# Optimizations of Dual Polarization FMCW Weather Radar Signal Processing on CUDA Platform

Satrio Adi Rukmono[1], Achmad Imam Kistijantoro[2], Riza Satria Perdana[3]
*School of Electrical Engineering and Informatics*
*Institut Teknologi Bandung*
Bandung, Indonesia
[1]sar@itb.ac.id, [2]imam@stei.itb.ac.id, [3]riza@stei.itb.ac.id

***Abstract*—Weather radar is a system that utilizes advanced radio wave engineering to detect precipitation in the atmosphere. One of the wave generation technique used in weather radar is frequency-modulated continuous wave (FMCW), with dual polarization for differentiating detected precipitation types by its shape and size. Weather radar signal processing is usually performed using digital signal processing and field-programmable gate array (FPGA), that performs well but with difficulty in system development and deployment. Software implementation of weather radar signal processing enables easier and faster development and deployment with the cost of performance when done serially. Parallel implementation using general purpose graphics processing units (GP-GPU) may provide best of both worlds with easier development and deployment compared to hardware-based solutions but with better performance than serial CPU implementations. In this paper, implementation of various optimization strategies weather signal radar processing in GP-GPU environment on the Nvidia CUDA platform is shown. Performance measurements show that among optimization strategies implemented, only the utilization of multiple CUDA streams give significant performance gain. This paper contributes in attempts to build full weather radar signal processing stack on GPU.**

***Keywords—weather radar, signal processing, parallel programming, GP-GPU, Nvidia CUDA***

## I. Introduction

Radar, or radio detection and ranging, is a system that utilizes radio wave to determine the existence of an object and its range from the radar itself [1]. Originally radars were designed to detect aircraft targets, with weather precipitations considered noise, but more advanced radio wave engineering enables radar to be used to map weather, with precipitations as radar target [2].

Radar signal processing is usually done using digital signal processor or field-programmable gate arrays [3][4], which performs well but with drawbacks of difficulty in system development and deployment. Software-based implementations of radar signal processing enables simpler development and faster deployment but does not perform well when implemented serially. The state of general purpose graphics processing units for high performance computing may provide best of both world with ease of development and deployment, but also with better performance compared to serial CPU implementation.

Graphics processing units (GPU), originally designed to process graphical display of computer systems, are now being used for high performance, general purpose computing. Since graphics processing usually involves same operations applied on different elements of the graphic, known as pixels, GPU's are designed to be good at processing simple instructions on multiple data at once. Weather radar signal data are usually represented as matrices and its processing involves algorithms with some amount of parallelism, which justifies the use of general purpose GPU for weather radar signal processing. This paper focuses on signal processing of frequency-modulated continuous wave (FMCW) weather radars with dual polarization.

## II. Weather Radar Signal Processing

Radars work by emitting radio waves and receiving back the echoes caused by the waves hitting objects. Time difference between transmitting the wave and receiving the echo is used to determine the *distance* between the radar and the detected object, or *target*. Other than distance, the position of a target is also defined by its *azimuth* which in rotating radars such as the FMCW radar can be determined directly from which direction the antenna is facing when transmitting the wave that is reflected by the object. Additionally, *elevation* angle of the antenna may be used to determine the altitude at which the target is positioned.

In radar signal processing, one full sweep of radar antenna (360°) is divided into several discrete *sectors* that each represents a certain range of azimuth. The received signal of each sector is represented by a 1024×512 matrix, with number of columns (512) representing the number of samples taken on each sector and the number of rows (1024) representing received signal in frequency domain for FMCW radar. This frequency-domain signal can then be separated into 512 *range bins*, which are discrete representation of distance, after application of fast-Fourier transform. Fig. 1 illustrates the relationship between sectors and range bins.

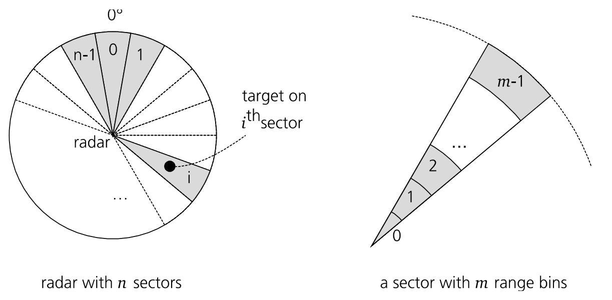

Fig. 1. Sectors and range bins.

Dual polarization radars such as the focus of this paper transmits signals in both horizontal and vertical polarizations [5]. This configuration allows the radar system to determine target shape in addition to its size. Echoes of both polarizations are received by both horizontal- and vertical-polarized antennas, resulting in four kinds of signals: ones transmitted by the horizontal antenna and received by horizontal receiver (HH), its vertical counterpart (VV), ones transmitted by the horizontal antenna and received by vertical receiver (HV), and the other way around (VH). Since HV and VH essentially contain the same information, only one of the two is necessarily processed. As such there are three 1024×512 matrices to be processed in each sector in weather radar signal processing.

The process applied to said matrices are divided into three stages according to the output size of each stage. The first stage takes the 1024×512 matrices and produces 512×512 matrices. The second stage takes the result of the first stage and reduces it into vectors sized 512. The third stage does calculation on the resulted vectors to get desired outcomes that is reflectivity, differential reflectivity, linear depolarization ratio, and mean Doppler velocity. For this paper, only first two outcomes are implemented. The full stages implemented are shown in Table I.

TABLE I. STAGES IN DUAL-POLARIZED FMCW WEATHER RADAR SIGNAL PROCESSING.

| ***Stage*** | ***Processes*** |
|---|---|
| I | Hamming window calculation |
| | FFT on range profile |
| | Clutter suppression |
| | FFT on Doppler profile |
| | Further clutter suppression |
| II | Power calculation on Doppler profile |
| | Doppler cell filtering |
| | Range bin filtering |
| III | Reflectivity calculation |
| | Differential reflectivity calculation |

## III. PARALLEL COMPUTATION WITH GPU

GPU is a part of computer system which were originally only used for processing graphical data for computer displays. GPU gives better performance compared to CPU for graphics processing by using many cores which execute the exact same instruction for different graphics elements simultaneously. GPU's are also designed to be more efficient at doing simple arithmetic and logical operations instead of controlling program flow, conforming to the characteristics of graphics processing.

As GPU technology continues to develop, current GPU's have theoretical performance that is multiples of CPU's. GPU's are now not only used for processing graphics data but also for general purpose computing, from which the term *general-purpose graphics processing unit* emerges. GP-GPU's are used for high performance computing such as deep learning.

CUDA is a GP-GPU-based computing platform developed by Nvidia, first introduced in 2006. CUDA enables GPU-based software development using high-level languages such as C and C++ [6].

## IV. RELATED WORKS

There are many works attempting to use GPU for radar signal processing, and they show that GPU indeed gives good performance. These works differ from this paper in terms of the types of radar in question and kinds of algorithms implemented. Specifically, they focus on synthetic aperture radar [7], air surveillance radars [8][10], pulsed-Doppler radar [3], modular UHF ionosphere radar [9], and passive radar [11]. Some of these works concern with the full stack of process [7][8][9] while the others only focus on part of the process such as constant false alarm rate [3][11] or space-time adaptive processing [10][11]. Venter's work [3] influenced this paper in terms of comparing different implementation strategies, although it compares different task division strategies while this paper compares different optimization strategies. Fallen's work [9] is unique in that it compares two different GPU environments for the exact same case.

In this paper, both baseline serial implementation and parallel implementations are developed based on CPU implementation of weather radar signal processing as shown in [12].

## V. PERFORMANCE CONSIDERATIONS IN THE GPU

In designing parallel algorithms that utilizes GPU optimally, there are several things to consider [13].

### A. *Warp partitioning*

GPU executes single instruction simultaneously for all threads. When a diverse conditional occurs in a GPU kernel, threads no longer execute the same instructions and thus, threads that take one path of the conditional must idly wait for the others to finish before continuing. Ideally there should be no conditional branching in a kernel to have maximum GPU utilization. If conditionals are unavoidable, the diversity and body of conditional blocks must be carefully designed.

### B. Memory coalescing

In CUDA programming model, memory latency is one cause of bottleneck. How one organizes data within hierarchies of memory is crucial to the efficiency of memory access. Data that are logically "nearby" in the scope of the algorithm should also be placed contiguous in the memory.

### C. Dynamic partitioning of shared memory resource

Data can be partitioned multiple ways in a parallel algorithm. In some cases, using many thread blocks with few threads gives better performance than using few thread blocks with many threads, but in others the other way around, depending on the size of the data and what processes are performed on them. In many case, dynamic partitioning is necessary to gain better performance.

### D. Data prefetching

Memory latency can be hidden by overlapping data transfer with data processing. One technique is to load next data while processing current data.

### E. Instruction mix

Loop controls such as loop conditionals and loop counters are additional computations that uses GPU resources. Loop unrolling can be used in loops with definite number of iterations to shave off some computational baggage.

### F. Thread granularity

Conceptually, more threads mean more data being processed at the same time, i.e. faster execution time. But, number of threads is not linear to performance improvements due to thread overheads. In some cases, using few large threads is better than using many small ones.

### G. Instruction-level parallelism

Instruction-level parallelism is another way to hide memory latency. Having fewer threads that have independent operations may give better utilization and improve performance up to 2× [14].

## VI. Implementation of Optimization Strategies

Eleven kernels were made for processing weather radar signal with two of them eventually merged into one, combined with calls to existing library cuFFT for fast-Fourier transform parts of the processing. Five of the kernels belong to Stage I of the processing, four to Stage II, and one to Stage III. This implementation is considered complete in terms of GPU usage, since no part of processing is done in the CPU and thus the only memory copy operations are in the beginning of sector processing for copying received signals and in the end of sector processing for copying the outcomes.

First implementation is a naïve one, done by the book from algorithm design without taking much considerations to GPU optimizations. After the first implementation, dubbed the G0 version is proven to yield correct results, optimization strategies are applied to see how much performance gain can be achieved. Optimization strategies applied follows.

### A. Block size adjustments

Two of the kernels implemented uses too few threads within a block, so they are revised to employ fewer blocks with more threads within them. Specifically, kernel calls for clipping are modified from using 1024 blocks of 2 threads into 2 blocks of 1024 threads, while kernel calls for reflectivity and differential reflectivity calculations are modified from using 512 blocks of 1 thread into 1 block of 512 threads. This version, named version G1, corresponds with performance consideration F (*thread granularity*).

### B. Instruction-level parallelism

Instruction-level parallelism is achieved in two forms in this implementation, one in the reduction kernels to process several elements within a kernel, and one by merging two kernels together.

In reduction kernels, it is achieved by processing several elements in one kernel using unrolled loops. Experiments were done with two, four, eight and 16 elements per kernel. This reduction kernel optimization is implemented in version G2 and again in version G4.

In the second form, the kernel that is used for computing complex number conjugates and the kernel for doing FFT shifts are merged. Conjugates are achieved by multiplying the imaginary part of a complex number with -1, while FFT shifts are done with common swap procedure using temporary buffer. In the merged kernel, two temporary buffers are used instead of one, and the conjugation is performed while loading elements into the buffers. This version is called version G6. Table II shows how the two kernels merge into one.

This optimization strategy corresponds with performance consideration G (*instruction-level parallelism*).

TABLE II. TWO KERNELS IN G5 MERGED INTO ONE IN G6.

| G5 | G6 |
|---|---|
| kernel conjugate: | kernel conjshift: |
| in$_{i,j}$.imag=in$_{i,j}$.imag*-1 | tmp1 = in$_{i,j}$ |
| end kernel. | tmp2 = in$_{i,j+n/2}$ |
| | tmp1.imag = tmp1.imag*-1 |
| kernel shift: | tmp2.imag = tmp2.imag*-1 |
| temp = in$_{i,j}$ | in$_{i,j}$ = tmp2 |
| in$_{i,j}$ = in$_{i,j+n/2}$ | in$_{i,j+n/2}$ = tmp1 |
| in$_{i,j+n/2}$ = temp | end kernel. |
| end kernel. | |

### C. Shared memory usage

Copying parts of data into shared memory may give better performance if said data are accessed multiple times within a kernel. This is proven in the optimization to the reduction kernels version G3. The modifications in G2 and G3 are then merged into G4 to give even better performance.

In naïve reduction implementations, matrix elements are copied to the output buffer and the procedures for reduction is done on the output buffer. In the modified version, matrix elements are copied to shared memory and the procedures are performed on the shared memory buffer. Finally, the reduction result is copied to the output buffer. Table III shows the principle differences of reduction kernels in versions G1 to G4.

TABLE III. REDUCTIONS IN VERSIONS G1, G2, G3, AND G4.

| G1 | G2 |
|---|---|
| $out_{i,j}$=$in_{i,j}$<br>for s=[…]:<br>  $out_{i,j}$ += $out_{i,j+s}$ | $out_{i,j}$=$in_{i,j}$<br>$out_{i,j+1}$=$in_{i,j+1}$<br>for s=[…]:<br>  $out_{i,j}$ += $out_{i,j+s}$<br>  $out_{i,j+1}$ += $out_{i,j+1+s}$ |

| G3 | G4 |
|---|---|
| $shared_{j}$=$in_{i,j}$<br>for s=[…]:<br>  $shared_{j}$ += $shared_{j+s}$<br>$out_{i,0}$ = $shared_{0}$ | $shared_{0,j}$=$in_{i+0,j}$<br>$shared_{1,j}$=$in_{i+1,j}$<br>for s=[…]:<br>  $shared_{0,j}$ += $shared_{0,j+s}$<br>  $shared_{1,j}$ += $shared_{1,j+s}$<br>$out_{i+0,0}$ = $shared_{0,0}$<br>$out_{i+1,0}$ = $shared_{1,0}$ |

Another form of reduction kernel in this implementation is the in-place reduction kernel, where the reduction result of each row is placed in the same memory buffer as the input, specifically on the first column of each row. In this version there is no copying to output buffer in the naïve implementation and therefore no memory latency cost, but in the revised version memory latency is introduced in copying input to shared memory. There is still positive performance gain, though, because subsequent accesses to the shared memory produces lower latency.

### D. *Constant memory usage*

Weather radar signal processing in this paper involves the use of two constant coefficients in the form of a matrix and a vector. Such coefficients can potentially be stored in constant memory to decrease latency, but since constant memory capacity is very limited, only the second coefficients are stored in there.

The Hamming window coefficient consists of 1024×512 floating point elements, which translates to 2 megabytes of data, while the constant memory of tested device is limited to 64 kilobytes. The moving average coefficient, on the other hand, consists of 512 complex numbers, which are represented as a pair of floats. This translates to 4 kilobytes of data, which still fit in the constant memory.

This optimization is called version G5.

### E. *Data prefetching with CUDA streams*

In this optimization, data for the next sector is copied to the device memory while processing data for the current sector, resulting in version G7. The overlap of memory copy and data processing is achieved using CUDA streams.

To achieve this, the main program loop is modified as follow: where previously the loop body begins with loading sector data followed by data processing and finally copying the result back to the host memory, the new main program starts with loading the first sector before the main loop, with the main loop body consisting of processing current sector data followed by loading next sector data and finally copying the result of current sector to the host memory. This difference is shown in pseudocode form in Table IV.

TABLE IV. COMPARISON OF MAIN PROGRAM LOGIC IN PREVIOUS VERSIONS AND THE STREAMED VERSION

| Non-streamed version | Streamed version |
|---|---|
| for *i*=[0, *NUMSECTORS*):<br>  load sector *i*;<br>  copy sector *i* to dmem;<br>  process sector *i*;<br>  copy result of sector *i*<br>    to hmem;<br>end for. | s = 0;<br>load sector 0;<br>copy sector 0 to dmem on<br>    stream *s*;<br>for *i*=[0, *NUMSECTORS*):<br>  process sector *i* on<br>    stream *s*;<br>  if *i*<*NUMSECTORS*-1:<br>    load sector *i*+1;<br>    copy sector *i*+1 to dmem<br>     on stream (*s*+1) mod<br>     *NUMSTREAMS*;<br>  end if;<br>  copy result of sector *i*<br>     to hmem on stream *s*;<br>  *s* = (*s*+1) mod *NUMSTREAMS*;<br>end for. |

Experiments were done using two to sixteen streams. In the following sections, versions coded G7-*x* denotes that the G7 version is run with *x* number of CUDA threads.

This optimization strategy corresponds with performance consideration D (*data prefetching*).

## VII. UNUSED OPTIMIZATION STRATEGIES

As can be seen from the previous section, not all performance considerations are implemented in the final, optimized versions. This is because experiments with some optimization strategies does not yield better performance, or the strategy is not applicable for the process concerned in this paper. This section discusses such optimization strategies.

Performance consideration A, *warp partitioning*, is not applicable for this research since there are no conditionals in the algorithm.

For performance consideration B, *memory coalescing*, the naïve base implementation is already coalesced according to the characteristics of the process. The algorithm includes both row-wise and column-wise processes, but the latter is only done once for each matrix. A such, arranging the matrix column-first does not yield better performance than row-first as in the base implementation.

While optimization strategies *shared memory usage* and *constant memory usage* seem related to performance consideration C (*dynamic partitioning of shared memory resource*), it is actually not applicable in this research because the data size is always constant, i.e. 1024×512 for each matrix, and therefore there are no room for dynamic partitioning. An optimal partitioning scheme for one input sample is applicable for all input samples.

Finally, in Hamming window calculation, experiments were done to apply performance consideration E, *instruction mix*, in which the kernel applies Hamming coefficient to several matrix cells. This experiment yields worse performance compared to the naïve version of Hamming window calculation.

## VIII. Results

### A. Correctness

The calculation result of the parallel implementation is found to be correct by using the formula of L2 relative error, compared to the calculation result of baseline serial implementation. Calculated error of one sector is $1.99995 \times 10^{-6}$, which is considered good enough.

### B. Performance analysis

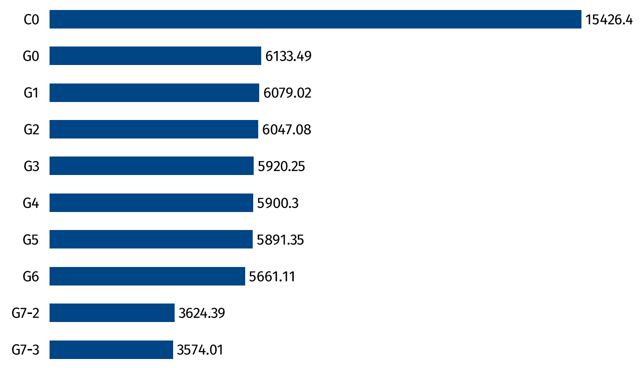

Fig. 2. Execution time of various implementation versions, in miliseconds.

Performance comparison of CPU implementation, dubbed version C0, and the various GPU implementations is shown in Fig. 2, while speedups as calculated by previous execution time divided by optimized execution time, is shown in Table V. Performance measurements are taken on Intel® Core™ i5-6200U (2 cores, 4 threads) machine with 4 gigabytes of RAM and NVIDIA® GeForce® 930M graphics card.

Table V. Speedup of various implementation versions

| Version | Remarks | *Immediate speedup* | *Cumulative speedup* |
|---|---|---|---|
| G0 | Naïve parallel implementation | 2.52 | 2.52 |
| G1 | Block size adjustment | 1.01 | 2.54 |
| G2 | Instruction-level parallelism | 1.01 | 2.55 |
| G3 | Shared memory usage | 1.02 | 2.61 |
| G4 | Combination of G2 and G3 | 1.00 | 2.61 |
| G5 | Constant memory usage | 1.00 | 2.62 |
| G6 | Kernel merge | 1.04 | 2.72 |
| G7-2 | Data prefetching with two streams | 1.56 | 4.26 |
| G7-3 | Data prefetching with three streams | 1.01 | 4.32 |

Below are several remarks about performance gains achieved by implementing optimization strategies discussed before:

1. Block size adjustments work because previously there were kernels that use only one or two threads per block. This is highly inefficient and thus optimized by instead having one or two blocks with hundreds of threads.
2. Using loop unrolling to achieve instruction-level parallelism in reduction kernels work but only to a small degree. This is most likely because data is not coalesced for this part of the processing but restructuring data to achieve coalescing in this part may impact performance in other parts.
3. Utilization of shared memory in reduction kernels give performance gain even when the cost of copying data to the shared memory is accounted for. This is because reducing a large number of elements such as in this case requires multiple accesses to the same element in one kernel, and each of those accesses require less time when stored in shared memory.
4. Utilization of device constant memory does not yield performance gain because each element within the constant memory slot is accessed only once per kernel.
5. Overlapping data transfers between host and device memory with computations gives significant performance gain by reducing GPU idle time (Fig. 3). Data loading which was previously started only after the previous sector has been processed can now be done parallel to it. The performance gain stops at three streams, though, because overhead in CPU-side data loading eclipses streaming advantages.

## IX. Conclusions

From the experiments, it can be concluded that GPU is indeed fitting for processing dual polarization FMCW weather radar signal, since signal data are represented as matrices with dimensions always in the form $2^n$, and processes include many highly parallel algorithms and algorithms with known optimized parallel implementations.

Another drawn conclusion is that most optimization strategies applied to the parallel implementations in this research only give minor performance boosts with exception of the utilization of multiple CUDA streams for data prefetching. This strategy gives 1.72 speedup compared to naïve parallel implementation, or 4.32 speedup compared to serial implementation.

Further research is necessary to implement other parts of weather radar signal processing that is not covered in this paper, such as the calculation of linear depolarization ratio and mean Doppler velocity. Comparing parallel implementations in GPU environment with parallel implementations in CPU environment may also worth researching.

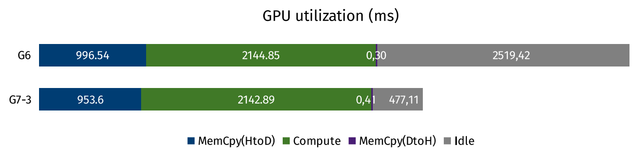

Fig. 3. Execution time breakdown of G6 and G7 with three CUDA streams, in miliseconds.

## References

[1] M. I. Skolnik, Radar Handbook. New York: McGraw-Hill, 2008.

[2] V. N. Bringi and V. Chandrasekar, Polarimetric Doppler weather radar principles and applications. Cambridge: Cambridge Univ. Press, 2005.

[3] C. J. Venter, H. Grobler, and K. A. AlMalki, "Implementation of the CA-CFAR algorithm for pulsed-Doppler radar on a GPU architecture," IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT), pp. 1–6, 2011.

[4] H. Jianxin, X. Mingyuan, and Z. Yi, "The signal processing system design of polarimetric weather radar using DSP and FPGA," IEEE International Symposium on Microwave, Antenna, Propagation and EMC Technologies for Wireless Communications. Vol. 2, 1245–1248, 2005.

[5] National Weather Service, "Dual Polarization Radar," https://www.weather.gov/bmx/radar_dualpol, Sept. 2016 [visited on 11/01/2017].

[6] *CUDA C Programming Guide*. v8.0, Nvidia, 2017.

[7] C. Clemente, M. Di Bisceglie, M. Di Santo, N. Ranaldo, and M. Spinelli, "Processing of synthetic Aperture Radar data with GPGPU," IEEE Workshop on Signal Processing Systems, pp. 309–314, 2009.

[8] P. Monsurró, A. Trifiletti, and F. Lannutti, "Implementing radar algorithms on CUDA hardware," Proceedings of the 21st International Conference Mixed Design of Integrated Circuits and Systems (MIXDES), pp. 455–458, 2014.

[9] C. T. Fallen, B. V. C. Bellamy, G. B. Newby, and B. J. Watkins, "GPU Performance Comparison for Accelerated Radar Data Processing," Symposium on Application Accelerators in High-Performance Computing, pp. 84–92, 2011.

[10] T. M. Benson, R. K. Hersey, and E. Culpepper, "GPU-based space-time adaptive processing (STAP) for radar," IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6, 2013.

[11] M. Bernaschi, A. Di Lallo, R. Fulcoli, E. Gallo, and L. Timmoneri, "Combined use of graphics processing unit (GPU) and Central Processing Unit (CPU) for passive radar signal & data elaboration," 12th International Radar Symposium (IRS), pp. 315–320, 2011.

[12] M. N. Haryasena, A. D. Setiawan, H. Muhaimin, and A. Munir, "Signal processing of position plan indicator display for weather radar," 22nd Asia-Pacific Conference on Communications (APCC), pp. 375–378, 2016.

[13] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2010.

[14] V. Volkov, "Better performance at lower occupancy," UC Berkeley, Sept. 2010.