Optimizing An Innovative SAR Post-Processing Algorithm for Multi-Core Processors: A Case Study

Peter Carlston, Intel Corporation, peter.carlston @intel.com; Dave Murray, N.A. Software Ltd., davem @nasoftware.co.uk

Project Background

A major US defense contractor asked NA Software, Ltd. (NASL) and Intel Corporation's Embedded and Communications Group to multi-thread a new Synthetic Aperture Radar (SAR) post-processing algorithm developed by Dr. Chris Oliver, CBE, of InfoSAR. The goal was to understand how the performance of the original serial algorithm would scale when it was multi-threaded and run across 1-24 processor cores. Dr. Oliver's "SARMTI" algorithm is able to detect a large number of moving objects at full SAR resolution, determine the motion of all objects at about 10X the accuracy of Moving Target Indication (MTI) systems, and then automatically register their accurate position on the background SAR image. It does this from the raw SAR data image itself, so there is no need for a separate MTI radar system, with the manual registration and correlation of MTI and SAR images collected during different time periods that has often been required in the past.

We illustrate typical SARMTI performance below. Figure 1 shows a portion of a simulated SAR image of four moving targets against a realistic clutter background, with target positions demoted by the green ovals. Each target has a combination of the three motion components. Note how target movement both blurs and shifts its observed image. This well-known phenomena occurs because SAR systems measure low frequency information.

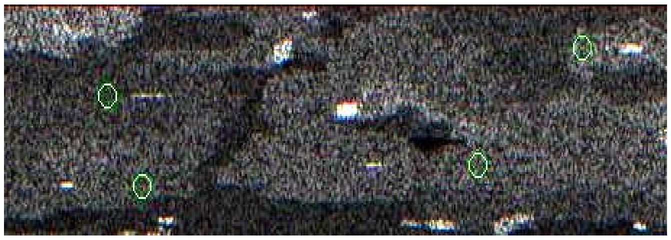

Figure 1: Part of a typical SAR image showing shifting and blurring of four moving targets.

Figure 2 shows the results after the SARMTI algorithm is used to process the same SAR image data. SARMTI utilizes all frequency information in the radar return, so it is able to measure and report across-track acceleration and along-track velocity, in addition to the across-track velocity measured in typical MTI systems. SARMTI has also automatically registered the actual locations of the targets on the background SAR image (red circles).

Note that the target positions are accurate to within about 1 pixel. The three associated motions are also measured within the predicted accuracy.

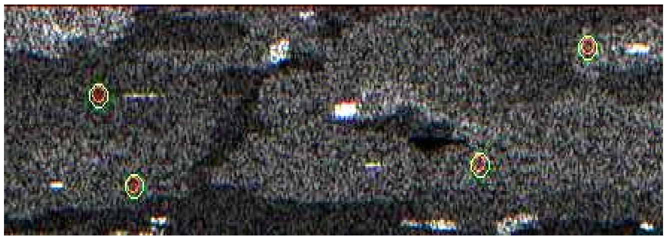

Figure 2: SARMTI has correctly determined the actual positions of the moving targets (red circles) from the same SAR image data

Method

We first used the GNU profiler 'gprof' to determine the percentage of time being spent in each function of the serial (non-threaded) code. The data showed that compressing complex data (FFT function calls and other compression) accounted for 64% of the overall time; 30% of the time was spent detecting targets. NASL then threaded the compression and detection portions of the code over a period of about six weeks. They have worked closely with Dr. Oliver for some time, and so were also able to optimize the algorithm during the threading process. The resulting overall algorithm structure is shown in Figure 3.

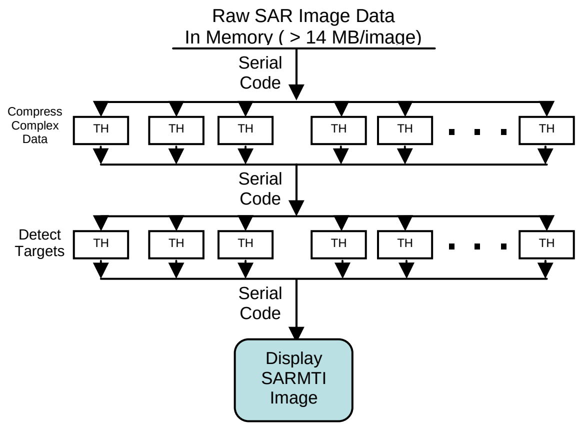

Figure 3: Conceptual diagram of SARMTI

The SARMTI post-processing algorithm begins after a raw SAR image (>14 MB) is loaded into memory. Some serial (non-threaded) processing is done at the beginning, then again during a small synchronization process in the middle and then at the end to display the image. But during the data compression and target detection phases, data tiles are processed independently on each core (represented by the TH[read] boxes.) NASL used processor affinity to assign specific threads to specific cores or processors; they also tried letting Linux dynamically place each process on the core with the least load. Analysis showed that core utilization was in fact quite balanced either way, with no significant variation in performance.

SARMTI contains many billions of FFT and math operations, so we next turned our attention to optimizing those algorithms. The original SARMTI code used FFT performance libraries from FFTW[1], so the Intel Math Kernel Library (MKL) "FFTW Wrappers" were substituted. In addition, the original C versions of complex vector add, conjugate, and multiply operations were replaced with the corresponding functions in the Intel MKL. Testing showed that speed up from utilizing MKL ranged from 14.7 to 18.4 percent.

After the initial threading and benchmarking on a system with 16 processor cores (four processors of four cores each), we used gprof to re-profiled the code in preparation for running it on a 24-core system (four processors of six cores each). gprof, however, shows only the time spent in each routine; when a routine is not at the end of a processing chain, this can be misleading. NASL therefore also timed the code stages. The results showed that the (serial) target record creation function was a good candidate for threading when the number of cores approached 24 core system used for the final benchmarking It turned out that threading this routine increased the speed of the SARMTI algorithm by 10% for 24 threads

Results With Simulated SAR Image

Table 1 summarizes the overall results of these efforts when the multi-threaded algorithm was run on the four-socket Intel rack mount server[2].

Test Case	Serial Time (sec)	Time in Seconds per Hardware Threads (Cores)						Speed Up 1T→ 24T	Total Speed Up 0→ 24T
		1	2	4	8	16	24		
1	85.4	37.0	18.8	9.9	4.9	3.6	2.2	16.8X	38.8X
2	120.2	47.9	24.6	12.9	6.7	4.8	3.1	15.5X	38.8X
3	104	35.6	18.3	9.6	4.8	3.5	2.1	17.0X	49.5X
4	166.2	67.1	33.9	17.8	9.6	6.6	4.4	15.3X	37.8X

Figure 4: Total SARMTI Performance Increase: 0-24 Cores (in seconds) (4x Intel® Xeon® Processors MP7460, 6 cores each)

Four scenarios were tested:

- Test 1: No targets; flat background
- Test 2: 10 targets; flat background
- Test 3: No targets; structured rural background
- Test 4: 10 targets; structured rural background

The overall speed up from the original serial code to the multi-threaded code running with 24 threads across 24 cores ranged between 37.5 and 49.5 times. As mentioned, the performance gains from the original, serial code to the multi-threaded version (1T) were realized by optimizing the algorithm during the multi-threading process and by using the Intel MKL library. The speed-up from multi-threaded code running on 1 core to 24 threads running on 24 cores was between 15.3 and 17.0 for the four test scenarios.

Figure 3 graphs how performance scaled per core.

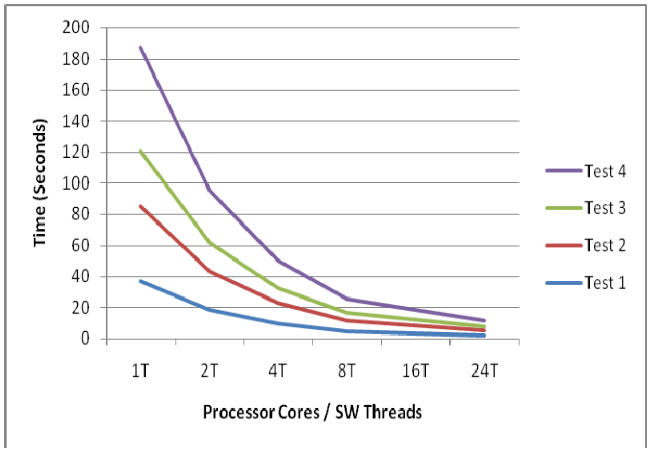

Figure 3: SARMTI Scalability graphed per number of cores

The slope of the curves shows that SARMTI scales quite well from one to eight cores. The rate of increase slows after eight threads/cores, but performance did continue to increase all the way out to 24 threads.

A number of areas were investigated to see if scaling per core could be increased. Neither front side bus nor memory bandwidth turned out to be issues. Cache thrashing was also not a problem since we had been careful to use localized memory for each thread. Early portions of the data compression stage are the only place where threads do process data from the same area of memory since they are all starting with the same input image. But changing the algorithm to make N copies of the image and then processing that unique memory block on each thread introduced overhead that actually increased execution times.

It turned out that some parts of the algorithm simply threaded more efficiently than others. Different portions of the algorithm use differently sized data sets, whose sizes change dynamically as the geometry changes. Some of the resulting data sets simply do not thread efficiently across 24 cores.

[1] See http://www.fftw.org
[2] Intel SFC4UR rack mount server, with 4 Intel Xeon Processors MP X7460 running at 2.66 GHz, with six cores each, and a 16 MB L2 cache shared across all six cores. The system's front side bus frequency was set at 1066 MHz The system was configured with 16MB of 667 MHz FBDIMMs. OS, Library, and compile flags information: Fedora Release 8 (Werewolf) for x86_64 architecture. Intel Math Kernel Library version 10.0. gcc 4.1.2 with compile flags -mfpmath=sse -msse4a -m64 -O4 -Wall