require over 20 fractional bits, ten variables require over 10 fractional bits and nine variables only required under 10 fractional bits.

Given the high quality that is possible to achieve even with considerably smaller wordlengths, it is clear the algorithm can benefit from approximate computing. This is the key idea of this work, reducing the precision of the data throughout the algorithm, leading to its optimization, even with slight degradation of the resulting image.

As previously mentioned, the precision study was applied to all variables of the algorithm and input data. SSIM values above 0.99 are considered acceptable, however, taking into consideration the dependency graph in Figure 1, severely reducing the wordlength in, for instance, `ant_x` would lead to error propagation throughout the algorithm. For instance, taking into consideration `ph_corr` variable, according to Table I, the variable needs at least 25 fractional bits to generate an image with a SSIM of 1.0, but only 4 fractional bits to generate an image with an SSIM of 0.99. The same happens for other variables, such as `ph_data`, `x_mat`, `y_mat`, among others. Choosing the smaller number of bits works for one variable, but does not work when every variable of the algorithm is represented by smaller wordlengths. For this reason, the smallest number of fractional bits that originates an image with a SSIM of 1.0 is the criteria chosen to decide the wordlength of each variable, reducing the error propagated. The final wordlength configurations are presented in Table III.

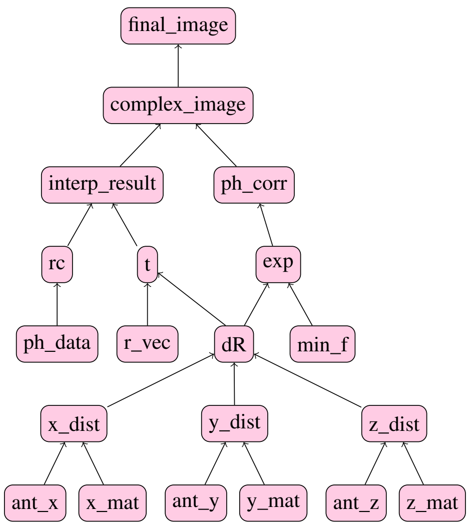
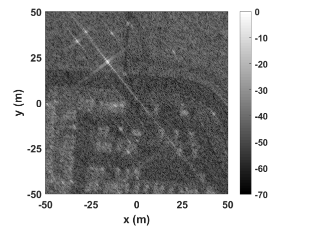
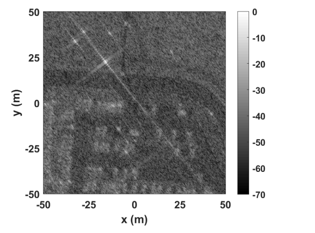


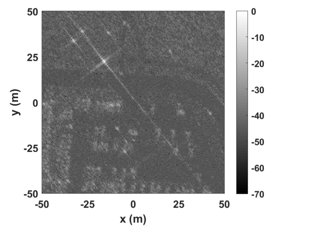
Fig. 1: Data dependency graph of the BP algorithm. Each node represents a different variable.



(a) The accumulator is represented as a fixed-point variable in Q1.45. Has a SSIM of 1.0.



(b) The accumulator is represented as a fixed-point variable in Q1.26. Has a SSIM of 0.99.



(c) The accumulator is represented as a fixed-point variable in Q1.22. Has a SSIM of 0.72.

Fig. 2: Images generated using the BP algorithm and the Gotcha Volumetric SAR dataset. The accumulator of the pulse contributions varies in the precision of the fixed-point variable.

## *B. Data Dependency Study*

The BP algorithm has a complexity of $X \times Y \times P$, where $X$ is the number of pixels in the x-axis, $Y$ is the number of pixels in the y-axis and $P$ is the number of pulses. This algorithm has the following input data: `x_mat`, `y_mat`, `z_mat`, `rc`,

TABLE II: Results of the precision study of the `image` variable, or accumulator variable. The quality of the image is 1.0 up to 46 fractional bits and over 0.99 up to 26 fractional bits. The chosen configuration was 45 fractional bits, the smallest number with a SSIM value of 1.0, and a total of 46 bits.

| Total Bits | Integer Bits | SSIM |
|---|---|---|
| 64 – **46** | **1** | **1.0** |
| 45 – 27 | 1 | [0.9944260789199652, 0.9999999792437367] |
| 26 | 1 | 0.9834842368999309 |
| 25 | 1 | 0.9480179848032538 |
| 24 | 1 | 0.8616897829956542 |
| 23 | 1 | 0.7245506273387262 |
| 22 – 1 | 1 | <= 0.6231567802348613 |

TABLE III: Final wordlength used for each variable used in the implementation.

| Variable | Total Bits | Integer Bits | Fractional Bits |
|---|---|---|---|
| **ant_x** | 25 | 14 | 11 |
| **ant_y** | 25 | 14 | 11 |
| **ant_z** | 25 | 14 | 11 |
| **dR** | 39 | 7 | 28 |
| **image** | 46 | 1 | 45 |
| **interp_res** | 42 | 1 | 41 |
| **ph_corr** | 27 | 2 | 25 |
| **r0** | 25 | 15 | 10 |
| **rc** | 32 | 1 | 31 |
| **r_vec** | 33 | 7 | 26 |
| **t** | 21 | 2 | 19 |
| **value** | 39 | 16 | 23 |
| **x_dist** | 40 | 26 | 14 |
| **y_dist** | 45 | 7 | 23 |
| **z_dist** | 43 | 27 | 16 |
| **x_mat** | 32 | 7 | 25 |
| **y_mat** | 30 | 7 | 23 |
| **z_mat** | 1 | 1 | 1 |
| **x_value** | 37 | 14 | 23 |
| **y_value** | 35 | 14 | 21 |
| **z_value** | 25 | 14 | 11 |

`r_vec`, `min_f`, `r0`, `ant_x`, `ant_y` and `ant_z`. In this study, we consider the GOTCHA input dataset [5]. The GOTCHA dataset is a real data dataset for Spotlight SAR with 117 pulses and 424 frequency samples per pulse, which is used to generate a 501 × 501 image. In order to optimize the data communication between the memory and the IP, we analysed the loops in the algorithm [3], finding the sequence of loops that requires less data transmissions. There are two possible sequences, Listing 2 and Listing 3.

```
    for (p = 0; p < input_data.np; p++) {
    // ...
        for (x = 0; x < NX; x++) {
            for (y = 0; y < NY; y++) {
```

Listing 2: Loops in the BP algorithm, where `p` is the outer loop variable.

TABLE IV: Memory occupied by the variables that depend on `p`.

| Variable | Dimensions | Memory |
|---|---|---|
| **min_f** | 117 * 35bits | 0.5KB |
| **r0** | 117 * 25bits | 0.36KB |
| **ant_x** | 117 * 25bits | 0.36KB |
| **ant_y** | 117 * 25bits | 0.36KB |
| **ant_z** | 117 * 25bits | 0.36KB |
| **rc** | 117 * 4096 * 32bits * 2 | 3744KB |
| **r_vec** | 4096 * 33bits | 16.5KB |
| **Total** | | 3762.44KB |

TABLE V: Memory occupied by the variables that depend on `x` and `y`.

| Variable | Dimensions | Memory |
|---|---|---|
| **x_mat** | 501 * 501 * 32bits | 980KB |
| **y_mat** | 501 * 501 * 30bits | 919KB |
| **z_mat** | 501 * 501 * 1bit | 30KB |
| **image** | 501 * 501 * 46bits * 2 | 2818.9KB |
| **rc[p]** | 4096 * 32bits * 2 | 32KB |
| **r_vec** | | 16.5KB |
| **Total** | | 4796.4KB |

```
    for (x = 0; x < NX; x++) {
        for (y = 0; y < NY; y++) {
            for (p = 0; p < input_data.np; p++) {
                // ...
```

Listing 3: Loops in the BP algorithm, where `p` is the inner loop variable.

In Listing 2, the variables that iterate over `p` are the following: `min_f`, `r0`, `ant_x`, `ant_y`, `ant_z`, `rc` and `r_vec`. Taking into consideration the wordlength configurations chosen in the previous section, the memory occupied by these data is 3.8MB, as can be observed in Table IV.

In listing 3, the variables that iterate over `x` and `x` are the following: `x_mat`, `y_mat`, `z_mat`, `image`, `rc` and `r_vec`. Taking into consideration the wordlength configurations chosen in the previous section, the memory occupied by these data is 4.8MB, as can be observed in Table V.

The second option, Listing 3 requires an additional 1MB. Also, `rc` variable is accessed randomly, depending on the interpolation results, unlike the other variables, making it impossible to load only smaller batches of the variable. For these reasons, the chosen implementation is Listing 2, with `p` as the outer loop variable.

## IV. BP Hardware Design

The results of the analysis described in the previous section are used to design the accelerator. The circuit was synthesized using Vivado High-Level Synthesis (HLS), from a C++ hardware specification, allowing the development of hardware in a faster and easier way compared to traditional hardware development.

Vivado HLS also facilitated the study mentioned above, with Arbitrary Precision Fixed-Point Data Types, or `ap_fixed` types. These data types allow the definition of arbitrary precision integer or fixed-point values, as seen in Listing 4.

```

#define IMAGE_TBITS 46 // number of total bits
#define IMAGE_IBITS  1 // number of integer bits

ap_fixed<IMAGE_TBITS, IMAGE_IBITS> image;
```

Listing 4: Definition of the `image` variable using the `ap_fixed` types from Vivado HLS with the chosen configuration after the study.

`ap_fixed` types are used to represent the input data of the hardware accelerator, with the wordlengths already defined as seen in Section III-A and Table III.

The following alterations were applied to the code in Listing 1:

- the pixel calculation is performed in blocks, with the order decided in section III-B, as seen in Listing 5, to prevent the transfer of large quantities of data to the accelerator. The functions `read_rc()` and `read_x_mat` transfer data as needed.
- the Inverse Fast Fourier Transform (IFFT) is performed in fixed-point using the IP developed by Xilinx.

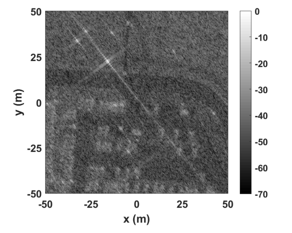
```
void bp(/* ... args in ap_fixed format ...*/)
{
    // aux variables declaration

    for (int b = 0; b < NR_BLOCKS; b++) {
        init_block(image);
        for (int p = 0; p < NP; p++) {
            read_rc(p, data_rc, rc);
            for (int x = 0; x < NX; x++) {
                read_x_mat(x, data_x_mat, x_mat);
                for (int y = b * BLOCK_SIZE; (y < (b + 1) *
     BLOCK_SIZE) && (y < NY); y++) {
                    // content of the loop,
                    // as seen in Listing 1
                }
            }
        }
        write_block(b, image, data_image);
    }
```

Listing 5: BP algorithm implementation in C.

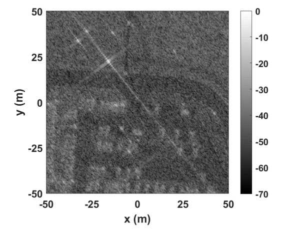
## V. Results

The precision analysis performed on every variable of the BP algorithm allowed the reduction of the wordlength at a variable-level. In other words, instead of choosing a fixed size for all variables, every variable has the minimum number of bits it requires to generate a high-quality image. Variables were originally represented by double-precision floating-point variables, that is, 64-bit variables. The custom approach developed and used in this work resulted in an average of 31.8 bits per variable, a reduction of 49.7%. The smallest variable has 21 bits and the largest 46 bits. The image generated using the results of this study has a SSIM value of 0.99, the value we considered high-quality. A comparison of the original image with full-precision with the image generated using the optimized BP algorithm implementation is presented in Figure 3. To the naked eye, there is no discernible difference between the original image, Figure 3a, and the image generated using the optimized architecture, Figure 3b.

The developed accelerator was synthesized, simulated and fully validated in Vivado HLS. The resource consumption reports were obtained considering the target device, a Zynq UltraScale+ MPSoC ZCU104. The estimate clock frequency for the accelerator is 114 MHz, and the resource estimates are presented in Table VI. The data transferred from memory to the accelerator, the variables in Table IV, is stored in BRAMs and URAMs and is used to generate the image. The low percentage of used LUTs and DSPs, 32% and 3%, respectively, also indicates it that there are enough resources available for parallel units.



(a) Original image generated using the BP algorithm implementation in C with double-precision variables.



(b) Image generated using the optimized algorithm with customized wordlength. Has a SSIM of 0.99.

Fig. 3: Original image, generated using the algorithm with double-precision images compared with the image generated using the optimized algorithm with customized wordlength. There is no discernible difference between these two images.

TABLE VI: Resources estimation according to Vivado HLS.

| **BRAMs** | | **URAMs** | | **LUTs** | | **DSPs** | |
|---|---|---|---|---|---|---|---|
| **Used** | **%** | **Used** | **%** | **Used** | **%** | **Used** | **%** |
| 44 | 7 | 39 | 40 | 74827 | 32 | 63 | 3 |

## VI. Related Work

Several works have addressed the implementation of the BP algorithm in hardware. These works target different technologies, and employ different design methodologies to achieve different end goals.

An area efficient implementation of the BP algorithm has been described in [6], where the algorithm is divided into blocks. The precision necessary for each of these blocks is then evaluated, by modifying the mantissa width between 1 and 52 bits. The minimum exponent width is also determined. The quality of the image is evaluated using two different metrics, Peak Signal-to-Noise Ratio (PSNR) (in dB) and SSIM (value between 0 and 1). The main difference between these two metrics is that while PSNR measures the ratio between the maximum signal power and the noise corrupting the image, SSIM is based on the human visual perception of an image and how similar two images appear to the naked eye. In [6] it was considered that images with a SSIM above 0.99 were acceptable, which was able to achieve with mantissa widths between 6 to 27 bits. This work targeted Application-Specific Integrated Circuit (ASIC) technology and its main concern is area reduction. There is no reference to the execution times of the algorithm or power consumption. Contrary to [6], the work described in this paper uses custom wordlengths for each variable, offering a more fine-tuned solution. Instead of fixed-point, this work also uses fixed-point values.

An optimized software implementation of the BP algorithm was developed in [7], where the cosine and sine operations are replaced with approximations. The approximations are used as a reduced-precision redundancy mechanism, however, they could be used to optimize the algorithm, introducing slight degradation. The approximations are used only in the sine and cosine operations, representing a subset of the operations targeted in this work.

A work regarding custom floating-point units is described in [8], where custom arithmetic units were developed and tested with several benchmark programs. For the majority of benchmark programs, the accuracy of the result was not affected when the width of the mantissa was reduced by half.

Real-time BP implementations have also been presented already [9]–[11], however they target Graphical Processing Unit (GPU) clusters or many-core processors, which do not meet the requirements for power and size that an on-board system requires.

All in all, there are many works dedicated to optimized implementations of the BP. However, none of these works has employed any optimization methods such as the one described in this paper.

## VII. Conclusions and Future Work

This paper presented an optimization methodology for the development of custom signal processing architectures. While we used the BP algorithm as an example for the methodology, other algorithms can benefit from it, to generate optimized architectures for hardware. The optimization methodology lead to a reduction of 49.7% of the wordlength of the variables used in the algorithm while still producing high-quality images. Besides custom wordlength optimization, the data dependency study provided crucial information for the implementation of the architecture, minimizing data transfers between the accelerator and memory.

Future work will involve further optimization of the architecture, such as parallelization of the algorithm, especially considering the low resource estimation occupied by the accelerator, and the data transferred to the accelerator can be used by many units simultaneously.

## Acknowledgment

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UIDB/50021/2020 and PTDC/EEI-HAC/31819/2017 (SARRROCA). Helena Cruz would like to acknowledge Fundação para a Ciência e a Tecnologia for the support through grant SFRH/BD/144133/2019.

## References

[1] M. Soumekh, *Synthetic Aperture Radar Signal Processing with MATLAB Algorithms*, J. W. t. Sons and Inc., Eds. New York: John Wiley & Sons, 1999. [Online]. Available: https://www.wiley.com/en-pt/Synthetic+Aperture+Radar+Signal+Processing+with+MATLAB+Algorithms-p-9780471297062

[2] I. G. Cumming and F. Wong, "Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation." Artech House, 2005.

[3] L. A. Gorham and L. J. Moore, "SAR image formation toolbox for MATLAB," vol. 7699, 2010, pp. 46 —- 58. [Online]. Available: https://doi.org/10.1117/12.855375

[4] D. Pritsker, "Efficient Global Back-Projection on an FPGA," *2015 IEEE Radar Conference (RadarCon)*, pp. 0204—0209, 2015.

[5] C. H. C. Jr., L. A. Gorham, M. J. Minardi, S. M. Scarborough, K. D. Naidu, and U. K. Majumder, "A challenge problem for 2D/3D imaging of targets from a volumetric data set in an urban environment," vol. 6568, 2007, pp. 97 —- 103. [Online]. Available: https://doi.org/10.1117/12.731457

[6] J. J. Pimentel, A. Stillmaker, B. Bohnenstiehl, and B. M. Baas, "Area efficient backprojection computation with reduced floating-point word width for SAR image formation," *2015 49th Asilomar Conference on Signals, Systems and Computers*, pp. 732—726, 2015.

[7] H. Cruz, "On-Board Multi-Core Fault-Tolerant SAR Imaging Architecture," 2018.

[8] J. Tong, D. Nagle, and R. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273—286, 2000.

[9] M. Gocho, N. Oishi, and A. Ozaki, "Distributed Parallel Backprojection for Real-Time Stripmap SAR Imaging on GPU Clusters," *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 619—620, 2017.

[10] T. M. Benson, D. P. Campbell, and D. A. Cook, "Gigapixel Spotlight Synthetic Aperture Radar Backprojection Using Clusters of GPUs and CUDA," *2012 IEEE Radar Conference*, pp. 0853—0858, 2012.

[11] J. Park, P. T. P. Tang, M. Smelyanskiy, D. Kim, and T. Benson, "Efficient backprojection-based synthetic aperture radar computation with many-core processors," *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1—11, 2012.