Profiling Techniques for Optimizing Synthetic Aperture Radar (SAR) Image Formation

By Norihito Naka, ECE '16

Current existing implementations of the image formation algorithm for UAV SARs take both multiple days and over 15 Gigabytes of memory to run: both the space and time complexity of running MATLAB® code, which exhaustively seeks to sharpen a 3-dimensional image, is unreasonable in its applications for defense. This note explores the numerous methods of optimizing the algorithm and the profiling tools used to measure the performance improvements as compared to previous iterations.

Background

In collaboration with MIT Lincoln Laboratories, the Blue Team seeks to develop a robust, real-time, framework that enables UAVs to sweep 2-dimensional apertures to construct radar images that contain height information. Lincoln Laboratories seeks to develop a UAV SAR drone that can scan different topographies using the PulsOn® board from TimeDomain® and generate a 3-dimensional image. The existing algorithm is written in MATLAB® and takes the back-projected pulses and exhaustively determine the appropriate phase off sets to decrease the distortion in the image, measured using entropy.

Many of the problems with the current algorithm are due to the fact that it applies brute force; in other words, it takes the back-projected pulses and calculates all possible entropy associated with a large range of phase offsets. Entropy measures the distortion that exists in an image and indicates the efficacy of the phase error correction: smaller entropy values imply better quality images. Through doing this computation, the algorithm pinpoints the phase offset that results in the lowest entropy and thereby focuses the three-dimensional image, decreasing the noise introduced as perturbations in the phase offsets.

In theory, this method may be a fine as it would determine the true minimum entropy possible. In practice, however, for any image of an aperture larger than 50 by 50 by 50, the computation not only takes days, but also requires greater than 15 Gigabytes of memory. In the spirit of making an image formation algorithm that will run within the order of hours while computing on an average laptop device, optimization for time and space is necessary.

This takes multiple forms: first, the Blue Team truly implement an algorithm that closely matches the mathematical derivation of minimum entropy autofocus; this will fundamentally improve the performance over the exhaustive search method describe above. Next, through profiling and testing the code, problematic areas are identified and refactored. Finally, migrating much of the computation from MATLAB® to C++ further memory and time optimizations were recorded.

Optimizations

Space

One of the primary methods of optimizing the algorithm was writing partial data to a file and writing to disk instead of memory. This prevented the RAM from being overloaded. Additionally, through selecting smaller back projection slices and completing computation of those before moving on to iterate through the rest of the pulses also proved to be an effective method of space optimization.

Time

Central Processing Unit (CPU)

In MATLAB®, vectorization of the four-dimensional data as well as preallocating space was crucial in saving computation time. By distributing the processes of precomputable values and reading from memory over disk enabled the step-size algorithm to effectively minimize the computations necessary, thereby improving the time complexity of the new algorithm.

Graphics Processing Unit (GPU)

CUDA® from NVIDIA has enabled further parallel computing using the GPU. Since the GPU has multiple blocks, each containing numerous threads, much of the incremental computations can happen at the same time. Figure 1 illustrates how each thread can be allocated to compute one simple calculation and are contained within blocks.

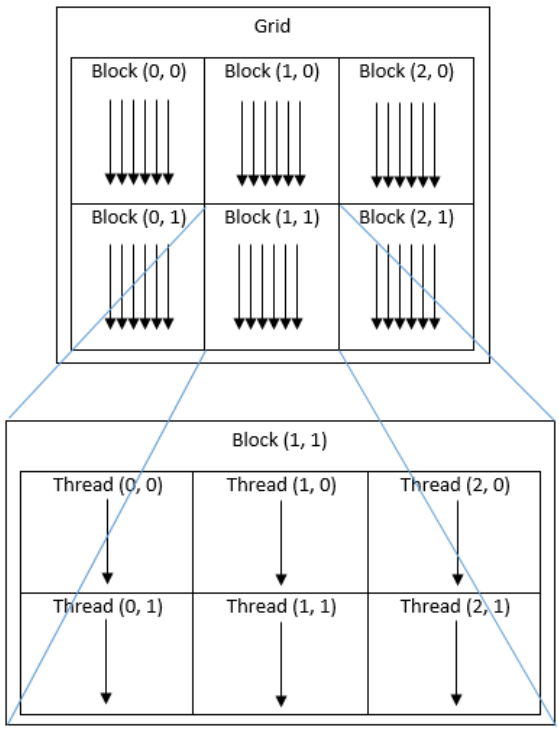

Figure 1. Each GPU has a grid which contains a number of blocks which in turn contain the threads. Each thread is able to do a basic mathematical computation.

Software Profiling

Execution Path Analysis (Native Profiling)

The native profiler in MATLAB® provided a springboard off to base future iterations of code. Examining the "Run and time" functionality of MATLAB® and its outcomes lead to an understanding that certain functions were taking more clock cycles than others. In C and C++ the "get_time" functions were called in order to simulate similar timing capabilities.

Static Analysis (Algorithmic Profiling)

Static analysis was used in implementing the autofocus algorithm from its mathematical model to concrete code. Calculating the optimal decreases in entropy and taking the appropriate step sizes down to the ideal phase offset would therefore perform strictly faster than the brute force method.

Results and Conclusion

Through the optimization methods outline above, initial speed improvements using multithreads were recorded. Moreover, these speed and memory improvement were further enhanced through the porting of the code from MATLAB® to C++

Through putting the minimum entropy gradient descent algorithm on the GPU, the radar data obtained from flying the UAV through a 2-dimensional aperture yields a 3-dimentional image approximately exponentially faster than execution times seen in the brute force algorithm.

The tools laid out above help identify problems areas in the Blue Team's code such that we could find better methods of optimizing for space and time complexity by over a factor of 50.

References

Chivers, I. & Sleightholme, J. (2015). *An Introduction to Algorithms and the Big O Notation.* Switzerland: Springer International Publishing. doi:10.1007/978-3-319-17701-4_23

Florian Brandner, S. H. (2014). Criticality: static profiling for real-time programs. *Real-Time Systems*, pp 377-410. doi: 10.1007/s11241-013-9196-y

Fuad, M. M., Deb, D. P., & Baek, J. (2013). Static Analysis, Code Transformation and Runtime Profiling for Self-healing. *Journal of Computers, 8*(5), 1127-1135. doi: 10.4304/jcp.8.5.1127-1135

Holzmann, G. J. (2015). Assertive Testing [Reliable Code]. *IEEE Software, 32*(3), 12. doi:10.1109/MS.2015.60

Kim, J. W. (2014). *Accelerating Matlab with GPUs*. Morgan Kaufmann, Boston, Pages 45-72. doi: 10.1016/B978-0-12-408080-5.00003-1

Kukunas, J. (2015). *Power and Performance*. Morgan Kaufmann, Pages 105-118. doi: 10.1016/B978-0-12-800726-6.09995-X

Mili, A., & Tchier, F. (2015). *Software Testing: Concepts and Operations*. John Wiley & Sons. Retrieved from http://proquest.safaribooksonline.com/book/software-engineering-and-development/software-testing/9781119065593

Steven Homer, A. L. (2011). *Computability and Complexity Theory*. New York: Springer US. doi:10.1007/978-1-4614-0682-2

Youfeng Wu, Y.-F. L. (2005-09). Hardware-Software Collaborative Techniques for Runtime Profiling and Phase Transition Detection. *Journal of Computer Science and Technology* , pp 665-675 . doi:10.1007/s11390-005-0665-1