Using GPU VSIPL & CUDA to Accelerate RF Clutter Simulation

Dan Campbell, Mark McCans, Mike Davis, Mike Brinkmann
Georgia Tech Research Institute
Smyrna, Georgia, United States of America
{dan.campbell, mark.mccans, mike.davis, mike.brinkmann}@gtri.gatech.edu

***Abstract*-This paper describes a flexible simulator for background Radio Frequency clutter developed at the Georgia Tech Research Institute, and how this simulation was accelerated with the use of nVidia GPUs using GPU VSIPL. The paper describes the mathematical basis for the simulation and how it can be used to simulate RF environments and scenarios; introduces the VSIPL API; describes the porting and validation process; highlights challenges raised by the conversion from double to single precision, and how they were met; and describes the techniques used to obtain improved execution speed, achieving 70x improvement over the original simulation.**

I. RF Clutter Simulation

The RF clutter simulation is a portion of a testbed to evaluate the performance of radar components and radar algorithms. In order to present realistic simulated radar return data, realistic simulated background clutter must be generated. GTRI maintains a flexible radar environment simulator developed in MATLAB, which contains a module to create this simulated data.

A radar system senses its environment by transmitting radio frequency (RF) energy and observing the echoes from objects in the surveillance area. In many scenarios of interest, the performance of such a system is limited by the strong return from the ground, which can mask smaller targets. In order to understand the impact of this so called radar clutter, and evaluate methods to suppress it, high fidelity simulations are required.

While it is straightforward to simulate radar returns from discrete targets, simulation of the return from every point on the ground that is visible to a radar is far more challenging. In many cases radar systems are physically capable of resolving patches on the ground that are less than a meter on each side. A high fidelity clutter simulation must simulate scatterers finer than this spacing over points on the earth that can extend for thousands of kilometers.

This approach is necessary since the contribution due to each resolvable clutter patch will have distinct angle and Doppler properties. These properties must be faithfully reproduced since it this structure that allows a radar signal processor to suppress clutter. The complexity of radar clutter simulations is due precisely to this fact: each point on the ground must be treated independently in order to model the salient features of the corresponding radar data.

In contrast to the computational complexity, the concept behind the radar clutter simulation is relatively simple. A radar is essentially only capable of measuring range. Because all targets at a particular range will be observed by the radar at the same time, the clutter simulation first divides the ground into range rings. Due to the distributed nature of clutter, each range ring may be processed independently. This range ring is then further sub-divided into a number of clutter patches. Each clutter patch may be described according to its range, azimuth, and elevation relative to the platform.

In order to determine the contribution of a clutter patch, a model of the radar system is then employed. The radar range equation maps these (range, azimuth, elevation) coordinates along with the radar system parameters to determine how much energy each patch contributes. This process is repeated for each clutter patch in a given range ring, for each range ring, and ultimately for each of several to many pulses.

II. GPU VSIPL

The Vector Signal Image Processing Library (VSIPL)[1] is a portable API for implementing high-performance signal processing applications while retaining platform independence. VSIPL supports memory abstractions for utilizing coprocessors with disjoint memory spaces. A signal processing application may structure input data in a block, admit it once to VSIPL's memory management, perform computations on that data, and release only the block containing the final result. Intermediate results are not transferred between system and coprocessor memory, avoiding unnecessary latencies and communication overheads. This capability distinguishes VSIPL from other numerical libraries that permit random access to data.

VSIPL consists of management functions for memory and data, and mathematical operations. The baseline VSIPL memory abstraction is the block, which is opaque, linearly addressable storage. Blocks may optionally be associated with a host memory location, and may be in admitted or released states. When in the admitted state, block data may only be accessed via get and put functions. If associated with host memory, state change optionally forces consistency, thus controlling the occurrence of copies between memory spaces, as well as the availability of data for host-based operations. The mathematical operations are scalar, vector, and matrix based, ranging in granularity from basic operators, to system solvers. Two subsets of the VSIPL API are defined for incomplete implementations, the Core Profile[2], and the Core Lite Profile[3].

GPU VSIPL is an implementation of the VSIPL API that is accelerated with nVidia GPUs. GPU VSIPL achieves high performance by performing all processing on available GPUs. By requiring all data blocks be admitted to VSIPL before operations may be executed on them, VSIPL effectively hides the disjoint memory spaces from client applications. New and existing applications written with VSIPL may leverage the GPU acceleration simply by linking with GPU VSIPL.

III. Porting to VSIPL

In order to ensure a correct port, our first task was to create a validation mechanism and correctness criteria. Since the simulation includes some random elements, and because we anticipated changing from double to single precision, a bitwise comparison of outputs was not suitable. We encapsulated the random number generator in MATLAB to optionally record the random number set generated, and treated this set as an optional input to the ported simulation. We ran the baseline simulation with 1600 different random number seeds and recorded the random sets and simulation output to use for validation. After considering the error and noise present in other elements of the system, we selected several correctness criteria, described below:

$$\frac{(CNRm - CNRv)}{CNRm} < 10^{-3} \quad (1)$$

$$20\log_{10}\left(\frac{norm(M-V)}{norm(M)}\right) < -60dB \quad (2)$$

$$mean(|(FFT(M) - FFT(V)|^2) < 10^{-3} \quad (3)$$

Where M represents the data from the original simulation and V represents data from the VSIPL version. CNR represents the overall clutter to noise ratio of the generated data set.

VSIPL supports a wide range of precisions, and implementations may provide different precision levels. Many optimized VSIPL implementations, including GPU VSIPL, support single precision floating point, but not double precision floating point. In order to easily change precisions and VSIPL implementations, we made the precision used selectable at compile time by means of a preprocessor directive.

Because the VSIPL API allows intuitive operation on mathematical data objects, the initial port of the simulation was straightforward. A professional C programmer that had no prior experience with VSIPL or tuning applications for execution speed performed the port in approximately three weeks. As an intermediate step, and to aid in validation, we tested the ported version by linking with TASP VSIPL[4], a CPU-based reference implementation of VSIPL, and used double precision throughout. The initial port of the simulation using double precision VSIPL met the correctness criteria on the first validation attempt, demonstrating the productivity of VSIPL.

When we compiled the simulation for single precision VSIPL, we found that criteria (2) and (3) were not met – yielding 2.9dB and 10^4 respectively. We determined that the source of error was quantization noise in the calculation of phase of return waveforms from adjacent clutter patches. In our test case, each clutter patch is less than a meter away from adjacent patches, and the scene center is dozens of kilometers away from the transmitter. As a result, at single precision, range detail was lost, which created large errors in return phase. We corrected this by changing range and phase calculations to use a far field approximation with a single double precision calculation for the entire scene, and single precision calculations for relative range and phase driving the creation of simulated returns. This change brought the VSIPL single precision implementation into compliance with our validation criteria.

IV. VSIPL Performance and Optimization

Once we had completed a validated single precision VSIPL implementation of the simulation, we linked with GPU VSIPL instead of TASP VSIPL with the goal of achieving minimum runtime consistent with correct output.

Our single precision VSIPL based implementation produced output data with identical correctness metrics using TASP and GPU VSIPL, demonstrating the functional portability of programming with the VSIPL API. The prototype implementation executed more slowly when linked with GPU VSIPL than it did when linked with TASP VSIPL. The test scenario executed in 162.5 seconds in MATLAB, 36.1 seconds in C using TASP VSIPL, and 150.2 seconds in C using GPU VSIPL executing on a nVidia 9800GX2 GPU. Based on prior porting efforts, we expected the GPU VSIPL-linked simulation to execute approximately 10x faster than the simulation linked with TASP VSIPL.

The same engineer that developed the prototype port of the simulation tuned the VSIPL implementation for use with GPU VSIPL over the course of about 2 months. Several optimizations with the greatest impact are described below.

The primary differences between TASP and GPU VSIPL with respect to application speed were related to effective use of the weakly consistent VSIPL memory model. Since TASP VSIPL is meant to be a maximally portable reference implementation of VSIPL, it uses only host system memory. As a result, admit and release operations are essentially free, and get and put operations are no more expensive than regular memory accesses. For platforms with distinct VSIPL and host memory spaces, such as GPU VSIPL, admit, release, get, and put operations are highly latent and must be minimized to reduce execution time. The simulation was reorganized to consolidate host access to simulation data, and to minimize admit and release operations.

The simulation contained several regions of small scalar or vector operations that were repeated over all or most points in the return data set. These operations were recast to increase dimensionality, reducing the overall total number of VSIPL function calls. GPU computing incurs a small overhead per kernel invocation. Casting operations with the higher dimensionality allows this overhead to be amortized over a greater number of operations, improving overall efficiency.

Several operations in the prototype were reorganized to allow consistency of row-major and column-major operations. GPU VSIPL attempts vectorized operations where a unit stride dimension is available, but TASP VSIPL does not. Providing a consistency of unit stride dimension improved the performance of the TASP-linked version due to cache coherencey, but had a greater impact on the GPU VSIPL-linked version due to increased vectorization and fewer hidden corner turns.

The results of this optimization effort are summarized in Figure 1, below. The uppermost, constant line represents the execution time of the original simulation; the red line shows the execution time of the ported simulation linked with TASP VSIPL; the green line shows the progress of the execution time of the GPU VSIPL-linked version of the simulation over the span of the optimization effort. The final runtimes achieved are summarized in **Error! Reference source not found.**, below.

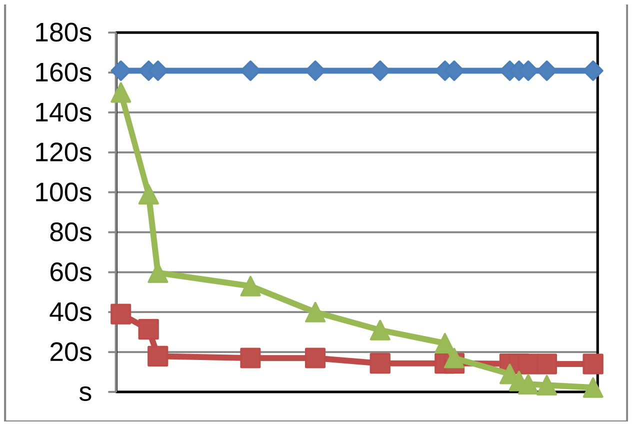

Figure 1 - Simulation Runtimes

V. CONCLUSION

After optimization by a relatively new VSIPL programmer with no prior optimization experience, our implementation achieved a speedup of over 70x relative to the original simulation, speeding the generation of realistic clutter for our test scenario from over 160 seconds to under 2.5 seconds. We found that VSIPL provides an intuitive and natural programming approach for porting and accelerating MATLAB based simulations. We found that VSIPL provides highly portable software that can be rapidly be redeployed onto different platforms, but that platform awareness is required to achieve optimum execution speed.

Table 1 - Final Run Times

Version	Runtime(s)	Speedup
MATLAB	**162.5**	**1x**
TASP VSIPL	**14.0**	**11.6x**
GPU VSIPL	**2.2**	**73.8x**

REFERENCES

[1] VSIPL Forum, *VSIPL API Specification 1.3*, http://www.vsipl.org/VSIPL1p3.pdf

[2] VSIPL Forum, *VSIPL Core Profile*, http://www.vsipl.org/coreprofile.pdf

[3] VSIPL Forum, *VSIPL Core Lite Profile*, http://www.vsipl.org/coreliteprofile.pdf

[4] Judd, R., TASP VSIPL, public domain software available for download at http://www.vsipl.org/software/tvcpp0p86.tgz