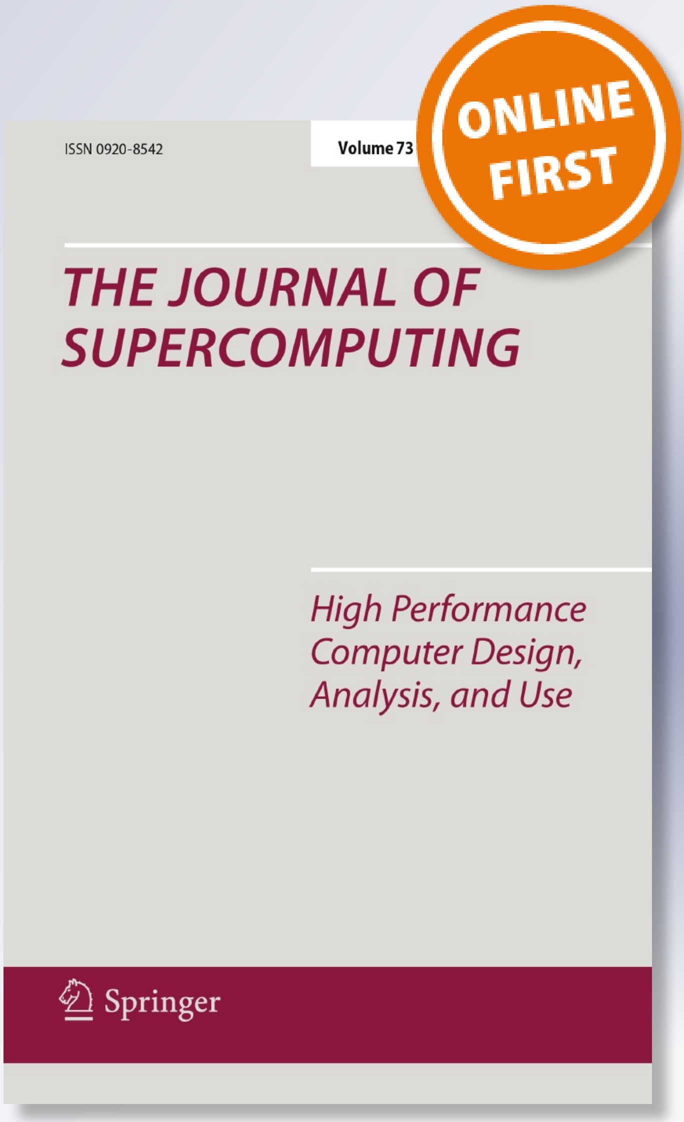
Weather radar data processing on graphic cards

Mónica Denham, Enrico Lamperti & Javier Areta

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-017-2166-8


ISSN 0920-8542
Volume 73
ONLINE FIRST
THE JOURNAL OF SUPERCOMPUTING
High Performance Computer Design, Analysis, and Use
Springer

J Supercomput
DOI 10.1007/s11227-017-2166-8

Weather radar data processing on graphic cards

Mónica Denham[1,2] · Enrico Lamperti[1] ·
Javier Areta[1,2]

Abstract Weather radar operation generates data at a high rate that requires prompt processing. The operations performed on data for weather product generation are repeated in each resolution cell and thus are naturally prone to parallelization. Parallel processing using graphic cards is an emerging technology that allows for implementation of high-throughput algorithms at a low cost. In this paper, the parallel implementation of the main product of a polarimetric weather radar using GPU is presented, focusing on its optimization. A speedup exceeding 20× is obtained when compared to the serial implementation. Also processing is found to be memory bound, which results in a counter-intuitive performance improvement when the number of threads per job is reduced.

Keywords Signal processing · Weather radar · High-performance computing · GPGPU

✉ Mónica Denham
mdenham@unrn.edu.ar

Enrico Lamperti
mail@elamperti.com

Javier Areta
jareta@unrn.edu.ar

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

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

Published online: 30 October 2017

1 Introduction

Weather radars (WR) have become a fundamental tool in weather forecast and severe weather warning applications. Accurate, real-time forecast and description of severe weather are fundamental to reduce and prevent weather disasters. The quality of radar data is essential to assure the performance of meteorological applications such as severe weather detection, precipitation estimation, and weather forecasting [2].

Polarimetric radar (PR) has allowed for better and richer weather variables, which have been thoroughly modeled in the literature [7,17]. The PR raw data processing is crucial to obtain useful information for the meteorological models used to forecast weather [10]. This processing involves the range compression of the signal [16], the ground clutter filtering [7] and the generation of PR variables [17] (box "WR Products" in Fig. 1). The latter depends also on the polarization scheme used by the radar [1], which may alternate H and V polarizations, send them together, or other possible schemes that need to be accounted for in the processing stage.

The high rate signal processing operations applied to raw data, mainly range compression, are implemented in hardware, either in application-specific integrated circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). A posterior decimation operation reduces the data rate for the forthcoming data processing stages. Even considering this data rate reduction, both clutter filtering and the PR variables calculation are not usually performed in real time unless very powerful processors are used. This fact delays the forecast process as data is not processed in real time. With the advent of the Graphical Processing Units (GPU) as computation devices, a great number of applications prone to parallelization found a technology that allowed their efficient implementation in real time [13].

The usage of GPU in WR applications is recent and focuses mostly on the implementation of meteorological models [9] or visualization [18]. The structure of raw WR data suits the GPU architecture well, since each radar cell maps to a number in the radar data cube [14], and processing is usually decoupled. In this paper, a number of PR variables are generated from raw data in order to verify that the standard algorithms can be parallelized and a real-time implementation achieved using general purpose computers and a relatively simple graphics card.

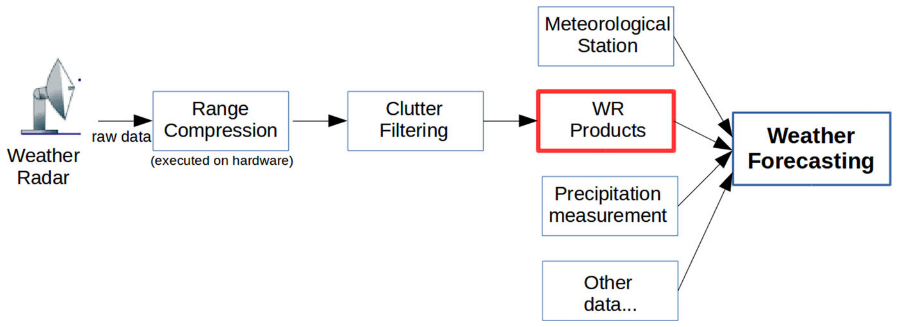


Fig. 1 Conceptual model of our work: the box WR Products indicates polarimetric radar products context

Weather radar operation implies the generation of huge amounts of data. A single radar turn can generate almost 0.5 GB of data in a matter of seconds (a standard turn rate is 6 rpm [15]). For almost-real-time operation, this data has to be processed in tenths of a second, before the data from the next turn arrives. This shows the need of efficient implementations to reduce total runtime.

In this work, a parallel implementation of the raw data processing for product generation is presented. The optimizations performed are clearly detailed as this is the key step for improving the performance of the algorithm. Real weather radar data is used, and the speedup obtained is higher than $20\times$.

In Sect. 2, an overview of GPU parallel implementations for WR data is presented. In Sect. 3, the meteorological products are presented as well as their parallel implementation. Section 4 devotes to the results of implementing these algorithms for processing real WR data. Finally, the conclusions are presented.

2 Related work

Data processing for WR can be divided in two levels: raw data processing and weather variables processing. The former involves processing the echoes from the radar in order to obtain meaningful variables that convey information about the weather state in the surveyed space. This involves large amounts of data at a fast rate that require fairly simple calculations and similar processing steps per resolution cell. The latter involves processing the weather variables, also called products, using physical models of the atmosphere dynamics. Measurements from other sensors are usually also incorporated. This involves a much smaller amount of data but requires complex models in the form of differential equations that are hard to solve.

There is a vast amount of work on WR raw data processing, which has been mostly developed since the 1990s. There is agreement in that data processing applications have to deal with large arrays of numbers (2D or 3D grids of cells) and thus processing requires high computational power and memory, matched with algorithm efficiency, to perform in a reasonable amount of time. Although the problem is well suited for GPU implementation, there is scarce evidence about its usage for this problem. On the other hand, the solution of the weather forecast models using GPU has received more attention.

In [18], the authors propose a fast weather radar data processing based on a parallel implementation using CUDA on a GPU. The workflow proposed consists of reading and formatting the raw data followed by normalization and interpolation. The main goal of the work is to present a parallel interpolation technique in the raw data domain to complete the missing pixels, obtaining better radar images. The interpolation step is the most computational intensive process in the algorithm and thus the optimization focus was placed there. A bilinear interpolation has been implemented, where each pixel is calculated in parallel. There are no data dependencies, and pixels are calculated simultaneously using CUDA C on GPU.

This work restates the well known fact that in order to design a high-performance application for GPU, three issues should be considered: avoid unnecessary data transmission between different memory spaces, maximize the use of the available bandwidth and assign threads appropriately. Following these guidelines, they were able to accelerate the interpolation step and achieve an acceleration of at least 4×. This reinforces that correct design of CUDA applications is fundamental for obtaining maximum efficiency.

In [12], the update of Colorado State University's (CSU) Pawnee Doppler radar system was presented. Part of the project dealt with the selection of a hardware accelerator for the implementation of the Parametric Time Domain Method (PTDM) clutter mitigation algorithm. The algorithm is based on determinant and inverses of large matrices. PTDM works in time domain, and no FFTs are used. The report presents a comparison between FPGA (Field Programmable Gate Array) and GPU as parallel architectures to accelerate PTDM. When FPGA were analyzed, they concluded that the main drawback of this approach is the development time along with the high cost of custom circuit design. The authors conclude that using GPU is the better approach since a significant amount of development time has already been spent optimizing matrix calculation, making the development time of the implementation significantly shorter.

In [11], the Weather Research Forecast (WRF) model was implemented on CUDA C. The non-optimized algorithm developed reached 17× acceleration.

A computationally intensive module from the WRF model was selected and adapted from FORTRAN to run on a NVIDIA GPU. The chosen function is WSM5 (WRF Single Moment 5 tracer), which accounts for only a 0.4 percent of the WRF source code but consumes a quarter of total run time on a single processor. This model represents condensation, fallout of various types of precipitation and related thermodynamic effects of latent heat release.

An initial validation and benchmark results comparing the original WSM5 code with the GPU version is presented, using the Storm of the Century Test-case consisting of a 3D grid with 115.000 cells.

The results show a parallel speedup greater than 17× relative to the host CPU (including data transfer) even though the application was not optimized. By using thread and memory optimization steps, a better performance and almost-real-time operation could be achieved [5,6].

In [2], a spectrum-based processing framework called STEP (Spectrum-Time Estimation and Processing) is presented. It integrates three novel algorithms to perform clutter identification, clutter filtering and noise reduction. The algorithm has been extensively tested and shows good performance, improving the quality of polarimetric radar data.

Garrido et al. [8] present a parallel solution for a forecast model which works with several atmospheric phenomenons named PROMES. The amount of parameters, the need for spatial resolution, accuracy and reasonable run times make it necessary to use parallel platforms. They use a distributed memory parallel platform: a cluster of 16 PCs (one processor per PC). The parallelization consists of dividing the domain in subdomains and distributing them across all processors. Once the domain has been divided, the processors just exchange the frontier information. Due to communication penalties, the best results in terms of speedup and efficiency are obtained for just 2 processors. This work shows that distributed memory is not the best parallel platform for this problem, concluding that domain division is better when processors share memory spaces, such as graphic processing units do.

Garrido et al. [9] recognize that vectorial multiprocessors are perhaps the best parallel platform to solve weather models, as a consequence of how these can handle the data domain. GPUs are a return to vectorial machines and thus seem to be the natural choice for solving these kinds of problems.

3 Polarimetric radar variables

In this work, different PR variables, also called products, are presented and their implementation discussed. These products must be of the highest possible quality as they are fed to weather forecasting models, where information fusion takes part with data coming from diverse sensors. The products that are object of the present study are: reflectivity, doppler frequency/autocorrelation, differential phase shift and correlation coefficient. These are obtained from direct manipulation of the raw radar data using estimators that convey information of the atmosphere in each resolution cell by extracting it from the scattering and transmission matrices [17].

3.1 Polarimetric radar product calculation

There is vast literature on polarimetric variable calculations [1,17]. Raw radar data consists of samples of the antenna induced voltage, $v_i[n]$, which are proportional to the hydrometeor scattered electromagnetic wave previously emitted by the radar. The received signal contains information about the hydrometeor state, which is to be extracted by the processor. In a polarimetric radar, the first step is to generate an estimate of the scattering matrix S which condenses the relevant information. Here, the focus is the parallel implementation rather than the radar echo modeling, a brief description of each follows.

3.1.1 Reflectivity

The reflectivity in each of the polarization channels is proportional to the hydrometeors cross section, integrated over a resolution volume cell. Calibration of the radar is important to properly estimate this product, although not knowing the calibration constant for a radar does not affect its calculation as it only implies a multiplication by a constant. The theoretical expression for the reflectivity is

$$Z_i = 10 \log \left(\frac{4\lambda^4}{\pi^4 |K|^2} \langle |S_{ii}|^2 \rangle \right) [dBz] \tag{1}$$

where λ is the signal wavelength, K is the calibration constant, and S_{ii} is the diagonal element of the scattering matrix corresponding to polarization $i \in \{h, v\}$ and dBZ is a relative reflectivity logarithmic dimensionless unit of a radar signal reflected off a remote object (in mm^6 per m^3) to the return of a droplet of rain with a diameter of 1 mm (1 mm^6/1 m^3).

In practice, the reflectivity estimator for polarization i is implemented averaging N signal samples

$$\hat{P}_i = \frac{1}{N} \sum_{n=1}^{N} |v_i[n]|^2 \tag{2}$$

3.1.2 Doppler frequency

The Doppler frequency measured by a weather radar represents the projection of the wind's speed in the direction of the radar beam. The Doppler frequency estimation is used to reconstruct the wind field in the radar coverage region. Two radars are needed to fully estimate the wind field, given that a single radar measures only a projection of the wind velocity vector. The usual scheme for measuring wind speed is modeling it as a stochastic process and obtaining an estimator for its correlation $R(t)$, which contains information about the average spectra of the process [1]. At lag T_S, the autocorrelation is

$$R(T_s) = e^{j2\pi f_d T_s} \int_{-T_s/2}^{T_s/2} S(f) e^{j2\pi(f-f_d)T_s} \mathrm{d}f \tag{3}$$

where f_d is the Doppler shift of the power spectral density of the process $S(f)$. To estimate the autocorrelation, it is considered that the process is ergodic, thus

$$\hat{R}(T_s) = \frac{1}{M} \sum_{m=1}^{M} v^*[m]v[m+1] \tag{4}$$

As the phase of this correlation is dependent on the desired mean Doppler frequency, obtaining it only requires analyzing its argument

$$\hat{v} = -(\lambda/4\pi T_s) \arg(\hat{R}(T_s)). \tag{5}$$

3.1.3 Differential phase shift

The propagation of the vertical and horizontal polarization signals depends on the media characteristics. The difference of the phase in the received signals can thus be used to estimate these characteristics. The differential phase is defined as $\phi_{dp} = \phi_{hh} - \phi_{vv}$ the difference of the phase in the horizontal and vertical polarization signals.

The rate of change of this phase is known as K_{dp} and contains information about the anisotropy of the propagation medium.

As with the Doppler frequency, the differential phase can be estimated using the cross-correlation between the vertical and horizontal polarization signals

$$\hat{R}_{hv}[0] = \frac{1}{N} \sum_{n=1}^{N} v_v[n] v_h^*[n] \tag{6}$$

the angle estimator is given by

$$\hat{\phi}_{dt} = arg(\hat{R}_{hv}[0]) \tag{7}$$

3.1.4 Correlation coefficient

Another product related to the difference between the polarization signals is the correlation coefficient. It considers the magnitude rather than the phase as previously presented. By definition

$$|\rho_{hv}(0)| = \frac{\langle S_{vv} S_{hh} \rangle}{\sqrt{\langle |S_{hh}|^2 \rangle \langle |S_{vv}|^2 \rangle}} \tag{8}$$

In practice this is estimated from the cross-correlation and the reflectivity

$$|\hat{\rho}_{hv}(0)| = \frac{|R_{hv}[0]|}{\sqrt{\hat{P}_v \hat{P}_h}} \tag{9}$$

3.2 RMA data

In this work, we used real data from Radar Meteorológico Argentino (RMA-0) located in San Carlos de Bariloche, Argentina ($41°08'23.0'' S$ $71°08'59.3'' W$). The data, $v_i[n]$, generated by the radar is stored in a $N \times M \times 2$ matrix. Columns correspond to the echoes of the emitted pulses in different angular positions. Rows are the samples, in fast time domain, of those echoes. The third dimension denotes polarization (horizontal and vertical). In this paper, data size corresponding to a full turn of the radar which is $2242 \times 12960 \times 2$.

As the radar turns continuously, a number of contiguous pulses are averaged to provide a measurement in each beam pointing direction. In this implementation, 36 consecutive pulses are launched within the beam direction (radar elevation and azimuth direction) which results in a one degree beam. Each set of 36 consecutive echoes is deemed a group and different operations are performed over these groups. Figure 2 shows the input data template.

Next, processing of this data cube is performed in order to obtain the desired products.

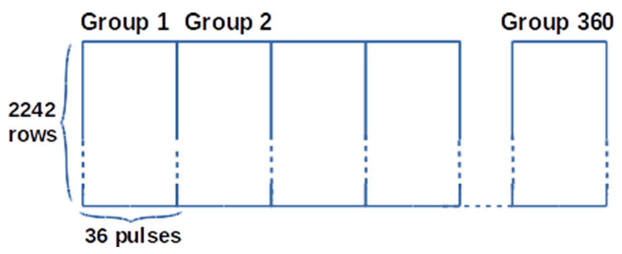


Fig. 2 RMA input: groups of 36 pulses are used altogether for each calculus

3.3 RMA product calculation

For each group, the reflectivity is calculated as:

$$\hat{Ref} = diag(A * A^{\mathsf{H}}) \tag{10}$$

where A is the 2242 × 36 matrix formed by the pulses of a group. A^{H} is the conjugate transpose matrix. This calculus is made for both H and V polarization, and the vector $\hat{Ref}$ is stored as a row of an output matrix.

Note that this operation works only with the diagonal of the matrix multiplication result. In order to accelerate this calculation, only the diagonal values are calculated rather than a full matrix multiplication.

$$\hat{Ref}_i = \sum_{j=0}^{36-1} (A_{i,j} * A^{\mathsf{H}}_{i,j}) \tag{11}$$

Here, i is the raw input data group and j corresponds to each pulse samples within the group.

In a similar way, frequency is calculated as

$$\hat{Freq}_i = -angle\left(\sum_{j=0}^{36-1} (A_{i,j} * A^{\mathsf{H}}_{i,j+1})\right) / (2\pi T), \tag{12}$$

where *angle* function is the two-quadrant arctangent function applied on each value.

The autocorrelation is obtained as

$$\hat{R} = (1/p) * \sum_{j=0}^{36-1} (A_{i,j} * A^{\mathsf{H}}_{i,j}), \tag{13}$$

where p is the number of pulses in the group.

In a similar fashion, the cross-correlation is obtained as

$$\hat{R}_{vh} = (1/p) * \sum_{j=0}^{36-1} (A_{i,j} * B^{\mathsf{H}}_{i,j}), \tag{14}$$

where A denotes data in the H polarization matrix and B in the V polarization matrix.

Then, the differential phase shift function is implemented as

$$\hat{Phi} = angle(\hat{R}_{vh}), \tag{15}$$

where $\hat{R}_{vh}$ is the cross-correlation of the polarized signals, calculated in Eq. 14.

In turn, the correlation coefficient is implemented as

$$\hat{\rho_{hv}} = abs\left(\frac{\hat{R}_{vh}}{\sqrt{Pov. * Poh}}\right), \tag{16}$$

where $Pov.*Poh$ is the element-wise product of the H and V $Power$ vector elements.

The results of these calculations are stored in matrices of 360 × 2242 cells (360 groups × 2242 columns).

3.4 Sequential and parallel algorithms

The code development workflow is as follows. A first MATLAB implementation is done for each product; it serves as a benchmark for the implementation of the sequential C functions. The rapid prototyping capabilities of this high-level language allows for much simpler debugging of C function implementations.

The sequential C implementation serves as a baseline to compare against and allows the identification of computational requirements, parallelization opportunities, bottlenecks, data dependencies, communication requirements, memory access patterns for load/store operations. This knowledge is used to properly plan a high-performance CUDA C implementation.

Algorithm 1 implements Eq. 11. For each group, each row is used in order to calculate one element of the resulting matrix. The sequential C application consists of a loop that processes the 360 groups of 36 pulses (Algorithm 1 line 2). During the group processing, the resulting products are vectors of 2242 elements, which are stored as a row in the corresponding output matrices (Algorithm 1 line 11, where `out` is a matrix of 360 × 2242 and `row_out` is a vector of size 2242).

The complexity of this product is $O(NM)$, where N is the number of columns and M the number of rows.

The analysis of Eqs. 11–16 shows that the remaining products can be implemented in a similar way. The same ideas are applied to sequential as well as parallel solutions.

Algorithm 1 Sequential Reflectivity

```
1:  procedure REFLECTIVITY($in$, $out$)
2:    for <g in 1 .. number of groups> do                          ▷ 360 groups
3:      for <r in 1 .. number of rows> do                ▷ For each row of the group
4:        total ← 0,0i
5:        offset ← row r (of matrix $in$) + first column of group $g$
6:        for <k in 1 .. number of pulses per group> do
7:          total ← total + $in$[offset + k] * $\overline{in}$[offset + k]
8:        end for
9:        row_out[r] ← total                        ▷ Element $r$ of the resultant vector
10:     end for
11:     $out$[g] ← row_out                             ▷ Copy of row $g$ to the output
12:   end for
13: end procedure
```

In a parallel implementation, the processing of each input group is independent of the computation of the other groups. Different products can be processed simultaneously.

Grids of 360 × 2242 threads are launched to obtain different products. This threads layout matches the output data matrices. All products share this thread arrangement. Algorithm 2 shows the operations in each thread.

The comparison of Algorithms 1 and 2 reflects a very important CUDA capacity: `for` loops in lines 1 and 2 of sequential algorithm (Algorithm 1) that iterates over groups and rows are replaced by the matrix of threads (parallel algorithm). Then, the code of these loops is executed "simultaneously" by kernel threads.

This fact reduces the computational complexity to $O(s)$ where s is the number of pulses of each group: 36 pulses in this application. Note that this considers an infinite number of possible concurrent threads. A more realistic analysis is to consider $O(NM/p)$ where p is the number of available cores.

Algorithm 2 Parallel Reflectivity

```
1: procedure KERNEL PARALLEL REFLECTIVITY($in$, $out$)            ▷ thread i,j
2:     output_row ← thread's row index (j)         ▷ index of $out$ matrix access (output write)
3:     output_col ← thread's column index (i)       ▷ index of $ot$ matrix access (output write)

4:     group ← output_row (j)                 ▷ The group is the row of the thread (output_row)
5:     input_row ← output_col ∗ N                           ▷ $in$ matrix access (input read)
6:     input_col ← pulses per group ∗ group                 ▷ $in$ matrix access (input read)

7:     total ← 0,0i
8:     for <k in 1 .. pulses per group> do
9:         total ← total + $in$[input_row + input_col + k] * $\overline{in}$[input_row + input_col + k]
10:    end for
11:    $out$[output_row + output_col] ← total
12: end procedure
```

Input data consists of complex matrices. For this data types addition, multiplication and division operators need to be implemented as they are not implemented in CUDA.

To check the correctness of the parallel implementation, the difference of the same products in different implementations (parallel or serial) is analyzed. Differences found are in the order of numerical accuracy, confirming that there are no inconsistencies in the different implementations. Figure 3 shows MATLAB and C outputs for the frequency product.

4 Results and discussion

In this section, the runtimes of the sequential and initial parallel implementations are compared. Parallel execution time includes data transfer between CPU main memory and GPU global memory.

Tests shown in this section (Table 1) were executed in a CPU Intel Core i5-2500K @ 3.30 GHz, Ubuntu 12.04LTS/LINUX. The graphic card is a GPU NVIDIA GeForce GTX 570, with 480 CUDA Cores, 1280MB of memory, and a processing power of 1504.4 GFLOPS in single precision. CUDA version 5.5, and CUDA compiler V5.5.0.

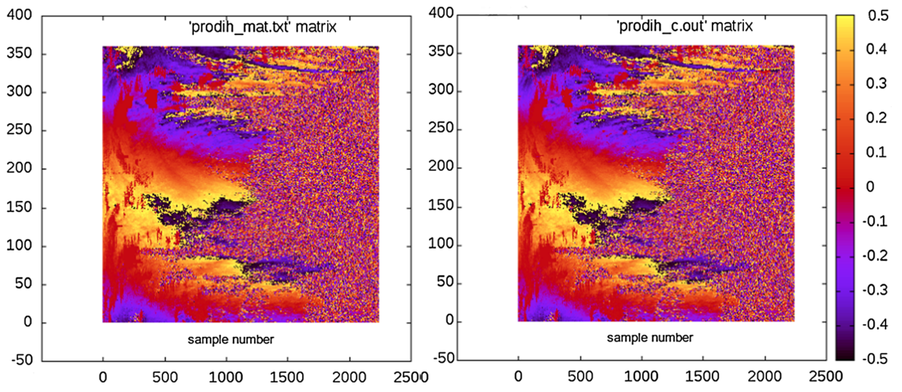


Fig. 3 MATLAB and sequential C results for the normalized Doppler frequency ($T = 1$) product

Table 1 RMA run times (in ms) using a Geforce GTX 570

	Sequential	Parallel	Acceleration
Reflectivity	1111.63	50.45	22×
Frequency	1169.92	70.49	16×
Autocorrelation	547.2	51.35	10×
Phase shift	34.21	0.16	213×
Coef. of correlation	32.45	0.27	120×
Data communication	–	81.6	–

Table 1 shows run times for sequential and non-optimized parallel functions for each of the weather radar products. These results are the average of 5 RMA iterations. In all executions, runtimes are very similar and thus no standard deviation is presented.

Total sequential runtime is 3268 ms, while parallel CUDA C application takes 264 ms. Parallel runtime includes CPU-GPU data transfer. This fact shows that although data movement consumes a big portion of total parallel runtime (near 29%), the simplest parallel implementation of the algorithm obtains a useful runtime reduction. In practical implementations where more meteorological radar products are to be calculated, the data movement should become less significant for the overall application runtime.

Both phase shift product and correlation coefficient products are fast, and hence they do not dominate the total runtime. Acceleration with CUDA is significant for these (213× and 120× approximately). Both kernels are simple, and they are based on arithmetic operators over input data (stored on GPU global memory).

For the other three products, CUDA acceleration is less than expected. These are based on a *for loop* over all samples of a group of pulses (Algorithm 2). Each product is based on performing the same operation with different portions of raw data in a SIMD (Single Instruction Multiple Data) fashion. These products dominate the total runtime of the application.

Since kernel operations are based on arithmetic operations between matrices, the expected acceleration for this kernels was higher, theoretically in the order of the

Table 2 Nvprof profiling results Geforce GTX 570

Time (%)	Time	Calls	Name
29.62	74.61 ms	2	CUDA memcpy H2D
26.94	70.21 ms	2	kernel_frequency()
19.67	51.28 ms	1	kernel_autocorrelation
19.29	50.28 ms	2	kernel_reflectivity()
5.35	13.94 ms	7	CUDA memcpy D2H
0.08	205.65 µs	1	kernel_correlation_coefficient
0.05	117.3 µs	1	kernel_phase_shift

number of cores. Based on this, an extended study of the performance of these kernels follows.

Section 4.1 shows the performance analysis of the code, with emphasis on the memory access. In Sect. 4.2, the kernel performance study using NVIDIA profiling tools is presented. Section 4.3 shows the use of pinned memory for improving application performance. Finally, a scalability analysis comparing different GPU architectures is presented in Sect. 4.4.

4.1 Parallel application and kernels performance

Performance analysis is crucial in GPU applications, and direct implementations of sequential algorithms have bottlenecks that require optimization. Two fundamental performance measures are considered [3]: application performance (overall GPU utilization and efficiency, and memory copy efficiency) and kernel performance (looking for instruction and memory latency reduction, efficient use of memory bandwidth, efficient use of computer resources, etc).

The NVIDIA profiling tools Nsight Eclipse Edition, NVIDIA visual profiler (nvvp) and nvprof command line profiler [3] are used for this purpose.

Table 2 shows the application timing using nvprof profiler. For each kernel and GPU activity, total execution time and percentage of time for each function are listed. These results are consistent with the results shown in Table 1.

To obtain more insight on the application performance, the memory efficiency is analyzed using different metrics. In CUDA profiling, an event is a countable activity that corresponds to a hardware counter collected during kernel execution. A metric is a characteristic of a kernel calculated from one or more events [3].

As mentioned earlier, the data matrix used for the calculations is large. Neither texture memory nor constant memory are big enough to allocate it. Because of this, global memory was used in this application. The drawback is that global memory has the highest latency and lowest bandwidth in the GPU memory hierarchy. Shared memory is limited too but one test—presented later—was performed using this memory.

Global memory efficiency was analyzed using the metrics: `gld_efficiency` and `gst_efficiency`, which are the ratio of requested global memory access and true required global memory accesses (which includes replay of transactions when

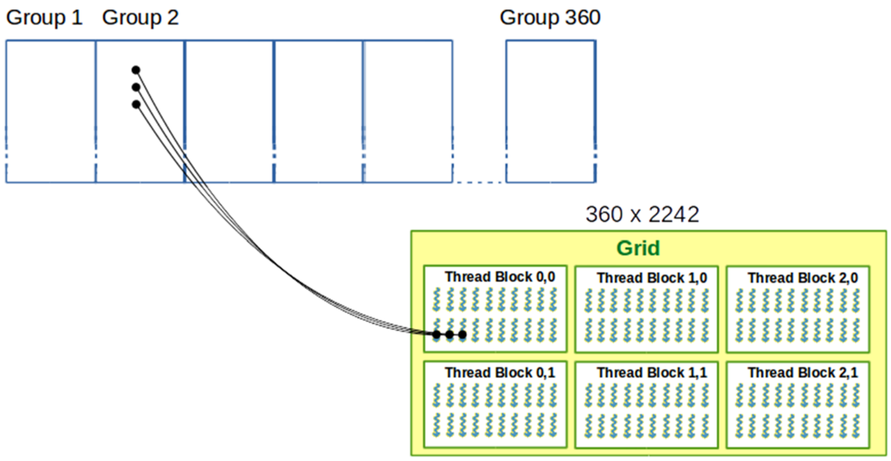


Fig. 4 Global memory access pattern for reading input data

not all bandwidth is used with useful data due to misaligned or not coalesced memory access). It could be seen that a very low percentage of bandwidth use was achieved, approximately 6.25%. That is, that most of the memory transactions had to be replayed and each access is handled by more than one memory transaction. The reason is that all the kernels have the same access pattern. Contiguous threads access contiguous data rows. That means that contiguous threads in a warp access data with an offset equal to the amount of input columns (Fig. 4).

Global memory access is efficient if aligned and coalesced. Memory accesses are aligned if the first address of a memory access is a multiple of block size (that depends on whether the access is cached or not cached in each architecture). Memory access is coalesced if a warp of threads accesses a contiguous chunk of data [3,4].

As a consequence, this initial implementation is memory bounded, and thus global memory access penalizes the application runtime in a very significant manner.

In order to improve memory access efficiency, some modifications were proposed and tested.

The first modification was changing the thread template for launching kernels. After several tries and evaluations, no improvement was found. It is not possible to improve access pattern due to input data format and "per group" operation mode. Load and store accesses have important and unavoidable offsets (groups, rows) that affect this behavior.

In order to check if a more efficient memory usage could be attained, a dummy kernel was implemented that reads contiguous data in global memory (Fig. 5). This test kernel exploits the access pattern of a single chunk of memory but has no use in the current application, and it is meant just for verification purposes. In this case, the global memory efficiency improved, from 6% to near 53% for load transactions. This result shows that the access pattern is fundamental for memory efficiency and that the access pattern of this application is not favorable for global memory usage.

Based on this result, a further modification of this dummy kernel was done. It consists of using float data instead of complex data. Using the proposed `gld` and `gst`

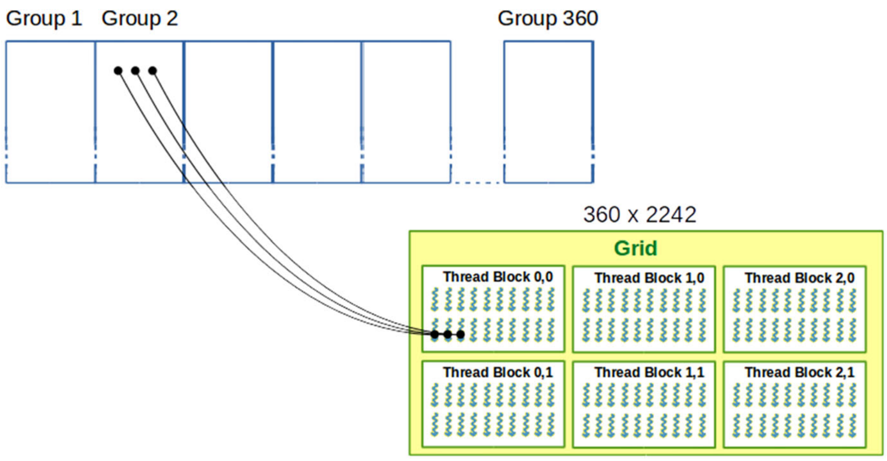


Fig. 5 Dummy kernel: coalesced access to global memory

metrics, this kernel achieved an efficiency of 73%. This improved memory efficiency can be properly explained using the analysis presented in [3], where the difference of memory pattern accesses using an array of structures vs a structure of arrays (for storing complex data, that means real and imaginary fields) is shown. Storing arrays of structures, the same field of data is not contiguous in memory. In CUDA C programming, the use of a structure of arrays is typically preferred because data elements are pre-arranged for efficient coalesced access to global memory. (Data elements of the same field are stored adjacent in memory.)

Then, for an efficient usage of the available memory bandwidth, the accesses must be aligned and coalesced. In this application, the global memory access patterns are not coalesced nor aligned due to the layout of the raw data format.

Another test performed consisted in using shared memory to perform the internal thread loops. In this case, data is loaded from global memory to shared memory. Given that this data is not reused by kernel threads, the overall result is not a noticeable performance improvement. The fastest access time (reading shared memory) is somehow canceled by the memory copy operation from global memory. For this reason, shared memory is not used in any implementation.

4.2 Kernels block size

The initial implementation for the parallel version of the code defined a block size of 16×16 for every application kernel. Even though the theory states that there should be as many threads running as possible in a block, in this application the increase in block size (and therefore the amount of threads) resulted in a runtime impairment, as Table 3 shows. It can be seen that using smaller blocks yielded faster times, up to a point where the block size was too small to achieve a better runtime.

To understand this counter-intuitive behavior, the memory access efficiency by kernel was thoroughly analyzed, taking into consideration the uncoalesced access pattern of this application. Using `l2_l1_read_hit_rate` metric, it was found

Table 3 Runtime for different block sizes (ms)

Product	32 × 32	16 × 16	8 × 8	4 × 4	2 × 2	Sequential
Reflectivity	49.7	50.6	40.1	20.5	31.8	1111.6
Frequency	51.6	70.3	46.7	29.2	33.3	1169.9
Autocorr.	51.6	51.3	50.3	26.6	18.8	547.2
Corr. coeff.	0.23	0.27	0.3	0.66	2.13	32.45
Phase shift	0.64	0.16	0.15	0.51	1.73	34.21
Total*	247.54	252.83	217.9	158.03	168.33	3268.89

*Using pinned memory (see Sect. 4.3)

that there is an increase of L1 to L2 cache misses as the block size increases, generating more requests to global memory, which are costly. Given this situation, a test kernel was created to maximize memory access efficiency (favoring coalescence). The contrast between the test kernel's efficiency and the one of the weather product's indicates there is additional data being brought to L2 memory in the latter case which is not required afterward by the L1 cache. Reducing the block size implicitly allows more granular memory reads, improving the overall latency by increasing hits in L2 cache and therefore reducing the amount of reads from global memory.

When thread block size was reduced, better kernel performance was attained (mainly due to L1 and L2 hit ratio). Considering the best kernel configuration, when 4×4 thread blocks are used, the reflectivity kernel accelerates $35\times$ (vs $22\times$ for 16×16 blocks), frequency $35\times$ (vs $16\times$), autocorrelation to $29\times$ (vs $10\times$). Using this thread configuration, the total application acceleration, accounting for communication time, is about $20\times$.

4.3 Pinned memory

Host allocated memory is by default pageable, that is, the operating system can move memory pages to different physical locations (virtual memory). The GPU cannot safely access data in pageable memory because it has no control over when the host operating system moves it. When transferring data from pageable host memory to device memory, the CUDA driver first allocates temporary pinned host memory (page-locked), copies the source host data to pinned memory, and then transfers the data from pinned memory to device memory.

CUDA allow to directly allocate pinned host memory, avoiding the default transfer from pageable to pinned memory. Since pinned memory can be accessed directly by the device, it can be read and written with much higher bandwidth than pageable memory [3].

When pageable memory was used, CPU-GPU communication lasts near 88.55 ms, representing near the 35% of total time of GPU activity. When pinned memory is used, data communication consumes 81.61 ms which accounts for about 31% of the total GPU activity. It is worth noting that using pinned memory has its drawback. Allocating excessive amounts of it might degrade the host system performance since it reduces the amount of pageable memory used for implementing virtual memory [3].

Table 4 Additional tests hardware architecture

Name	CUDA cores	Processing power[a]	Cache L2 (KB)	Memory (GB/s)[b]	Runtime 4×4 (ms)	Runtime 16×16 (ms)
GTX 570	480	1405	640	152	158.03	252.83
Tesla K40c	2880	4291	1536	288	147.87	193.79
GTX 780	2304	3977	1536	288	136.46	144.57
GTX 780Ti	2880	5046	1535	336	128.16	137.47
GTX 970	1664	3494	1792	224	93.87	160.02

[a] Single precision GFLOPS peak
[b] Memory bandwidth

In the final implementation, pinned memory was used. Host data (radar raw data and products results) was allocated in pinned memory. Results using this memory are presented in Sect. 4. In particular, total times are shown in Table 3.

4.4 Scalability study

Applications with high computational requirements are usually designed and programmed taking into account hardware features to achieve the most efficient use of the architecture resources. Constant advances in technology force the software to remain useful and efficient when computer capabilities increase, which enters in contradiction with the previous point.

CUDA model and GPU hardware are meant to be a highly scalable platform. That is, CUDA applications should remain efficient when run in different cards. One of the reasons for this is that GPU schedulers deliver workload through Streaming Multiprocessors, and within a multiprocessor, the best occupancy of cores and different processing units is obtained.

To test the scalability of the developed implementation, the program was tested and executed in different GPU platforms. Table 4 shows the runtimes of the application when tested in different graphic cards.

Four different GPUs were used in order to evaluate application scalability. For each graphic card name, processing power (just single precision because our application uses single precision numbers) is shown. Furthermore, memory bandwidth is listed (in GB per second, fourth column). The last two columns show the application total runtime (including CPU-GPU data transfers). Previous results, using the GeForce GTX 570 architecture, are included in order to make comparison easier.

Runtimes show that the application is scalable. When better graphic cards are used, the total runtime decreases. Since the application is memory bounded, both L2 cache and memory bandwidth explain the performance improvement.

A notable outlier is the GTX 970 card. The very low runtime is a consequence of a much lower data transfer time, about 30 ms lower than the rest of the cards. Even when this difference is taken into account, the runtime is still the best, and can be explained as it is the card with biggest L2 cache.

5 Conclusion and further work

The implementation of polarimetric radar products on a GPU was studied, and an efficient implementation was found. This allows for a speedup of almost 20× in a cheap GPU card when compared to an efficient serial implementation in C. This is a promising result that could improve the processing time and lower the cost of weather forecasting systems.

The procedure consisted of three steps: first implementing the algorithms in MATLAB due to its rapid prototyping characteristics; then finding an efficient sequential C implementation where functions are analyzed and tested in order to discover functions computation requirements, data dependencies, potential application bottlenecks, data communication, functionality dependencies, memory requirements and memory pattern access for store and load operations; and finally moving to the parallel implementation in two phases, the first being a direct implementation in CUDA C of the sequential algorithm and then, after profiling and analysis, proceeded to implement an optimized version.

The direct parallel version was found to accelerate more than 10×, including CPU-GPU communication time. The optimized version accelerates slightly more than 20× in a quite basic card.

A comprehensive profiling of the application was needed to find that it is memory bound and that by using very few threads per block (4×4 threads per block) the runtime improves notably. This is counter-intuitive as the general rule states that the number of threads per block should be high in order to improve performance through maximum usage of the processors. In this application, when 4×4 blocks were used, the achieved occupancy (ratio of active warps to the maximum number of warps supported on a multiprocessor) was 16.6 and a 50% warp execution efficiency, while when 16×16 thread blocks were used the achieved occupancy was about 96.2% and warp execution efficiency 99.4%. Even though these numbers suggested that the larger blocks should result in better performance, the L2 cache missed reads dominated. Reducing the block size allowed more granular memory reads that in turn decreased memory latency by increasing hits in L2 cache and reducing the amount of reads from global memory.

It was also found that although pinned memory is more expensive to allocate and deallocate than pageable memory, it provides higher transfer throughput for large data transfers.

A scalability analysis was performed using different NVIDIA graphic cards. When better architectures were used, lower application runtimes were achieved. Using different profiling tools, it was found that this is a memory bound application. This result was confirmed in the scalability study, where cards with larger L2 cache an higher memory bandwidth present the best overall runtime.

Data transfer turned out to be a quite important factor in the overall runtime. Further work with the usage of streams for overlapping data transfer time with kernel execution is planned. In this case, a pre-formatting of the raw data has to be done, in which case a data layout that favors coalesced memory readings will be sought.

References

1. Bringi VN, Hendry A (1990) Technology of polarization diversity radars for meteorology. American Meteorological Society, Boston, pp 153–190
2. Cao Q, Zhang G, Palmer RD, Knight M, May R, Stafford RJ (2012) Spectrum-time estimation and processing (step) for improving weather radar data quality. IEEE Trans Geosci Remote Sens 50(11):4670–4683
3. Cheng J, Grossman M, McKercher T (2014) Professional CUDA C programming. Wrox, Birmingham
4. Cook S (2013) CUDA programming. A developer's guide to parallel computing with GPUs. Morgan Kaufmann Publishers Inc, Burlington
5. Denham M, Areta J, Tinetti FG (2015) Synthetic Aperture Radar signal processing in parallel using GPGPU. J Supercomput 72(2):451–467. https://doi.org/10.1007/s11227-015-1572-z
6. Denham M, Areta J, Vaquila I, Tinetti F (2014) Synthetic aperture radar signal processing using GPGPU. CARLA 2014 First HPCLATAM–CLCAR joint conference latin american high performance computing conference. Oral communication
7. Doviak R, Zrnić D (1984) Doppler radar and weather observations. Academic Press, San Diego
8. Garrido JE, Arias E, Cazorla D, Cuartero F, Fernandez I, Gallardo C (2009) PROMESPAR: a parallel implementation of the regional atmospheric model PROMES, vol. 1
9. Garrido JE, Arias E, Cazorla D, Cuartero F, Fernandez I, Gallardo C (2010) PROMESPAR: a high performance computing implementation of the regional atmospheric model PROMES. Springer, Dordrecht, pp 527–538. https://doi.org/10.1007/978-90-481-8776-8_45
10. Meischner P (2003) Weather Radar. Springer, New York
11. Michalakes J, Vachharajani M (2008) GPU acceleration of numerical weather prediction. IEEE international symposium on parallel and distributed processing, 2008. IPDPS 2008. pp 1–7
12. Neuberg M, Picard C (2007) Radar signal processing: Hardware accelerator and hardware update. Master's thesis, Department of Electrical and Computer Engineering
13. Owens JD, Luebke D, Govindaraju N, Harris M, Krger J, Lefohn A, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Comput Graph Forum 26(1):80–113
14. Richards MA (2005) Fundamentals of radar signal processing. McGraw-Hill, New York
15. Rodríguez A, Lacunza C, Serra J, Saulo C, Ciapessoni H, Caranti G, Bertoni JC, Martina A (2017) SiNaRaMe: El primer sistema integrado de radares hidro-meteorológicos de latinoamérica. Revista de la Facultad de Ciencias Exactas, Físicas y Naturales, Universidad Nacional de Córdoba, Argentina 4(1):41–48
16. Skolnik MI (2000) RADAR systems. McGraw-Hill, New York
17. Straka JM, Zrnić DS, Ryzhkov AV (2000) Bulk hydrometeor classification and quantification using polarimetric radar data: synthesis of relations. J Appl Meteorol 39(8):1341–1372
18. Yang L, Jang BJ, Lim S, Kwon KC, Lee SH, Kwon KR (2015) Weather radar image generation method using interpolation based on CUDA. J Korea Multimed Soc 18(4):473–482