Real-Time Implementations of Ordered-Statistic CFAR

M.R. Bales, T. Benson, R. Dickerson,
D. Campbell, and R. Hersey

Sensors and Electromagnetic Applications Laboratory
Georgia Tech Research Institute
Atlanta, GA, USA

E. Culpepper

Sensors Directorate
Air Force Research Laboratory
Wright-Patterson AFB, OH, USA

Abstract—Ordered-statistic constant false alarm rate (OS-CFAR) detectors provide improved robustness over cell-averaging CFAR (CA-CFAR) detectors in multiple target and heterogeneous clutter environments. However, this benefit comes at the cost of generally increased processing time due to the need for a rank-ordering of the CFAR training data. Real-time implementations of OS-CFAR must consider this additional processing burden.

In this paper, we present real-time FPGA and CPU/GPU implementations of OS-CFAR. A novel sorting architecture that scales linearly with window size is presented alongside traditional compare-and-swap and rank-only architectures in an FPGA. A rank-only GPU implementation is demonstrated alongside multi-threaded sorting and rank-only CPU implementations. Effects of training window size on throughput and power consumption are considered.

I. INTRODUCTION

Radar systems commonly employ constant false alarm rate (CFAR) detectors to estimate the background interference level and set the detector threshold to yield the desired false alarm rate. CFAR techniques estimate the background interference level by training over adjacent range and Doppler bins.

CA-CFAR is the most direct training implementation, where the estimated interference level is set to the arithmetic mean of the training samples. While CA-CFAR produces the maximum likelihood estimate [1], it requires homogeneous training data with statistics similar to the cell under test. Heterogeneous training data, which can result from clutter variations and closely spaced targets, can result in target masking and false alarms [1].

As an alternative to CA-CFAR, OS-CFAR rank orders the training data and chooses a pre-determined, rank-order position for the estimated interference level. As a result of this rank order process, the estimated interference level is affected less by targets and clutter edges in the training window. As a result, OS-CFAR has been shown to be more robust in multiple target and heterogeneous clutter environments [2-3].

Real-time implementations of OS-CFAR must consider the additional processing burden of rank ordering the training data. For each CFAR estimate, the CA-CFAR implementation requires $N-1$ additions and one multiplication, where N is the length of the CFAR training window. The brute force OS-CFAR implementation, on the other hand, requires on the order of N^2 computations because of the sorting requirement. This additional computational requirement becomes significant considering that a typical radar system with a sliding-window CFAR implementation calculates the estimated interference level on the order of one million times for each coherent processing interval.

II. FPGA IMPLEMENTATIONS

Several techniques for OS-CFAR calculation have been considered for FPGA implementation. A common related application is median filtering for noise removal in image processing, which usually entails small two-dimensional windows and assumes 8-bit integer representations. These constraints are exploited to allow fast comparison networks [4-5] or histogram implementations [6]. Radar OS-CFAR applications require longer window lengths and greater dynamic range (such as single-precision floating point numbers), and thus cannot take advantage of these approaches. FPGA implementations have been demonstrated that avoid sorting by successively finding and zeroing out the window's maximum value until the desired rank is found [7], and by exploiting the persistence of the window data and recursively subdividing the window until converging on the desired rank [8]. Our goal is to compute ordered statistics of a series of single-precision floating-point numbers with X training bins and Y guard bins. The total filter window length is therefore $X+Y+1$, and is applied as a sliding window over the sample data. For several filter lengths, we compare traditional brute force sorting, a proposed new sorting approach, and an architecture that performs OS-CFAR without sorting. We use four guard bins ($Y=4$) for each experiment.

A. Brute Force

Traditional brute-force computation of ordered statistics for a set of numbers is performed by compare-and-swap networks, in which pairs of numbers are compared and their positions in the array swapped as necessary to obtain the

proper order. In an FPGA, pairs of numbers can be compared simultaneously. Sorting N numbers requires a maximum of $X-1$ sorting stages, with each stage consisting of $X/2$ or $(X/2)-1$ swap modules. In such a configuration, the FPGA implementation can be fully pipelined to produce a result every clock cycle. However, as the length of the sorting window increases, the amount of logic fabric consumed by the design grows on the order of X^2 , which can quickly become prohibitively large. Guard bins are easily accommodated by leaving guard bins of the shift register unconnected from the swapping network.

In applications where the FPGA maximum operating frequency exceeds the required result frequency by at least an even multiple Z , the repetitive structure of brute force sorting can be exploited to produce a design approximately $1/Z$ the size of the original. The pipeline can be time-multiplexed, sending a set of inputs through the processor and reprocessing the output Z times. For our experiments, $Z=4$. Figure 1 shows an example of such a network with 8 training bins to be sorted. Normally 8 bins would require 7 compare-and-swap stages, but by running the network at four times the needed solution rate, we can accept a new vector as input on the first clock, and use the following three clocks to process previous vectors at intermediate stages of completion.

B. Time-To-Live

The two-dimensional structure of the brute force approach quickly consumes fabric space, and comprises a great deal of redundancy since a vector of X inputs must be copied $X-1$ times as the data propagate through the swapping network. In the proposed technique (called Time-To-Live, or TTL), we sort values in place at the expense of a few clock cycles, thereby requiring only X nodes to sort X values. Since inputs are not kept in their original order, it is necessary to track the number of sort cycles in which each input has participated. Each input has an associated time to live that is initialized to the length of the sort window.

An array of $X+Y+1$ nodes is constructed. Figure 2 shows the node structure. Each node contains data and TTL registers, and shares these with its immediate neighbors along with its desire to move in either direction. A master controller dictates the current state; the state machine is shown in Figure 3.

At the beginning of a sort cycle, nodes decrement their TTL values to make room for a new input. Every sort cycle, exactly one node's TTL reaches zero and its data value is effectively shifted out of the window. The node announces its death to a one-hot encoder, which broadcasts location of the death to the rest of the chain. As new data is shifted into the filter window, each node compares its data value with the new data, considers the location of the dead node, and shifts its data and TTL values up or down to a neighboring node as appropriate to make room for the new data. New data is thus inserted directly into its proper ordered position in the array.

Figure 1. Example of swapping network with extra register layers for time-multiplexed brute force method.

Figure 2. A single node used in TTL sorting.

Figure 3. TTL State Machine.

Figure 4. Example of rank-only network.

A small, fixed number of guard bins can be accommodated in the TTL technique by polling the nodes for TTL values that correspond to guard bin positions. Each node makes a backup of its data and TTL values. Then the nodes are polled for each guard bin TTL value. All nodes in the chain below the guard node shift up simultaneously, causing that guard node to drop out of the list. After each guard bin has been polled and removed, the upper N nodes contain the desired sorted values. The nodes then restore their original data and TTL values so the next input may be shifted in. One poll-and-shift cycle is needed for each guard bin; this would be prohibitive for many guard bins, but handles small numbers of guard bins effectively.

The TTL method takes multiple clock cycles to create a solution, but it scales on the order of X .

C. Rank-Only

OS-CFAR can be calculated without performing a sort of the sample data. While the median or other ordered statistic of a set of numbers is not explicitly computed, this method determines whether a bin under test is above or below a designated position in the array. As shown in Figure 4, each training bin value is compared with a scaled bin under test. The number of bins exceeding the scaled bin under test is counted, and this number is compared with a threshold. This technique can be pipelined to produce one result per clock, and scales linearly with the number of training bins. It does not sort the data, so it is not sufficient when sorted values are needed (e.g., to calculate SNR).

D. Comparison of Methods

The designs discussed are implemented using Xilinx ISE 12.2 tools, targeting a Virtex 6 LX550T part. Implementation strategies are set to the default balance of speed/space optimization. Cores for floating point comparison operations

Figure 5. FPGA Slice Utilization of three OS filter implementations for filter lengths of 8, 16, 32, 64, 128, and 256.

Figure 6. Throughput of three OS filter implementations for filter lengths of 8, 16, 32, 64, 128, and 256.

are built using Xilinx COREGenerator. Each technique is implemented for filter lengths of 8, 16, 32, 64, 128, and 256. Four guard bins are used in all cases. Even with time multiplexing, the brute force filter causes the place and route tools to fail during the routing stage for filter lengths greater than 64.

Figure 5 shows how FPGA slice usage increases with filter length for each technique. Due to the overhead of the node and controller structures of the TTL technique, brute force computation uses fewer FPGA resources for window lengths smaller than 16. Rank-only and TTL both scale on the order of X with window length, but rank-only does not produce sorted data. Throughput is plotted in Figure 6 for the three methods. Throughput declines with increasing window length; as the sorting networks consume more space, the FPGA's maximum obtainable clock rate decreases. The brute force and rank-only techniques each produce one solution per clock. When accommodating guard bins, the TTL structure requires $7+Y$ clock cycles to complete one sort, reducing its throughput to approximately 10% of that of brute force and rank-only. Only 3 clocks per TTL sort are needed when guard bins are not required, yielding a latency of about 15 ns per solution. The TTL technique provides a fully sorted, compact option that can be easily placed in the logic fabric among other processing elements.

Power consumption estimates are generated using the Xilinx XPower Analyzer for each FPGA method and window size. In Figure 7, we separate power consumption into

contributions due to clock switching (solid lines) and logic and

Figure 7. Power consumption as a function of training window size, separated into clock and logic+signal power contributions.

signal switching (dashed lines). Each TTL node carries out several comparisons, condition checks, and communications with neighboring nodes and the controller, resulting in the highest logic power component. The brute force method consists of relatively simple compare-and-swap modules with limited fanout, yielding modest logic power use despite its X^2 area cost. However, the area cost of the brute force method results in significantly higher clock net fanout—and therefore a larger clock power component—than the other methods, causing brute force to consume the most power overall. Clock power contributions dominate for brute force and rank-only methods as window size increases. The TTL method is the only method that consumes more power due to logic switching than to clock switching.

III. CPU AND GPU IMPLEMENTATIONS

We also consider OS-CFAR implementations on general-purpose CPUs and graphics processing units (GPUs). Many of the issues discussed for the FPGA implementations also apply to the CPU and GPU implementations, but there are unique characteristics to consider for each computing architecture. For the CPU implementation, we must consider the impacts of multi-core parallelism and the CPU memory hierarchy. For the GPU implementation, we must additionally consider much higher levels of parallelism and the potential for reducing global memory access frequency via utilization of shared memory. The level of programmability presented by CPUs and GPUs provides significant advantages in terms of flexibility and development time, albeit often with larger size, weight, and power requirements than an FPGA implementation. However, the increased programmability often offers certain optimizations that may otherwise be difficult to exploit. For example, in the case of rank-only OS-CFAR, we can selectively sort only the training bin sets corresponding to detections in order to compute the necessary ordered statistics without sorting the training data for the non-detections.

A. CPU OS-CFAR With Sorting

We now consider the family of OS-CFAR CPU implementations that involve sorting. While there are many

candidate sorting routines available, we here evaluate insertion sort and the standard sorting routine (`std::sort`) as included in the C++ standard template library (STL). The former sorting algorithm exhibits n^2 average and worst case run-time complexity, while the latter is a modified merge sort with $n \log n$ average and worst case complexity. However, given small training sets, it is not clear that the latter sorting routine will outperform the former.

Our implementations are parallelized via OpenMP in order to exploit the multiple cores offered by modern CPUs. We distribute the work such that one thread processes all fast-time samples for one or more Doppler-angle pairs. Furthermore, we employ dynamic scheduling within the OpenMP thread group to evenly distribute work. As we progress to each new cell under test, we copy the requisite training samples into a working array, sort the array, and extract the necessary ordered statistic from the sorted array. Given small training sets, this working array will easily fit within low-level caches during sorting on most CPUs. There is typically significant overlap in training data for adjacent cells and thus we could modify the insertion sort approach to populate the workspace array for the next cell under test with the already sorted entries in the intersection of the current and next training sets. Given that insertion sorting performs well on substantially sorted arrays, this approach may increase performance, albeit with additional bookkeeping. Furthermore, algorithms exist that offer linear performance in the worst case for selection of a single ordered statistic [9]. Such linear time selection algorithms could be used to generate the ordered statistic required per cell under test. However, we do not explore these options further because we instead later consider a linear time rank-only algorithm similar to that presented in Section II.C. While both the rank-only and linear time select algorithms offer worst case linear complexity, the latter approach involves a calculation of the median-of-medians of a partitioned set and thus likely exhibits a substantially higher complexity constant than that associated with the direct rank-only approach.

B. CPU Rank-Only OS-CFAR

The CPU implementation of the method described in Section V.C can operate directly on the original data and thus does not require copying training bins to a workspace array. We also parallelize the rank-only CPU OS-CFAR implementation via OpenMP with a distribution strategy similar to that presented in Section III.A.

As before, this approach does not require the sorting operations, but also does not generate the ordered statistics that may be necessary for other purposes, including calculation of the SNR associated with a detection. However, with the CPU and GPU implementations, we can subsequently sort only those training sets corresponding to detections. While the run-time will then depend somewhat on the number of detections, the constant false alarm rate aspect of OS-CFAR should keep that number manageable provided that the number of true positives remains reasonable. Therefore, the set of cells under test on which

sorting is applied to the respective training samples is orders of magnitude smaller than the full sorting-based approaches.

C. GPU Rank-Only OS-CFAR

For the GPU implementation, we consider only the rank-only version of OS-CFAR. This is done for simplicity and because the CPU results indicate that the rank-only version exhibits significant performance advantages relative to the sorting-based approaches. However, high-performance sorting routines for the GPU that exploit the massive parallelism via parallelized merging operations have been developed [10]. In the case that we wish to generate the ordered statistics corresponding to a given cell under test, we could either employ a simple quadratic sorting approach or a merging based approach as described in the above reference.

We here consider the Fermi generation of GPUs offered by NVIDIA. These GPUs are characterized by up to 16 streaming multiprocessors running many active threads and high global memory bandwidth, especially in the case of coalesced data reads. Furthermore, each streaming multiprocessor (SM) contains 64KB of high-speed memory of which either 16KB or 48KB can be utilized for a user managed cache, with the remainder being used as a device-managed L1 cache.

In order to achieve near optimal performance on the GPU, we must expose high degrees of parallelism, minimize global memory access (e.g., by instead using shared memory), coalesce global memory accesses when possible, and minimize thread divergence, among other goals. We here consider several of these factors for the rank-only OS-CFAR GPU implementation.

For the GPU workload distribution, we launch one thread block per collection of fast-time range bin samples. Thus, for example, a data set with N Doppler bins, M angles, and L range bins would be split into $N*M$ thread blocks. The T threads within a thread block handle T contiguous cells under test and this window of width T progresses in fast-time in a sliding-window fashion until all L range bins have been processed for this block.

Because the memory accesses for the training bins of the T cells under test will include a great deal of overlap, we first load all of the necessary values for the T test cells into shared memory and subsequently all threads within the block will access the training bins using the shared memory copy. This utilization of shared memory in turn minimizes global memory bandwidth consumption.

D. Comparison of Methods

The CPU implementations were run on a server with dual socket hex-core 2.8GHz Xeon 5660 processors with 12MB L3 cache and 24GB DDR3-1333 RAM. The software was compiled with gcc 4.5.1 using optimization level -O3. The GPU versions were implemented using CUDA 3.2 and run on a single Tesla C2050 GPU.

CPU scalability as a function of thread count is shown in Figure 8 for the case of thirty-two training bins per cell under test. Execution times are taken to be the average of ten consecutive runs for both the CPU and GPU experiments.

The Xeon servers have twelve physical cores, and also support hyper-threading. The nearly linear speedup plateaus

Figure 8. Cell test throughput for OS-CFAR as a function of thread count. The host machine included twelve physical cores with hyperthreading.

Figure 9. Cell test throughput for CPU and GPU based CFAR implementations as a function of training configuration.

at twelve threads with only marginal increases, and some significant decreases, for higher thread counts. Lack of further benefit from hyperthreading implies that our implementation performs most memory access from lowest level cache. Furthermore, as expected, the rank-only version of OS-CFAR significantly outperforms both sorting versions.

Figure 9 presents the cell test throughput as a function of training configuration for each of the CPU-based implementations and the GPU-based implementation running on a C2050. In this case, both the GPU and non-sorting CPU implementations offer very high throughput. For a window of sixty-four training bins, the GPU implementation processes over 630 million cell tests per second whereas the CPU implementation processes approximately 180 million cell tests per second. When computing the GPU execution time, we did not include the data transfer from the host to the device over the PCI Express bus. Therefore, the GPU OS-CFAR implementation is best applied when upstream processing is also performed on the GPU and thus no transfer from host to device is required.

Figure 10 compares CPU, GPU, and FPGA implementations in terms of millions of solutions per second

produced per watt expended. Numbers for FPGA power consumption are produced using the Xilinx XPower Analyzer tool for post place-and-route designs. Power consumption for

Figure 10. Comparison of power efficiency (in millions of solutions per watt expended) for CPU, GPU, and FPGA implementations as a function of training window size.

CPU/GPU platforms are estimated from each chip's thermal design power specification: 190 W total for the CPU, and 238 W for the GPU. The CPU and GPU implementations are shown in solid lines, while the FPGA implementations are shown in dashed lines. The fully-sorting TTL method demonstrates efficiency similar to that of the GPU rank-only approach, while the FPGA brute force and rank-only implementations produce one to four orders of magnitude more solutions per watt than other methods. For this range of window sizes, the power efficiencies of the FPGA methods are less sensitive to window size than the fully sorting CPU methods.

IV. SUMMARY

Ordered-Statistic CFAR is a commonly used estimator in radar systems due to its improved robustness over Cell-Averaging CFAR. However, the sorting process incurs additional computational expense. We present real-time FPGA and CPU/GPU implementations of OS-CFAR for several training window sizes. Three FPGA architectures are analyzed in terms of logic fabric utilization and throughput. Rank-only uses the fewest resources, but it does not generate sorted data that are needed for some applications. Brute-force sorting uses the most resource but runs very quickly and produces sorted data. TTL is introduced as a node-oriented technique that reduces logic consumption by sorting inputs in place. TTL does not generate as many solutions per second as the others, but scales much better than brute force (X vs. X^2) and produces sorted data, unlike rank-only. Brute force sorting consumes the most power overall due to high clock fanout. The linear area scaling and complex logic operations of the TTL method consume more power due to logic switching than to clock switching.

Three CPU implementations and one GPU implementation demonstrate OS-CFAR performance scaling on multi-core and many-core platforms. Both the CPU and GPU memory access patterns are such that accesses of the training window data effectively exploit low level caches, thus enabling scaling to many cores. The $n \log n$ complexity of the CPU merge sort begins to show improvement over the n^2 complexity of insertion sort once the number of training bins exceeds 16. The rank-only CPU and GPU implementations scale uniformly with increases in training window size, and achieve throughputs an order of magnitude higher than the sorting CPU versions.

While the rank-only methods do not generate sorted lists of the training samples for later use, the programmable nature of the CPU and GPU implementations enable sorting after a detection has been made in cases where training window ordered statistics for detections are desirable. Given the small number of relative detections typical when employing OS-CFAR, such a selective sorting approach will only minimally impact throughput.

From an energy efficiency perspective, the FPGA implementations offer higher throughput per Watt and maintain more constant energy profiles for larger training window sizes. The rank-only implementation offers the most energy efficient implementation for the CPU and GPU with the GPU offering comparable energy efficiency to the FPGA TTL implementation with up to approximately 64 training window bins.

REFERENCES

- [1] M. Richards, Fundamentals of Radar Signal Processing, McGraw-Hill: New York, 2005, pp. 347-374.
- [2] M. Weis, "Analysis of some modified cell-averaging CFAR processors in multiple target situations," IEEE Transactions on Aerospace and Electronic Systems, vol. 1, pp. 102-113, January 1982.
- [3] M. Weis, "Radar CFAR thresholding in clutter and multiple target situations," IEEE Transactions on Aerospace and Electronic Systems, vol. 4, pp. 608-621, July 1983.
- [4] P. Wei, L. Zhang, C. Ma, and T.S. Yeo. "Fast median filtering algorithm based on FPGA," IEEE Proc. Int'l Conf. on Signal Processing, pp. 426-429, 2010.
- [5] Z. Vasicek and L. Sekanina. "Novel hardware implementation of adaptive median filters," IEEE Proc. Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp. 110-115, 2008.
- [6] S.A. Fahmy, P.Y.K. Cheung, and W. Luk. "Novel FPGA-based implementation of median and weighted median filters for image processing," Proc. Int'l Conf. on Field Programmable Logic and Applications, pp. 142-147, 2005.
- [7] B. Magaz and M.L. Bencheikh. "An efficient FPGA implementation of the OS-CFAR processor," IEEE Proc. International Radar Symposium, pp. 1-4, 2008.
- [8] Z. Ali, A. Arshad, U. Razzaq, S. Sana, A.H. Ahmed, and A.M. Harris. "Design and implementation of an OS-CFAR processor based on a new rank order filtering algorithm," Proc. Int'l Symposium on System-on-Chip, pp. 158-162, 2010.
- [9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms, 2nd ed., MIT Press: Cambridge, Mass, 2001, pp. 183-195.
- [10] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," Proc. IEEE International Symposium on Parallel and Distributed Processing, pp. 1-10, 2009.