Accepted version of: C.J. Venter, H. Grobler and K.A. AlMalki, "Implementation of the CA-CFAR Algorithm for Pulsed-Doppler Radar on a GPU Architecture," *2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies,* 6-8 December 2011, pp. 233-238. Published version available online:
http://ieeexplore.ieee.org/arnumber=6132514

©2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)

# Implementation of the CA-CFAR Algorithm for Pulsed-Doppler Radar on a GPU Architecture

C.J. Venter[1,2], H. Grobler[2] and K.A. AlMalki[3]
Email: cventer@csir.co.za, hans.grobler@up.ac.za, kalmalki@kacst.edu.sa
[1]Defence, Peace, Safety and Security, Council for Scientific and Industrial Research, South Africa
[2]Department of Electrical, Electronic and Computer Engineering, University of Pretoria, South Africa
[3]Electronics, Communications and Photonics, King Abdulaziz City for Science and Technology, Kingdom of Saudi Arabia

***Abstract* — The Cell-Averaging Constant False-Alarm Rate (CA-CFAR) algorithm was implemented and optimized in software on the NVIDIA Tesla C1060 GPU architecture for application in pulsed-Doppler radar signal processors. A systematic approach was followed to gradually explore opportunities for parallel execution and optimization by implementing the algorithm first in MATLAB (CPU), followed by native C (CPU) and finally NVIDIA CUDA (GPU) environments. Three techniques for implementing the CA-CFAR in software were identified and implemented, namely a naïve technique, sliding window technique and a new variant which employs the Summed-Area Table (SAT) algorithm. The naïve technique performed best on the GPU architecture. The SAT technique shows potential, especially for cases where very large CFAR windows are required. However, the results do not justify using the GPU architecture instead of the CPU architecture for this application when data transfer to and from the GPU is taken into consideration.**

***Keywords* — *CFAR; GPU; Doppler; radar; signal processing***

## I. Introduction

A Constant False-Alarm Rate (CFAR) detector in a radar system detects targets in varying interference by adjusting detection thresholds dynamically. The computational workload of calculating the interference estimate for every radar data sample in order to adjust the detection threshold can be very high. High throughput and low latency is required for radar applications, where the latter is especially important in tracking systems which need to maintain a tight tracking loop.

CFAR processing in modern radar signal processors is typically performed with Field-Programmable Gate Arrays (FPGAs) or Digital Signal Processors (DSPs) due to the processing and I/O performance that they provide. Current generation many-core GPUs can deliver massive computational performance on a single chip. The memory bandwidth between a GPU chip and its own high-speed memory is also very high as a result of very wide data bus widths and high memory clock rates. Although GPUs were originally designed to perform graphics processing, recent generations make provision for General Purpose Computing on the GPU (GPGPU). GPUs could therefore potentially be used to perform CFAR processing in pulsed-Doppler radar systems.

The authors would like to express their gratitude to the National Program for Electronics, Communications and Photonics at the King Abdulaziz City for Science and Technology for supporting this work.

An algorithmic optimization of the Cell-Averaging CFAR (CA-CFAR) algorithm was performed in an attempt to find an efficient algorithmic structure for software implementation on the GPU architecture. Three techniques were identified, implemented, optimized and evaluated using a systematic approach to move from high-level computing languages to the target GPU architecture. The GPU results were compared to results of our own implementations in native C, as well as another CA-CFAR software reference implementation.

The results shows that, in terms of kernel throughput and latency, the GPU implementations generally perform better than comparative CPU implementations for larger input sizes. However, in terms of total throughput and latency, which includes data transfer to and from the GPU, the performance results do not justify using the GPU architecture for the range of input sizes that were evaluated.

## II. CA-CFAR for Pulsed-Doppler Radar

A 2D data matrix is generally used to represent the sampled data for pulsed-Doppler radar systems with a single receiver front-end element. The 2D data matrix is called a burst and it has range and Doppler dimensions. The range dimension contains “fast-time” samples of the echo of a single received pulse. The Doppler dimension contains “slow-time” samples which represent the echo received from successive transmitted pulses in the burst.

A Constant False-Alarm Rate (CFAR) detector adjusts the detector threshold in real-time in order to maintain a constant, design-specific false-alarm rate with corresponding probability of $P_{fa}$. The threshold is determined by estimating the interference around the target. A Cell-Averaging CFAR (CA-CFAR) detector estimates the interference power for a Cell Under Test (CUT) by averaging the sample values of adjacent cells, under the assumption that interference statistics are homogeneous in a localized area.

### A. CA-CFAR Algorithm

The maximum likelihood estimate for the interference power, $\widehat{\beta^2}$ is obtained from the average of $N$ samples in the vicinity of the CUT [1] as shown in equation 1.

$$\widehat{\beta^2} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{1}$$

978-1-4577-1084-1/11/$26.00 ©2011 IEEE

A CFAR window is the set of $N$ cells around the CUT that are included in the estimation of the interference power. The CFAR window excludes a set of guard cells in the immediate vicinity of the CUT, since energy from a target in the CUT might occupy multiple cells around the CUT as well. In this case the energy in the cells immediately adjacent to the CUT do not represent interference alone and will cause the detector threshold to be raised artificially if included in the calculation. The threshold, $\widehat{T}$ is then calculated as a function of the estimated interference power and the desired false-alarm probability $\overline{P}_{fa}$ [1] as shown in equation 2 and 3.

$$\widehat{T} = \alpha\widehat{\beta^2} \tag{2}$$

$$\alpha = N(\overline{P}_{fa}^{-1/N} - 1) \tag{3}$$

### B. Existing CA-CFAR Software Implementations

The High Performance Embedded Computing (HPEC) Challenge Benchmark Suite [2] includes a CFAR benchmark for CPUs. The assessment in [3] compared their CA-CFAR GPU implementation to the HPEC CFAR benchmark CPU implementation. The implementation in [3] favours parallelism by minimizing communication and synchronization between elements. The problem effectively turns into an embarrassingly parallel problem using this approach.

A speedup factor of between 2.6 and 166 was achieved with the GPU implementation in [3] over the HPEC CPU reference implementation [4]. The strict kernel latency metric, as described in the HPEC Challenge benchmark guidelines [2], is used. The conclusion in [3] is that the overall run time is limited by global memory bandwidth.

The HPEC Challenge benchmark code for CPUs is publicly available, which allowed us to compile and run the code on the same platforms that were used to obtain our own results. No code is available for the other implementations that were found and we could therefore not do comparisons with these implementations on the target hardware platforms.

## III. High-Level Algorithmic Optimization

Performance optimization is typically performed on different levels ranging from high-level algorithmic optimization, to medium-level implementation optimization, down to low-level instruction and other other architecture-specific optimizations. The structure of an algorithm that is implemented in software can have a major impact its performance, based on how well-suited the chosen structure is for the target hardware architecture.

A process of optimizing the algorithmic structure of the CA-CFAR algorithm for software implementation on the GPU architecture was followed. The aim was to optimize the computational performance on the GPU architecture in order to increase throughput and reduce latency. Three techniques with different algorithmic structures were identified, namely naïve, sliding window and Summed-Area Table (SAT).

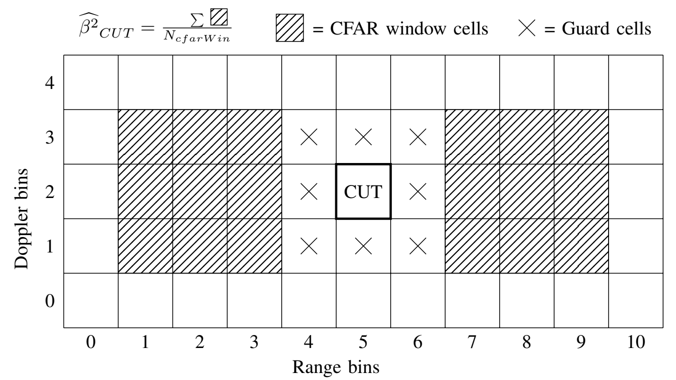


Figure 1. Calculation of $\widehat{\beta^2}_{CUT}$ from CFAR window for naïve technique.

### A. Naïve Technique

One of simplest and most intuitive ways of implementing the CA-CFAR is to simply sum the values for all cells in the CFAR window for each CUT given by

$$\widehat{\beta^2}(i,j) = \frac{1}{N_{cfarWin}} \sum_{k=1}^{N_{cfarWin}} CW(i,j,k), \tag{4}$$

where $CW(i,j,k)$ is the $k$th CFAR window cell of cell $(i,j)$ as shown in figure 1. We refer to this technique as the naïve technique due to the potential drawbacks associated with using such a simple, brute-force approach. Redundant summation will be performed, since there is a large overlap in the CFAR window cells for adjacent cells under test in the range-Doppler map. The computational complexity increases as the dimensions of either the input data or the CFAR window increases. The naïve technique provides an embarrassingly parallel structure which means that each cell can be evaluated independently without requiring synchronization with any other cells.

The time complexity for the naïve technique with a range-Doppler map of $N_r$ range bins by $N_d$ Doppler bins and with leading and lagging CFAR windows of $N_{rwin}$ by $N_{dwin}$ cells each, is given by

$$O(N_{rd} \times N_{cfarWin}), \tag{5}$$

where $N_{rd} = N_r \times N_d$, $N_{cfarWin} = 2 \times N_{rwin} \times N_{dwin}$.

### B. Sliding Window Technique

The CA-CFAR can also be implemented in software using a sliding window technique, which capitalizes on the redundancy in the calculation of the local noise estimate for adjacent cells [2] as given by

$$\widehat{\beta^2}(i+1,j) = \widehat{\beta^2}(i,j) + \Delta, \tag{6}$$

where $\Delta$ is calculated as shown in figure 2. It is more computationally efficient to slide the CFAR window by one cell at a time and only account for the difference in the local interference estimate, instead of naïvely performing redundant summation. Computational complexity is reduced at the cost of reducing the parallelism of the algorithm, as a result of the inherent data dependencies that are created.

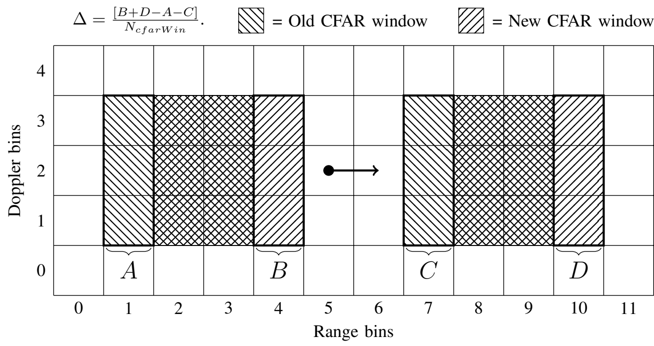


Figure 2. Calculation of $\Delta$ when sliding in range dimension using sliding window technique.

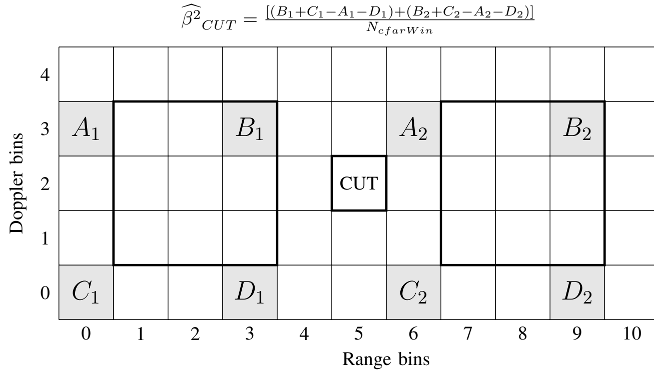


Figure 3. Calculation of $\widehat{\beta^2}_{CUT}$ using Summed-Area Table generated from range-Doppler map.

The time complexity for the sliding window technique with a range-Doppler map of $N_r$ range bins by $N_d$ Doppler bins and with leading and lagging CFAR windows of $N_{rwin}$ by $N_{dwin}$ cells each, is given by

$$O(N_{rd} \times \sqrt{N_{cfarWin}}), \tag{7}$$

where $N_{rd} = N_r \times N_d$, $N_{cfarWin} = 2 \times N_{rwin} \times N_{dwin}$.

### C. Summed-Area Table Technique

A Summed-Area Table (SAT) [5] or Integral Image [6], once generated, can be used to obtain the sum of values inside any sub grid of the original input data matrix by doing lookup and summation of 4 data elements. This technique can be applied to CA-CFAR by generating a SAT from the input range-Doppler map sample values. Once generated, the SAT can be used to compute the local interference estimate efficiently, by optimizing the summation calculation as shown in figure 3. The summation is performed in constant time, regardless of the size of the CFAR window. No evidence was found of this technique being used for CFAR in radar applications.

The time complexity for the SAT technique with a range-Doppler map of $N_r$ range bins by $N_d$ Doppler bins and an arbitrary CFAR window size, is given by

$$O(N_{rd}), \tag{8}$$

where $N_{rd} = N_r \times N_d$.

## IV. Low-Level Optimization for GPU Architecture

Following the high-level algorithmic optimization of the CA-CFAR algorithm, additional lower-level optimizations were performed for the target GPU architecture. The optimizations that were made are based on known techniques in GPGPU and computer vision fields for improving performance.

### A. Precision

FPGA and DSP implementations typically use 16-bit or 32-bit fixed-point representation for sample values. Single-precision floats were used instead for the GPU implementations. On the target GPU architecture that was used, the single-precision floating point performance for typical operations exceed performance that is obtained when using integer or double-precision formats. Single-precision floats also provide sufficient resolution for the application under consideration.

### B. Apron Generation

When Doppler ambiguities are present, the cells around the edges of the Doppler dimension of a range-Doppler map are adjacent in terms of velocity. As a result, CFAR algorithms often implement wrapping instead of clipping of the CFAR window in the Doppler dimension around the edges of the range-Doppler map. Given that GPU performance can degrade when conditional branches diverge, the need to check for all cases where wrapping can occur is a potential problem. A standard technique used for GPUs to avoid branching in similar situations, is to pad an image with an apron of pixels. For all GPU implementations an apron of redundant cells is replicated around the edge of the range-Doppler map, which mirrors the opposite edge, to allow for wrapping without divergent branching.

### C. Shared Memory Implementation

The target GPU architecture provides a large off-chip memory which is referred to as device memory or global memory. A small amount of on-chip shared memory is also available which needs to be explicitly loaded with data by the programmer. The shared memory is shared by and limited to the scope of the processor cores in the particular symmetric multiprocessor unit. Once data has been loaded into shared memory, the read latency is typically in the order of 100 times less than reading from global memory.

The basic naïve implementation reads all cells in the CFAR window of a particular CUT from global memory. A variation of the naïve technique was also implemented which uses shared memory to reduce the read latency for redundant data reads. The shared memory implementation also segments the summation into two stages where rows and columns are summed separately, allowing for more optimal use of the limited shared memory that is available on the target GPU. With this two-stage process the overlap only increases linearly with increases in the range and Doppler dimensions of the CFAR window, instead of in two dimensions simultaneously as with the global memory implementation.

The shared memory implementation has a limitation in terms of the maximum CFAR window size as a result of the limited amount of shared memory available on the target GPU. The maximum CFAR window size in both range and Doppler dimensions is limited to a maximum of 33 cells.

## V. Implementation

The optimization of the CA-CFAR algorithm was an iterative process which included implementation and evaluation cycles. The process, environment and methods that were used to develop, verify and benchmark the implementations is discussed in this section.

### A. Development Process

A systematic approach was followed by moving gradually from initial implementations in high-level computing environments to native implementations on the target GPU architecture.

Implementations were performed and verified in MATLAB [7], which was used as a rapid prototyping platform. Native C implementations for the CPU were then performed as a stepping stone to the final GPU target architecture. The initial native C implementations were single-core CPU implementations. Multi-core optimizations were then made for the native C implementations on the CPU architecture. The techniques were then implemented and optimized on the target NVIDIA GPU architecture.

A comparable level of optimization was performed for the native C and GPU implementations on the respective architectures for each technique.

### B. Environment

A GPGPU system with an NVIDIA Tesla [8] C1060 GPU was used as the development, test and benchmarking platform for all the GPU implementations. The CPU implementations were benchmarked on a system with dual Intel Xeon X5355 quad-core CPUs which yields a total of 8 processor cores.

OpenMP [9] was utilized to optimize the native C code in order to use multiple cores of the processor. Restructuring of the code was necessary in some cases in order to obtain maximum benefit of using OpenMP. Opportunities for exploiting parallelism in the algorithms were exposed using this approach and it also served as a stepping stone towards an efficient parallel implementation for the GPU.

NVIDIA Compute Unified Device Architecture (CUDA) [10] was utilized for all the GPU implementations. The CUDA Data Parallel Primitives (CUDPP) [11] library was used to generate the SAT for the GPU implementation that uses the SAT technique.

### C. Verification

The output produced by the MATLAB implementation for a particular test data set or test burst was used as a baseline to verify other implementations against.

NVIDIA provide a CUDA Visual Profiler [10] (computeprof) as part of the standard CUDA Toolkit release. The profiler utilizes hardware counters that are built into the NVIDIA chip sets for debugging and profiling purposes. The GPU kernels were profiled and analyzed during development using this profiler.

CUDA also provides access to hardware timers via GPU events in the API. These events can be used do accurate timing of GPU code relative to the internal GPU clock. These GPU events were utilized to obtain all the timing information for the results.

All implementations process a single burst of synthetic radar data which is placed in host (CPU) memory before timing begins. During execution this burst of data is transferred to the GPU, processed and transferred back to the host.

## VI. Results

The results show the throughput that was achieved, which is derived from the measured latency of processing a single burst and the data size of the burst. For the GPU implementations, both the kernel and total latency were measured and used to derive the kernel and total throughput statistics. Total latency is the time required to transfer the input data from the host memory to the GPU memory, perform the CA-CFAR processing on GPU and transfer the output data back from the GPU memory to the host memory. Kernel latency excludes the data transfer to and from the GPU memory in order to isolate processing and I/O performance results to some extent. Speedup graphs are not shown since they only show relative performance which is of limited use for the radar application, where actual throughput and latency are of primary concern.

### A. GPU Implementations

A CFAR window size of $N = 40$ cells was used as a typical value for the CFAR window size for the initial GPU implementation benchmarks. A CFAR window size of around $N = 32$ cells is suggested as being a typical size by [12] and [13]. The value of $N = 40$ was used instead to allow for a symmetrical CFAR window around the CUT, which is required for our implementations. The results for all three techniques on the GPU architecture are shown in figure 4.

*1) Effect of input size:* The sliding window technique performs the worst over the entire range of parameters, which is expected since this technique reduces parallelism in exchange for reduced computational complexity. The SAT technique performs better than the sliding window technique, but is still outperformed by the naïve technique in general for typical CFAR window sizes. The SAT technique has increased complexity and more overhead since it first needs to generate the SAT before it can subsequently benefit from the constant-time lookup.

*2) Effect of CFAR window size:* The basic naïve implementation, which uses global memory (GMEM), degrades markedly as the CFAR window size increases. The degradation be can attributed to the growth of the CFAR window in two dimensions as $N$ increases and the fact that both dimensions are summed simultaneously in a single phase. The overlap of CFAR windows for neighboring cells increases in both dimensions, which introduces significant read contention.

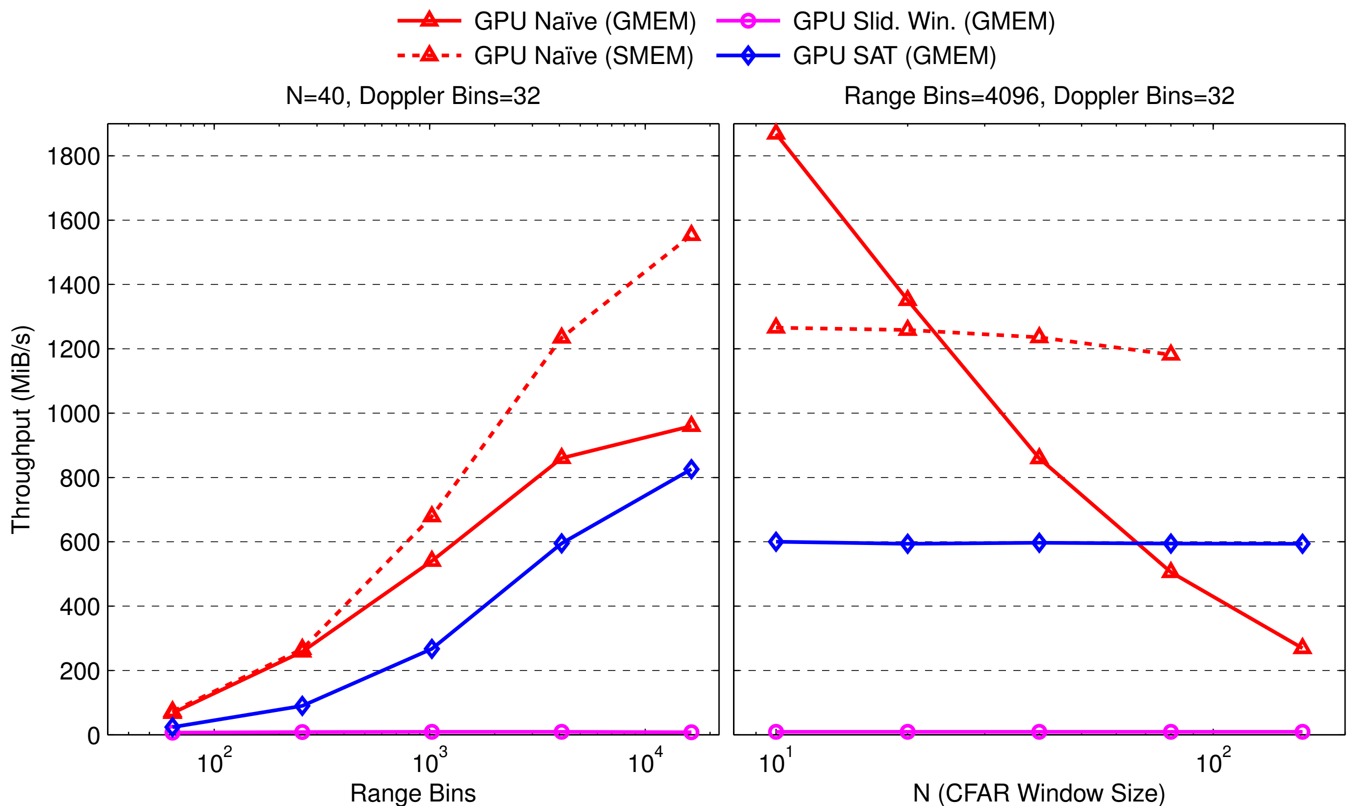


Figure 4. Kernel Performance of GPU Implementations using a 2D CFAR window.

This effect is still present in the shared memory (SMEM) implementation, even though it segments the processing into a rows and columns. However, the effect only manifests itself in a single dimension at a time. For larger $N$ the performance trend for this implementation is better than the naïve implementation which uses global memory. Again, note that this version is limited to a maximum CFAR window size of 33 cells in both range and Doppler dimensions.

The result for the Summed-Area Table implementation shows that it is essentially unaffected by the CFAR window size. The SAT requires a constant time to generate the SAT table according to the input size of the range-Doppler map. The SAT technique can therefore be particularly useful where window sizes become very large.

### B. Performance Comparison with C Implementations

The performance of our GPU and CPU implementations are compared with the HPEC CFAR Benchmark [2] in figure 5. The HPEC CFAR Benchmark only uses a 1D CFAR window, which acts only in the range dimension. Our implementations are optimized for a 2D CFAR window, but can be run in a 1D CFAR window mode by setting the Doppler dimension of the CFAR window to 1. The results were obtained using this configuration. The standard datasets for the HPEC CFAR Benchmark also uses a data cube, which has range, Doppler and beam dimensions. Multiple beams can be obtained by using multiple receiver elements in a radar system. Our implementations are designed for a data matrix which has range and Doppler dimensions only. Datasets were therefore generated for the HPEC CFAR Benchmark where the number of beams are set to 1. The HPEC CFAR Benchmark CPU code was also compiled and run on the target CPU platform.

### C. Data Throughput

For GPU implementations the throughput generally decreased as the input burst size decreases, which can be attributed to some extent to the fixed overheads associated with transferring data between the host to the device. The suboptimal occupancy, in terms of parallel units on the GPU that are utilized, for input burst sizes below a certain threshold also contributes to the trend that is observed for the throughput.

## VII. Conclusion

The results showed that optimized GPU implementations that favour reduced algorithmic complexity and increased parallelism over reduced computational complexity performed the best. The GPU implementation for the naïve technique, which uses shared memory performed better, in terms of kernel throughput and latency, than comparable CPU reference implementations for larger input burst sizes (greater than 8K cells) and typical CFAR window sizes. Even though the SAT technique is outperformed by the optimized naïve techniques for typical CFAR windows sizes, there is a crossover point where the SAT technique will perform better with increasing CFAR window size.

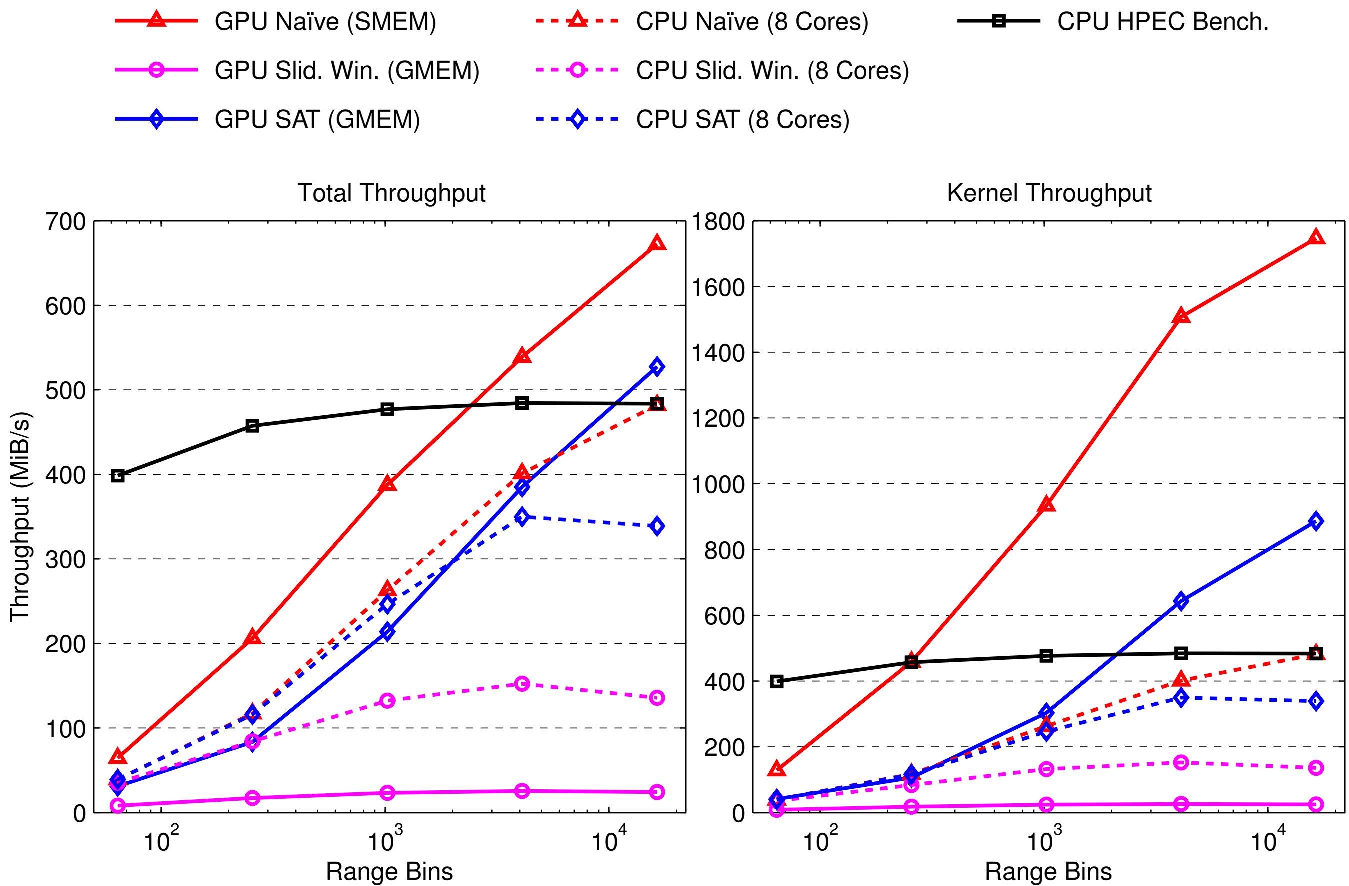


Figure 5. Performance Comparison for N=20, Doppler Bins=32 using a 1D CFAR window.

In terms of the total time required to transfer the input data to the GPU, perform the processing and transfer the output data back to the host, the results show that it is generally not worthwhile to use the GPU architecture for the CA-CFAR algorithm instead of a CPU based approach. The trend does however indicate that throughput increases as input size increases, the implication being that the use of the GPU architecture may still be viable for larger input sizes than investigated here. This conclusion is based on the assumption that a single burst of data is transferred and processed at a time in order to keep overall latency through the system reasonable. In this investigation the CA-CFAR processing was performed in isolation and not as part of a typical radar signal processing pipeline. In such a pipeline one could potentially amortize the cost of transferring the data to and from the GPU.

Further optimization could also be performed on the SAT technique by using shared memory, amongst other, given the improvement seen in case of the naïve techniques. Another aspect of the SAT technique that needs to be investigated further is the potential for loss of precision that can occur as a result of the increasing monotonic function associated with the generation of a SAT. There are techniques for compensating for this effect, such as subtracting an estimated or calculated mean from all cells prior to generating the SAT.

## REFERENCES

[1] M. Richards, *Fundamentals of Radar Signal Processing*. New York, NY: McGraw-Hill, 2005.

[2] HPEC Challenge. (2010) Constant false-alarm rate detection. [Online]. Available: http://www.ll.mit.edu/HPECchallenge/cfar.html

[3] A. Kerr, D. Campbell, and M. Richards, "GPU performance assessment with the HPEC challenge," in *Proc. 12th Annual HPEC Workshop*, Lexington, MA, Sept. 2008.

[4] HPEC Challenge. (2010) HPEC software reference implementation. [Online]. Available: http://www.ll.mit.edu/HPECchallenge/software.html

[5] F. C. Crow, "Summed-area tables for texture mapping," in *SIGGRAPH '84 Proc. 11th Annual Conf. on Computer Graphics and Interactive Techniques*, New York, NY, USA, July 1984, pp. 207–212.

[6] P. Viola and M. Jones, "Robust real-time object detection," in *Proc. 2nd Intl. Workshop on Statistical and Computational Theories of Vision*, Vancouver, Canada, July 2001.

[7] The MathWorks, Inc. MATLAB®. [Online]. Available: http://www.mathworks.com

[8] NVIDIA Corporation. (2010) Tesla™. [Online]. Available: http://www.nvidia.com/object/tesla_computing_solutions.html

[9] OpenMP. (2010) The OpenMP API specification for parallel programming. [Online]. Available: http://openmp.org/wp

[10] NVIDIA Corporation. (2010) CUDA zone. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[11] CUDPP. (2010) CUDA data parallel primitives library. [Online]. Available: http://code.google.com/p/cudpp

[12] M. Skolnik, *Introduction to Radar Systems*, 2nd ed. Singapore: McGraw-Hill, 1981.

[13] ——, *Radar Handbook*, 3rd ed. New York, NY: McGraw-Hill, 2008.