J Supercomput (2016) 72:451–467
DOI 10.1007/s11227-015-1572-z

Synthetic aperture radar signal processing in parallel using GPGPU

Mónica Denham[1,2] · **Javier Areta**[1,2] · **Fernando G. Tinetti**[3,4]

Published online: 22 December 2015

Abstract In this work an efficient parallel implementation of the Chirp Scaling Algorithm for Synthetic Aperture Radar processing is presented. The architecture selected for the implementation is the general purpose graphic processing unit, 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 speed-up. Details of the issues found are explained, and the performance improvement of their correction explicitly shown.

Keywords Synthetic aperture radar · High performance computing · GP-GPU

✉ Mónica Denham
mdenham@unrn.edu.ar

Javier Areta
jareta@unrn.edu.ar

Fernando G. Tinetti
fernando@lidi.unlp.edu.ar

1 CONICET-Consejo Nacional de Investigaciones Científicas y Técnicas, Buenos Aires, Argentina

2 Laboratorio de Procesamiento de Señales Aplicado y Computación de Alto Rendimiento, Universidad Nacional de Río Negro, Sede Andina, San Carlos de Bariloche, Río Negro, Argentina

3 Facultad de Informática, Instituto de Investigación en Informática LIDI, UNLP, La Plata, Buenos Aires, Argentina

4 Comisión de Investigación Científicas de la Provincia de Buenos Aires (CIC), La Plata, Argentina

1 Introduction

Synthetic aperture radar (SAR) [4,16,19,20] is a mature technology that combines radar and signal processing to enable high-resolution imaging of the earth's 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 ground point captured at different trajectory points contains different and predictable Doppler shifts. This Doppler information is embedded in the phase, which has to be properly processed to obtain a higher resolution image, which turns out to be independent of the range and proportional to the antenna aperture. 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 other points in space [4,9]. SAR signal memory is shown in Fig. 1a while Fig. 1b shows the memory usage structure. 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, usually referred to as fast and slow time dimensions respectively.

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 computing 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 can be implemented properly using a parallel scheme. In this regard, general purpose graphic processing unit (GPGPU) [7,12] is a powerful platform that allows the implementation of complex processing algorithms and is very well suited for this application [7,10].

One of the most important applications of SAR signal processing is the generation of high-resolution images of the earth's surface. These images are very useful for

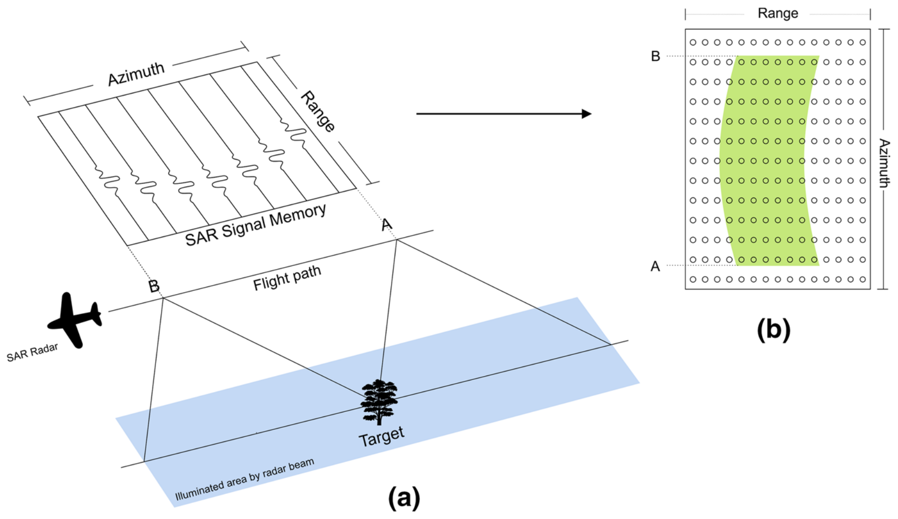

Fig. 1 **a** SAR radar movement and two dimensional signal formation [4]. **b** Two-dimensional matrix (raw data): each received echo is stored in a *matrix row* while *matrix columns* are formed by echoes samples [4]

cartography, remote sensing, hydrology, agronomy, earth change detection, study of ocean currents, etc. All these applications can benefit from real-time imaging.

There are several known algorithms for SAR image focusing, range Doppler algorithm (RDA), chirp scaling algorithm (CSA), ω-k algorithm, back-projection algorithm and others [4,9,20]. Since linearity is assumed in most of these models, the vast majority use both frequency and spatial coordinates in the processing stages. The discrete Fourier transform (DFT) is used for coordinate transformation since it has very efficient parallel implementations [8].

SAR image formation from raw data involves application of parallelizable operations; thus processing can be done efficiently using this programming paradigm. Programmable GPU has evolved into a highly parallel, multi-thread, multi-core processor with tremendous computational power and very high memory bandwidth compared to CPU [22], and is very well suited for this application.

In [22] three main high-resolution SAR processing algorithms are listed, RDA, CSA and ω-k. Focus is made in CSA for real-time systems due to its characteristics, less computation, easy parallelization and simple control flow, the main reason being that CSA requires only two simple phase multiplication operations to correct range cell migration (RCMC) while RDA and ω-k need interpolation—a more computational intensive operation—for this correction. The authors propose a real-time SAR imaging system which processes SAR raw data on the fly for the case that the image does not fit in GPU memory. Received data are thus divided into packets that fit the available memory. Each packet is processed in GPU applying CSA; then the focused partial image is spliced with previously focused packages in CPU involving a complex flow control. In order to achieve good GPU performance they use the CUFFT library for solving FFTs and IFFTs and perform transpose operations of the data matrix on shared memory. No other optimization is proposed to improve performance; focus is made in the sequential splicing process.

The work of Kraja et al. [13] states that SAR image formation applications can profit much more from the architecture and the capabilities of many-core GPUs than from modern multi-core CPUs. The CPUs are bandwidth constrained, so it is more difficult to extract all theoretical FLOPS from them. In turn, GPU FLOPS to bandwidth ratio is much more favourable. This work proposes a space-based architecture where having GPUs on-board would speed up applications which perform a significant number of floating point operations and have regular access to memory. Limitation of GPUs are memory transfers, due to limited bandwidth in the PCI express connection and the limitation in GPU board memory. These issues become relevant for applications with large data sets where data have to be partitioned to be processed one part at a time. To compensate for low PCI express bandwidth the authors propose overlapping transfers with computation. For benchmarking purposes they use a part of the scalable synthetic complex application SSCA#3 benchmark, where SAR image reconstruction is part of the benchmark. They work with large images where data have to be partitioned. For this purpose they use 2D tiles of 32×32 (due to maximum block size of 1024 threads). Each data block is processed by a thread block. To exploit the fine grain parallelism on the GPU having limited memory available they present different application approaches. Using multiple GPUs, they try to distribute the work for the reconstruction of the same image among two devices. In this case GPUs communication is needed, which requires using the CPU, thus slowing down the overall processing. To avoid communication overhead, they implemented a pipelined version, in which separate images are being reconstructed on separate GPU-devices, that showed the best performance. This work focuses on data partitioning in the case of raw data exceeding the GPU memory capacity; code optimization is not their goal.

Bhaumik and Nagendra [3] present GPU implementations of RDA and CSA algorithms. A coarse description of the implementations steps for their parallelization is presented but the fine details are not mentioned. Code optimization is not considered. In particular they do not tackle the fact that branch divergences may cause inactive threads due to SIMD warp thread execution. Global and shared memory features and use of registers are mentioned as optimization techniques, although they are mainly direct consequences of GPU architecture.

Rubin et al. [17] demonstrate GPU acceleration of SAR/ISAR processing. Using GPUs they greatly improve processing times of backprojection-based SAR/ISAR imaging. This work exposes weakness and strengths of main algorithms for SAR raw data focusing: RDA, CSA, backprojection algorithm (BPA), time domain correlation (TDC), etc. CSA is recognized as computationally efficient, but can limit scene size and image resolution. This work is based in the usage of a third party parallelization tool, combining MATLAB and Jacket. Jacket serves as nearly transparent middleware, allowing execution of MATLAB code on CUDA-capable NVIDIA GPUs directly from the MATLAB development environment. The use of this proprietary library does not allow for the fine tuning and optimization of the code. One can conclude that parallelization is a valuable tool for this problem, but the question of whether there is more room for improvement when tailor made code is used is not tackled.

In [5] an implementation of RDA for GPUs was proposed. Starting from a non optimized parallel implementation of the original algorithm optimizations were proposed, implemented and tested, based of detailed profiling and detection of penalties.

This related work encouraged us to pursue exhaustive and systematic parallel code optimization of CSA for SAR image focusing. Our work is based on the application of successive optimization steps, driven by performance/profiling metrics. An initial naive parallel CSA implementation was proposed and profiling tools were used to analyse algorithm efficiency and detecting bottlenecks. Once detected, optimization schemes were proposed, implemented and tested. An exhaustive study of performance of CSA parallel solutions is done resulting in an efficient parallel implementation. CSA implementation is carried out based on Moreira et al.'s [14] work.

The remainder of the paper is organized as follows: Section 2 is a review of CSA and the main operations are presented. Section 3 shows a first implementation of CSA in CUDA C. Sections 4 and 5 show the two proposed optimization phases and their results. Section 6 shows a detailed performance evaluation using *nvprof*, metrics and events. Section 7 shows additional tests which demonstrate application scalability when different GPU are used. Finally, Sect. 8 presents the conclusions.

2 Parallel chirp scaling algorithm

Processing raw SAR data consists of coherently combining the information of all received signals to form (focus) the image.

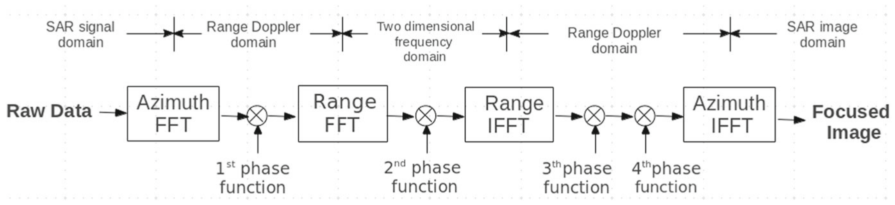

Fig. 2 Block diagram of CSA

CSA is based on modulation properties of chirp (linear FM) signals [4]. Chirp waveforms are frequently used by Radar systems as they can achieve high resolution. Chirp-like signals also appear naturally in the azimuth direction due to the approximately linear Doppler modulation resulting from radar platform movement.

CSA performs signal compression and correction of range cell migration (RCM) using matched filters and focuses the data over range and azimuth dimensions. These matched filters are implemented using so-called phase functions [14]; these are pre-calculated chirp functions based on the scenario parameters. In this algorithm, total RCM is divided into two parts: a "bulk RCM" and a "differential RCM". The bulk RCM is the same for all targets and is modelled as a single tone. Differential RCM is the remaining part, it is range dependent and smaller than bulk RCM; it is modelled as a chirp function. Each part is then corrected by multiplying the phase function to the signal [4, 14]. Figure 2 shows the main steps of CSA.

Appropriate matched filters (phase functions) are used in different domains performing range and azimuth compression as well as RCM corrections. In each operation two-dimensional matrices are initialized with the corresponding phases, see [14] for details. Considering that the data size is n, the phase function matrix initialization requires n^2 operations.

Table 1 depicts the main operations of the CSA presented in Fig. 2. The parallel implementation of the algorithm focuses on (a) phase function generation, (b) basic matrix operations and (c) Fourier transforms. The first implementation of the algorithm will serve as a basis to find bottlenecks and propose performance improvements.

3 Initial implementation

A first, not fully optimized, version of the CSA was implemented based on an own sequential implementation, taking into account several details that were a-priori considered to be relevant to achieve good parallel performance. The sequential implementation was also carefully crafted although its full optimization was not the goal of this work.

Synthetic raw data were used for testing the algorithms developed. Raw data were obtained from a SAR simulator developed in [2]. The test platform used has the characteristics shown in Table 2.

The radar parameters used for the simulations are exposure time of 3.4 s (the objective was in the illuminated area for 3.4 s), 600 Hz Pulse Repetition Frequency

Table 1 Chirp scaling algorithm

Function	Functionality	Pseudocode

Azimuth FFT	Time → frequency azimuth direction	A ← fftshift(A)
		A ← transpose(A)
		A ← fft(A)
		A ← transpose(A)
		A ← fftshift(A)

1st Phase function	Differential range cell migration correction	H1 ← matrix initialization $A_{i,j} \leftarrow A_{i,j} * H1_{i,j}$

Range FFT	Time → frequency range direction	A ← fftshift(A)
		A ← fft(A)
		A ← fftshift(A)

2nd Phase function	Range compression, SRC and bulk RCM correction	H2 ← matrix initialization $A_{i,j} \leftarrow A_{i,j} * H2_{i,j}$

Range IFFT	Frequency → time range direction	A ← fftshift(A)
		A ← ifft(A)
		A ← fftshift(A)

3th Phase function	Extra phase compression	H3 ← matrix initialization $A_{i,j} \leftarrow A_{i,j} * H3_{i,j}$
4th Phase function	Azimuth compression	H4 ← matrix initialization $A_{i,j} \leftarrow A_{i,j} * H4_{i,j}$

Azimuth IFFT	Frequency → time azimuth direction	A ← fftshift(A)
		A ← transpose(A)
		A ← ifft(A)
		A ← transpose(A)
		A ← fftshift(A)

Table 2 Hardware architecture characteristics

CPU	Intel Core i5-2500K @ 3.30 GHz
OS	Ubuntu 12.04LTS/LINUX
GPU	NVIDIA GeForce GTX 570. 480 CUDA Cores
Memory	1280 MB
Processing power	1504.4 GFLOPs (single precision)
CUDA	Version 5.5 (CUDA compiler version V5.5.0)

(PRF), sampling rate of 120 MHz. Raw data are stored in matrices of 2000 rows (range echoes) $\times$ 4000 columns (spatial samples).

Since this work is focused on achieving high-performance CSA for GPUs the imaging quality is simply measured as the accumulated pixel difference between the parallel implementation and the sequential implementation. Algorithms are implemented using single precision data types. The use of double precision may improve imaging quality but this study is not within the scope of this work.

In this initial parallel implementation of CSA several considerations were taken into account to obtain the highest efficiency we could, without feedback from profiling, that is, considering good coding practices and taking into account both the kind of operations involved and the GPU architecture. The following paragraphs describe implementation details of our CUDA C parallel CSA.

CUDA C kernels are executed by threads in a Single Instruction Multiple Thread way (SIMT). The programmer defines for each kernel a grid of parallel threads: threads are arranged into blocks (1D, 2D or 3D array of threads) and blocks form a grid (1D, 2D or 3D array of blocks). The user defines the best thread configuration for each kernel. This configuration is usually conditioned by the data structures used in the application: thread configuration impacts GPUs core utilization and other resource utilization. For kernels that solve matrix pointwise product, transpose, fftshifts, etc, one thread per matrix cell is launched. For these kernels 2D thread blocks are used and grids are configured by 2D array of blocks. Variants of optimized transpose and fftshifts are used (these variants are presented in Sect. 5.2). Several block dimensions were tested. At times it was seen that 1024 threads per block (maximum for current GPUs) are not the best block size because threads consumes device resources and the number of parallel active blocks may be reduced. For each kernel the best block size was chosen (most of the times 512 and 1024 threads per block was used). Furthermore, warp execution was taken into account for improving application performance. Warp execution efficiency analysis is presented in Sect. 6.

Fast Fourier Transform (FFT) and inverse FFT (IFFT) operations are used to change to range-Doppler domain, frequency domain or time domain. These are divide and conquer algorithms for efficiently computing DFTs of complex or real data sets. In this work the FFT and IFFT are solved using CUFFT library [15] and, since it is highly optimized, there is very little room for improvement [1]. The usage of this library implies the need to implement the fftshift operation before multiplication with the phase functions. The fftshift function shifts the zero-frequency element of the resulting vector to the center of the spectrum.

GPUs have a memory hierarchy that includes registers, local memory, shared memory, global, constant and texture memories. Each GPU memory has its own size, bandwidth and latency. Optimal use of several of them depends on patterns in which threads access application data. In order to maximize bandwidth and reduce latencies, these memories were analysed in order to improve application performance.

Global memory was used for allocating the raw SAR signal, the focused final image and partially processed data. That memory resides in device DRAM, for transfers between the host and device as well as for the data input and output from kernels. It can be accessed and modified from both the host and the device. Thread access pattern has to be aligned and coalesced for achieving low latencies and high bandwidth. This will be analysed thoroughly in the following sections.

Shared memory is located on-chip; thus access is much faster than local and global memory. Shared memory latency is roughly $100\times$ lower than uncached global memory latency, if there are no bank conflicts between threads. Each GPU Streaming Multiprocessor has its own shared memory. Due to shared memory dimension it is not enough for allocating application matrices. But this memory is used for an optimization presented later in Sect. 5.2.

Constant memory is small, just 64 KB in the device used. This memory space is cached; as a result a read operation costs one memory read from device memory only on a cache miss; otherwise, it just costs one read from the constant cache (for all threads of a half warp, reading from the constant cache is as fast as reading from a register). This memory is read only for threads in a kernel and achieves best performance when the accessed data are in constant cache and when all threads within a warp access the same data. In this case the accessed data are broadcasted to every thread in the warp, performing just one memory access. CSA raw data and processed data cannot be allocated in constant memory due to the space limitation.

Texture memory is a cached read-only memory space for kernel threads. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read closely located texture addresses will achieve best performance. This access pattern is not present in CSA; thus this memory is not used.

Registers are the fastest memory present in GPUs, but there is a small number of registers per kernel thread. Variables that do not fit in thread registers are allocated in local memory (thread private memory). Local memory is physically on global memory; thus there would be a penalization when trying to use excess registers.

The CSA requires operations with complex numbers. These are not implemented in CUDA, so implementation of functions for performing these operations were made, an issue also found in [13].

Since phase functions are constant once the operation parameters are defined, they are initialized in CPU and transferred to device global memory. After initialization and data transfer to GPU, the matrices are applied to the data as a point-wise product (each phase function is applied at a specific time as is shown in Fig. 2). This is done invoking a CUDA kernel that simply launches a thread for each matrix element, then $thread_{i,j}$ calculates the element corresponding to row i and column j of the resulting matrix

After considering these implementation details the first version (V1) of the code was executed ten times with different data sets. The number of repetitions is low given the low dispersion obtained, at about 1 %. The average runtime of these executions is shown in Fig. 3. This figure also shows results of the sequential implementation and the optimized versions. The black line shows algorithm runtime (data transfer time is not included). Data transfer time (including raw data CPU to GPU transfer and focused image GPU to CPU communication) is presented in red color. In order to gain clarity y axis is logarithmic scaled.

Figure 4 is an example of the dataset used; it shows a typical image used for generating raw data and the result of applying CSA to it.

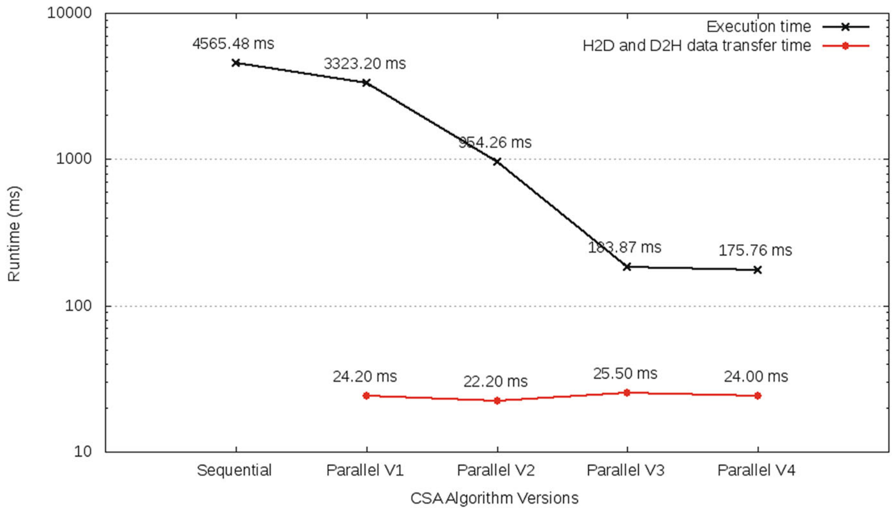

Fig. 3 Execution time for sequential and parallel algorithms. Y axis is logarithmically scaled

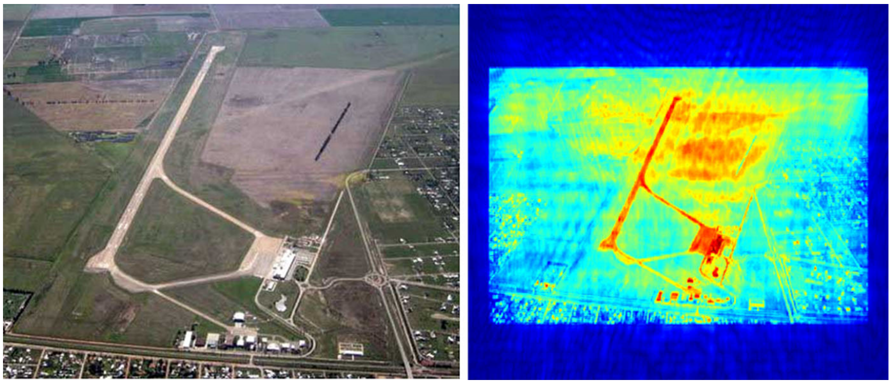

Fig. 4 CSA focused image of an airport

Runtime of V1 implementation is compared against the sequential implementation as this is, temporarily, the only available benchmark. The improvement obtained is definitely insufficient for such an a-priori parallelizable algorithm as CSA. This fact leads to a more detailed analysis of the initial implementation. Table 3 shows execution times for the main operations of CSA: FFT and IFFT of data and phase functions, fftshift operation, matrix transpose and point-wise product.

All the functions perform faster in GPU, although speedup values are quite dissimilar. In particular we note that both matrix transpose and fftshift operations take more time than expected. Both routines are simple value swaps, with no operation performed on the data (memory bound). As an extreme case, note that in the sequential implementation matrix fftshift is ten times faster than IFFT, while it is two times slower in the parallel counterpart.

Table 3 Execution time for sequential and parallel operations (in ms) and first parallel version speedups

Operation	Sequential	Parallel	Speedup
Matrix IFFT	299.88	2.23	134.47
Matrix FFT	217.6	2.41	90.29
Pointwise product	85.65	3.34	25.64
Matrix transpose	147.25	5.02	29.33
Matrix fftshift	28.78	4.54	6.33

Table 4 Matrix transpose routines execution times

Algorithm	Runtime (ms)
Sequential	147.25
cuBLAS	117.30
CUDA kernel	5.02

To improve matrix transpose operation performance, two functions were compared, an own CUDA implementation and a CUBLAS library function for matrix transposition. Table 4 shows runtimes for these options.

CUBLAS is a CUDA library that implements Basic Linear Algebra Subprogram (BLAS) operations; for each of its three levels of operations (vector–vector, vector–matrix and matrix–matrix operations). *cublasCgeam* function was used for complex data matrix transposition. This function 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. The low performance achieved when CUBLAS is used can be traced to the initialization and communication required by the library. In [6] an evaluation of JAVA operation in GPU is analysed with three representative operations (matrix product, stencil2D and FFT), and it shows that CUDA kernels achieve better performance (GFLOPs and runtime) than CUBLAS. This may be due reliance on JAVA routines (jcuda or arapi) which have an overhead of data movements between JAVA and GPU. Also, 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 becomes an important source of performance penalization that rules out this implementation as an option.

The CUDA kernel implementation of this operation is a very simple kernel where threads access a matrix cell and copy values to their cell destination. Different implementations of optimized matrix transpose in CUDA inspired on [18] are analysed in the next section.

It should be mentioned that matrix fftshift is a necessary operation due to the way CUFFT is implemented and the current implementation of phase function matrices arrangement. Matrix data are organized in a way that fftshift operation is needed for data consistency through the data processing chain.

Table 5 Phase function execution times (initializations and point wise product)

Phase function	V1	V2	V3
First	765.6	128.13	10.53
Second	380.6	135.8	8.18
Third	1098.6	264.9	14.94
Fourth	836.8	213.43	8.11

Execution times are in ms

4 First improvement phase

Since the first parallel version of CSA shows poor performance, a thorough analysis of the algorithm was performed since the reason for such performance loss can be multifactorial. The first issue identified after fixing the previously analysed matrix operations was the unexpectedly high runtimes related to phase functions. Table 5 shows phase function application times. These times include phase function initialization and pointwise product (this operation takes almost 3 ms as seen in Table 3). 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.

Runtimes of this second version are also shown in Fig. 3 and Table 5 as Parallel V2. An important reduction was achieved, which can be enhanced if we take into account data structures. Since the phase corrections are the same for each row, these matrices can be reduced to 1-dimensional arrays. This was implemented in version 3 of the parallel algorithm (Parallel V3 in Fig. 3; Table 5) greatly simplifying the kernels involved. Table 5 shows a 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 took 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 operation in these phase functions. 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. This lack of resources can be avoided by reducing the number of threads in the kernel launch.

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

5 Second improvement phase

The previous section showed first-order runtime improvements. In the following, second-order effects are analysed to further improve the performance. Two matrix operations are to be re-analysed, fftshift and matrix transpose.

5.1 CUDA FFTSHIFT

Two alternative implementations are considered.

The first is based in [21], where no data are moved but complex number properties are used to make this operation more efficient. It consists of multiplying the vector to be transformed by a sequence of 1 and -1 s which is equivalent to the multiplication by $e^{(-jn\pi)}$ and thus to a shift in π in the conjugate domain. This kernel must be called before and after the application of the FFTs or IFFTs. It turned out that this seemingly more complex sign exchange algorithm performed better than the explicit fftshift operations, reducing runtime from 4.54 to 1.3 ms.

A different optimization, which embeds the fftshift operation in the phase function initialization was also implemented. This removed the need for explicit fftshift function calls, which otherwise is called eight times during CSA processing for proper frequency domain multiplication and reduces to almost zero the overhead for this operation. Version 4 (V4) runtime using this optimization is shown in Fig. 3.

5.2 Matrix transpose optimized

When FFT or IFFT is applied to matrix columns, a transpose operation must be performed since CUFFT is natively a row operation. In [18] four versions of matrix transpose functions, naive transpose, coalesced transpose, bank conflict free and diagonal transpose are presented.

These versions were implemented for this problem and compared to the initial transpose function, to use the most efficient version. Table 6 shows runtime for different matrix transpose designs.

Our own parallel kernel is the initial approach used. Kernels with half the number of cell threads were launched, where each thread simply swaps two matrix values.

The use of Coalesced transpose using shared memory showed memory bank conflicts due to several read or write operations accessing the same memory bank. To overcome this the approach was making half-warp access 16 different banks simultaneously, achieving maximum memory bandwidth when all threads in a half-warp access different memory banks. This is done padding shared memory, adding a column to the tile matrix. In this way shared memory access of 16 threads are all in different banks.

In addition to shared memory bank conflicts due to access pattern, global memory has its own access conflicts. Global memory is divided into either 6 o 8 partitions (it depends on GPU serie) of 256-byte width. To use the global memory effectively, concurrent access to global memory by all active warps should be divided evenly

Table 6 Matrix transpose routines execution times

Kernel	Runtime
Parallel	4.05
Coalesced	3.54
Bank conflict free	3.6
Diagonal	5.02

Runtimes are in ms

amongst partitions. Diagonal transpose was implemented for testing global memory access pattern, but no improvement was achieved.

For this application none of the implementations showed a clear improvement over the others. This is in agreement with [18] which states that the methods efficiency depend on matrix size, GPGPU characteristics, etc.

6 Additional performance evaluation

A detailed runtime profiling study was done based on *nvprof* CUDA tool. This profiling tool is available since CUDA 5.1. It collects timeline information from application's CPU and GPU activity, including kernel execution, memory transfers and CUDA API calls [11].

Table 7 presents information collected by this profiling tool. It shows the most important kernels developed for CSA algorithm, total execution time (the sum of all calls of each function) and percentages of time that each operation represents. Both of them are relative to the whole application. There are more kernels and CUDA API calls, but this table shows only kernels that we implemented.

It is important to take into account that this table includes just kernel runtimes. No initialization or other operations that each kernel needs to perform its work are taken into account.

Total time is the amount of time that a kernel is executing, taking into account all calls through the application. Percentages of each kernel is again, relative to all application kernels.

Additional performance evaluation was performed. Events and metrics were used to improve profiling data. Events are hardware counters observed during the execution of an application. Metrics are calculated based on events and used to measure the efficiency of different operations, like how well the application is using memory, cores, etc. [11]. The following metrics were used: *gld_efficiency*, *gst_efficiency* and *warp_execution_efficiency*. Table 8 shows these metrics for the main kernels.

*gld_* and *gst_efficiency* are used to obtain the efficiency of global memory use (*g* for global memory, *st* for store transactions and *ld* for load operations).

Optimally, for maximizing memory bandwidth, global memory access should be coalesced and aligned. Any other pattern will result in a replay of memory request.

The metric *gld_efficiency* (*gst_efficiency*) is the ratio of requested global memory load (store) throughput to actual global memory load (store) throughput. When *gld_ gst_ efficiency* is less than 100 % it means that some requests are replayed, indicating that memory bandwidth is not optimally used. For example, an efficiency of 50 %

Table 7 Kernels analysis using *nvprof* executed on GeForce GTX 570 architecture

Kernel	Calls	Time	Time (%)
Matrix transpose	4	5.94	8.92
Fftshift	8	7.43	11.19
Pointwise product	4	5.49	8.26
Phase function init	4	9.12	6.06

Runtimes and time percentages are the sum of all calls of each function

Table 8 Kernels analysis using *nvprof* metrix executed on GeForce GTX 570 architecture

Kernel	Invocations	Metric name	Average (%)
Matrix point wise product	4	gld_efficiency	100.00
		gst_efficiency	100.00
		warp_execution_efficiency	100.00
Init phase functions	1	gld_efficiency	100.00
		gst_efficiency	100.00
		warp_execution_efficiency	100.00
Matrix transpose parallel kernel	4	gld_efficiency	100.00
		gst_efficiency	25.00
		warp_execution_efficiency	100.00
Matrix transpose 3 optimizations	4	gld_efficiency	100.00
		gst_efficiency	100.00
		warp_execution_efficiency	100.00
FFT_shift	8	gld_efficiency	100.00
		gst_efficiency	100.00
		warp_execution_efficiency	100.00
FFT_shift optimized	8	gld_efficiency	100.00
		gst_efficiency	100.00
		warp_execution_efficiency	100.00

indicates that load (or store) memory requests are replayed once. This can happen due to un-aligned memory access or not coalesced memory access. An efficiency of 100 % indicates that neither load nor store memory requests are replayed when processing the kernel; thus each access is handled by a single memory transaction.

The warp execution efficiency (nvprof metric *warp_execution_efficiency*) is the average percentage of active threads in a warp in each executed warp. Increasing warp execution efficiency will increase utilization of GPU's resources. Divergent branches and predicated instructions provoke warp efficiency to less than 100 %.

Table 8 shows load and store global memory access efficiency of CSA main kernels. When global memory efficiency is analysed, it can be seen that several of the kernels achieve 100 % memory efficiency for load or store operations. That means that global memory accesses are aligned and coalesced (they start in a multiple of memory granularity and accesses are contiguous within the warp). No replays are needed for solving an access.

When phase functions are initialized a 100.00 % memory store efficiency is obtained as we use auxiliary variables instead of directly accessing the phase function arrays. In Table 8 just one row is used for these initialization functions due to all of them showing the same behaviour.

Matrix transpose with no optimization techniques (matrix transpose parallel kernel in Table 8) achieves 25 % of store operation efficiency. This low memory access efficiency is due to writing the resulting matrix by columns that implies not coalesced writes (stores). Transpose matrix optimization avoids this not coalesced accesses, using shared memory or diagonal transpose solution (as presented in Sect. 5.2).

Table 9 Additional tests hardware architecture

	Name	C.C.	CUDA cores	Processing power[a]	Memory bandwidth[b]	Runtime (ms)
1	Tesla C2070	2.0	448	1030	144	150.74
2	Tesla C2075	2.0	448	1000	144	151.07
3	GeForce GTX 480	2.0	480	1344	177.4	171.66
4	Tesla K20c	3.5	2496	3520	208	145.54
5	GeForce GTX 780	3.5	2304	3977	288	134.16
6	GeForce GTX Titan	3.5	2880	5121	288	137.87

[a] Single precision GFLOPs peak
[b] Maximum bandwidth (in GB/s)

Finally, warp execution efficiency was analysed to study warp thread executions and GPU cores use. Implemented kernels show an efficiency of 100.00 % that means that threads within a warp can execute in a SIMD way, avoiding inactive threads within a warp. We can have divergences in different warps, but it does not cause thread inactivity.

It can be concluded that the proposed implementation of the CSA is based on SIMD operations and that the implemented kernels achieve high bandwidth memory use and high GPU core utilization.

7 Performance in different platforms

After the algorithm refinements presented in the previous sections, the optimized CSA was tested in different GPU architectures to analyse its scalability.

Table 9 shows the main features of the architectures used for these additional tests. Compute capability (C.C.), CUDA cores and runtime are shown. The runtimes presented do not include I/O communication times between CPU and GPU (comparable to black line in Fig. 3). Theoretical peak processing power for single precision and maximum bandwidth for each architecture is included.

The implemented algorithm shows scalability for these architectures, showing variable speedups as better processors do not necessarily show improvement over simpler ones. It can be seen that runtimes are in accordance with memory bandwidth. This is due to several CSA operators being memory bound. Fine tuning of block size and memory usage is needed to take advantage of each architecture, and in general the algorithm will require adaptation to the architecture in use to achieve the best possible performance.

8 Conclusions and further work

This work presents the implementation and optimization of the parallelization of CSA algorithm for CUDA C. The goal is to develop a high-performance CSA algorithm with real-time requirements in mind.

CSA turns out to be 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 and it operates with a great amount of data, performing complex operations over the data.

As a main contribution of this work, a parallel version of the CSA was implemented and examined, comparing its performance to a direct implementation. This first parallel version allowed bottleneck detection. An important speedup has been achieved after careful performance analysis and successive improvement of the parallel implementations.

Synthetic raw data were used for testing, with close to 10^6 pixels per image. The optimized algorithm execution times show that it can operate at—practically—real time, during the radar data acquisition process. For the proposed operational PRF, the system should be able to operate in real time. That is, every 3 s the radar can acquire and store a data matrix, and while the next matrix is collected, CSA processing is done, taking about a sixth of this time.

References

1. Abdellah M, Saleh S, Eldeib A, Shaarawi A (2012) High performance multi-dimensional (2D3D) FFTShift implementation on graphics processing units (GPUs). In: Cairo international biomedical engineering conference (CIBEC). doi:10.1109/CIBEC.2012.6473306
2. Areta J, Richter S (2011) Simulador de sistema SAR. Congreso; XIV Reunión de Trabajo en Procesamiento de la Información y Control
3. Bhaumik P, Nagendra G (2014) Parallel of synthetic aperture radar SAR imaging algorithms on GPU. Comput Sci J 5(1):143–146
4. Cumming I, Wong F (2005) Digital processing of synthetic aperture radar data. Artech House, Norwood
5. Denham M, Areta J, Vaquila I, Tinetti F (2013) Procesamiento de señales SAR: Algoritmo RDA para GPGPU. In: XIX Congreso Argentino de Ciencias de la Computación, Red de Universidades con Carreras de Informática (RedUNCI), pp 174–183
6. Docampo J, Ramos S, Taboada G, Exposito RJ, Touriño J (2013) Evaluación de Java para computación de propósito General en GPU
7. Farber R (2011) CUDA application design and development, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco
8. Frigo M, Johnson SG (2005) The design and implementation of FFTW3. Proc IEEE 93(2):216–231 (special issue on "Program generation, optimization, and platform adaptation")
9. Hein A (2004) Processing of SAR data. Springer, New York
10. Hwu W (2011) GPU Computing Gems Emerald Edition, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco
11. John Cheng TM, Grossman M (2014) Professional CUDA C programming. Wrox
12. Kirk DB, WmW Hwu (2010) Programming massively parallel processors: a hands-on approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco
13. Kraja F, Murarasu A, Acher G, Bode A (2012) Performance evaluation of SAR image reconstruction on CPUs and GPUs. In: IEEE (ed) Aerospace conference, 2012 IEEE. IEEE, pp 1–16
14. Moreira A, Mittermayer J, Scheiber R (1996) Extended chirp scaling algorithm forair- and spaceborne SAR data processing in stripmap and ScanSAR imaging modes. IEEE Trans Geosci Remote Sens 34(5):1123–1136
15. NVIDIA (2014) CUFFT library. User guide, NVIDIA
16. Richards MA (2005) Fundamentals of radar signal processing. McGraw-Hill, USA
17. Rubin G, Sager E, Berger D (2010) GPU acceleration of SAR/ISAR imaging algorithms. In: The antenna measurement techniques association (AMTA) 2010 preliminary program, Atlanta Georgia, 10–15 Oct 2010

18. Ruetsch G, Micikevicius P (2009) Optimazing matrix transpose in CUDA. http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf. Accessed July 2015
19. Skolnik MI (2000) RADAR systems. McGraw-Hill, USA
20. Soumekh M (1999) Synthetic aperture radar signal processing with MATLAB algorithms. Wiley-Interscience, New York
21. StackOverflow (2013) One dimensional fftshift in CUDA. http://stackoverflow.com/questions/14187481/one-dimensional-fftshift-in-cuda
22. Wu Y, Zhang H, Chen J (2012) A real-time SAR imaging system based on CPU–GPU heterogeneous platform. In: IEEE (ed) 11th international conference on signal processing (ICSP), 2012 IEEE, vol 1, pp 461–464. doi:10.1109/ICoSP.2012.6491524