An Efficient GPU-Based Implementation of the R-MSF-Algorithm for Remote Sensing Imagery

David Castro-Palazuelos[1,2,*], Daniel Robles-Valdez[1], and Deni Torres-Roman[1]

[1] Center for Advanced Research and Education of the National Polytechnic Institute of Mexico
CINVESTAV Unidad Guadalajara, Mexico
Avenida del Bosque #1145, Colonia el Bajío, 45019, Zapopan, Jalisco, México
[2] Culiacan Technological Institute, Department of Electrical-Electronic Engineering, Sinaloa, Mexico
dcastro@gdl.cinvestav.mx

Abstract. This paper presents an efficient real time implementation of the regularized matched spatial filter algorithm (R-MSF-Algorithm) for remote sensing (RS) imagery that employs the robust descriptive experiment design (DED) approach, using a graphics processing unit (GPU) as parallel architecture. The achieved performance is significantly greater than initial requirement of two image per second. The performance results are reported in terms of metrics as: number of operations, memory requirements, execution time, and speedup, which show the achieved improvements by the parallel version in comparison with the sequential version of the algorithm.

Keywords: Remote sensing, GPU, real time implementation.

1 Introduction

Currently, sensor array signal processing (SP) for imaging radars have been focus of great interest in many research works that now are available in [1, 2]. Such algorithms are computationally expensive; consequently, the majority of the sequential implementations are not suitable to achieve real time or near real time. As many of the required operations needed by these algorithms are: correlation, convolution, filtering, and matrix operations, parallelization technique and theory can be applied in order to improve their performances [3]. The implementation of real time systems in the field of high performance computing (HPC) is limited by the data dependencies of algorithms to be implemented and the features of graphic processing unit (GPU) used [4]. Hence, the advent of the GPU with user-friendly programming environments has allowed the employment of a GPU as parallel mathematical co-processor. In this document, the proposed DED framework based on the matched spatial filter (MSF) SP technique [2] are implemented (referred here as R-MSF-Algorithm). The R-MSF-Algorithm is composed of three parts: the average, the array correlation function, and the matched spatial filter (MSF). In addition, R-MSF-Algorithm is aimed to form a

[*] Corresponding author.

low-resolution image to detect multi-targets on a given test scenario. The algorithm is not fully parallelizable, however, there are some tasks (portions of code) that can be benefit greatly by being mapped/ported to a GPU, which leads toward very attractive speedups (real-time implementation). Three versions of the R-MSF-Algorithm were implemented: the first one (Seq-CPU) was implemented over the programming environment of Visual C++ (C++11) in a sequential way, the second one (One-Thread) was implemented on a GPU using CUDA and a single *thread*, and the third one (Multi-Thread) was implemented on a GPU using CUDA and multiple-*threads*. This implies challenges to map the algorithm on a GPU, which must be considered to achieve the best possible performance. Using the strategy shown in this paper were obtained speedup improvements of 15.5X to 22.5X (for image sizes of [128×128] to [512×512]) for the tested scenarios.

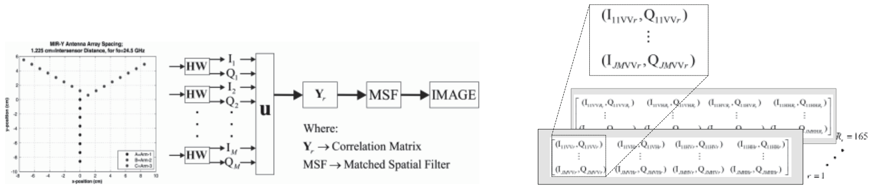
The rest of paper is organized as follows: in Section 2, a background of NVIDIA GPU devices and CUDA programming environment are presented. An analysis of the robust descriptive experiment design related matched spatial filter method is introduced, next, in Section 3. In Section 4, a parallelization of SP procedures is presented. In Section 5, the GPU implementation is described. In Section 6, a performance analysis is provided. Finally, the paper is concluded in Section 7.

2 NVIDIA GPU and CUDA

The GPUs devices are a large collection of multi-*threaded* cores, used to perform concurrently some computations of the algorithms. The compute unified device architecture (CUDA) of the NVIDIA company together with a graphic processor unit (GPU) card mounted on a personal computer (PC) are used to perform a wide variety of signal processing algorithms (SP), as: matrix addition, matrix subtraction, matrix product, among others. Then, CUDA is a general purpose parallel computing architecture, which, provide an application programming interface (API) based on Python/Java/Fortran/C for the instruction set architecture (ISA) and an engine of a GPU [5]. Hence, the CUDA C API environment allows to write programs in C as a high-level programming language, this program is able to be executed on a CPU (host) with the additional ability to run single input multiple *thread* (SIMT) code on a GPU (device). The term SIMT refers to the model in which many *threads* are running in parallel the same instructions at the same time, but with data specified by that *thread* [5]. Therefore, the basic unit of parallelism in CUDA is the *thread*. CUDA *threads* have their own program counter and registers. Furthermore, threads share the stream (flow) instructions and execute instructions in parallel. Therefore, CUDA devices can execute many *threads* simultaneously. Summarizing, a grid is a collection of blocks (of 1, 2 or 3 dimensions), and a block is a collection of *threads* (of 1, 2 or 3 dimensions). In this way, with CUDA is possible to launch multiple blocks to perform a mathematical operation on a GPU. Furthermore, the *compute capability* term is used by NVIDIA to describe the computational potential of a GPU [5]. Furthermore, a *kernel* is a function (as a C function) that the designer launch from the host and is executed on a GPU to perform some computation. When a *kernel* is launched, a grid of blocks is created and the blocks are placed into a queue to be executed by the GPU streaming multiprocessors (SMs).

3 Regularized Matched Spatial Filter Algorithm (R-MSF-Algorithm)

In Fig. 1(a) is shown the multi-sensor imaging (MIR) radar system, which has the R-MSF-Algorithm for RS imaging formation. The sensor array is composed of *M* sensors, configured in a Y-shaped GeoSTAR [1] multi-sensor imaging radar geometry (GeoSTAR MIR-Y), with 3 arms referenced beneath as A, B and C, with 8 equally spaced sensors in each arm. Where, each sensor element receives signals at V (vertical) and H (horizontal) polarizations for *J* pulse repetition times and R_r range gates.


MIR-Y Antenna Array Spacing: 1.225 cm=Intersensor Distance, for fo=24.5 GHz
HW
I_1
Q_1
I_2
Q_2
I_M
Q_M
u
Y_r
MSF
IMAGE
Where:
$Y_r \to$ Correlation Matrix
MSF $\to$ Matched Spatial Filter
(I_{11VVr}, Q_{11VVr})
(I_{JMVVr}, Q_{JMVVr})
$R_r = 165$
$r = 1$

(a) (b)

Fig. 1. MIR radar system. (a) General diagram, and (b) Measurement data format

Each sensor delivers two measurements In-phase (I_m) and Quadrature (Q_m) (for each pulse) as summarize in Fig. 1(b). Here, the first subindex $j = 1, \ldots, J$ indicates the number of the snapshot corresponding to the *j*th transmitted pulse (where, $J \geq M+1$ to form the full-rank sensor data covariance matrix $\mathbf{Y}_r$); the second subindex $m = 1, \ldots, M = 24$ corresponds to the *m*th sensor; the third and fourth subindexes correspond to four cross polarization modes VV, VH, HV, and HH, respectively; the last subindex $r = 1, \ldots, R_r$ corresponds to the range gate, see details in [2]. Once the measurement data has been recorded according to the previously specified format, an average of the four possible polarization modes measurements (VV, VH, HV, HH) are performed to only has an average complex number (I-Q) by each sensor in each PRT (*J*). The next step is to perform the following correlation function:

$$\mathbf{Y}_r = \frac{1}{J}\sum_{j=1}^{J} \mathbf{u}_r^{(j)} \mathbf{u}_r^{+(j)} \leftarrow \text{Averaged over } J \text{ Pulse Repetition Time for each range gate.} \tag{1}$$

The superscript (+) means the Hermitian transpose, which is the transpose conjugate (*T). And, the corresponding real ($\text{Re}\{\mathbf{Y}_r\}$) and imaginary ($\text{Im}\{\mathbf{Y}_r\}$) parts of the complex correlation coefficients become:

$$\text{Re}\{\mathbf{Y}_r\} = \frac{1}{J}\left[\sum_{j=1}^{J}\left(\mathbf{u}_{(\mathbf{I})r}^{(j)} \mathbf{u}_{(\mathbf{I})r}^{\mathbf{T}(j)}\right) + \sum_{j=1}^{J}\left(\mathbf{u}_{(\mathbf{Q})r}^{(j)} \mathbf{u}_{(\mathbf{Q})r}^{\mathbf{T}(j)}\right)\right] \tag{2}$$

$$\text{Im}\{\mathbf{Y}_r\} = \frac{1}{J}\left[\sum_{j=1}^{J}\left(\mathbf{u}_{(\mathbf{Q})r}^{(j)} \mathbf{u}_{(\mathbf{I})r}^{\mathbf{T}(j)}\right) - \sum_{j=1}^{J}\left(\mathbf{u}_{(\mathbf{I})r}^{(j)} \mathbf{u}_{(\mathbf{Q})r}^{\mathbf{T}(j)}\right)\right]. \tag{3}$$

The structure of the DED signal data correlation matrix $\mathbf{Y}_r$ is shown in Fig. 2. The matrix is composed of nine data blocks of 8-by-8 matrices, corresponding to correlations of sensors between arms (see Fig. 2).

$$\mathbf{Y}_r = \underbrace{\overbrace{\begin{bmatrix} \mathrm{corr(AA)} & \mathrm{corr(AB)} & \mathrm{corr(AC)} \\ \mathrm{corr(BA)} & \mathrm{corr(BB)} & \mathrm{corr(BC)} \\ \mathrm{corr(CA)} & \mathrm{corr(CB)} & \mathrm{corr(CC)} \end{bmatrix}}^{\mathit{Matrix}[M\times M]}}_{\mathit{Complex\ Numbers}}$$

Fig. 2. Structure of the DED data correlation matrix $\mathbf{Y}_r$

The robust MSF method for RS image formation implies the computation of each pixel of the image, employing the expression (4) that yields the MSF image of the scene (for details see [1], [2])

$$\hat{b}_{MSF}(q_1,q_2) = \left|\overset{+}{\mathbf{arg}}_{(q_1,q_2)} \mathbf{Y}_r \mathbf{arg}_{(q_1,q_2)}\right|;\ \text{where: } 0 \le q_1 \le N-1;\ 0 \le q_2 \le N-1 \tag{4}$$

$$\begin{aligned}\mathbf{arg}_{(q_1,q_2)} = \Big(&\exp\left[\mathrm{i}2\pi(u_{(1)}\theta_{x(q_1)} + v_{(1)}\theta_{y(q_2)})\right], \exp\left[\mathrm{i}2\pi(u_{(2)}\theta_{x(q_1)} + v_{(2)}\theta_{y(q_2)})\right], \ldots \\ &, \exp\left[\mathrm{i}2\pi(u_{(M)}\theta_{x(q_1)} + v_{(M)}\theta_{y(q_2)})\right]\Big)\end{aligned} \tag{5}$$

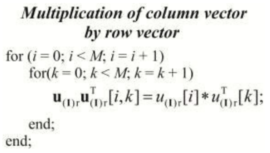
where, θ_x and θ_y are the directional cosines at a particular r from the range observation domain $r \in R$, and $N\times N$ represents the dimension of the final obtained image. The R-MSF-Algorithm is decomposed into three new small functions: the average method (Av-Function), the cross-correlation matrix $\mathbf{Y}_r$ algorithm (1)-(3) (Y-Function), and the last for the matched spatial filter technique (4) (MSF-Function). An analysis of the number of complex and arithmetic operations required by each of the implemented algorithms is shown in Table 1 (where, Mult = Multiplication, and Add = Additions).

Table 1. Number of operations

Algorithm	NumComplexMult	NumComplexAdd	NumArithmeticMult	NumArithmeticAdd	NumTotalArithmeticOperations
Av-Function	$JM/2$	$3JM$	$2JM$	$6JM$	$8JM$
Y-Function	$M^2(J+1/2)$	$M^2(2J-1)$	$2M^2(2J+1)$	$2M^2(2J-1)$	$8JM^2$
MSF-Function	$(M^2+M)N^2$	$(M^2-1)N^2$	$4(M^2+M)N^2$	$2N^2(2M^2+M-1)$	$2N^2(4M^2+3M-1)$

4 Parallelization of Some SP Procedures

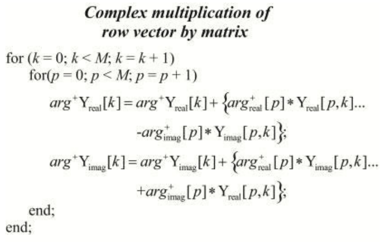
This section describes how to implement the required basic matrix operations for the R-MSF-Algorithm. From (2) and (3), two data $\mathbf{u}$ vectors ($\mathbf{u}_{(\mathbf{I})r} = \{u_{(\mathbf{I})r(1)}, u_{(\mathbf{I})r(2)}, \ldots, u_{(\mathbf{I})r(M)}\}$ and $\mathbf{u}_{(\mathbf{Q})r} = \{u_{(\mathbf{Q})r(1)}, u_{(\mathbf{Q})r(2)}, \ldots, u_{(\mathbf{Q})r(M)}\}$) are required to perform $4J$ multiplications of column vector by row vector, $4J+2$ matrix additions, and 2 scalar matrix multiplications. In addition, (5) are performed via off-line and previously stored in memory, and this is composed by N^2 **arg** vectors, where $\mathbf{arg} = \{arg_1, arg_2, \ldots, arg_M\}$. Furthermore, in (4) N^2 **arg** are provided to perform the N^2 matrix operations of row vector by a matrix multiplication followed by a row vector by column vector multiplication. Hence, here is described the data dependencies of the following operations: multiplication of column vector by row vector, row vector by a matrix multiplication, and row vector by column vector multiplication. Thus, the concept of data dependence can be explained as follows: given two sentences *S* and *T*, where *S* is executed before *T*, also, there is a memory location that is referenced (read or write) by both iterations in a nested loop **L** with (L_1, L_2). Then $S(i_1, i_2)$ denotes the instance of *S* for a iteration point (i_1, i_2), and $T(j_1, j_2)$ the instance of *T*. Therefore, the distance from $S(i_1, i_2)$ to $T(j_1, j_2)$ is defined to be the vector $(j_1 - i_1,\ j_2 - i_2)$ where (i_1, i_2) and (j_1, j_2) are the corresponding iteration points, respectively. Therefore, if an iteration of *T* depends on an iteration of *S*, then the difference ($\mathbf{d} = \mathbf{j} - \mathbf{i}$) between the two index values is called dependence distance vectors for the nested loop (**L**) [6].



Multiplication of column vector by row vector

```
for ($i = 0$; $i < M$; $i = i + 1$)
    for($k = 0$; $k < M$; $k = k + 1$)
        $\mathbf{u}_{(1)r}\mathbf{u}^{T}_{(1)r}[i,k] = u_{(1)r}[i] * u^{T}_{(1)r}[k]$;
    end;
end;
```

(a)



Complex multiplication of row vector by matrix

```
for ($k = 0$; $k < M$; $k = k + 1$)
    for($p = 0$; $p < M$; $p = p + 1$)
        $arg^{+}\mathrm{Y}_{real}[k] = arg^{+}\mathrm{Y}_{real}[k] + \{arg^{+}_{real}[p] * \mathrm{Y}_{real}[p,k]$...
                $-arg^{+}_{imag}[p] * \mathrm{Y}_{imag}[p,k]\}$;
        $arg^{+}\mathrm{Y}_{imag}[k] = arg^{+}\mathrm{Y}_{imag}[k] + \{arg^{+}_{real}[p] * \mathrm{Y}_{imag}[p,k]$...
                $+arg^{+}_{imag}[p] * \mathrm{Y}_{real}[p,k]\}$;
    end;
end;
```

(b)

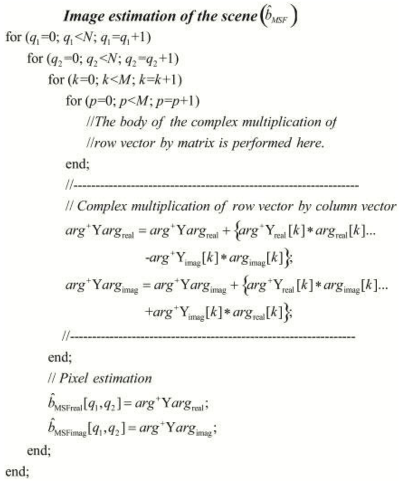


Image estimation of the scene $(\hat{b}_{MSF})$

```
for ($q_1$=0; $q_1$<$N$; $q_1$=$q_1$+1)
    for ($q_2$=0; $q_2$<$N$; $q_2$=$q_2$+1)
        for ($k$=0; $k$<$M$; $k$=$k$+1)
            for ($p$=0; $p$<$M$; $p$=$p$+1)
                //The body of the complex multiplication of
                //row vector by matrix is performed here.
            end;
            //-------------------------------------------------------------
            // Complex multiplication of row vector by column vector
            $arg^{+}\mathrm{Y}arg_{real} = arg^{+}\mathrm{Y}arg_{real} + \{arg^{+}\mathrm{Y}_{real}[k] * arg_{real}[k]$...
                    $-arg^{+}\mathrm{Y}_{imag}[k] * arg_{imag}[k]\}$;
            $arg^{+}\mathrm{Y}arg_{imag} = arg^{+}\mathrm{Y}arg_{imag} + \{arg^{+}\mathrm{Y}_{real}[k] * arg_{imag}[k]$...
                    $+arg^{+}\mathrm{Y}_{imag}[k] * arg_{real}[k]\}$;
            //-------------------------------------------------------------
        end;
        // Pixel estimation
        $\hat{b}_{\mathrm{MSFreal}}[q_1,q_2] = arg^{+}\mathrm{Y}arg_{real}$;
        $\hat{b}_{\mathrm{MSFimag}}[q_1,q_2] = arg^{+}\mathrm{Y}arg_{imag}$;
    end;
end;
```

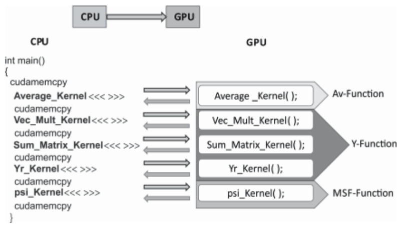
(c)

Fig. 3. Pseudo-codes of some SP procedures. (a) Multiplication of column vector by row vector, (b) Complex multiplication of row vector by matrix, and (c) Image estimation of the scene.

In Fig. 3(a) a multiplication of column vector by row vector pseudo-code is listed. The theory of dependence analysis proves that, there is no data dependency between any iterations of the program, allowing to run simultaneously the both loops (*i* and *k*). Next, Fig. 3(b) is listed the pseudo-code for complex multiplication of row vector by matrix, and the dependence distance vector is $\mathbf{d} = (k\text{-}k) = (0)$, which, allows to execute in a parallel way the loop of the index *k*. Furthermore, Fig3 (b) shows that the *k* index iterations will produce a read and then a write on the same memory locations $arg^{+}\mathrm{Y}_{\mathrm{real}}$ and $arg^{+}\mathrm{Y}_{\mathrm{imag}}$ defined by (*k*), which shows that there is an associated dependency data to the *k* index loop. Then, in Fig 3(c), a pseudo-code to perform the complex multiplication of row vector by column vector is listed. In addition, Fig3 (c) shows that the *k* index iterations will produce a read and then a write on the same memory locations $arg^{+}\mathrm{Y}arg_{\mathrm{real}}$ and $arg^{+}\mathrm{Y}arg_{\mathrm{imag}}$ defined by (*k*), which shows that there is an associated dependency data to the k index loop. Also, in the pseudo-code portion of the pixel estimation in Fig. 3(c), shows that the sentence will not write or read on the same memory location $\hat{b}_{\text{MSFreal}}$ and $\hat{b}_{\text{MSFimag}}$ defined by (q_1, q_2), hence, there are not associated any data dependency to the index loops of q_1 and q_2, which allows to run at the same time the both loops (q_1 and q_2). Finally, some matrix additions are required, if the matrix addition operation is between two matrices, no data dependency exist. However, if the matrix addition is between more of two matrices, then, a dependency associated to the outer loop will appear.

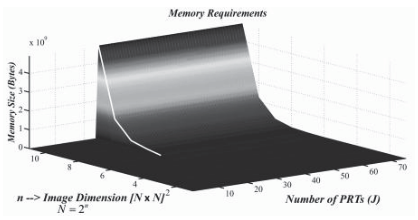
5 GPU Implementation

As previously mentioned in the introduction section, three versions of the R-MSF-Algorithm were implemented: Seq-CPU, OneThread, and MultiThread. These versions were implemented onto a PC with the following features: i7-2600K (3.4 GHz) with 4 cores, 12GB RAM (random access memory), and a Tesla C2075 GPU with 448 CUDA cores (14 multiprocessors x 32 CUDA cores), 1.15GHz clock rate, 65536 bytes of *constant memory*, 49152 bytes of *shared memory*, 32768 register per block,1536 maximum number of *threads* per multiprocessor, 1024 maximum number of *threads* per block, 1024x1024x64 maximum sizes of each dimension of a block, and 65535x65535x65535 maximum sizes of each dimension of a grid. Fig. 4(a) illustrates that the Av-Function algorithm was implemented using a single CUDA *kernel* named here as Average_Kernel; the Y-Function algorithm was implemented using three CUDA *kernels*: Vec_Mult_Kernel, Sum_Matrix_Kernel, and Yr_Kernel. In addition, the MSF-Function algorithm was implemented employing two steps: the first one performs the computation of the arguments (5) (previously stored in memory), and the second one performs the computation of each pixel of the output image (4), called here as psi_Kernel (CUDA *kernel*).



```
int main()
{
 cudamemcpy
 Average_Kernel<<< >>>
 cudamemcpy
 Vec_Mult_Kernel<<< >>>
 cudamemcpy
 Sum_Matrix_Kernel<<< >>>
 cudamemcpy
 Yr_Kernel<<< >>>
 cudamemcpy
 psi_Kernel<<< >>>
 cudamemcpy
}
```

(a)



(b)

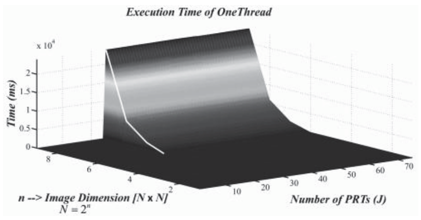
Fig. 4. GPU implementation. (a) GPU as parallel mathematical co-processor (master/slave as communication model) , (b) **.** Memory requirements for the R-MSF-Algorithm with $J = \{25, 26, \ldots, 75\}$, $M = 24$ and $N = \{2^7, 2^8, \ldots, 2^{11}\}$.

Hence, a GPU is employed as a mathematical co-processor of the CPU, where the CPU (host) is the master and a GPU (device) is the slave (see Fig. 4(a)). Therefore, the algorithm execution is as follows: Average_Kernel performs the average of the four possible polarization modes measurements ($\mathrm{I}_{jm\mathrm{VV}r}$, $\mathrm{I}_{jm\mathrm{VH}r}$, $\mathrm{I}_{jm\mathrm{HV}r}$, $\mathrm{I}_{jm\mathrm{HH}r}$ and $\mathrm{Q}_{jm\mathrm{VV}r}$,

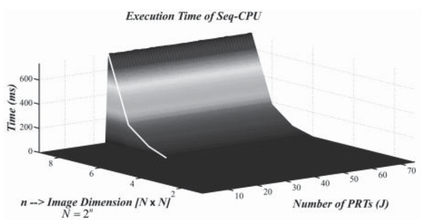
$Q_{jm\mathrm{VH}r}$, $Q_{jm\mathrm{HV}r}$, $Q_{jm\mathrm{HH}r}$) to obtain a complex number (I-Q) by each sensor in each PRT (J). Then, Vec_Mult_Kernel performs the required calculation of the four multiplications of column vector by a row vector, thereby, the J matrices with dimensions of [M×M] are obtained. Next, Sum_Matrix_Kernel performs the matrix addition of the J matrices that were obtained above, thus, four matrices with size of [M×M] are obtained. Therefore, Yr_Kernel performs two matrix additions and two average calculations of the I and Q parts, correspondingly, thereby, two matrices with length of [M×M] are produced (I and Q matrices of $\mathbf{Y}_r$). Finally, psi_Kernel performs multiplications of row vector by a matrix and multiplications of row vector by a column vector required to define each pixel value of the output image of dimension [N×N]. Once the R-MSF-Algorithm has been implemented, an analysis of memory requirements was performed to know the behavior of the memory re-size for different values of J ($J \geqslant M + 1$, number of PRTs), M (number of sensors), and N (image dimension) (see Fig. 4(b)). From (4), it follows that the parameter N has the greatest influence over the growth of memory requirement, as shown in Fig. 4(b). For example: if J =25, M =24 and N = {128, 256, 512}, hence, the total memory requirements were 19804032 bytes, 77606784 bytes, and 308817792 bytes, respectively.

6 Performance Analysis

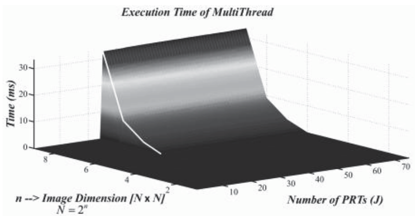
The execution time of a task is defined as the amount of time required by the R-MSF-Algorithm to accomplish this task. Hence, when N = 128 the average execution times achieved by each algorithm versions were: 2.7146ms by the MultiThread version, 47.4840ms by the Seq-CPU version, and 1549.8419ms by OneThread version. Then, when N = 256, the average execution times achieved by each algorithm versions were: 8.8912ms by the MultiThread version, 183.2489ms by the Seq-CPU version, and 6035.2695ms by the OneThread version.



(a)



(b)



(c)

Fig. 5. Execution time by each R-MSF-Algorithm with J = {25, 26,..., 75}, M = 24, and N = {128, 256, 512} for 100 iterations. (a) OneThread version, (b) Seq-CPU version, and (c) MultiThread version.

Furthermore, with N = 512, the average execution times achieved were: 23971.644ms by the OneThread version, 732.5262ms by the Seq-CPU version, and 33.1519ms by the MultiThread version. In addition, Fig. 5(a) to Fig. 5(c) show the general behavior of execution time of the three deployed versions when $J = \{25, 26, ... , 75\}$, $M = 24$, $N = \{2^7, 2^8, 2^9\}$, respectively. The latter shows that the parameter that most influences the execution time is N. The speedup (S_p) metric [7] of a parallel computation that employ p processing elements is defined as the rate: $S_p = T_1 / T_p$, where T_1 is the required execution time to perform the computation of the algorithm on one processing element (best sequential version, Seq-CPU), and T_p is the required execution time to perform the computation of the algorithm on p processing elements (MultiThread version). Therefore, the speedup metric shows the speed gain of the parallel computation of the algorithm over the sequential algorithm computation.

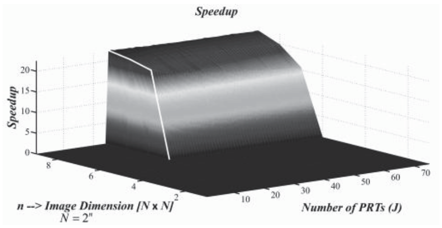


Fig. 6. Speedup with $J = \{25, 26,\dots, 75\}$, $M = 24$, and $N = \{128, 256, 512\}$ for 100 iterations

Hence, the achieved average speedup by the MultiThread version was: 17.57 for N = 128, 20.6187 for N = 256, and 22.0969 for N = 512. In addition, Fig. 6 shows the speedup general behavior of the deployed algorithm when $J = \{25, 26, ... , 75\}$, M = 24, $N = \{2^7, 2^8, 2^9\}$, respectively.

7 Conclusions

This paper presents an efficient real time implementation of the R-MSF-Algorithm that is employed to solve the remote sensing imaging problems. The improvement of the execution time was achieved by transformation of some code segments and mapped on a GPU in a parallel way. Therefore, the parallelized implementation is highly scalable on multi-processors and for different: number of sensors (M), number of pulse repetition times (J), and different image sizes [$N \times N$]. The MultiThread version achieved excellent speedups of 15.5X and 22.5X (for image sizes of [128 × 128], [256 × 256] and [512 × 512]), allowing to obtain 50, 22 and 6 images per second respectively. Furthermore, the GPU parallelized implementation is portable across others GPU architectures.

References

1. Tanner, A.B., et al.: Initial Results of the Geosynchronous Synthetic Thinned Aperture Radiometer (GeoSTAR). In: IEEE Inern. Symposium on Geoscience and Remote Sensing, IGARSS 2006, pp. 3951–3954. IEEE (2006), ISBN 0-7803-9510-7/06
2. Shkvarko, Y., Espadas, V., Castro, D.: Descriptive Experiment Design Optimization of GeoSTAR Configured Multisensor Imaging Radar. In: 4th International Radio Electronics Forum (IREF 2011), Kharkov, Ukraine, vol. I, pp. 76–81 (October 2011)
3. Ponomaryov, V.I.: Real-time 2D–3D filtering using order statistics based algorithms. Journal Real-Time Image Processing 1, 173–194 (2007)
4. Hwu, W.: GPU Computing Gems Emerald Edition, 1st edn. Morgan Kaufmann (2011)
5. NVIDIA, CUDA C Programming Guide, version 6.0 (2014)
6. Banerjee, U.: Loop Transformations for Restructuring Compilers: The Foundations, 1993th edn. Springer (January 31, 1993)
7. Moldovan, D.I.: Parallel Processing, From Applications to System. Morgan Kaufmann Publishers, San Mateo California, U.S.A.