The Challenge of Onboard SAR Processing: A GPU Opportunity

Diego Romano[1(✉)], Valeria Mele[2], and Marco Lapegna[2]

[1] Institute for High Performance Computing and Networking (ICAR), CNR, Naples, Italy
diego.romano@cnr.it
[2] University of Naples Federico II, Naples, Italy
{valeria.mele,marco.lapegna}@unina.it

Abstract. Data acquired by a Synthetic Aperture Radar (SAR), onboard a satellite or an airborne platform, must be processed to produce a visible image. For this reason, data must be transferred to the ground station and processed through a time/computing-consuming focusing algorithm. Thanks to the advances in avionic technology, now GPUs are available for onboard processing, and an opportunity for SAR focusing opened. Due to the unavailability of avionic platforms for this research, we developed a GPU-parallel algorithm on commercial off-the-shelf graphics cards, and with the help of a proper scaling factor, we projected execution times for the case of an avionic GPU. We evaluated performance using ENVISAT (Environmental Satellite) ASAR Image Mode level 0 on both NVIDIA Kepler and Turing architectures.

Keywords: Onboard SAR focusing · GPU-parallel · Range-Doppler algorithm

1 Introduction

In the domain of environmental monitoring, Synthetic Aperture Radar (SAR) plays an important role. It is an active microwave imaging technology for remote sensing, which can be employed for observations in all-day and all-weather contexts. Satellites and aircraft have limited space for a radar antenna, therefore a SAR sensor creates a synthetic aperture by exploiting their motion. As a platform moves along a direction (called *azimuth* direction), the sensor transmits pulses at right angles (along *range* direction) and then records their echo from the ground (see Fig. 1).

Thanks to its synthetic aperture, SAR systems can acquire very long land swaths organized in proper data structures. However, to form a comprehensible final image, a processing procedure (*focusing*) is needed.

The focusing of a SAR image can be seen as an inherently space-variant two-dimensional correlation of the received echo data with the impulse response of the system. Radar echo data and the resulting Single-Look Complex (SLC)

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12139, pp. 46–59, 2020.
https://doi.org/10.1007/978-3-030-50420-5_4

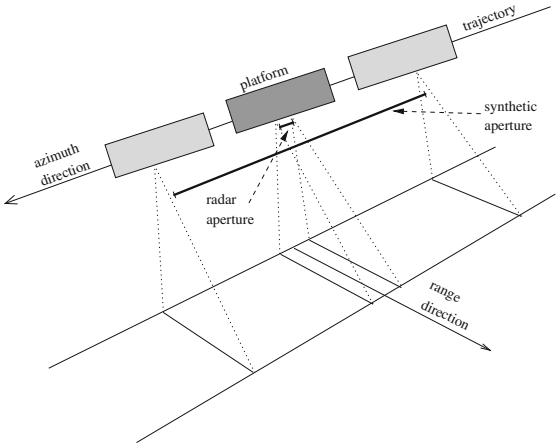

Fig. 1. Representation of the synthetic aperture created by a moving platform provided with a sensor.

image are stored in matrices of complex numbers representing the in-phase and quadrature (i/q) components of the SAR signal. Several processors are available, based on three main algorithms: Range-Doppler, ωk, and Chirp Scaling [7].

Usually, this processing takes time and needs HPC algorithms in order to process data quickly. Heretofore, considering the limited computing hardware onboard, data had been transmitted to ground stations for further processing. Nevertheless, the vast amount of acquired data and the severely limited downlink transfer bandwidth imply that any SAR system also needs an efficient raw data compression tool. Because of structures with apparent higher entropy, a quasi-independence of in-phase and quadrature components showing histograms with nearly Gaussian shape and identical variance, conventional image compression techniques are ill-suited, and resulting compression rates are low.

Thanks to advances in the development of avionic specialized computing accelerators (GPUs) [1,12], now the onboard SAR processing with real-time GPU-parallel focusing algorithms is possible. These could improve sensor data usability on both strategic and tactical points of view. For example, we can think of an onboard computer provided with a GPU directly connected to both a ground transmitter and a SAR sensor through GPUDirect [13] RDMA [5] technology.

Several efforts have been made to implement GPU SAR processors for different raw SAR data using CUDA Toolkit. In [4], the focusing of an ERS2 image with $26,880 \times 4,912$ samples on an NVIDIA Tesla C1060 was obtained in 4.4 s using a Range-Doppler algorithm. A similar result is presented in [14], where a COSMO-SkyMed image of $16,384 \times 8,192$ samples has been processed employing both Range-Doppler and ωk algorithms in 6.7 s. Another implementation of the ωk algorithm, described in [20], focused a Sentinel-1 image with $22,018 \times 18,903$ in 10.87 s on a single Tesla K40, and 6.48 s in a two GPUs

configuration. In [15], a ωk-based SAR processor implemented in OpenCL and run on four Tesla K20 has been used to focus an ENVISAT ASAR IM image of $30,000 \times 6,000$ samples in 8.5 s and a Sentinel-1 IW image of $52,500 \times 20,000$ samples in 65 s. All these results have accurately analyzed the ground station case, where one or more Tesla GPU products have been used.

Our idea is to exploit the onboard avionic GPU computing resources, which are usually more limited than the Tesla series. For example, on the one hand, the avionic EXK107 GPU of the Kepler generation is provided with 2 Streaming Multiprocessors (SMs), each with 192 CUDA core. On the other hand, the Tesla K20c, of the same architecture generation, has 13 SMs, also with 192 CUDA core each.

Historically, the development of SAR processors has been characteristic of the industrial sector, and therefore there is little availability of open-source processors. This work is based on the *esarp* processor within the GMTSAR processing system [17], a focuser written in C and implementing a Range-Doppler algorithm (Fig. 2) for ERS-1/2, ENVISAT, ALOS-1, TerraSAR-X, COSMOS-SkyMed, Radarsat-2, Sentinel-1A/B, and ALOS-2 data. For testing convenience, the GPU-parallel processor herein presented is limited to ENVISAT ASAR Image Mode level 0 data [18], but with a reasonably little effort, it can be adapted to other sensors raw data.

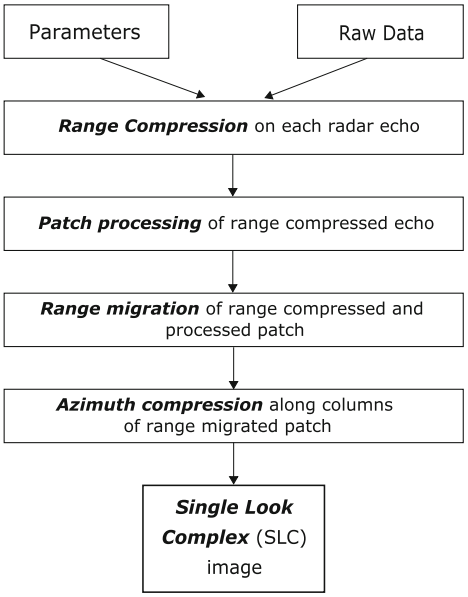

Fig. 2. Range-Doppler Algorithm flow in *esarp* processor

This paper shares the experiences gathered during the testing of a prototype HPC platform, whose details are subject to a non-disclosure agreement and therefore excluded from this presentation. However, several insights can be useful to discuss new approaches in the design of SAR processing procedures and strategies. Indeed, from previous experiences in GPU computing, which also included special devices ([8–11,16]), we can make some assumptions. Furthermore, the reasoning made when dealing with an off-the-shelf hardware solution can be in some way translated to an avionic product, accepting that the algorithmic logic does not change. In order to develop and test our algorithm, with the intent to exploit the massive parallelism of GPUs, we applied the approach proposed in [2].

In the next section, we provide a schematic description of the Range-Doppler algorithm, and we focus on data-parallel kernels that can be efficiently implemented on a GPU. Section 3 presents the actual kernels implemented and their relative footprint in the perspective of avionic hardware. Testing is presented in Sect. 4, with an estimation of the execution time on an avionic GPU. Finally, we discuss results and conclude in Sect. 5.

2 Range-Doppler Algorithm and Identification of Data-Parallel Kernels

The GMTSAR processing system relies on precise orbits (sub-meter accuracy) to simplify the processing algorithms, and techniques such as *clutterlock* and *autofocus* are not necessary to derive the orbital parameters from the data.

In the *esarp* focusing component, data are processed by patches in order not to overload the computing platform. Each patch contains all the samples along the range direction and a partial record along the azimuth direction. Several patches are concatenated to obtain the image for the complete strip.

1. **Range Compression** – In the ENVISAT signal, there are 5681 points along the range direction that must be recovered in a sharp radar pulse by deconvolution with the chirp used during signal transmission. The operation is done in the frequency domain: firstly, the chirp is transformed, then the complex product of each row with the conjugate of the chirp is computed. A Fast Fourier Transform (FFT) is therefore needed before and after the product. In order to take advantage of the speed of radix-2 FFT, data are zero-padded to the length of 8192. This procedure allows obtaining phase information for a longer strip, which will be later reduced to 6144 points for further processing.
2. **Patch Processing** – In order to focus the image in the azimuth direction, data must be transformed in the range-doppler domain, which means in the frequency domain for the azimuth direction, by applying an FFT on the transposed matrix representing the range compressed image. For the ENVISAT radar, the synthetic aperture is 2800 points long. Again, to exploit the speed of radix-2 FFT, 4096 rows are loaded and processed, consisting of a patch. The last 1296 rows are overlapped with the following patch.

3. **Range Migration** – As the platform moves along the flight path, the distance between the antenna and a point target changes, and that point appears as a hyperbolic-shaped reflection. To compensate for this effect, we should implement a remapping of samples in the range-doppler domain through a sort of interpolator. Such a migration path can be computed from the orbital information required by the GMTSAR implementation and must be applied to all the samples in the range direction.
4. **Azimuth Compression** – To complete the focusing in the azimuth direction, a procedure similar to the Range Compression is implemented. In the range-doppler domain, a frequency-modulated chirp is created to filter the phase shift of the target. This chirp depends on: the pulse repetition frequency, the range, and the velocity along the azimuth direction. As before, after the complex product, the result is inversely Fourier transformed back to the spatial domain to provide the focused image.

In the four steps described above, many operations can be organized appropriately, respecting their mutual independence [3]. As shown in Fig. 3, each sub-algorithm corresponds to a GPU kernel exploiting possible data parallelism. The several planned FFTs can be efficiently implemented through cuFFT batching. If the raw data matrix is memorized in a 1-dimensional array with row-major order, all the FFTs in range direction can be executed in efficient batches [19]. When the FFTs runs in the azimuth direction, a pre- and post-processing matrix transpose becomes necessary.

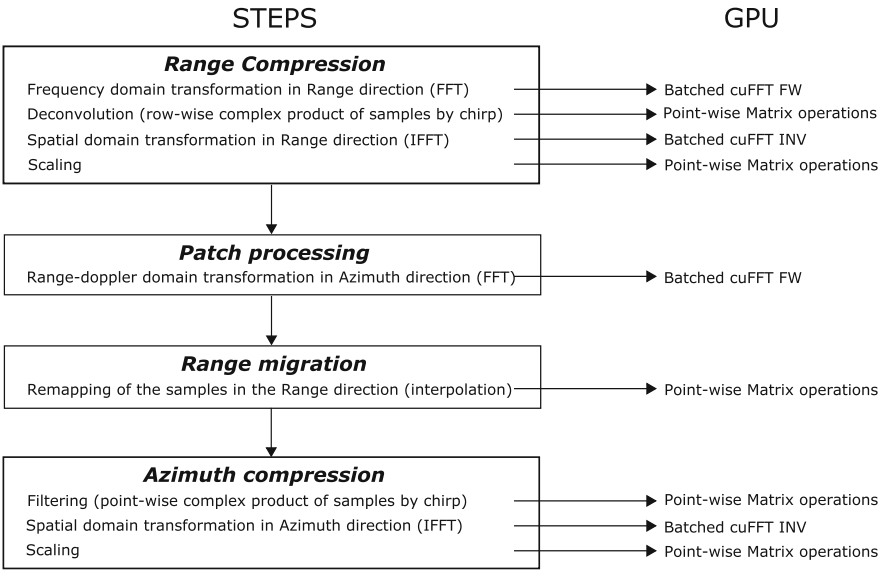

Fig. 3. Steps of the Range-Doppler Algorithm and correspondence with possible data-parallel GPU operations

The filtering sub-algorithms can be easily organized as point-wise matrix operations, assuming that the chirps are available in the device memory for reading. This step is efficiently achievable by building the range chirp directly on the GPU, as it consists of a mono-dimensional array with spatial properties, and by subsequently transforming it in the frequency domain through a proper FFT. Similarly, the azimuth chirp can be built and transformed directly on the GPU, but this time it is a 2-D array.

About the mapping of the samples in the Range direction, assuming enough memory is available for storing the migrated samples, it can be seen as point-wise matrix operation, as each sample corresponds to a previously patch processed data subject to operations involving orbital information.

3 GPU Kernels and Memory Footprint

In order to evaluate the feasibility of onboard processing, we present an analysis of the resources needed.

Firstly, let us observe that cuFFT proposes a convenient function to get an accurate estimate of the additional work area size needed to run a batched plan. Since the dimensions used in the Range-Doppler algorithm for ENVISAT data are a power of 2, that is 8192 complex numbers of 8 bytes each in the range direction for 4096 rows, the additional work area consists of 256 MBytes for the batches in the range direction. Similarly, in the azimuth direction, the batches are organized in 6144 columns of 4096 points, and the additional work area required is about 192 MBytes.

In Algorithm 1, a GPU-parallel pseudo-code presents the kernels and the cuFFT runs of the GPU-parallel version of *esarp*. In the following, we analyze the kernels with their possible sources of Algorithmic Overhead [3] and their memory footprint.

- **d_orbit_coef**: in order to remap the range samples and to compensate platform movement within the range migration step, for each sample in the range, there are 8 parameters describing the orbit characteristics and their influence on the migration. These parameters are the same for each row of the patch, and they are scaled considering the position in the synthetic aperture, that is the position in the azimuth direction. They are also useful to put up the chirp in the azimuth direction. To save useless recomputing, this kernel precomputes 8 arrays of 6144 elements with a corresponding memory footprint of 384 KBytes. Their values can be computed independently by 6144 threads in an appropriate thread-block configuration that takes into account the number of SMs in the GPU.
- **d_ref_rng**: this kernel populates an array with the chirp in range direction based on the pulse emitted by the sensor. The array is also zero-padded to the length of the nearest power of 2 to exploit subsequent radix-2 FFT efficiency. For the ENVISAT data, the array consists of 8192 complex numbers of 8 bytes each, i.e., 64 Kbytes. The workload of this kernel is proportional to the number of elements in the array. Moreover, each element can be processed

Algorithm 1: esarp on GPU

```
Result: SAR focused image
initialization;
d_orbit_coef(coef) ;          // kernel to create arrays with orbital info
d_ref_rng(r_ref) ;                          // kernel to set up range chirp
cuFFT(r_ref,FW) ;          // transform range chirp in frequency domain
d_ref_az(a_ref) ;                        // kernel to set up azimuth chirp
cuFFT(a_ref,FW) ;        // transform azimuth chirp in frequency domain
while patches to be focused do
    receive patch;
    // Range compression
    cuFFT(patch,FW) ;               // transf. freq. in range direction
    d_mul_r(patch,r_ref) ;       // kernel for deconvolution in range dir.
    cuFFT(patch,INV) ;              // transform back in spatial domain
    d_scale(patch) ;                // kernel for scaling partial results

    // Patch processing
    d_trans_mat(patch) ;                      // kernel to transpose patch
    cuFFT(patch,FW) ;             // transf. freq. in azimuth direction

    // Range migration
    d_intp_tot(patch,coef) ;      // kernel to remap samples in range dir.

    // Azimuth Compression
    d_mul_a(patch,a_ref) ;         // kernel for filtering in azimuth dir.
    cuFFT(patch,INV) ;              // transform back in spatial domain
    d_scale(patch) ;                       // kernel for scaling results
    d_trans_mat(patch) ;                    // kernel to transpose patch
end
```

independently of the others, meaning that the workload can be split among threads. If those are organized in a number of blocks, which is multiple of the number of SMs present in the GPU, we can have a good occupancy of the devices. Also, the divergence induced by the zero-padding can be minimized during thread-block configuration.

– **d_ref_az**: by using previously calculated orbital parameters, a 2-D array of the same size of the patch is populated with the chirp in the azimuth direction, which is different for each column. Hence, the memory footprint is $6144 \cdot 4096 \cdot 8 = 192$ MBytes. Beforehand, the array is reset to zero values since not all the samples are involved in the filtering. To limit divergence, each element in the array can be assigned to a thread that populates the array if necessary, or it waits for completion. Since the same stored orbital parameters are used for each row, the threads can be arranged in blocks with column-wise memory access in mind in order to limit collisions among different SMs. Hence, the execution configuration can be organized in a 2-D memory grid with blocks of threads on the same column.

- **d_mul_r**: implements a point-wise multiplication of each row of the patch by the conjugate of the chirp in the frequency domain. The workload can be assigned to independent threads with coalescent memory accesses. Following reasoning similar to d_ref_az, with the idea of limiting memory collisions, each thread in a block can compute one column of the patch in a *for* cycle, realizing a coalesced write of the results with the other threads in the same warp. This kernel does not require additional memory occupation.
- **d_scale**: after the inverse FFT needed to transform the patch back to the spatial domain, a point-wise scaling is needed. As before, independent threads can work with coalescent memory accesses, and efficient workload assignments can be configured.
- **d_trans_mat**: this kernel follows the highly efficient sample proposed in [6]. In this case, the memory footprint corresponds to a new array with the same dimension of the patch, i.e., 192 MBytes.
- **d_intp_tot**: the remapping of the samples is carried on in a point-wise procedure. The output patch must be in a different memory location, and therefore the memory footprint consists again of an additional 192 MBytes. Making similar reasoning on the memory accesses as we did for the d_ref_az kernel, we can configure the execution to minimize global memory collisions, optimizing block dimensions for occupancy.
- **d_mul_a**: this kernel filters the patch to focus the final image in the frequency domain. The operations consist of element-wise matrix products and do not need additional work area in memory. An efficient thread-block configuration can follow the reasoning made for the previous kernel.

To summarize the analysis of the memory footprint for the whole procedure to focus a patch: 192×2 MBytes are necessary to swap the patch for transposing and remapping data in several kernels, 256 MBytes are necessary for the most demanding FFT, and the preliminary computing of chirps and orbit data require $\approx$192.5 MBytes. The total is less than 1 GByte of memory, which is a fair amount available on every GPU.

4 Testing on Workstation and Reasoning on Avionic Platform

As mentioned in the introduction, we had access to a prototype avionic platform for testing purposes, and we had the opportunity to run our algorithm repeatedly. Even if we cannot disclose details about platform architecture and testing outcomes due to an NDA, we can refer to the GPU installed, which is an Nvidia EXK107 with Kepler architecture.

In this section, we will present the results collected on a workstation with a Kepler architecture GPU (see Table 1), to propose some reasoning on the avionic platform with the help of a scale factor, and on another workstation with a Turing architecture GPU (see Table 2) to evaluate the running time on a more recent device.

Table 1. Workstation used for testing on Kepler architecture

	Workstation Kepler
OS	Ubuntu 18.04
CPU	Intel Core i5 650 @3.20 GHz
RAM	6 GB DDR3 1333 MT/s
GPU	GeForce GTX 780 (12 SMs with 192 cores each)

Let us consider the execution time of our GPU version of the esarp processor, excluding any memory transfer between host and device, i.e., considering data already on the GPU memory. Such is a fair assumption since all the focusing steps are executed locally without memory transfers between host and device. In an avionic setting, only two RDMA transfers happen: the input of a raw patch from the sensor, the output of a focused patch to the transmitter (Fig. 4).

If we call t_{wk} the execution time for focusing a patch on the *Workstation Kepler*, and t_a the execution time to focus a patch on an avionic platform provided with an EXK107 GPU, from our testing we noticed a constant scale factor:

$$s_f = \frac{t_{wk}}{t_a} = 0.23$$

It should not be considered a universal scale factor for whatever kernel run on both devices. However, it is a constant behavior on the total execution time to focus whatever patch from ENVISAT ASAR IM data using our GPU-parallel version of the esarp processor. Therefore s_f is useful to estimate the time needed to focus a swath on an avionic platform using such application.

To verify the functionalities of the focusing algorithm, we used data freely available from http://eo-virtual-archive4.esa.int. Measures presented in this section are relative to the processing of the image in Fig. 5 subdivided in 9 patches.

Table 2. Workstation used for testing on Turing architecture

	Workstation Turing
OS	CentOS 7.6
CPU	Gold Intel Xeon 5215
RAM	94 GB
GPU	Quadro RTX 6000 (72 SMs with 64 cores each)

In Table 3 we present the execution times of the GPU-esarp software, relatively to the steps of the Range-Doppler algorithm, on Workstation Kepler. The preliminary processing step, which includes the creation of arrays containing orbital information and chirps in both range and azimuth direction, is executed

just for the first patch, as the precomputed data do not change for other patches within the same swath. The total execution time needed to focus the whole image is $t_{wk} = 1.12\,\text{s}$, excluding input-output overhead and relative memory transfers between host and device.

We can, therefore, expect that the execution time needed on the avionic platform is:

$$t_a = \frac{t_{wk}}{s_f} = 4.87\ \text{s}$$

which is less than the ENVISAT stripmap acquisition time $t_{in} \approx 16\,\text{s}$ for the relative dataset. Moreover, if each sample of the resulting image consists of a complex number of 16 bits, the total size of the output is $\approx 295\,\text{MBytes}$. In a pipelined representation of a hypothetical avionic system, as pictured in Fig. 4, all data transfers are subject to their respective connection bandwidth. Considering that the payload communication subsystem of the ENVISAT mission had a dedicated bandwidth for SAR equipment of 100 Mbit/s, the time necessary to transmit the result to the ground would be $t_{out} \approx 24\,\text{s}$. That is, we can suppose that:

$$t_a < t_{in} < t_{out}$$

hence, we have an expected GPU-parallel focusing algorithm able to satisfy real-time requirements on an EXK107 device.

If we consider the execution times on Workstation Turing (Table 4), we see that the total time needed to focus the whole image is $t_{wt} = 0.208\,\text{s}$, excluding input-output transfers, which is very promising for the next generation of avionic GPUs. Moreover, considering the spare time available for further processing during down-link transmission, we can think about computing Azimuth FM rate and Doppler Centroid estimators. Those algorithms are useful to provide parameters for Range Migration, and Azimuth Compression steps in case of non-uniform movements of the platform, as it happens on airborne SAR.

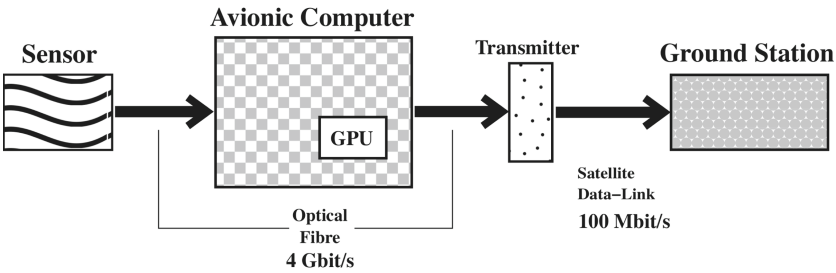

Fig. 4. Transfer data rates in an avionic system: sensors are usually connected to the computer unit through Optical Fibre, which allow rates of the Gbit/s magnitude or more; within the Avionic Computer, GPUs allows transfers at rates with a magnitude of Gbit/s; at the end of this pipeline, a data-link connection to the ground station can transfer with a maximum rate of 100 Mbit/s with current technology.

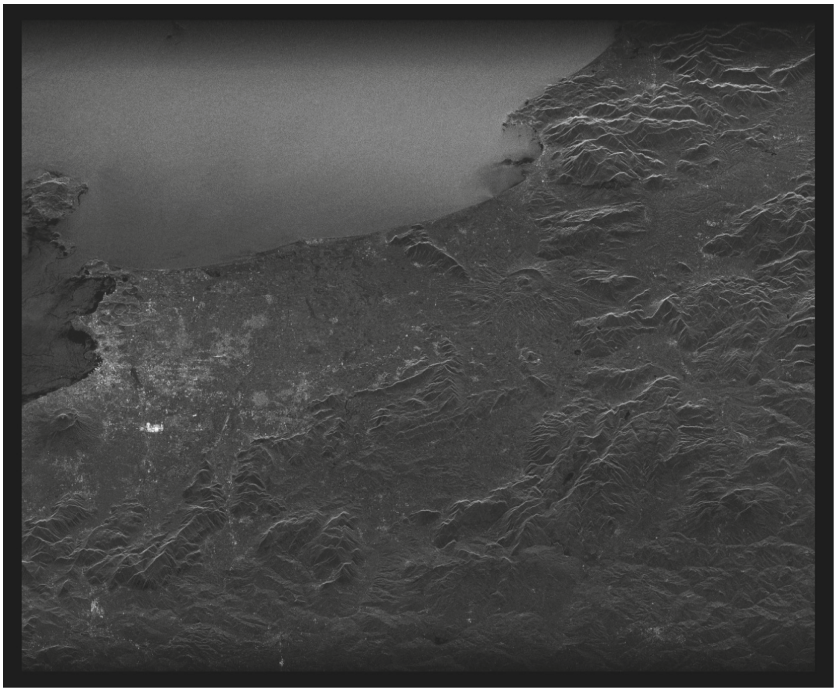

Fig. 5. Focused SAR image of Napoli area, consisting of 6144 samples in the range direction and 25200 samples in the azimuth direction. The sampled area is $106 \times 129\,\mathrm{Km}^2$, with an Azimuth resolution of 5 m. For rendering purposes, here the image is proposed with vertical range direction and with the azimuth direction squeezed to map on square pixels.

Table 3. Execution times in milliseconds for each step of the GPU-esarp software on the Workstation Kepler

	Execution time in milliseconds								
Preliminary processing	21.7								
Range compression	46.9	46.2	46.2	46.1	46.3	46	45.8	45.9	46
Patch processing	4.8	4.8	4.8	4.8	4.8	4.8	4.8	4.7	4.7
Range migration	47.8	47.1	46.9	46.9	47.2	46.9	47.1	47.2	47.3
Azimuth compression	24.4	24.1	24.1	24.2	24.3	24.2	24.8	24.1	24.1
Total (excl. I/O)	145.4	122.2	122	122	122.6	121.9	122.5	121.9	122.1
Patch	1	2	3	4	5	6	7	8	9

Table 4. Execution times in milliseconds of the GPU-esarp software on the Workstation Turing

	Execution time in milliseconds								
Total (excl. I/O)	28.5	24.3	22.9	22.3	22.3	22.2	22	22	22
Patch	1	2	3	4	5	6	7	8	9

5 Conclusions

When thinking about SAR sensing, a common approach is to consider it as an instrument for delayed operational support. Usually, SAR raw data are compressed, down-linked, and processed in the ground stations to support several earth sciences research activities, as well as disaster relief and military operations. In some cases, timely information could be advisable, and onboard processing is becoming an approach feasible thanks to advances in GPU-technology with reduced power consumption.

In this work, we developed a GPU-parallel algorithm based on the Range-Doppler algorithm as implemented in the open-source GMTSAR processing system. The results, in terms of execution time on off-the-shelf graphics cards, are encouraging if scaled to proper avionic products. Even if we did not present actual results on an avionic GPU, thanks to some insights acquired during testing of a prototype avionic computing platform and a constant scale factor, we showed that onboard processing is possible when an efficient GPU-parallel algorithm is employed.

Since this result is based on the algorithmic assumption that orbital information is available, some processing techniques such as *clutterlock* and *autofocus* have been avoided. That is the case for many satellite SAR sensors, but further experiments must be carried on to verify the feasibility of onboard processing on airborne platforms, where parameters like altitude and velocity may slightly change during data acquisition. In this sense, as future work, we plan to implement a GPU-parallel algorithm for parameters estimation.

References

1. GRA112 graphics board, July 2018. https://www.abaco.com/products/gra112-graphics-board
2. D'Amore, L., Laccetti, G., Romano, D., Scotti, G., Murli, A.: Towards a parallel component in a GPU–CUDA environment: a case study with the L-BFGS harwell routine. Int. J. Comput. Math. **92**(1), 59–76 (2015). https://doi.org/10.1080/00207160.2014.899589
3. D'Amore, L., Mele, V., Romano, D., Laccetti, G.: Multilevel algebraic approach for performance analysis of parallel algorithms. Comput. Inform. **38**(4), 817–850 (2019). https://doi.org/10.31577/cai_2019_4_817
4. di Bisceglie, M., Di Santo, M., Galdi, C., Lanari, R., Ranaldo, N.: Synthetic aperture radar processing with GPGPU. IEEE Signal Process. Mag. **27**(2), 69–78 (2010). https://doi.org/10.1109/MSP.2009.935383

5. Franklin, D.: Exploiting GPGPU RDMA capabilities overcomes performance limits. COTS J. **15**(4), 16–20 (2013)
6. Harris, M.: An efficient matrix transpose in CUDA C/C++, February 2013. https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/
7. Hein, A.: Processing of SAR Data Fundamentals, Signal Processing, Interferometry, 1st edn. Springer, Heidelberg (2010)
8. Laccetti, G., Lapegna, M., Mele, V., Montella, R.: An adaptive algorithm for high-dimensional integrals on heterogeneous CPU-GPU systems. Concurr. Comput.: Pract. Exper. **31**(19), e4945 (2019). https://doi.org/10.1002/cpe.4945. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4945, e4945 cpe.4945
9. Laccetti, G., Lapegna, M., Mele, V., Romano, D.: A study on adaptive algorithms for numerical quadrature on heterogeneous GPU and multicore based systems. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013. LNCS, vol. 8384, pp. 704–713. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55224-3_66
10. Marcellino, L., et al.: Using GPGPU accelerated interpolation algorithms for marine bathymetry processing with on-premises and cloud based computational resources. In: Wyrzykowski, R., Dongarra, J., Deelman, E., Karczewski, K. (eds.) PPAM 2017. LNCS, vol. 10778, pp. 14–24. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78054-2_2
11. Montella, R., Giunta, G., Laccetti, G.: Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. Cluster Comput. **17**(1), 139–152 (2014). https://doi.org/10.1007/s10586-013-0341-0
12. Munir, A., Ranka, S., Gordon-Ross, A.: High-performance energy-efficient multicore embedded computing. IEEE Trans. Parallel Distrib. Syst. **23**(4), 684–700 (2012). https://doi.org/10.1109/TPDS.2011.214
13. NVIDIA Corporation: Developing a Linux Kernel Module Using RDMA for GPUDirect (2019). http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, version 10.1
14. Passerone, C., Sansoè, C., Maggiora, R.: High performance SAR focusing algorithm and implementation. In: 2014 IEEE Aerospace Conference, pp. 1–10, March 2014. https://doi.org/10.1109/AERO.2014.6836383
15. Peternier, A., Boncori, J.P.M., Pasquali, P.: Near-real-time focusing of ENVISAT ASAR Stripmap and Sentinel-1 TOPS imagery exploiting OpenCL GPGPU technology. Remote Sens. Environ. **202**, 45–53 (2017). https://doi.org/10.1016/j.rse.2017.04.006. Big Remotely Sensed Data: Tools, Applications and Experiences
16. Rea, D., Perrino, G., di Bernardo, D., Marcellino, L., Romano, D.: A GPU algorithm for tracking yeast cells in phase-contrast microscopy images. Int. J. High Perform. Comput. Appl. **33**(4), 651–659 (2019). https://doi.org/10.1177/1094342018801482
17. Sandwell, D., Mellors, R., Tong, X., Wei, M., Wessel, P.: GMTSAR: an InSAR processing system based on generic mapping tools (2011)
18. Schättler, B.: ASAR level 0 product analysis for image, wide-swath and wave mode. In: Proceedings of the ENVISAT Calibration Review. Citeseer (2002)

19. Střelák, D., Filipovič, J.: Performance analysis and autotuning setup of the cuFFT library. In: Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems, ANDARE 2018. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3295816.3295817
20. Tiriticco, D., Fratarcangeli, M., Ferrara, R., Marra, S.: Near real-time multi-GPU ωk algorithm for SAR processing. In: Agency-Esrin, E.S. (ed.) Big Data from Space (BiDS), pp. 277–280, October 2014. https://doi.org/10.2788/1823