# ACCELAERATED SAR IMAGE GENERATION ON GPGPU PLATFORM

AK Agrawal, C Bhattacharya
Defence Institute of Advanced Technology (DIAT)
Pune 411025, India
Email: anidiv@gmail.com, cbhat0@diat.ac.in

P Somawanshi, M Khadtare, SK Karandikar
Computational Research Lab (CRL)
Pune 411004, India
Email: prakalp.somawanshi@crlindia.com, shrirang@crlindia.com

***Abstract*—Image generation from satellite-borne synthetic aperture radar (SAR) video data requires numerical computations of the order of gigaflops (GFLOPs). Current parallel processing schemes distribute computation load among a number of parallel processing elements (PEs) for improving efficiency in computation. In order to handle the huge computational task for real-time image generation, we report here the use of general purpose graphics processing units (GPGPUs) that have shown excellent results for data-parallel kinds of operations. In this paper, we describe the architecture, and highlight the steps needed to extract parallelism in implementation of the algorithm for accelerated SAR image generation.**

## I. Introduction

Image generation from satellite-borne synthetic aperture radar (SAR) video raw data requires elaborate computational procedures. For example, nominal footprint size in fine beam mode of operation of RADARSAT-1, an earth observation satellite having a C-band SAR sensor as payload is 50Km in range and 3.6Km in azimuth. The scale of numerical computations is in the order of gigaflops (GFLOPs) that may be appreciated from the fact that such ground footprint results an image with 7.2m ground range resolution and 5.26m fine spatial resolution in azimuth [1].

One direct way of improving efficiency in SAR video data processing is to distribute image formation process among several processing elements (PEs). Such schemes essentially depend on the configuration of the PEs, their interconnectedness, message passing protocols devised and data transmission bandwidth among PEs [2]. Although such schemes of distributed processing with improved communication protocols enhance speed of computation they do not evolve or include any parallel signal processing algorithms for SAR image formation.

A recent realization of a computational platform targeted towards highly data-parallel computations is the general purpose graphics processor units (GPGPUs). These platforms available from NVIDIA, ATI and Intel have a large number of processors (of the order of a few hundred) structured to allow multiple threads of execution. This massively parallel architecture results in speed-ups of 100X, and more easily obtained when an algorithm of implementation is mapped appropriately for execution in the GPGPUs. However, in cases where the structure of the algorithm does not map directly onto the architecture, we need to develop new methods to extract parallelism, and correspondingly improve performance. In this work, we use GPGPUs from NVIDIA, and utilize compute unified device architecture (CUDA) as the programming platform. In the following sections we describe a parallel algorithm of implementation of SAR video data processing that substantially improves the data parallelism of CUDA operations on the GPGPU.

## II. Algorithm for accelerated SAR image generation

In coherent receivers such as in a SAR system the principle of matching the received signal phase with the transmitted signal phase at a stable frequency of transmission is applied that depends heavily on fast Fourier transform (FFT) based techniques. The traditional approach what we call brute force correlation cause serious overhead on the latency of the system for large receive data window. We generate SAR image from RADARSAT-1 video data by an inherently parallel algorithm for matched filtering of block data vectors [3]. Matched filter implementation in this partitioned block correlator algorithm is done by dividing the impulse response function of matched filter into several blocks of equal lengths. The partial convolution results from FT domain operations may be shifted and summed up to produce the final output utilizing the linearity of operation.

Consider $u^*(-n)$, $y(n)$ to be the matched filter vector and the received data vector from ground footprint respectively. The matched filter is the conjugate of time-reversed transmitted complex linear FM chirp envelope $u(n)$ of finite duration $T_P$ sampled at the rate of $B$, the transmitted bandwidth. The finite vectors $u(n)$, $y(n)$ are partitioned respectively in $P$ and $Q$ numbers of non-overlapping data blocks shown in Fig. 1, each block being of $K$ samples.

$$u(n) = \Sigma_{i=0}^{P-1} u_i(n), y(n) = \Sigma_{j=0}^{Q-1} y_j(n) \tag{1}$$

where $u_i(n) = u(n), iK \leq n \leq (i+1)K-1$, $y_j(n) = y(n), jK \leq n \leq (j+1)K-1$, $P < Q$. Matched filtering being a linear process is the complex correlation between the block vectors. Summation of outer product of blocks of $u_i(n)$ and $y_j(n+l)$ for a lag of $l$ samples produce identical results as in the correlation of $u(n)$, $y(n)$.

$$r(l) = \Sigma_{j=0}^{Q-1} \Sigma_{i=0}^{P-1} \Sigma_{n=0}^{2K-|l|-2} y_j(n+l) u_i(n). \tag{2}$$

Time domain correlation $r_{ij}(l)$ between any two blocks $u_i(n)$, $y_j(n+l)$ in (2) is implemented by FFT in the transform domain.

$$r_{ij}(l) = IDFT[U_i^*(k)Y_j(k)], 0 \leq k \leq (2K-1). \quad (3)$$

The first $K$ samples in (3) account for $r_{ij}(l)$ with one zero padding for $l < 0$, the rest $K$ samples are for $r_{ij}(l)$ with $l > 0$ as shown in Fig. 1. The first outer sum in (2) is the partial correlation of $P$ blocks of $u(n)$ with each block of $y_j(n+l)$ depending on the shift index $l$. The outermost sum over all blocks of $r_{ij}(l)$ is realized by a block matrix operator $\mathbf{A}$ of size $(P+Q) \times 2PQ$ and a block correlation vector matrix $\mathbf{T}$ whose non-zero elements in a block of $K$ columns are $r_{ij}(l)$ derived from (2). The operator matrix $\mathbf{A}$ produces the partial outer sums by weighting the corresponding blocks of $r_{ij}(l)$ in the rows of $\mathbf{T}$. For example, $\mathbf{A}$ is of size $7 \times 24$ for $P$ to be 3 and $Q$ to be 4 blocks of data and $\mathbf{T}$ is of size $24 \times 7K$. Row elements of $\mathbf{A}$ are one corresponding to the blocks of $r_{ij}(l)$ in rows of $\mathbf{T}$ those are added up to produce the partial sums in final correlation vector matrix $\mathbf{X} = \mathbf{AT}$. Matched filter output of the block correlator algorithm is derived from the non-zero elements of rows in $\mathbf{X}$ for sample indices $l > 0$ as filtering here is a causal process.

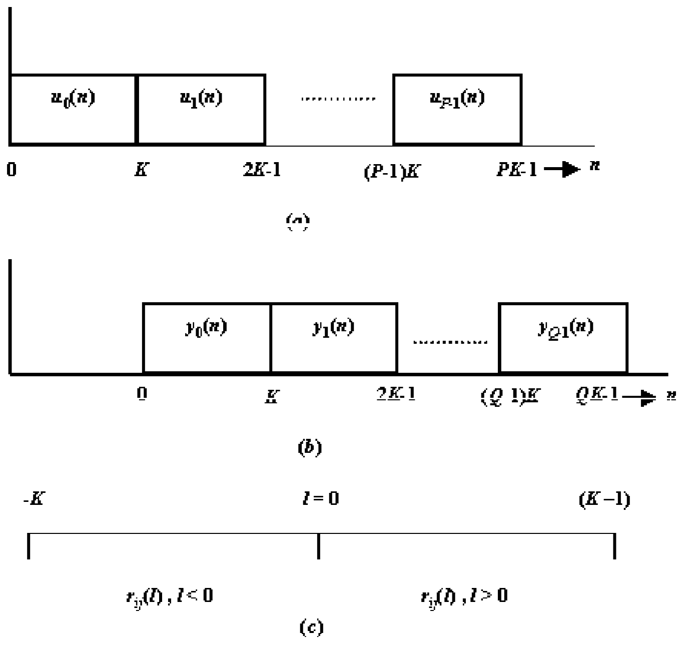

Fig. 1. Partitioned block correlator algorithm; (a) filter data blocks , (b) received data blocks, (c) correlator output vector block.

Following RADARSAT sensor parameters $u(n)$ in range is of size 1349 samples, and the synthetic aperture in azimuth is of size 701 samples. In range, radix-2 FFT size $N = 512$ is chosen for implementation of the partitioned block correlator algorithm, and for correlation in azimuth $N = 256$ is taken. This results in $P = 3$ partitioned blocks for $u(n)$ in both range and azimuth dimensions. The length of $y(n)$ in range is 2048 samples and in azimuth is 1024 samples so that the number of blocks $Q$ of $y_j(n+l)$ remains 4 in both dimensions. Therefore, the same kernel of operations for correlation of block vectors may be utilized in deriving matched filtered output results both in range and azimuth. The actual size of $\mathbf{A}$ is reduced to $4 \times 15$ in implementation. The required matched filter output being only of the length of received data vector, the number of correlation data blocks $r_{ij}(l)$ taken into account in producing the partial sums being 15. The operator matrix $\mathbf{A}$ for this example of implementation is

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (4)$$

Valid output data vector of 2048 samples in range and of 1024 samples in azimuth are derived from $\mathbf{X} = \mathbf{AT}$ by concatenating $N$ non-zero samples from each of the 4 rows of $\mathbf{X}$ in range and azimuth matched filtered operation. Rows of range-compressed data are to be written along columns before proceeding to azimuth matched filtering and a corner turn of data memory locations is necessary.

## III. Implementation on GPGPU Platforms

Unlike the previous GPU architecture, CUDA incorporates a unified computing unit called streaming processor (SP). For example, the NVIDIA 8800GTX contains 128 SPs. Every 8 streaming processors constitute a streaming multiprocessors (SM). The SM has the single instruction, multiple thread (SIMT) architecture that executes the same instructions on different data. Four types of on-chip memory are associated with each SM: (1) a set of local 32-bits general propose registers per processor, (2) a 16-bank shared memory that is shared by all SPs within a single SM that allows SPs to communicate with each other without passing data outside the chip, (3) a read-only constant cache that maps to the constant memory of the device DRAM accessible by all SPs, and (4) a read-only texture cache that maps to the texture memory of the device DRAM accessible by all SPs. In addition to the constant memory and texture memory that are read-only memory for SPs there is a global on-device memory that is both readable and writable by SPs.

CUDA allows developers to harness the underlying massively parallel compute engine with C-like programming language. Programming model in CUDA differs significantly from single threaded CPU code and even the parallel code for GPUs before CUDA. In a single-threaded model, the CPU fetches a single instruction stream that operates serially on the data. A superscalar CPU may route the instruction stream through multiple pipelines, but theres still only one instruction stream, and the degree of instruction parallelism is severely limited by data and resource dependencies. Notably, the differences in developing in CUDA compared to the standard C language are implied parallel execution of threads and memory hierarchy management.

In CUDA programming, the original program is first compiled to conform to the CUDA device instruction set and it becomes a parallelized new program that is called kernel. The kernel is downloaded to the GPU device that acts as a coprocessor to the host (CPU). It is executed by the mechanism of threads that are organized in thread block. The threads within a thread block can co-work with each other through the shared memory and can synchronize their execution to coordinate their memory access. The maximum number of threads within a thread block is limited; however, thread blocks that execute the same kernel can be batched together to form a grid of blocks. Therefore, the total number of threads that execute a single kernel can be much larger. The SM may execute one or more thread blocks concurrently depending on the shared memory and register occupancy [4]. Data parallelism is a key concept for GPU operation. An algorithm is broken down into CUDA threads which is scheduled for concurrent execution. For full occupancy of numerous execution units within a GPU creation of thousands of threads are recommended that is ideal for multithreaded GPU architecture.

These threads executed on GPU are organized into a three-level hierarchy. Threads are grouped in blocks and many blocks are run in a grid. The codes defined in a kernel function are to be executed by each of these threads, and process the data stored in the device memory. Programming by CUDA allows shared memory and barrier synchronization. Threads in the same block can cooperate among themselves through barrier synchronization and shared access to a memory space private to the block [5]. For example, the Tesla- GeForce GTX 275 GPU card produce 512 number of threads per block of execution. The maximum size of a grid of thread blocks is 65535.

Signal processing for SAR image formation is done mainly in two steps, the first one is processing of the input raw video data in range dimension followed by processing the range image processed data in the azimuth dimension. The flow chart in Fig. 2 actually articulates our approach towards handling data parallelism, and computational procedures of the SAR algorithm described in Section II over the hybrid hardware platform like X86 as a main processor and GPGPU unit as a co-processor. Since SAR processing involves intensive use of FFT and IFFT operations, we make direct utilization of NVIDIA's CUFFT libraries. The library functions deliver a parallel implementation of complex and real data transformation in one, two or three dimensions.

Video data block of size $(1024 \times 2048)$ and transmit data vector of size $(1349 \times 1)$ are first read into CPU in this example of SAR data processing. The transmit vector is rearranged in $P = 3$ partitioned data blocks, each block is of 512 samples. Video data is rearranged in $Q = 4$ partitioned data blocks. In all, twelve blocks of partitioned correlation operations are to be performed in this example for every range line of video data. The data blocks once partitioned are sent to GPU for FFT related operations shown in Fig. 2. Kernels are created to perform the micro-operations shown in the flow chart. For optimum usage of on-chip memory in SP devices and to reduce latency in communication from CPU to GPU following optimization are done :

- threads are identified corresponding to the algorithms those are computationally intensive ( e.g., data conversion, FFT, complex transpose, etc.)
- prudent memory usage and data transfer are adopted to reduce time in host-device memory transfer
- dynamic assignment of thread blocks are done to SMs of GPU enabling efficient utilization of resources.

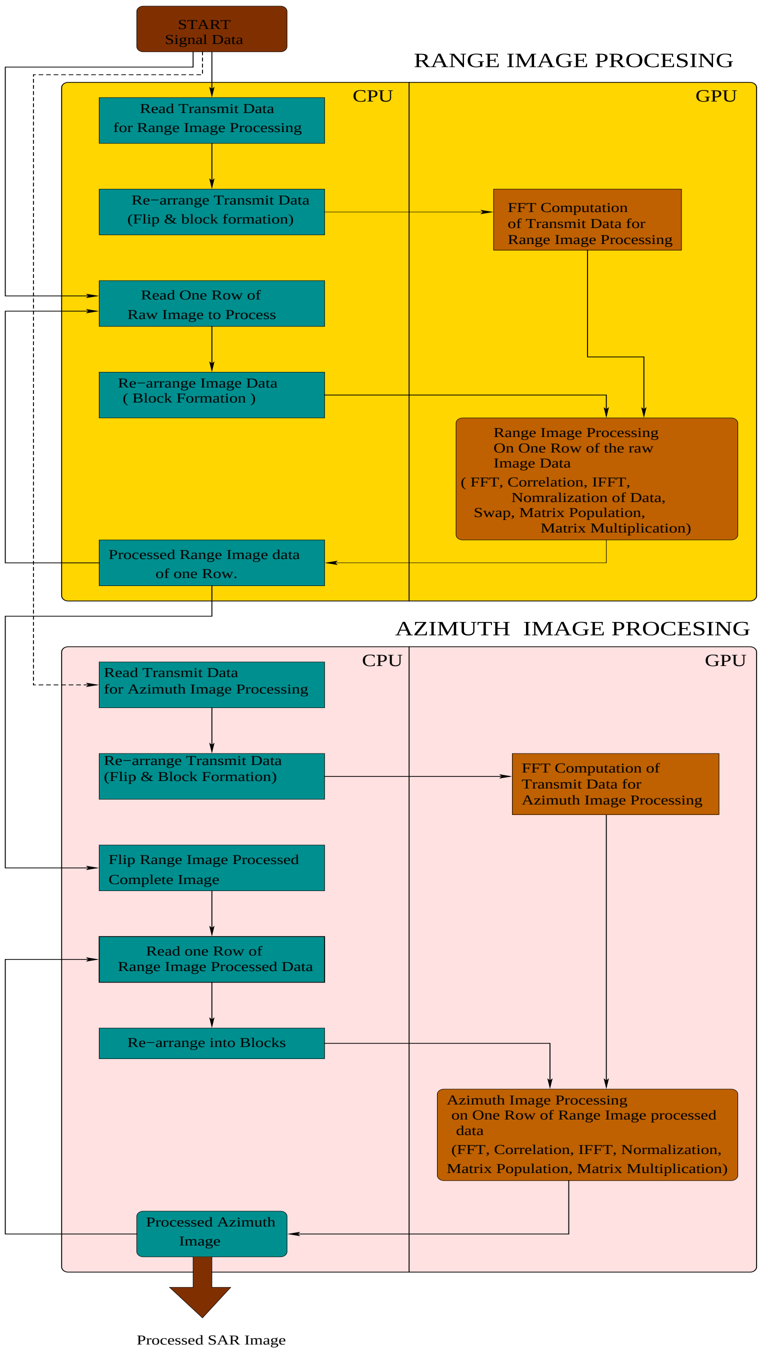
START
Signal Data
RANGE IMAGE PROCESING
CPU
GPU
Read Transmit Data for Range Image Processing
Re−arrange Transmit Data (Flip & block formation)
FFT Computation of Transmit Data for Range Image Processing
Read One Row of Raw Image to Process
Re−arrange Image Data ( Block Formation )
Range Image Processing On One Row of the raw Image Data ( FFT, Correlation, IFFT, Nomralization of Data, Swap, Matrix Population, Matrix Multiplication)
Processed Range Image data of one Row.
AZIMUTH IMAGE PROCESING
CPU
GPU
Read Transmit Data for Azimuth Image Processing
Re−arrange Transmit Data (Flip & Block Formation)
FFT Computation of Transmit Data for Azimuth Image Processing
Flip Range Image Processed Complete Image
Read one Row of Range Image Processed Data
Re−arrange into Blocks
Azimuth Image Processing on One Row of Range Image processed data (FFT, Correlation, IFFT, Normalization, Matrix Population, Matrix Multiplication)
Processed Azimuth Image
Processed SAR Image

Fig. 2. GPU implemnetation flow of SAR image generation.

Once the partitioned matched filtered output vector is there population of matrix $\mathbf{T}$ is done over a device with the help of the swapped output data. Matrix multiplication is the next important activity carried out in the GPU those are of size

TABLE I
RANGE PROCESSING PERFORMANCE FOR $(1024 \times 2048)$ BLOCK.

| CPU / GPU | Core Details | Timing(sec) | Speedup |
|---|---|---|---|
| AMD Athlon X2 Dual Core | 1 | 45.337 | 1x |
| Fx1800 | 64 | 19.50 | 2.32x |
| Tesla GeForce GTX 275 | 192 | 1.8751 | 24.17x |

TABLE II
AZIMUTH PROCESSING PERFORMANCE FOR $(1024 \times 2048)$ BLOCK.

| CPU / GPU | Core Details | Timing(sec) | Speedup |
|---|---|---|---|
| AMD Athlon X2 Dual Core | 1 | 50.065 | 1x |
| Fx1800 | 64 | 36.60 | 1.368x |
| Tesla GeForce GTX | 192 | 3.725 | 13.44x |

TABLE III
COMPLETE PROCESSING PERFORMANCE FOR $(1024 \times 2048)$ BLOCK.

| CPU / GPU | Core Details | Timing(sec) | Speedup |
|---|---|---|---|
| AMD Athlon X2 Dual Core | 1 | 95.00 | 1x |
| Fx1800 | 64 | 36.60 | 2.6x |
| Tesla GeForce GTX | 192 | 5.563 | 17.07x |

$\mathbf{A}(4 \times 15)$ and $\mathbf{T}(15 \times 3584)$. We used CUBLAS routine to get the complex valued matrix multiplication done over GPU. Every range line processed row data is returned to the host memory space. After completion of processing in range dimension we start processing in azimuth dimension on the range image processed elements. Azimuth signal processing executes almost in a similar way as range signal processing is done.

Results of SAR image generation in different GPGPU platforms such as Fx1800, Tesla C870 are compared with performance of workstations with AMD Athlon(tm) 64 X2 Dual Core Processor 4600 at 2.411GHz. The accelerated speedup in execution of range, azimuth , and complete image generation are shown in Table I-III for a block of $(1024 \times 2048)$ RADARSAT-1 SAR video complex data. The final RADARSAT SAR magnitude image chip generated by the CUDA flow diagram is displayed in Fig. 3.

## IV. CONCLUSION

In this paper, we proved that the goal of reaching near real-time processing of the radar system to accommodate field operation is feasible with the aid of GPU accelerators. Moreover, GPU computing makes the power of high performance computing possible and accessible in physically smaller systems.

GPUs are specialized to calculate single-precision mathematic operations, and has the potential to work in parallel with the CPU in asynchronous manner. The combined power of a CPU and GPU improves processing speed and complements existing computing resources.

The high efficiency of the GPU-based processor provides a promising way to address the complicated problems of high performance SAR processing in two aspects. First, given the GPU-based processor is suitable for compute intensity processing, it can be designed to increase arithmetic intensity to get more precise images. Second, considering the accelerated performance of these processors real time SAR image generation with fine grained parallelism becomes a distinct possibility.

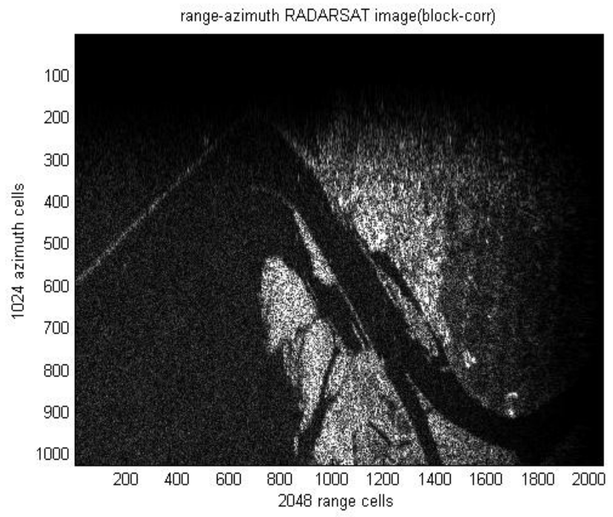

Fig. 3. Processed SAR image from RADARSAT-1 video data in CUDA programming.

## ACKNOWLEDGMENT

The authors from CRL acknowledge the kind support received from CRL in registering and presenting the paper.

## REFERENCES

[1] I. G. Cumming and F. H. Wong , *Digital Processing of Synthetic-Aperture Radar Data:Algorithms and Implementations*, Artech House, 2005.
[2] J. Suh , U. Monte, and V. K. Prasanna, *Parallel implementation of synthetic aperture radar on high performance computing platforms*, Proc. IEEE-ICAPP 97, 3rd Int. Conf. on Algorithms and Architectures for Parallel Processing, pp. 557 - 570, Dec 1997.
[3] C. Bhattacharya, *Parallel Processing of Satellite-Borne SAR Data for Accurate and Efficient Image Formation*, Proc. EUSAR 2010, Germany, pp. 10461049, July 2010.
[4] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors A Hands-on Approach*, NVIDIA, 2010.
[5] J. Nickolls, I. Buck, M. Garland and K. Skadron, *Scalable parallel programming with CUDA* , Queue, ACM, vol.6, no. 2, pp. 40-53, 2008.