Synthetic Aperture Radar Signal Processing using GPGPU

Mónica Denham[1,2], Javier Areta[1,2], Isidoro Vaquila[2,5], and Fernando G. Tinetti[3,4]

[1] Ingeniería Electrónica, Sede Andina, Universidad Nacional de Río Negro, Argentina
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[3] Instituto de Investigación en Informática LIDI, Facultad de Informática - UNLP, La Plata, Buenos Aires, Argentina
[4] Comisión de Investigación Científicas de la Provincia de Buenos Aires (CIC), Argentina
[5] INVAP S.E., Argentina
mdenham@unrn.edu.ar, jareta@unrn.edu.ar, ivaquila@invap.com.ar, fernando@lidi.unlp.edu.ar

Abstract. In this work an efficient parallel implementation of the Chirp Scaling Algorithm (CSA) for Synthetic Aperture Radar (SAR) processing is presented. The architecture selected for the implementation is General Purpose Graphic Processing Unit (GPGPU), as it is well suited for scientific applications and real time implementation of algorithms. The analysis of a first implementation led to several improvements which resulted in an important final speedup. Details of the issues found are explained, and the performance improvement of their correction explicitly shown.

1 Introduction

Synthetic Aperture Radar [3] [22] [19] [21] is a mature technology that combines radar and signal processing to enable high resolution imaging of the earth surface. The principle of operation is based on the coherency of the different radar images obtained during a known trajectory -usually linear and constant speed- and the fact that the same point captured at different spatial points contains different and predictable doppler shifts. This doppler information is embedded in the phase, which has to be properly processed in order to obtain a higher quality image, which turns out to be independent of the antenna aperture and the range. The measurements are usually arranged in two dimensional arrays, matrices, that contain the sampled echoes of the signals emitted at a fixed point in space in columns while rows correspond to sampled echoes taken at another point in space [7] [3]. Due to the time constants involved, range echoes are sampled at a rate of millions per second (MHz) while spatial -or azimuth- samples are taken in the order of seconds, its common to refer them as fast and slow time dimensions.

The obtained data matrix, usually deemed raw data, contains in the order of millions of elements. Processing this data to obtain a focused image takes a considerable amount of processor load and consequently may not be suited for real time operation under the usual paradigm of sequential programming. Due to the nature of the problem it is well suited for parallel implementation. In this regard, General Purpose Graphic Processing Unit (GPGPU) [9] [5] [13] is a powerful platform that allows the implementation of complex processing algorithms and is very well suited for this application [8] [5].

There are several known algorithms for SAR image focusing, RDA (Range Doppler Algorithm), CSA (Chirp Scaling Algorithm), ω-K algorithm, Back-Projection algorithm and others [3] [22] [7] [10]. Most of them work both on frequency and spatial coordinates, so the use of Discrete Fourier Transform is required, which itselfs has very efficient parallel implementations.

In this work an implementation of CSA is carried out based on [12], analyzing its behaviour and proposing several improvements in order to take advantage of parallelization. The remainder of the paper is organized as follows. Section 2 introduces the GPGPU architecture and the C-CUDA programming tool. In section 3 the CSA is reviewed and the main operations required are presented. Section 4 shows the experimental results and the analysis of the algorithm implementation, showing the performance improvement obtained after correcting the detected issues. Finally section 5 presents the conclusions and the future work on this topic.

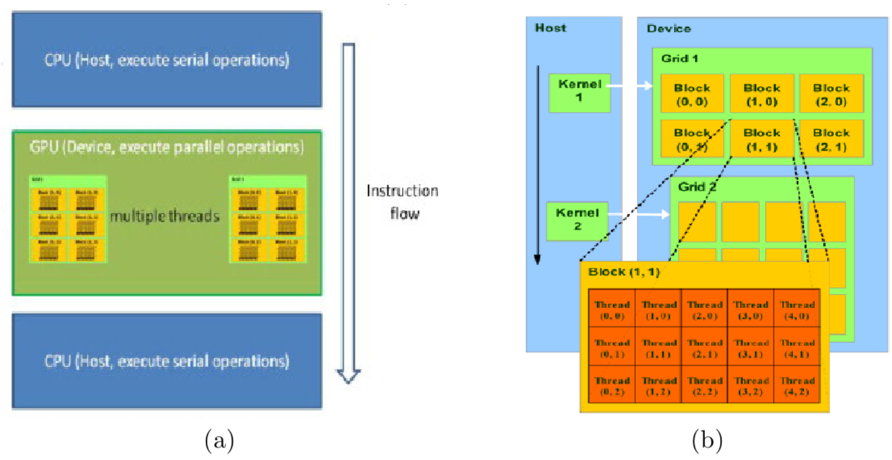
2 GPGPU and C-CUDA

Originally GPUs were developed for the video game industry. These cards were designed to achieve high performance in video game applications, where many similar simple computations had to be done in parallel, thus several basic ALUs were used in an independent graphics chip (deemed graphics processing unit, GPU) in order to alleviate the main CPU load. As their computational power grew up while keeping prices low, GPU became an atractive architecture for High Performance Computing. Simple and dedicated cores became complex and general purpose cores. First generation GPU cores could perform some specific operations suited for graphical pipeline. Current GPUs cores have grown in complexity and can perform a wide set of operations. This revamped architecture is called GPGPU (General Purpose GPU) [18] [9] [2].

The broader use of GPGPU architectures was motivated by the creation and definition of several tools for developing and programming GPGPU applications. Perhaps the most extended of such applications is CUDA (Compute Unified Device Architecture), which is a language (extensions of other languajes such us C, C++, phyton, etc), a compiler and a programming model [20] [9] [5] [16] [2].

Graphic cards are used in conjunction with a CPU, which governs GPU execution. GPU applications are hybrid programs combining sequential and parallel code. Sequential code is executed on CPU and parallel code is executed on the graphic card. Figure 1 (a) shows the structure of such a hybrid CUDA program.

GPU arquitectures have a memory hierarchy composed of different memory kinds. Each kind of GPU memory has its own access velocity, latency, bandwidth,



(a) (b)

Fig. 1. (a) Hybrid program CUDA model [16]. (b) Kernel arrange example [16].

access pattern, cost, etc. It is necessary to take this hierarchy into account in order to achieve high performance.

Parallel CUDA code is executed in kernels. A kernel is a function that is executed by several threads. These threads are arranged in arrays of 1, 2 or 3 dimentions. A kernel launch implies the creation of large amount of threads. Thread layout is another important design key for achieving high performance.

CUDA applications are suited for applications with the following characteristics: high computational requirements (quadratic or higher), high workload for each thread, small data dependencies, low CPU-GPU data communication, no critical sections. Almost all of these features compensate the cost of data communication between CPU and GPU, which is a resource consuming operation. Critical sections deserve to be executed in sequential order. This fact attempts against applications throughput and performance [18] [5].

Ideally, CUDA model implies a data initialization in CPU, data transfer from CPU to GPU, parallel GPU execution and finally, communication of results from GPU to CPU.

In the following sections we will show how these steps apply to a particular application, namely the Chirp Scaling Algorithm for SAR image formation. Several SAR signal processing algorithms share these characteristics as they make heavy use of discrete Fourier tansform.

3 Parallel Chirp Scaling Algorithm

SAR can be viewed as the combination of a coherent Radar system and the posterior application of signal processing techniques. This posterior processing allows for the improvement of the azimuth resolution, making it independent of range. A moving platform with a radar mounted on it sends succesive pulses towards an area of interest -defined by the antenna lobe-, different sets of pulses are transmitted at different positions in space given by the ship trajectory and the spatial sampling rate. Consecutive echoes, deemed raw data, are collected and stored in matrix form. In this matrix the azimuth dimension corresponding to the different trajectory samples are stored in rows, while the consecutive pulses/echoes from a single point in space that represent range are stored in columns. Processing raw SAR data consists in coherntly combining the information of all received signals in order to form (focus) the image.

Chirp Scaling Algorithm (CSA) is based on properties of chirp (linear FM) signals [3] which appears naturally in the azimuth direction, and also in the range direction as usually chirp waveforms are used by the Radar system. Chirp signals achieve high resolution using short pulses with large bandwidth.

CSA performs the signal compression and correction of range cell migration (RCM) using matched filters and focuses the data over range and azimuth dimentions. These matched filters are implemented using so called Phase Functions, that are pre calculated based on the scenario parameters and making use of the aforementioned chirp signal properties. In this algorithm, total RCM is divided in two parts: a “bulk RCM” and a “differential RCM”. The bulk RCM is the same for all targets and it is range invariant. Differential RCM is the remaining part, it is range dependent and is smaller than bulk RCM. Each part is then corrected using different types of operations [3] [12].

Image 2 shows the main steps of Chirp Scaling Algorithm.

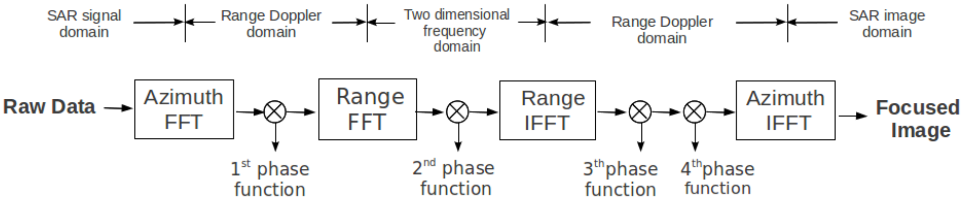


Fig. 2. Block diagram of Chirp Scaling Algorithm.

In this algorithm appropriate matched filters are used in different domains. FFTs and IFFTs are used in order to change to range-doppler domain, frequency domain or time domain. For solving these operations, we used the CUFFT CUDA library. Next subsection introduces the main features of this library.

CUFFT library: this library solves the Fast Fourier Tranforms using C CUDA [17]. FFT and inverse FFT are divide and conquer algorithms for efficiently computing discrete Fourier transforms of complex or real data sets. CUFFT is based on FFTW3 [6] [11] routines which are efficient secuencial implementations of the FFT. To use it, the user creates a plan with the main features of the FFT (tranform dimensions, array sizes, data types and layout, direction for executing FFT or IFFT, etc). Then the CUFFT library creates the best threads configuration for the FFT operation. Furthermore, each plan can be reused for several transforms. In this way the best FFT configuration is transparent to the user and it is performed each time.

Once the CUFFT library has been introduced, we are ready to present the parallel implementation of the CSA algorithm. Next sections present main characteristics of CSA steps seen in Figure 2. Some operators are used more than once, so their presentation is detailed.

Azimuth FFT: Raw data is transformed to Range-Doppler domain applying and FFT in the azimuth dimension, i.e. a FFT is performed over each of the columns of the raw data. Due to FFT characteristics data may be shifted before and after FFT operator, a fftshift routine was implemented in parallel for our application to compensate this shifting. Furthermore, taking into account that CUFFT routine for Fourier Tranform operates along the rows of a matrix, a transpose matrix operation is performed before and after the FFT. When the FFT plan is created, the batch parameters allow to specify how many FFT must be calculated. Each column will be transformed independently and stored back to its location.

First phase function this phase correction is applied in order to correct the differencial range cell migration. After creation it is applied to the data as a pointwise product. The proposed solution is to perform this product invoking a CUDA kernel that launches a thread for each matrix element. Then, $thread_{i,j}$ calculates the element corresponding to row i and column j, resulting in $(c_{i,j} \leftarrow a_{i,j} * b_{i,j})$. In this way, the secuencial pointwise product is $O(n^2)$ while the parallel version is $O(1)$ because of all cells are calculated in parallel in a single call.

Range FFT: Range FFT solution is straightforward, each matrix row is transformed invoking the CUFFT library. A 1D transform plan is created, indicating row size and in this case, the number of rows in order to perform all this FFT independently by CUFFT library. After this operation data is in frequency-frequency domain. As well as with Azimuth FFT, a fftshift operation is applied after and before FFT routine.

Second phase function: this phase correction is applied to perform range compression, secondary range compression (SRC), and bulk RCM correction in the same operation. Similar to first phase function, after creation this phase function is multiplied pointwise with the data.

Range IFFT: is performed in the rows of the data matrix, in order to transform data back to range-Doppler domain, invoking CUFFT routine for executing FFT in the inverse direction, setting the appropriate parameter.

Third phase function: an extra phase correction is required as a result of the chirp scaling applied in step 2. Once again, the same procedure is applied as with previous phase functions.

Fourth phase function: a phase multiply is performed to apply azimuth compression with a range varying matched filter.

Azimuth IFFT: an azimuth IFFT is applied in order to transform the compressed data back to the two dimensional time domain. Data is, again, trasposed in order to execute FFT over the rows (columns) of the matrix. Once IFFT is executed, a new traspose operation is needed.

In brief, this algorithm is based in the parallelization of phase funcion generation, simple matrix operations and fourier transformations. This is a first implementation of the algorithm which will serve as a basis to finding the bottlenecks and propose improvements, in order to achieve the highest performance possible.

The next section shows the results of this first implementation.

4 Experimental Results

Sequential and parallel versions of the Chirp Scaling Algorithm will be analized. There are four different implementations for which CSA runtimes are shown: C sequential algorithm, C-CUDA parallel algorithm and two improvements of the parallel algorithm.

Sinthetic raw data was used for these tests. Raw data was obtained from a SAR simulator developed in [1] and corresponds to a pointwise objective. We hope to use real raw data in the near future.

Hardware platform used for testing has the following characteristics:

CPU	Intel Core i5-2500K @ 3.30GHz
Memory	7.8GiB
GPU	NVIDIA GeForce GTX 570. 480 CUDA Cores.
Memory	1280GiB
OS	Ubuntu 12.04LTS / LINUX
CUDA	Version 5.5

Sequential and parallel codes were instrumented in order to obtain executions times. Each test was performed five times and the average of these executions is presented. The number of repetitions is low given that these times were always very similar, sometimes they were equal through different executions.

The radar parameters used for the simulations are: exposure time of 3.4 seconds (the objetive was in the illuminated area for 3.4 seconds), 600Hz Pulse

Repetition Frequency (PRF), sampling rate of 120MHz. Raw data is stored in matrices of 2000 rows (range echoes) x 4000 columns (spatial samples).

Total runtime of Chirp Scaling Algorithm is showed in figure 3, where all steps of the algorithm (figure 2) are included. Three cases are shown: sequential algorithm and 2 versions of parallel algorithm: a straightforward implementation and an optimized version based on detected penalties or bottle necks. The following paragraphs describe implementation details and detail the proposed improvements.

Time reduction of the sequential and the first (non optimized) parallel algorithm is close to 31%. This improvement is definitely insufficient as a much larger speedup is expected for such an a -priori parallelizable algorithm as CSA. This fact lead us to analyze in more detail the initial implementation, decomposing de problem in subproblems for which runtimes were analyzed.

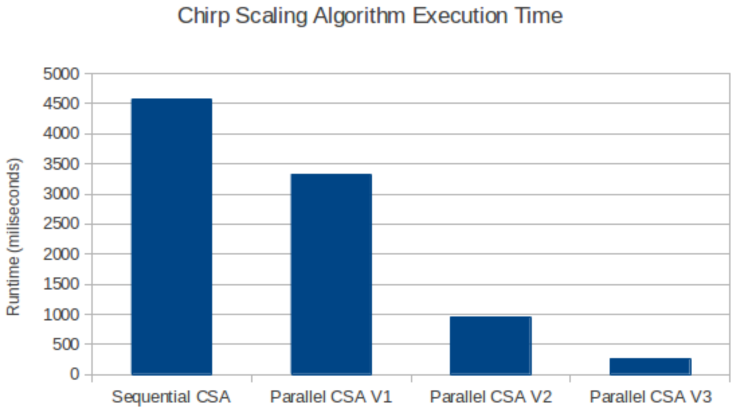


Fig. 3. Execution time for sequential and parallel algorithms.

Figure 4 shows execution times for the central operations of CSA: FFTs and IFFTs applied through all columns or rows of matrices, fftshift operation, tranpose and pointwise product. In order to improve figure understanding, run times (in miliseconds) are included for each experimentation.

This figure shows that each of the parallelized operation achieve an important reduction of runtime. Taking secuencial time as reference, a reduction of 99.26% for matrix IFFT is achieved, reduction of 98.89% for matrix FFT, 96.11% for matrix pointwise product, 96.6% for matrix transpose, and finally the 84.23% reduction time for matrix fftshift.

Although an important speedup has been achived, we note that both matrix transpose and fftshift operation last more time than expected. Both routines are simply value swaps, without no operation performed on the data. It can be seen that in the sequential implementation matrix fftshift is 10 times faster than IFFT, while it is two times slower in the parallel counterpart.

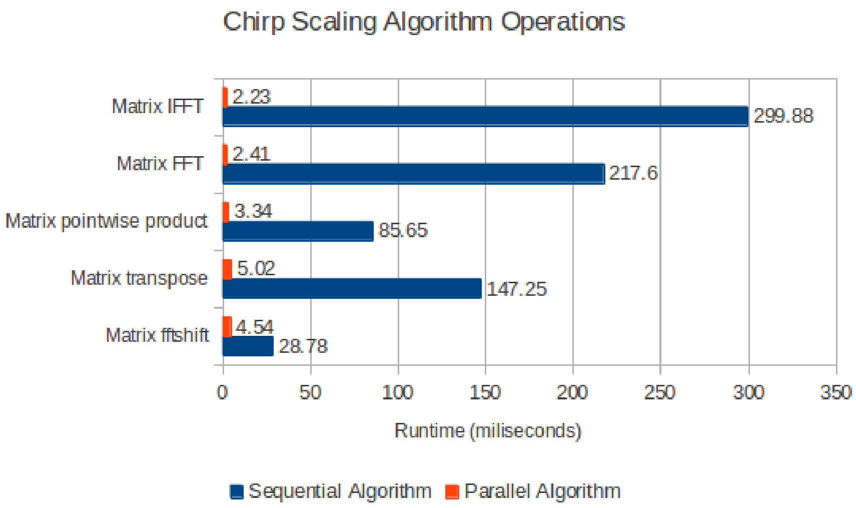


Fig. 4. Execution time for sequential and parallel operations.

In an attemp to improve matrix transpose operation performance, 3 versions of that operation were developed and tested: C sequential algorithm, CUDA parallel algorithm and lastly, a CUBLAS library [15] function for matrix transposition was used. Figure 5 shows runtimes using these options.

CUBLAS is a CUDA library that implements BLAS (Basic Linear Algebra Subprograms) operations, for each of its three levels of operations (vector-vector, vector-matrix and matrix-matrix operations). Our algorithm uses *cublas_X_geam* routine for solving transpose operator. Option *cublasCgeam* was used for complex data matrix.

Function *cublasCgeam* is based on a static method of jcublas class (a JAVA class). Transpose operation requires only data movement and is not considered a computation operation. Figure 5 shows the low performance achieved when CUBLAS is used.

In [4] an evaluation of JAVA operation in GPU is analysed with three representative operations (matrix product, stencil2D and FFT). This work shows that CUDA kernels achieves the best performance (GFLOPS and runtime). JAVA routines (jcuda or arapi) have an overhead of data movements between JAVA and GPU.

Moreover, when CUBLAS library is used, it requires the use of *cublasCreate()* and *cublasDestroy()* operations, in order to allocate or release hardware resources on the host and device. These functions implicitly call *cublasDeviceSynchronize()*. This overhead is an important source of performance penalization [14].

With respect to matrix fftshift, this is a necessary operation due to the current implementation of phase function matrices arrangement: matrix data is

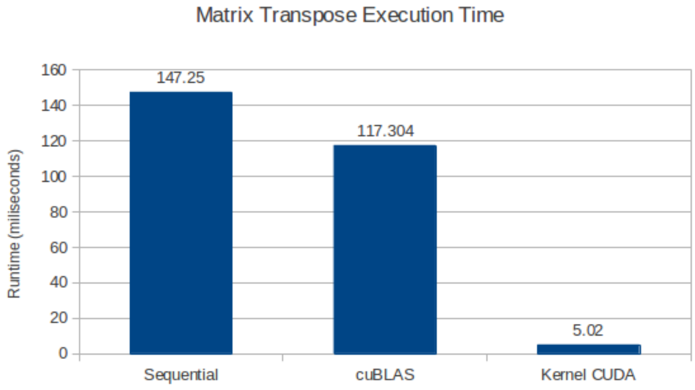


Fig. 5. Matrix transpose routines execution times.

organized in a way that fftshift operation is needed for data consistency through the data processing chain. This operation could be avoided if phase function matrices are modified accordingly, a point to be addressed in the near future.

As mentioned above, our first parallel version of CSA (Parallel CSA V1 in figure 3) shows low performance.

The reason for such performance loss can be multifactorial. The first issue identified was related to the unexpectedly high runtimes of phase functions. Figure 6 shows phase function application times. These times include phase function initialization and pointwise product (this operation takes almost 3 miliseconds as we can see in figure 4). The slowdown cause boiled down to the phase function (a constant matrix once the scenario parameters are fixed) initialization being performed in CPU. This issue was addressed in the next version of the parallel algorithm (V2) where phase functions are initialized on GPU to exploit the parallel nature of this calculation. Each phase function was allocated on a matrix and one thread per cell was launched to initialize its value.

Times of this second version are showed in figure 6 as Parallel V2. An important runtime reduction was achieved, which can be enhanced if we take into account data structures.

Due to the phase corrections being the same for each row, these matrices can be reduced to 1 dimentional arrays. This was implemented in version 3 of the parallel algorithm (Parallel V3 in the figure) which deals with 1D arrays. Then, each kernel works with arrays and each of them can access its columns or rows through arrays.

Figure 6 shows an important reduction of time for this optimized initialization scheme. Now an important runtime reduction is observed for the phase function filter implementation.

It should be noted that both second and fourth phase functions take longer than the first and third phase functions. The analysis of the slower phase function initialization showed that the slowdown had to do with the use of a division

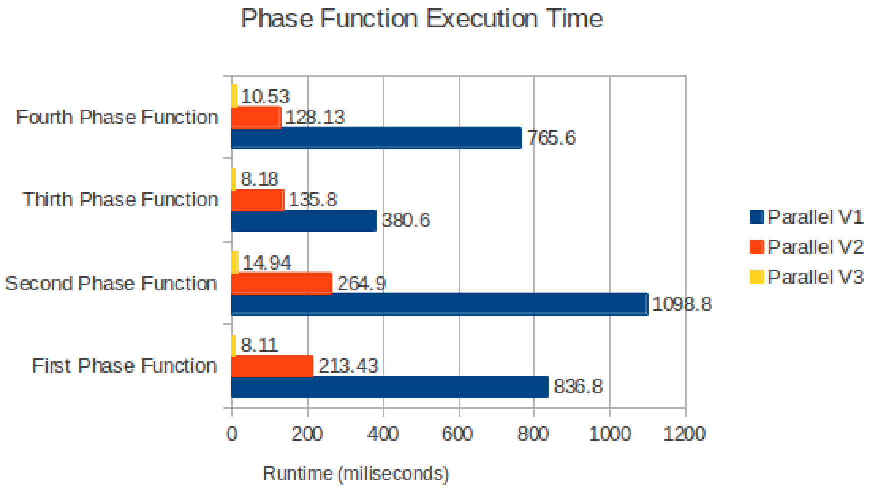


Fig. 6. Runtime of Phase Functions.

operation. Given there is no hardware support for floating point divisions (or integer divisions) on the GPU these operations are implemented as software subroutines that require additional registers for temporary storage [13]. This lack of resources can be avoided if we reduce the number of threads in the kernel launch.

To alleviate this, both second and fourth phase function initializations were executed by grids of threads of different sizes. First executions launched 32x32 threads blocks. This kernel was not able to execute because of lack of registers. Then, blocks of 16x16 threads were used showing good performance. The latter configuration was used for Parallel V3.

5 Conclusions and further work

This work presents the first steps for the parallelization of CSA algorithm for C-CUDA. Our goal is to develop a high performance CSA algorithm thinking on real time requirements.

CSA algorithm is a highly parallelizable problem, and GPUs are very convenient hardware architecture for this algorithm. CSA satisfies most of the requirements for an application to be suitable for executing on GPUs: there is no data dependency, no communication is needed, it operates with a great amount of data, performing complex operations over the data.

As CUDA programation model requires, there exists 2 data transfer: at the beginning, when raw data is read from disk and then tranferred from CPU to GPU. Afterwards, all calculations are performed in GPU, focusing raw data to obtain the final image. This final image is transmitted back from GPU to CPU. Data communication between CPU and GPU (and viceversa) penalise applications througput and efficiency (due to memory latencies, bandwidth of pci express, etc.).

As a main contribution of this work, a parallel version of the CSA was implemented and analized, comparing its performance to a sequential implementation. An important speedup has been achieved after analysis and improvement of the first parallel implementation. There are still points to improve, in particular the extensive use of fftshift and transpose operations that should be avoided, also memory data transfers could be avoided if some data is allocated into GPU global memory for performing phase function multiply. Block size is another important issue to address, for optimizing workload and augmenting throughput.

This work covers the first steps of our challenge of achieving a high performance application for SAR data processing. Next steps include the use of different GPU memories, the implementation of better phase function initializations and the removal of -at least- the fftshift operation as well as the use of field SAR raw data allowing for comparison of different problem sizes and the optimizations required for each of them.

The speedup (sequential runtime/parallel runtime) using our last parallel algorithm is close to 17. This is an important improvement and we hope to further reduce it by implementing the optimizations mentioned above.

References

1. Areta, J., Richter, S.: Simulador de sistema SAR. Congreso; XIV Reunión de Trabajo en Procesamiento de la Información y Control; (2011)
2. Cook, S.: CUDA Programming. A developer's guide to parallel computing with GPUs. Morgan Kaufmann Publishers Inc. (2013)
3. Cumming, I., Wong, F.: Digital Processing of Synthetic Aperture Radar Data. Artech House (2005)
4. Docampo, J., Ramos, S., Taboada, G., Exposito, R.J., Touriño, J., Doallo, R.: Evaluación de Java para computación de propósito General en GPU (2013)
5. Farber, R.: CUDA Application Design and Development. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2011)
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93(2), 216–231 (2005), special issue on "Program Generation, Optimization, and Platform Adaptation"
7. Hein, A.: Processing of SAR Data. Springer (2004)
8. Hwu, W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2011)
9. Kirk, D.B., Hwu, W.m.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2010)
10. Martinez, A., Fraile, F., Martinez, A., Mallorqui, J.J., Nogueira, L., Gabaldá, Broquetas, A., González, A.: PASAR - a SAR Processor Implemented in a Workstation Cluster. In: Guyenne, T.D. (ed.) Data Systems in Aerospace - DASIA 97. ESA Special Publication, vol. 409, p. 103 (Aug 1997)
11. MIT: Fastest Fourier Transform in the West, http://www.fftw.org/, acceded on July 2014

12. Moreira, A., Mittermayer, J., Scheiber, R.: Extended Chirp Scaling Algorithm for Air- and Spaceborne SAR Data Processing in Stripmap and ScanSAR Imaging Modes. Geoscience and Remote Sensing, IEEE Transactions on 34(5), 1123–1136 (1996)
13. NVIDIA: Nvidia - developer zone, https://devtalk.nvidia.com/default/topic/567328/cuda-programming-and-performance/division-problem-in-cuda-kernel/, acceded on July 2014.
14. NVIDIA: CUBLAS Library (2013), http://docs.nvidia.com/cuda/cublas/cublascreate, acceded on July 2014
15. NVIDIA: CUBLAS Library User Guide (2013), http://docs.nvidia.com/cuda/cublas/index.htmlaxzz36RGPzRzq
16. NVIDIA: CUDA C Programming Guide (February 2014), http://docs.nvidia.com/cuda/cuda-c-programming-guide/
17. NVIDIA: CUFFT Library. User Guide. NVIDIA (February 2014)
18. Piccoli, M.F.: Computación de Alto Desempeño en GPU. Universidad Nacional de San Luis (2011)
19. Richards, M.A.: Fundamentals of Radar Signal Processing. McGraw-Hill (2005)
20. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 1st edn. (2010)
21. Skolnik, M.I.: RADAR Systems. McGraw-Hill (2000)
22. Soumekh, M.: Synthetic Aperture Radar Signal Processing with MATLAB Algorithms. Wiley-Interscience (1999)