Minesweeper: A Novel and Fast Ordered-Statistic CFAR Algorithm

Carl L. Colena, Michael J. Russell, Stephen A. Braun
Spectrum Systems Laboratory
Lockheed Martin Advanced Technology Laboratories
Cherry Hill, New Jersey, USA
{carl.l.colena,michael.j1.russell,stephen.a.braun}@lmco.com

***Abstract*— A novel algorithm named '*Minesweeper*' was developed for computing the Ordered Statistic Constant False Alarm Rate (CFAR) in a computationally efficient and novel way. OS-CFAR processing chains are used in radar applications for noise-floor estimation and target discrimination. Unlike other approaches, this algorithm aims to minimize data reuse by using training cell geometry and an accumulation matrix to compute the noise estimate. Computing the OS-CFAR in this manner affords some unique efficiencies that are novel for this application. This includes runtime invariance by bit-depth of the input data and by training geometry. Three implementations of *Minesweeper* were developed and benchmarked. The Optimized GPU Implementation (GPU-OPT) performed the best in both throughput and latency for large inputs. This algorithm has potential for use in real-time GPU-accelerated SDR applications.**

Keywords—Radar, OS-CFAR, GPU, Algorithm, Filter

I. Introduction

In radar applications, CFAR (Constant False Alarm Rate) detectors serve an important role. They estimate the noise environment for computing SNR and/or target discrimination. [1]. There are various techniques to perform this type of detection, including Cell-Averaging CFARs (CA-CFAR). In applications that require multi-target or high clutter performance, the Ordered-Statistic CFAR (OS-CFAR) is the most robust for this task [2].

Compared to other detectors, the OS-CFAR detector is the most computationally intensive, and therefore often prohibitive to implement in real-time and embedded radar platforms. However in applications where robust multi-target and clutter performance is required, such as processing 2D Range-Doppler Maps (RDM), there is a need for a performant OS-CFAR detection algorithm. Here a novel algorithm called '*Minesweeper*' is presented as a viable candidate for performing OS-CFAR detection in real-time.

II. Minesweeper Algorithm

A. Background

Computing the OS-CFAR requires an intensive step that traverses through the input matrix and sorts the training bins around the Cell Under Test (CUT). For large input sizes and many training bins, this becomes computationally prohibitive to perform in real-time. One approach by Bales *et al.* addresses this by counting the number of cells that were above or below the scaled CUT [3]. While this is a fast approach, one drawback of this method is that the noise estimate cannot computed. This is because the algorithm only computes the rank selection, as opposed to the ordered-statistic noise estimate for the CUT, so SNR cannot be calculated.

Another approach employs the use of image-processing techniques to quickly compute the OS-CFAR. This approach by Villa *et al.* [4] leverages a median filtering method by [5] to compute the OS-CFAR in constant time with respect to window size. This is particularity useful in 2D OS-CFAR applications such as SAR and RDM processing. However a drawback of their approach is that the algorithm is sensitive to bit depth, with space complexity scaling at $O(2^b)$, where b = bit depth. This restricts the general applicability of this approach to low bit-depth inputs such as 8-bit and 16-bit. Floating points and non-primitives are inapplicable for this histogram based approach.

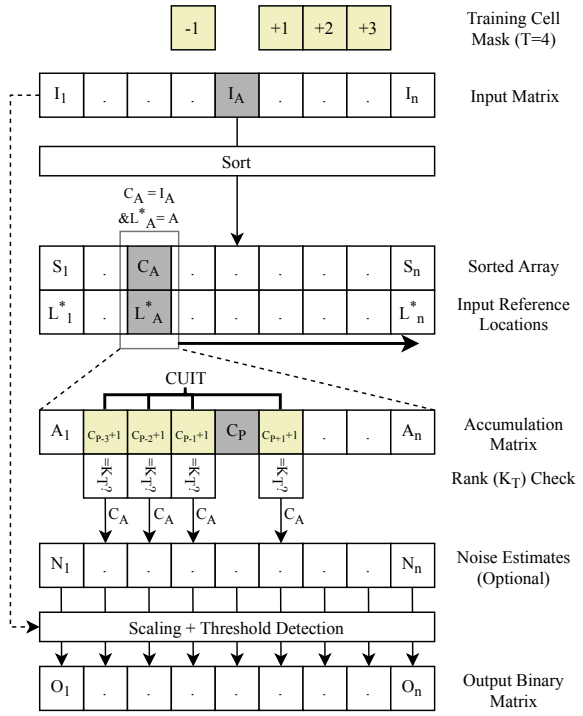

Fig. 1 Diagram of Minesweeper Algorithm performing OS-CFAR processing with an asymmetric Training Cell Mask with 4 training cells.

To address large input sizes, the ability to derive SNR, and scale well with variable bit-depth while still performing close to constant against window sizes, the algorithm named *Minesweeper* was created.

Conceptually *Minesweeper* is similar to another OS-CFAR detector algorithm by Hyun & Lee in 2011 [6]. Both algorithms employ a general sorting pre-processing step, as well as a sliding register that traverses the sorted matrix as seen in the top half of Fig. 1. However their algorithm computes the noise estimate values with a cost (excluding sort) of $O(N\log^e T)$. The $\log^e T$ cost comes from treating each cell in the sliding window as the Cell Under Test (CUT) and performing a tree search for each CUT. *Minesweeper* approaches reference cell selection in a different way, which when parallelized, reduces this cost to $O(T/t)$. For sufficiently large t (such as on GPUs and many-core processors), this can be can be practically treated as close to constant scaling. This process is described in the next section.

B. Noise Estimation Method

The process is illustrated in the bottom half of Fig. 1 and starts with the Sorted Array (SA) of values, the associated reference locations of each cell back to the Input Matrix (IM), and the training cell mask M, comprising of T cells. A Ranked Comparator value, K_T, is set based on the K^{th} ranked element of T cells. For example, if given a training mask of 3x3 (minus CUT) and a rank of 75th percentile, then $T = 8$, and $K = 6$. The value of K_T in this case will be 2 (or $T - K$) instead of 6. This is explained in the next section.

This process iterates across the SA in order, one cell at a time. The iteration order depends on whether $T - K < K$. If true, then descending order is used, and $K_T = T - K$. Else $K_T = K$ and ascending order is used. By doing this, the upper bound of the Rank Check count will be $T/2$ for any given OS-CFAR. By extension, if $K_T = T/2$, then either ascending or descending order may be chosen, as the distance is the same.

Each iterated cell is treated as the current Anchor Cell (C_A). The Anchor Cell represents a noise estimate candidate for T neighboring cells. The geometric location of C_A is used to derive the Pivot Cell (C_P) location in the Accumulation Matrix (AM). In other words, the original geometric location of C_A in the IM should be the same geometric location of the C_P in AM. Here AM serves as a proxy for IM that tracks state in the form of incrementing counters.

For each C_A, the location reference is obtained, and from this location (C_P), the negated form of the mask ($-M$) is applied to T cell locations. Each of these T cell locations refers to a Cell Under Incremental Test (CUIT). These cells are candidates that potentially have C_A as its noise estimate. This is determined by the geometrical observation that for a given neighborhood M centered around a cell C, if all cells in this space have the same geometric neighborhood M centered around each cell, then the set of cells that contain C as a member of their own M are the cells that belong to the $-M$ neighborhood of cell C.

Put formally, if given C_A with position (x_a, y_a), and a C_{CUIT} with position (x_b, y_b), then we consider that C_{CUIT} contains C_A as a member of the set of training cells for C_{CUIT} *iif* there exists a member in M (x_m, y_m) such that $(x_a - x_b, y_a - y_b) - (-x_m, -y_m) = (0, 0)$. Therefore, we can find C_{CUIT} when given C_A and M using $(x_a + -x_m, y_a + -y_m) = (x_b, y_b)$. This is demonstrated in Fig. 1 with an asymmetric mask of T cells. For symmetric masks, $M = -M$.

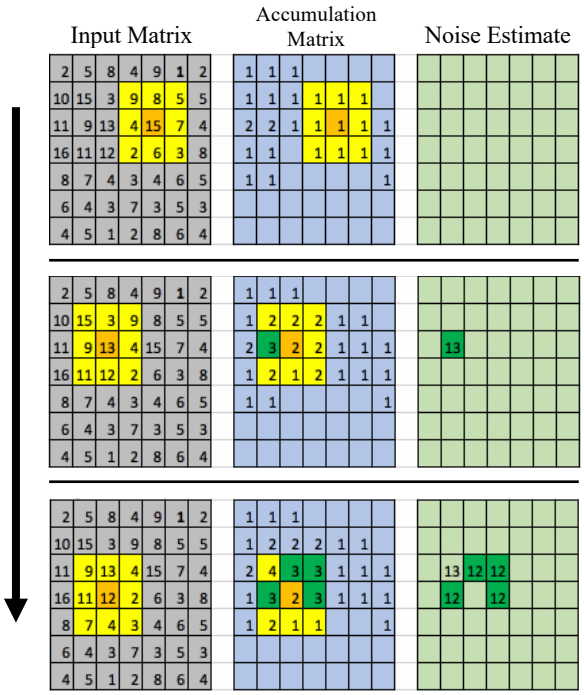

Fig. 2 Graphical representation of 3 consecutive steps of *Minesweeper's* noise estimation operating on a 2D IM. Here a 3x3 kernel (T=9) is used (no guard cell for CUT) on a 7x7 IM. $K_T = 3$ with iteration order descending. Noise estimates are saved when the K_T check passes (CUIT = K_T).

From here, the negated mask $-M$ is applied onto the AM at the anchor point. The T cells selected in the AM represent the CUITs who have C_A as a member of their M with respect to the IM. The CUIT are incremented by 1, and the count saved to the AM. If, after incrementing, one or more cells in the CUITs have a value equal to K_T, then C_A is selected as the noise estimate for those cells. This process continues until the SA is fully traversed, or if the count of the number of cells that passed the K_T check equals the size of the IM. Keeping track of the count that passes the K_T check is a greedy optimization that we can make because cells in the AM can only pass the K_T check once.

This process is illustrated in Fig. 2 operating over a 2D matrix. The orange cell represents the C_A selected for the given iteration, as well as it's corresponding C_P in AM. Notice how in the third iteration that 4 CUIT's pass the K_T check at once, resulting in the noise estimate value 12 being set for all 4 CUIT's. This pattern is where the name *Minesweeper* comes from, as the way this algorithm searches for noise estimates is visually analogous to the classic computer game *Minesweeper*. This also highlights a key trait of this algorithm, where a single iteration could yield the result of multiple CUITs at once. By combining this trait with the greedy optimization of counting K_T checks, the runtime of this algorithm can be considered to be at worst $O(N)$ with the expected number of iterations required $o(N)$. Analysis of this is in Section V Results.

The correctness of this noise estimate method relies on traversing the SA in-order. If the SA is not traversed in order, then the noise estimate output is not guaranteed to be correct. This is because the counts of each cell in the AM are valid only

as long as all previous cells in the SA have already been processed. The SA ensures that the global maxima/minima are captured to ensure the correctness of the ordered-statistic result. Thereby, if the incremented CUIT = K_T, then the current C_A is guaranteed to be the ordered statistic noise estimate.

III. Algorithm Analysis

A. Complexity Analysis

1) Storage Complexity: The space complexity of *Minesweeper* as described in Fig. 1 is estimated to be *4N+T*, where *N* is the IM size, and *T* is the mask size (number of training cells). Each *N* in this estimate represents the IM, AM, SA, and Output Binary Matrix. If SNR is desired to be kept, an additional *N* of space will need to be allocated. This is a conservative estimate that is larger than most OS CFAR implementations, but this has potential to be smaller through some design optimizations discussed in Section VI Extensions for Improvement, such as streaming sort. One can also reduce the estimate if a lower bit depth than IM's bit depth can be assumed for the AM and Output Binary Matrix. This is certainly possible for the AM if *T* can fit in a smaller bit depth than IM's bit depth. For example, if $T = 15^2$, then AM can be represented as an 8-bit matrix ($15^2 <= 2^8$-1).

2) Time Complexity: The time complexity for a purely sequential implementation of *Minesweeper* is

$$O(sort(N)) + O(NT)$$

Sequentially each increment of a CUIT counts as an operation, as opposed to all CUIT's at once being incremented. If parallelism is introduced, the runtime cost reduces significantly.

The time complexity for a parallelized implementation of *Minesweeper* across *T* is

$$O\big(sort(N)\big) + O\left(N\frac{T}{t}\right)$$

where *t* represents the number of concurrent threads available to perform incrementation and testing across *T* cells. On a GPU, *t* can be quite large, which can make the runtime asymptotically close to constant against *T*. For CPU's, SIMD vectorized instruction sets such as AVX and NEON can contribute to *t* at the thread-level, and multi-threading can also contribute to *t* at the processor level.

One can additionally parallelize across *N*. This could be done by partitioning the dataset *N* into *p* independent partitions. This would enable multiple independent pipelines to divide and conquer across the dataset. The estimated time complexity becomes

$$O\big(sort(N)\big) + O\left(\frac{N}{p}\frac{T}{t}\right)$$

There may be a hidden cost to this approach however, since depending on implementation, overlaps between independent partitions may incur costs from data reuse. This is because overlapping regions between partitions inevitably share the same set of cells, and if the partitions are to be processed independently, each partition must have its own copy of these shared cells. This cost is a function of *T* and the geometric distribution of *M.* Therefore performance will be a balance between the throughput gains from parallelism and the latency cost from the added computational load.

The expected runtime will fluctuate if the count of K_T checks is used to exit the noise estimation process early. One variable that can affect this is value of K_T with respect to *T*. The likelihood that all results are computed before traversing the whole SA will depend on the value of K_T. As a general rule, the strict upper bound on counts required to match a CUIT to K_T is *T*/2. This means that the worst-case runtime performance is expected when $K_T = T/2$. This is a special case of ordered statistic filters, and is referred to as a median filter.

Another variable that affects this is the geometric distribution of values in the input matrix. The algorithm performs it's best when the incrementation occurs across multiple CUIT that haven't yielded an estimate yet. As the process continues, more CUIT will yield results. For each iteration, the ratio of CUIT that are incremented that have already yielded results will decrease the amount of useful work done per iteration. This in turn increases the expected number of iterations to complete towards *N*.

Lastly, it is noted that all runtime estimates contain *O*(*sort(N)*) as an additive cost to the algorithm, where *sort(N)* is the cost of the given sorting algorithm used. While this may seem problematic, the significance of this is reviewed in Section V Results, and a method for potentially amortizing the cost of sorting is described in Section VI Extensions for Improvement.

B. Bit-depth Invariance

One of the useful traits of this algorithm is its time-complexity invariance against the bit-depth of the IM. The number of operations to operate over 8-bit data is equal to that of operating over 64-bit doubles. This is because the algorithm operates based on incidence counts as opposed to the values in themselves. This algorithm also scales linearly in storage complexity with respect to bit depth, which is in contrast with the $O(2^b)$ space complexity by [4]. This enables OS-CFAR processing on inputs that traditionally would be otherwise unfit for real-time operation, such as double-precision floating point and non-primitive data types.

C. Geometry Invariance

Another benefit of this approach is the invariance of training cell geometry. For *Minesweeper*, the geometric distribution of *M* does not generally affect the number of operations required to perform the ordered statistic noise estimation. The one exception to this is an increased data reuse cost when independently parallelizing across *N*. Geometrical invariance is generally not discussed in the existing literature for CFAR applications. Therefore we claim that *Minesweeper's* property of geometric invariance is novel for this application.

A practical benefit of geometry invariance is that the algorithm does not require special customization for non-standard kernels or masks. For algorithms like [4], since the histograms operate over contiguous columns of data, if one was to introduce variable geometry, the implementation becomes

more complex, and the algorithm becomes tailored for specific geometric applications. Since *Minesweeper* is geometrically agnostic, this provides opportunity for applications that can benefit from the flexibility of training bin geometry, such as fine-tuning a CFAR detector's training bins to the target waveform's ambiguity function for a much closer fit.

IV. Experimental Setup

To analyze the correctness and runtime performance of this algorithm, three implementations of this algorithm were developed and tested against a MATLAB implementation. These implementations were developed to operate over 2D datasets (e.g. RDM).

A. CPU-MS

A serial CPU-based implementation of *Minesweeper* was written in C++ which will be referred to as 'CPU-MS'. This implements the *Minesweeper* design described in the *Complexity Analysis* earlier. CPU-MS serves as a baseline serial implementation against the parallelized implementations we developed.

For the sorting algorithm, The *sort_by_key* function of the C++ Accelerator Library *Thrust* was used [7]. For this implementation, the host execution policy was used. Sort direction is determined by T - K < K. For the noise estimation process, two nested for-loops are used. The outer loop iterates against N, and the inner loop iterates against T. For each iteration, the K_T check is performed. Count is kept of how many results were obtained, and if the count reaches N, it breaks out of the loop and exits.

Wrap-around behavior was employed to handle boundary conditions. Because of this, for any given kernel size the output matrix is equal the input matrix size. No cropping or bounding of the compute area was done.

B. GPU-MS

The first GPU implementation of *Minesweeper* will be referred to as 'GPU-MS'. GPU-MS implements parallelism over T as described in the *Complexity Analysis* earlier. GPU-MS was written in C++/CUDA for running on NVIDIA GPUs.

Like CPU-MS, the *sort_by_key* function in *Thrust* was used for the sorting preprocessing step [7]. For this implementation, the device (GPU) execution policy was used for parallelized sorting. This implementation is similar to CPU-MS, with the primary difference being that there is only one for loop which iterates over N. T threads are used to perform K_T checks simultaneously.

Due to implementation limitations, GPU-MS is restricted to a maximum 32^2 kernel size, which will bound our comparative analysis later.

C. GPU-OPT

Lastly, a further optimized version of the original GPU-MS was developed, to be referred to as 'GPU-OPT'. This implementation adds parallelization across N and attempts to address high memory latency observed in GPU-MS.

To implement parallelization across N, a fixed number of blocks was used to divide the IM horizontally into p partitions. Each partition was then sorted individually to produce an independent SA specific for that region. Then, p blocks are spawned that each iterate through its own SA and perform T simultaneous K_T checks to produce the result.

The ordered-statistic noise estimate time complexity is the same as mentioned in Section III for parallelism across N. However the sort complexity for this implementation is not $O(sort(N))$. Instead the sort complexity for the optimized version with k blocks is

$$O\left(k\ sort\left(\frac{N}{k} + wT_h\right)\right)$$

where T_h is the height of the kernel and w is the width of the IM. The sorting step is broken into k sorts, but each block contains an extra T_h in height to allow for data continuity.

In addition to this, low memory transfer throughput was observed in GPU-MS. The primary cost stemmed from fetching a single C_A on each iteration from global memory. To address this, chunks of 64 C_A's from global memory were loaded into shared memory for use by the T threads. This reduced the memory cost and improved throughput.

A minor cost of implementing this is the increased data reuse at the sort and compute stage. The sort step and the ordered statistic noise estimation step has to process more elements than GPU-MS. The magnitude of reuse is proportional to the value and geometric distribution of T. Unlike GPU-MS, we would expect increased runtime complexity at higher kernel dimensions with GPU-OPT.

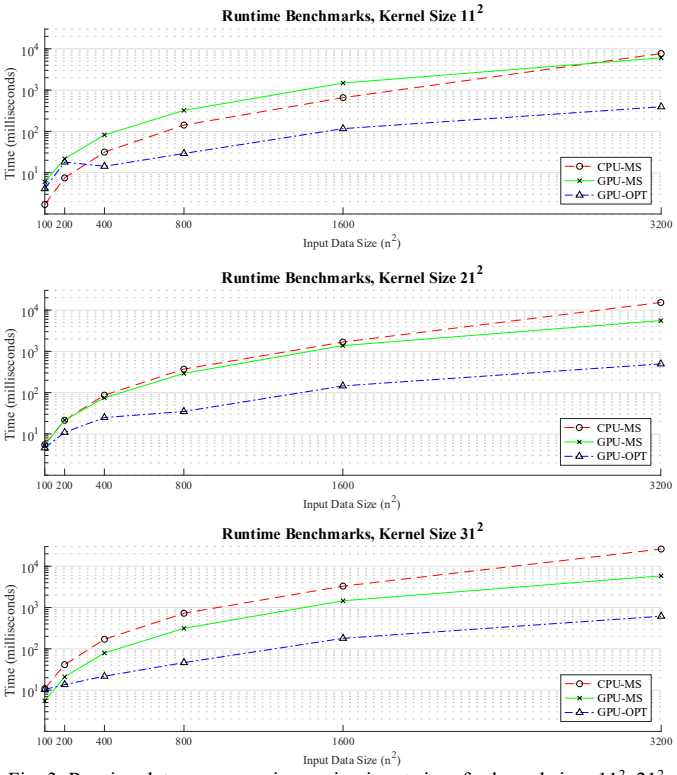

Fig. 3 Runtime latency across increasing input sizes for kernel sizes 11^2, 21^2, and 31^2. In all three plots, GPU-OPT outperforms all other implementations tested at input sizes > 100^2.

TABLE I
BENCHMARKS: KERNEL SIZE 11^2
ROW 1: RUNTIME LATENCY (DURATION IN MILLISECONDS)
ROW 2: RUNTIME THROUGHPUT (MB/S)

Test Case	Input Size (n^2)					
	100	**200**	**400**	**800**	**1600**	**3200**
CPU-MS	1.7	7.5	31.5	142.1	656.2	7690
	23.5	21.3	20.3	18.0	15.6	5.3
GPU-MS	6	21.7	82.2	322.7	1481.3	6048.7
	6.7	7.4	7.8	7.9	6.9	6.8
GPU-OPT	4.1	18.1	14.3	29.2	116.6	394
	9.8	8.8	44.8	87.7	87.8	104

TABLE II
BENCHMARKS: KERNEL SIZE 21^2
ROW 1: RUNTIME LATENCY (DURATION IN MILLISECONDS)
ROW 2: RUNTIME THROUGHPUT (MB/S)

Test Case	Input Size (n^2)					
	100	**200**	**400**	**800**	**1600**	**3200**
CPU-MS	5.5	21.4	86.9	371.6	1680	15298
	7.3	7.5	7.4	6.9	6.1	2.7
GPU-MS	5.3	21.6	75.1	293.1	1380.2	5567.5
	7.5	7.4	8.5	8.7	7.4	7.4
GPU-OPT	4.6	10.9	24.9	35	145.6	496.8
	8.7	14.7	25.7	73.1	70.3	82.4

TABLE III
BENCHMARKS: KERNEL SIZE 31^2
ROW 1: RUNTIME LATENCY (DURATION IN MILLISECONDS)
ROW 2: RUNTIME THROUGHPUT (MB/S)

Test Case	Input Size (n^2)					
	100	**200**	**400**	**800**	**1600**	**3200**
CPU-MS	10.9	41.4	170.4	727.2	3290.8	26065.2
	3.7	3.9	3.8	3.5	3.1	1.6
GPU-MS	5.5	21	79.5	313.1	1457.9	5842
	7.3	7.6	8.1	8.2	7	7
GPU-OPT	10.4	13.6	21.7	46.4	179.5	615.9
	3.8	11.8	29.5	55.2	57	66.5

D. Test Setup

To test the implementations, 2D *n x n* uniformly-distributed input data of 32-bit single-precision floating points were generated with values between 1 and 10^6. All implementations were tested against the same generated dataset. The tests involved running a *m x m* kernel on the *n x n* dataset and computing the average runtime to process a single *n x n* matrix. These implementations were all tested against the worse-case rank scenario ($K_T = T/2$). Any rank other than the median would result in quicker runtimes for *Minesweeper*.

To prove correctness, the results of the three *Minesweeper* implementations were validated against a reference MATLAB implementation. The version of MATLAB used was R2017b. The CPU and GPU implementations of *Minesweeper* ran on an Intel Core i7-7920HQ CPU with 32GB RAM and an NVIDIA Quadro P4000 running on 64-bit Ubuntu 16.04. For GPU-OPT, a block size of 28 was chosen for the Quadro P4000. The GPU implementations were compiled with GCC 7.4.0 and NVIDIA CUDA 10.2.

TABLE IV
PERCENTAGE (%) OF RUNTIME SPENT BY SORTING

Test Case	Input Size (n^2)					
	100	**200**	**400**	**800**	**1600**	**3200**
CPU-MS	5.88	8.53	11.43	10.91	7.2	4.75
GPU-MS	15.09	3.33	1.46	0.88	0.67	0.57
GPU-OPT	83.65	65.44	42.4	18.97	15.88	8.88

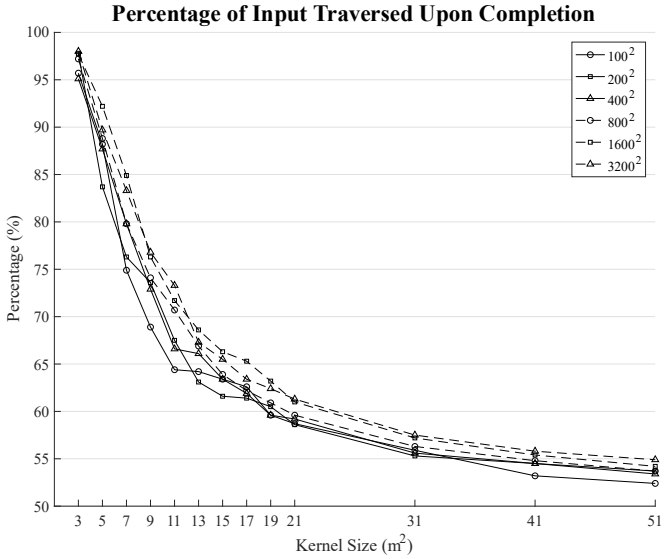

Fig. 4 Iterations as a percentage of *N* that the noise estimation step completed before exiting. Lower percentage is better. It is observed that for $K_T = T/2$, the best case iteration lower bound is at $\omega(N/2)$.

V. RESULTS

A. Runtime Latency

Runtime latency results are shown in Fig. 3 and Tables I, II, and III. For kernel size 11^2, it is observed that GPU-MS overall performed worse than CPU-MS, except at 3200^2. This is chiefly due to the costs of memory latency exceeding the gains from parallelism. This trend starts to wane at kernel size 21^2, and reverses completely at 31^2, where GPU-MS outperforms CPU-MS at every input size. It is worth noting that for all three kernel sizes, GPU-MS's runtime latency is relatively consistent. This is because a single CUDA thread block was created consisting of up to 1024 threads. Since this block iterates over the same *N* cells and *T* is amortized into the thread count, the runtimes are flat with respect to *T*.

For all three kernel sizes, at larger *N* GPU-OPT outperforms all other implementations in runtime. The speedup against GPU-MS was up to 15x faster, with up to 42x improvement against CPU-MS. For smaller *N*, CPU-MS outperformed the GPU implementations due to lower latency, and because of the increased startup cost of initializing and transferring data between the GPU and CPU, which is throughput-efficient but not latency-efficient.

B. Runtime Throughput

Throughput metrics are shown in Tables I, II, and III. In all three tables, we observe an interesting trend. For CPU-MS, as *N* and *T* increases, throughput decreases. This is unsurprising,

given the sequential nature of CPU-MS. For GPU-MS, the throughput remains constant across N and T, which is also expected. For GPU-OPT, it is observed that for increasing N, throughput increases, but for increasing T, throughput decreases. This is interesting to note, and is likely related to the data reuse cost increasing from larger T.

For large N, GPU-OPT had the highest throughput of the three implementations. For $T = 11^2$, CPU-OPT had about 2x higher bandwidth than GPU-OPT for N at 100^2 and 200^2. In practice, this may suggest that CPU-OPT is a more suitable option for smaller datasets with small training kernels.

C. Sort Runtime Contribution

Since *Minesweeper* relies on a sorting preprocessing step, it belongs to the category of OS-CFAR implementations that utilize sorting. However for this implementation, it is not intuitive whether sorting is a major cost to the overall runtime of the algorithm. Metrics for the contributing percentage of runtime that the sort step utilizes is laid out on Table IV.

For implementations CPU-MS and GPU-MS, the sort runtimes were constant across kernel sizes, so the average sort time was used with the lowest runtime observed across the three kernel sizes to compute a worst-case percentage. For GPU-OPT, because sort times are now affected by kernel size, kernel size 31^2 was used for these percentages.

For both GPU-MS and CPU-MS, the percentage of utilization incurred by sort is relatively low, reaching as high as 15% for GPU-MS at 100^2, but decreasing rapidly at higher N. However, for GPU-OPT it is interesting to note that the majority of runtime for input sizes 100^2 and 200^2 comes from the sorting step. This is likely due to the increased cost of sorting due to partitioning. For both GPU implementations, as N increases the runtime contribution of sort decreases.

D. Iterations To Compete

Since all three *Minesweeper* implementations use the K_T count optimization, is it worth observing the degree of effect that this has on iterations completed before all results were computed.

In Fig. 4 a plot of percentages completed upon exit is shown against kernel sizes for 6 sizes of N. These percentages were computed using CPU-MS, but the same results would be expected for GPU-MS, despite the kernel size limitation. For all 6 cases, the curve of decrease appears to fit an inverse exponential trend, approaching 50% (it was observed that for $K_T = T/2$, 50% was the best case bound). However what is most interesting is how similar all 6 cases are in terms of percentage, despite the significant difference in sizes for N.

For instance, at $T = 21^2$, $N = 100^2$ yields a percentage of 58.7%. This T represents 21% of N. However for $N = 3200^2$, the percentage is 61.3% at the same T, but this T represents just 0.0043% of N. Here it is observed that despite being 32x larger than 100^2, the percentage required to iterate only went up by 2.3%. It is observed that just the magnitude of T had significantly more effect on iterations required than what percentage T was with respect to N. It is clear that this optimization has a significant and measurable effect on iterations required to complete.

VI. Extensions for Improvement

This algorithm has a number of avenues for runtime and efficiency improvement.

A. Greedy Geometry

Geometrically it may be possible to determine distances between contiguous CA's in the SA such that two or more CA's could be processed simultaneously if their relative distances are large enough to not affect each other's AM neighborhood. This would be a greedy optimization that would rely on the general distribution of data for performance gains.

A. Streaming Sort

A streaming sort algorithm would reduce latency and allow for concurrent processing of both sort and noise estimation. Since *Minesweeper* doesn't need to process the entire SA at once, this would allow the noise estimation step to begin as soon as the first ordered element returns from the streaming sort. This amortizes the cost of sorting into the ordered-statistic noise estimation.

VII. Conclusion

The algorithm presented has strong potential for use in novel real-time processing pipelines. The ability for users to employ variable, nontraditional and asymmetric training bin masks is a significant trait that opens the possibility for better tuned and higher-sensitivity OS-CFAR detectors. In addition, being able to leverage higher bit-depth inputs and datatypes that were previously impractical is a potentially important benefit for advanced applications that require higher degrees of precision.

Given the GPU implementations tested, this algorithm is well suited for next-generation embedded Software Defined Radio (SDR) applications that are capable of leveraging parallelism from GPUs. There are a growing number of GPU-enabled compute systems available on the market (e.g. NVIDIA Jetson series of SOMs). For these platforms, this algorithm has the most promise for faster and better tuned radar processing chains in the future.

References

[1] G. Brooker, "Detection of Signals in Noise," in *Introduction to Sensors for Ranging and Imaging*, 2009, pp. 357-388.

[2] H. Rohling, "Ordered statistic CFAR technique - an overview," *2011 12th International Radar Symposium (IRS)*, Leipzig, 2011, pp. 631-638.

[3] M. R. Bales, T. Benson, R. Dickerson, D. Campbell, R. Hersey and E. Culpepper, "Real-time implementations of ordered-statistic CFAR," *2012 IEEE Radar Conference*, Atlanta, GA, 2012, pp. 0896-0901, doi: 10.1109/RADAR.2012.6212264.

[4] S. A. Villar, B. V. Menna, S. Torcida and G. G. Acosta, "Efficient approach for OS-CFAR 2D technique using distributive histograms and breakdown point optimal concept applied to acoustic images," in *IET Radar, Sonar & Navigation*, vol. 13, no. 12, pp. 2071-2082, 12 2019, doi: 10.1049/iet-rsn.2018.5619.

[5] S. Perreault and P. Hebert, "Median Filtering in Constant Time," in *IEEE Transactions on Image Processing*, vol. 16, no. 9, pp. 2389-2394, Sept. 2007, doi: 10.1109/TIP.2007.902329.

[6] E. Hyun and J. Lee, "A New OS-CFAR Detector Design," *2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering*, Jeju Island, 2011, pp. 133-136, doi: 10.1109/CNSI.2011.16.

[7] N. Bell and J. Hoberock, *Thrust: A Productivity-Oriented Library for CUDA*. Montgomery, AL, USA: GPU, 2012.