Channelization and Resampling Using a Graphics Processing Unit

Ambrose Slone, Paul Otto, Aqsa Kuraishi
Innovative Technology Office
Space and Geospatial Intelligence Business Unit, SAIC
14668 Lee Road, Chantilly, VA, 20151
ambrose.j.slone@saic.com, paul.r.otto.jr@saic.com, aqsa.i.kuraishi@saic.com

Introduction

Channelization and resampling of digital signals are common elements in a wide range of signal processing applications. Depending upon the requirements of the system, these basic tasks can consume a significant portion of the overall system computation budget, and can limit the data rates at which the system can operate.

In this paper, we present efficient implementations of a channelizer and resampler using a Graphics Processing Unit (GPU). By formulating our channelizer and resampler implementations in data-parallel forms, we are able to use the parallelism of the GPU to achieve a significant improvement in processing time compared with general-purpose CPU implementations of the same algorithms.

Software-Defined/Cognitive Radio Applications

Channelization and resampling are key components in a Software-Defined Radio (SDR) - Cognitive Radio (CR) system. These systems have become cost-effective and computationally viable in many applications, including public safety communications systems [1] and commercial wireless PCS devices. [2]

SDR systems leverage the computational power of modern hardware to perform in software much of the radio processing that has been traditionally been implemented using fixed hardware.

Cognitive radio systems utilize the dynamic abilities of SDR platforms to implement adaptive logic to optimize the performance of the radio in a given environment. CR systems perform real-time analysis of their working spectrum and make decisions based on this analysis to modify the radio parameters of its associated SDR.

For SDR/CR systems to characterize their environments, many channelize the wideband operating spectrum and perform occupancy analysis and identification of the narrowband signals within the environment. This task must be performed efficiently so the information can be used by the CR logic to make adjustments to the radio waveforms.

Data resampling is another processor-intensive task in SDR/CR systems. SDR/CR systems inherently must deal with a wide range of data rates in a dynamic manner. These resampling rates can often involve rational resampling rates that involve large integer up- and down-sample factors, requiring efficient processing in order to maintain real-time performance.

Polyphase Processing

A common approach to performing channelization and resampling is by utilizing a polyphase filterbank implementation (see [2], [3]). For a filterbank of Q equally-spaced channels, each at a rate of P/Q times the input sample rate and using a FIR filter of length KQ, a polyphase implementation is more efficient by a factor of :

$$\frac{KPQ(\log(K)+\log(Q))}{K(\log(K)-\log(P))+P\log(Q)}$$

over a simple tune-upsample-filter-downsample approach. (for cases where Q > P, K > P, and where filtering is done using FFT processing).

For a polyphase implementation of a rate (P/Q) resampler using a filter length of (KQ), the improvement is:

$$\frac{P(\log(K)+\log(Q))}{\log(K)-\log(P)}$$

compared with a simple upsample-filter-downsample approach. (assuming here Q > P, K > P)

The polyphase formulation of channelization and resampling also lends itself well to a parallel processing implementation. (Figure 1) The core processing stages of a polyphase filterbank can be preformed as large blocks of parallel FFTs, allowing the use of highly-optimized software libraries.

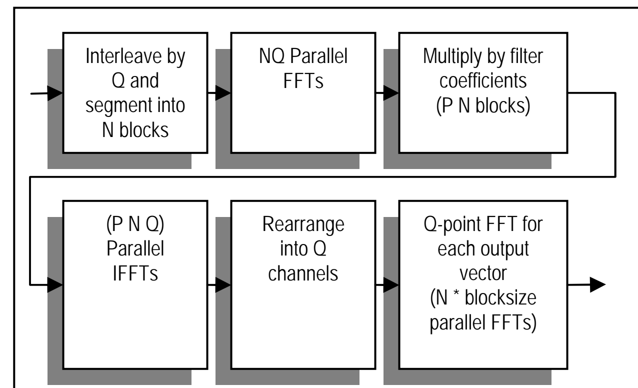

Figure 1. Parallel Processing Chain for Q-channel Channelization at Oversampled Rate P

GPU Processing

For all of our GPU development in this paper we utilized a NVIDIA 8800 GTX (see [5] for specifications) running under 32-bit Red Hat Enterprise Linux 4. We used the NVIDIA CUDA toolkit [6] and the CUFFT 1.1 library [7] to create our GPU channelizer and resampler implementations.

We wrote custom GPU kernel code using CUDA to perform our data conversions (integer/signed/endian), and to perform parallel data rearranging and multiplexing

between the FFT stages. We also wrote a kernel for performing a tailored parallel binary reduction operation for the resampler application.

CPU Processing

For comparison, we created reference implementations of both the channelizer and resampler using CPU code. We implemented these libraries in C/C++ using GCC 3.4.5 under 32-bit Red Hat Enterprise Linux 4 (kernel 2.6.9-34) running on a Dell Precision 690 workstation with two dual-core Intel Xeon 2.33 GHz processors (5140 Woodcrest, 4MB L2 cache), and 2GB of system memory.

We used single-precision, SSE-enabled, multithreaded FFTW (version 3.1.2) libraries built from source to perform our CPU FFT computations. Whenever appropriate, we used parallel FFTW plans to maximize our CPU FFT performance. We configured FFTW to use four threads for generating its FFT plans in order to match the number of cores on our system.

Test Cases

As our baseline test case, we used a 16-bit, signed complex data file stored on disk, containing 2^{28} samples (1GB file size). We ran a range of channelization tests, with the number of channels processed ranging from 2-1024.

Because our processing is heavily I/O bound by the file writing process, our timing results presented here represent no writing of the resulting data to disk, thereby highlighting the relative performance advantage of the GPU. These numbers do, however, include reading data from the file and transferring data to and from the GPU device.

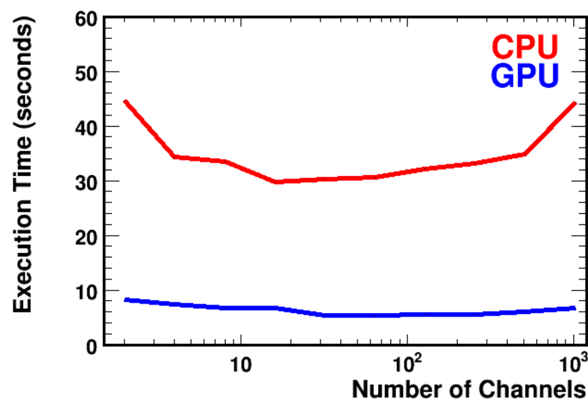

Figure 2. Channelization Time for 1GB 16-bit Complex Signed Data File, Filter Length = (number channels * 63)

Results

From Figure 2, we can see that the GPU implementation was typically on the order of 4-10 times faster than the associated CPU polyphase implementation across a wide range of channelization levels.

Figure 3 shows timing results for a range of decimation rates for the polyphase resampler. In this example, the resample ratio was a simple decimation by a factor of 1/N. These results show that the GPU processing advantage over the CPU grows as the decimation rate increases, with a 10x advantage for a decimation rate of 1024. Even for lower decimation rates, however, the GPU outperforms the CPU by at least a factor of two.

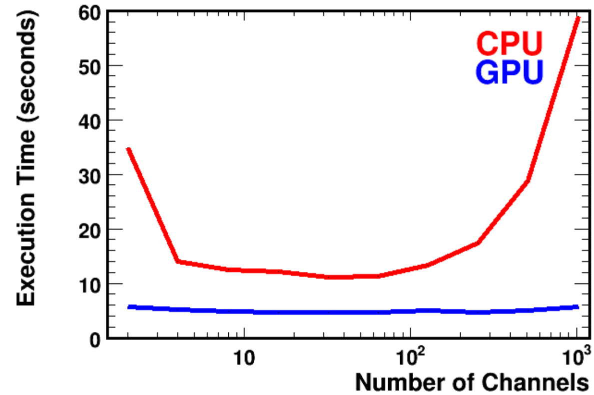

Figure 3. Resample Time for 1GB 16-bit Complex, Signed Data File, Filter Length = (decimation rate * 63)

Conclusions

Based on our results, using a GPU in channelization and resampling applications can provide a significant improvement in processing time. This advantage is especially pronounced for higher channelization and decimation rates.

To further take advantage the GPU's parallelism, we plan to extend the processing on the channelized data while still on the device to include filtering, subspace processing, equalization, and demodulation. We have implemented GPU approaches to some of these signal processing tasks already and need only apply them in parallel to the channelized streams.

References

[1] *Use Cases for Cognitive Applications in Public Safety Communications Systems – Vol. 1 Review of the 7 July Bombing of the London Underground,* http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-07-P-0019-V1_0_0.pdf

[2] *Business Model for Wireless PCS*, SDR Forum, 2003, http://www.sdrforum.org/pages/documentLibrary/documents/SDRF-03-P-0001-V1_0_0_Business_Case.pdf

[3] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[4] F. J. Harris, C. Dick, and M. Rice, "Digital Receivers and Transmitters Using Polyphase Filter Banks for Wireless Communications," *IEEE Transactions on Microwave Theory and Techniques*, Vol. 51, No. 4, April 2003.

[5] http://www.nvidia.com/page/geforce_8800.html

[6] *CUDA Programming Guide, Version 1.1*, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[7] *CUDA CUFFT Library User's Manual, Version 1.1,* http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf

[8] *FFTW3 User's Guide*, http://www.fftw.org/fftw3.pdf