

Implementing Hilbert Transform for Digital Signal Processing on Epiphany Many-Core Coprocessor

Kyle L. Labowski
Technical and Project Engineering, LLC
U.S. Army Research Lab
Aberdeen Proving Ground, MD
kyle.l.labowski.ctr@mail.mil

Patrick W. Jungwirth
U.S. Army Research Lab
Aberdeen Proving Ground, MD
patrick.w.jungwirth.civ@mail.mil

James A. Ross
U.S. Army Research Lab
Aberdeen Proving Ground, MD
james.a.ross176.civ@mail.mil

David A. Richie
Brown Deer Technology
Forest Hill, MD
driche@browndeertechnology.com

Abstract—The Adapteva Epiphany MIMD architecture is a scalable 2D array of RISC cores with a fast network-on-chip (NoC) for parallel processing. The work presented here discusses the suitability of the architecture to handle software defined radio (SDR) applications such as Finite Impulse Response (FIR) filters. This paper discusses implementation of the Hilbert filter through using the COPRTHR 2.0 SDK which includes Pthread-like interface for offloading the thread function. We present timing and performance results for our implementation.

Keywords—2D RISC Array; Software Defined Radio; Parallella; Adapteva Epiphany; Hilbert Filter

I. INTRODUCTION

The Adapteva Epiphany MIMD architecture is a multicore parallel computing mesh network-on-chip (NoC). Each mesh node consists of a RISC CPU, a DMA Engine, 32 KB of shared local memory, and a mesh network interface. The entire architecture is scalable to thousands of cores in a single chip [1]. The 16-core Epiphany III coprocessor was integrated into the Parallella minicomputer platform [2]. The Parallella board combines the Epiphany coprocessor with a dual core Advanced RISC Machine (ARM) Central Processing Unit (CPU) in a Zynq 7010, and asymmetric shared-memory access to off-chip global memory.

The Epiphany III peak single precision performance is 19.2 GFLOPS at 0.5 watts power consumption. Although this performance is low compared to CPU and Graphics Processing Unit (GPU) architectures, the low power consumption and the architecture's scalability make it a promising architecture for highly parallel applications with low power requirements. The roadmap for the Epiphany architecture projects scaling the chip to greater than 1,000 cores in the near future. One of the drawbacks of many current architectures, such as GPUs, is the amount of power required to run the hardware. With the rise of mobile computation requirements, for example, software defined radio (SDR) or neural networks for self-driving cars, computations need to be fast while maintaining power efficiency. A reasonable speedup at a modest power increase may far outweigh the cost

of the large power budget for a larger speed increase.

To take advantage of the system architecture requires a programming model suitable for the Epiphany architecture. The COPRTHR 2.0 SDK from Brown Deer Technology provides a lightweight interface and Pthread syntax [3]. Threaded MPI was written specifically for the Epiphany architecture. The limited resources available on the chip prevented a full process image or program from executing on each core as in traditional MPI. The device also must be accessed as a coprocessor, so coprocessor offload semantics are used to launch concurrent threads. Conventional MPI semantics are then used for inter-thread communication.

The fastest speedup is achieved when the data and code reside on the executing core. There is a performance penalty for accessing data off the Epiphany chip. Approaches that maximize the on-chip processing while minimizing the communication off-chip will see the largest speed increase. Data reuse and inter-core communication to minimize off-chip resources are central to programming for this architecture.

Power efficient high speed processing is ideal for mobile applications that rely on data processing. Software Defined Radio (SDR) is a rapidly growing application that relies on real-time processing of radio signals, often on handheld devices or in mobile platforms. SDRs provide the flexibility of software reconfiguration for signal processing functions. Prior to SDR, changing frequencies involved physically changing hardware, or maintaining redundant hardware to span the entire frequency range of interest. This SDR work focuses on implementing a Hilbert transform on the Epiphany processor. Lessons learned will feed into future work on more complex DSP algorithms and applications.

II. THREADED MPI

Threaded MPI is a compact version of MPI for the Epiphany coprocessor architecture [4], which we have previously demonstrated for several image processing tasks [5]. Threaded MPI uses coprocessor semantics to access the resources on the Epiphany RISC cores. Each thread has to

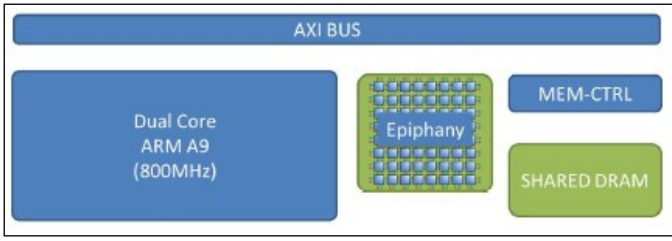


Fig. 1. Parallella High Level Architecture

operate under strict resource constraints as each core only has 32 KB for instructions and data. There is not enough space to contain a full process image or program. The result is that unlike traditional MPI that is launched from a command prompt, threaded MPI is launched from a host program operating on the platform CPU. Coprocessor offload semantics are used to launch concurrent threads, and conventional MPI semantics are used for inter-thread communication.

The host code is responsible for explicit device control and memory management tasks. The host code makes the call to the `coprthr_mpiexec()` function, which launches the parallel code in a fork-join model. The calls to the MPI functions are embedded within a larger application that can be running other routines on the platform CPU. The MPI code that runs on the Epiphany architecture is written as a thread in a separate kernel, and that uses **Pthread** semantics for passing arguments. The host code can make the call to `coprthr_mpiexec()` multiple times within the main code, and the code can employ multiple MPI kernels [4].

Another distinction between traditional MPI and MPI on the Epiphany architecture is the memory buffer. Normally large buffers are used for message queues tuned for the specific host and network parameters. The Epiphany RISC cores only have 32 KB per core for program instructions, local storage, and message buffers. There is a much larger shared memory region in global DRAM of 32 MB, but there is a high access penalty due to the access latency associated with off-chip resources. The access penalty prevents the use of DRAM for inter-core MPI buffers. Fig. 1 shows a simplified high level diagram of the Parallella architecture [2].

The Threaded MPI implementation was developed based on the Epiphany support provided by the COPRTHR SDK. The COPRTHR run-time supports coprocessor device management and offloading threads to the coprocessor. Threaded MPI supports a minimal subset of the MPI standard. Support was provided for basic initialization, the creation of Cartesian topologies, blocking send/receive pairs, and a combined send/receive/replace call. The **MPI_Send** and **MPI_Recv** calls are implemented as tightly coupled zero-copy communication routines. The Epiphany architecture supports direct read and write transactions (write has priority over read) from one core to the local memory of another.

A recent update to the COPRTHR 2.0 library included improvements to the algorithm that initializes the coprocessor and transfers the kernel program to the Epiphany processor cores. Initial testing with the previous library showed an initialization time of approximately 160 milliseconds. The

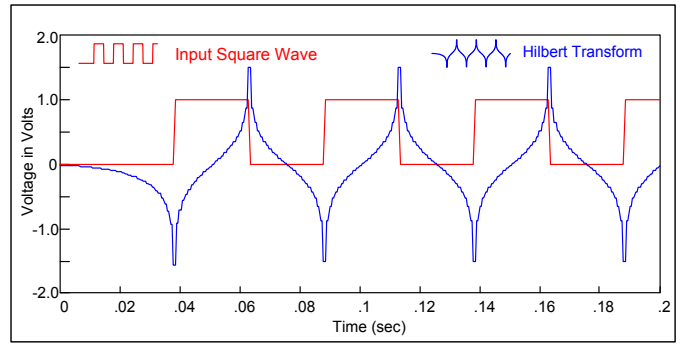


Fig. 2. Square Wave and Hilbert Transform

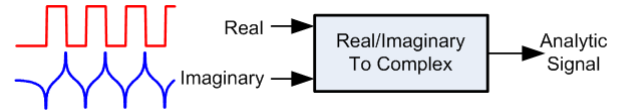


Fig. 3. Analytic Signal

$$x(t) = \cos(2\pi f_1 t) \cdot \cos(2\pi f_2 t) = \frac{1}{2} \cos(2\pi t(f_2 \pm f_1)) \quad (1)$$

$$u(t) = e^{j2\pi f_1 t} \cdot e^{j2\pi f_2 t} = e^{j2\pi(f_2 + f_1)t} \quad (2)$$

$$v(t) = e^{j2\pi f_1 t} \cdot e^{j\phi} = e^{j(2\pi f_1 t + \phi)} \quad (3)$$

slow initialization time was due to the serial transfers of the kernel program to each core from the main memory. The improved loader now copies the kernel program to a single core from main memory, then performs a tree loading algorithm between cores. Inter-core communication is significantly faster, resulting in initialization times of nearly 30X faster than previous version of the library. The new library also allows the use of persistent threads to constantly look for data to run on the coprocessor, but these improvements were not exploited at the time of this paper.

III. REVIEW OF HILBERT TRANSFORM

The real signal, $x(t)$, in (1) is a product of cosine terms. The result consists of sum and difference frequencies. For a complex exponential, $u(t)$, in (2), the product only contains one term, the sum frequency, $f_2 + f_1$. The complex signal, $v(t)$, shows phase can be shifted with a multiplication. Complex exponentials only require multiplication for frequency translation and phase shift. In real functions, frequency translation and phase shift are much more complicated.

Fig. 2 shows a square wave function and its Hilbert transform. As illustrated in Fig. 3, setting the real input equal to the square wave and the imaginary input equal to the Hilbert transform, we create an *analytic* signal. An analytic signal has the same frequency translation and phase shift properties as Equations (2) and (3). Fig. 4 shows the power spectral density of a square wave. Note the real signal has a symmetric spectrum. Fig. 5 shows the analytic signal

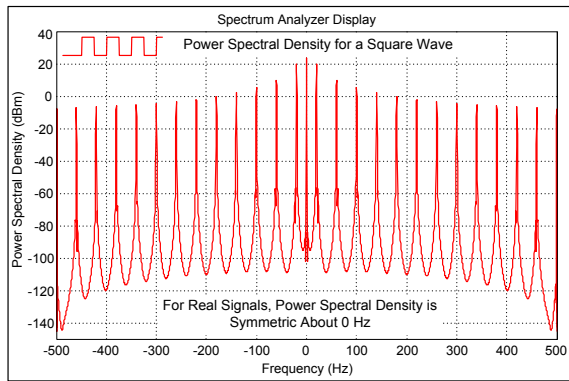


Fig. 4. Square Wave Power Spectral Density

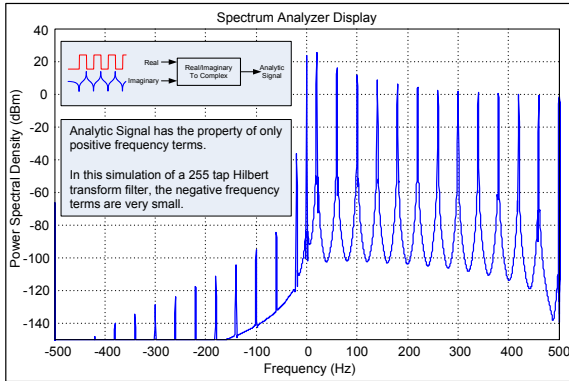


Fig. 5. Analytic Signal's Power Spectral Density

computed using Fig. 3. Note the spectrum only has positive exponential terms just like (2).

IV. HILBERT FILTER ALGORITHM ON EPIPHANY

We chose a Finite Impulse Response (FIR) filter for our initial implementation on the Epiphany III architecture. As described in section III, the Hilbert transform is used to convert a real signal to an analytical signal. Analytic signals have the same frequency translation and phase shift properties as Equations (2) and (3). Our implementation is based off of the code from [6] and the GNU radio libraries [7] [8].

There are two main approaches to the implementation of the Hilbert Filter in software. One approach is a direct convolution based approach. Since convolution in the time domain is multiplication in the frequency domain, the other approach is a Fast Fourier Transform (FFT) based approach. Digital Signal Processing (DSP) approaches often use the convolution based approaches for high speed streaming data. FFT approaches are faster for offline processing where additional zero padding can be added to take advantage of calculation symmetry. These problems often require an input length that is a multiple of two raised to the power of N. Active data streaming with an FFT based technique requires reading a portion of the data into a buffer, forward transforming the data into the frequency domain, multiplication by the filter transfer function, inverse transform back to the time domain for further processing, and then pulling in more data. In contrast, convolution based approaches require reading in data into a buffer, convolution

$$f[n] \otimes h[n] = \sum_{m=-\infty}^{\infty} f[m] h[n-m] \quad (4)$$

$$h(t) = \frac{1}{\pi t} \quad (5) \quad \mathcal{H}[g(t)] = g(t) \otimes \frac{1}{\pi t} \quad (6)$$

$$\mathcal{H}[\text{rect}(t)] = \frac{1}{\pi} \ln \left[\frac{2t+1}{2t-1} \right] \quad (7)$$

$$a(t) = x(t) + j\mathcal{H}[x(t)] \quad (8)$$

TABLE I. Sampling Rate and Bit Rate for Applications

Application	Sampling Rate (Hz)	Bit Rate (Kbps)
Telephone [12]	8,000	96
Audio CD [12]	44,100	706
720p Video [10]	30	1500 to 4000
1020p Video [10]	30	3000 to 6000

with the filter kernel, which is a multiply and add step, and then pulling in more data [9]. Discrete convolution is shown in Equation (4). The convolution filter kernel for the Hilbert transform, $h(t)$, is shown in Equation (5). Equation (6) defines the Hilbert Transform of a function, $g(t)$, as the convolution of $g(t)$ with the filter kernel $h(t)$ [10]. The Hilbert transform of a rectangular function is found in Equation (7).

The convolution summation is performed over the length of the filter kernel (same as the number of FIR taps in low pass FIR filter implementation). The output of the convolution is the imaginary part of the analytic signal (see III). The analytic signal, $a(t)$ in (8) shows the real part is $x(t)$ and the imaginary part is the Hilbert transform of $x(t)$ (see Fig. 3).

V. APPROACH

The goal of this SDR application study was to evaluate the performance of a Hilbert transform implemented on an Epiphany coprocessor. SDR applications include low quality voice transmission to High Definition video transmission. Some example applications and data rates are provided in Table 1. For this study, the main processor will generate a square wave signal and send it to a memory location in DRAM that is accessible to the Epiphany chip. The main code also initializes the coprocessor and sends the kernel program to the cores for execution. From there, the kernel code reads the data from DRAM, writes the filter coefficients, and performs the convolution. A successful result means that the entire end-to-end processing time must be faster than the required data rate for each application. So in other words the algorithm needs to process X samples per second and return data every Y seconds. The processing time includes the time it takes to stream data into the application, process it, and stream the data from the Epiphany chip.

A. Hilbert Transform Kernel

The implementation of the Hilbert Filter focused on streaming data in the context of the available `coprth` library

and the Parallella board. The main processor created the input signal and wrote it to DRAM where the Epiphany coprocessor can access the data. The Hilbert Filter Kernel running on each core copied the input signal from the DRAM to on-core memory, calculated the Hilbert Filter coefficients once per thread, performed the convolution, and finally wrote the data back to DRAM. An identical kernel was run on each core. MPI group semantics handled coordination of the data processing between cores. The input data was split equally amongst the available cores.

Passing data between cores was not implemented as part of this algorithm even though the data does overlap between cores. The convolution of the current point in time relies upon data from the successive time steps for the length of the filter. Longer filter lengths therefore require more data overlap from future time steps. Larger filter coefficient arrays are used to increase the accuracy of the resulting signal, reduce ripple in the filter's passband, and other benefits that depend upon the signal processing algorithm. Passing data between cores, especially for longer filter lengths, could reduce run time by reducing data access to off-core resources. On-chip data transfer is significantly faster than off-chip transfers.

The MPI send and receive functions were not utilized as part of the code, however future algorithms could make use of them to decrease processing overhead. For now, each core reads in the additional input signal data directly. The Hilbert Filter coefficients were also calculated on each core to remove the overhead required to transfer the coefficients to each core. The convolution is a series of multiply accumulates on the input signal. The **for loop** in the kernel code that executed the convolution was unrolled to execute four multiply accumulates per loop.

The **MPI_exec()** utilizes a fork and join approach to program execution and thread coordination. The main code initiates the kernel code execution and waits for the threads to complete. In practice this would be a continuous loop during data streaming, but for this study we only looked at a single execution. For testing purposes we looked at the throughput and timing results for several different input signal lengths and filter lengths. Each core was restricted to reading in 1914 floating point numbers to prevent loading too much data to each core. If the buffer size in DRAM was larger than 30,625 floating point numbers then the kernel looped over the entire signal until it was complete. Although this eliminates reading and writing data from the main processor to DRAM, this focuses on the timing of the Epiphany chip and assumes data is streaming to buffers that can be read in directly by the Epiphany chip. The current fork and join model controls the thread execution via the main code, which means every iteration comes back to the main processor to join together the data. Recent updates allow for the control to move to one of the cores, which opens up the persistent thread approach. A persistent thread approach would allow one of the Epiphany cores to operate on a work queue and spawn new threads as data algorithm, and will be the subject of a future paper.

The per core input data stream size was fixed to 1914 floating point values per iteration. This ensured there was

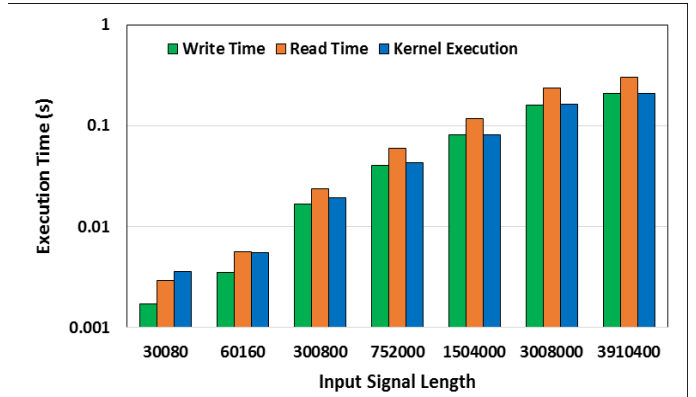


Fig. 6. Program Execution Times with Filter Length of 40 taps

enough space for the output buffer, the kernel program, and the filter. The filter lengths varied between 40 and 800 floating point values, which ensured that at least one full filter length fit on each core without overrunning the internal memory. Fig. 6 shows a breakdown of the timing for the filter execution with a filter length of 40 taps and increasing input signal length. The write and read times refer to data access from the global DRAM, and do not include the time for each core of the Epiphany chip to copy data to on-core memory. The read and write times are also for sending all the data at once, and not separate read/write operations. The MPI execution time refers to the execution of the kernel program on the Epiphany chip and includes the time to read and write data from DRAM to the local on-core memory for each processor core of the Epiphany chip, the time to calculate the filter coefficients, and the convolution time. Any input signal lengths greater than 30,625 floating point numbers required multiple transfers between DRAM and the Epiphany chip.

Fig. 6 shows that with the small filter length of 40 taps, as the problem size increases the time it takes to read the data values back to the main processor starts to dominate the run time. If we look only at the MPI execution time, and assume the buffer contents are updated independently of the kernel, this corresponds to a sample rate of approximately 8.5 MHz for the input signal with a buffer size of 30,000 samples. With a buffer of 4 million samples, the sample rate is 16.6 MHz. The difference in sample rate with increasing buffer size is partially due to some of the overhead involved in the initialization of the code. There is a fixed amount of time spent transferring the kernel to all of the cores, setting up variables and MPI communication variables, and initializing the filter coefficients. The larger buffer size allows the kernel to set up the variables once, then loop over reading in the data from DRAM, convolving the signal with the filter, and write out to RAM. Using a small filter size, the calculation intensity of the kernel program is also relatively low. Fig. 6 shows that the bottleneck for the code in this configuration is the read and write times for the input and output data from DRAM to the main chip. Fig. 7 compares the execution time for the convolution on the host ARM processor (in red), the Epiphany coprocessor (in blue), and the start-to-finish Epiphany execution plus data transfer (in green). The red line represents execution time on the host ARM processor only, and does not include the coprocessor. The blue Epiphany line includes the

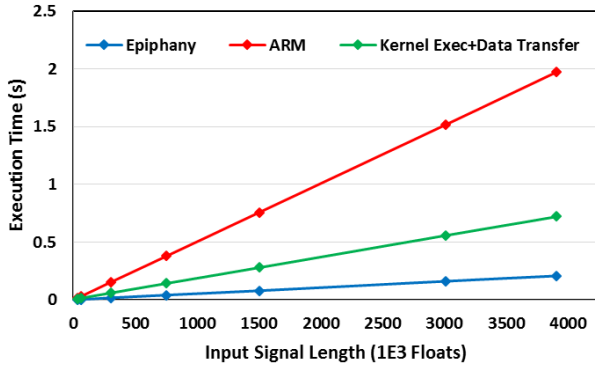


Fig. 7. Convolution Execution Time for Filter Length of 40 taps

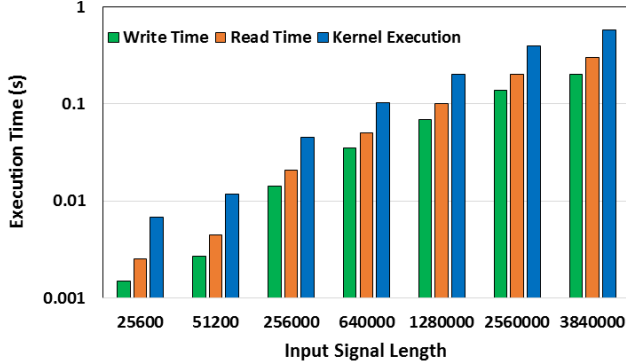


Fig. 8. Program Execution Times with Filter Length of 800 taps

time for data transfers from DRAM to on-core memory in addition to the convolution. It also represents the time for one core to execute the code sequence on its portion of the input signal, with the entire signal distributed across all 16 cores. The green line is the total execution time using the Epiphany co-processor. The timing for the green line includes reading and writing data from the main chip to the DRAM and the full execution of the kernel program, including reading and writing data to the local cores on the Epiphany. The green line was included for reference to show the entire cost of using the Epiphany cores vs. processing the data on the host ARM processor only.

Fig. 8 shows the timing breakdown using a filter length of 800 taps. Note that the input signal sizes are slightly different lengths than those in Fig. 6 due to the increased filter length and the limited memory available on each core of the Epiphany co-processor. The greater filter length increases the computational intensity of the convolution because each point is a summation over the entire filter length. There is a significant increase in the kernel execution time. The read and write times for transferring data from the main core to DRAM are similar to those shown in Fig. 6. Unlike the smaller filter length, the larger filter length spends more time in the actual convolution calculation. In this case the actual data processing dominates the calculation time. The sample rate is 3.8 MHz for the 25,600 floating point buffer size, and 6.6 MHz for the 3.8 million floating point buffer size.

Fig. 9 compares the execution time for the convolution on the host ARM processor in red, and the convolution plus the read and write time to and from on-core memory for the

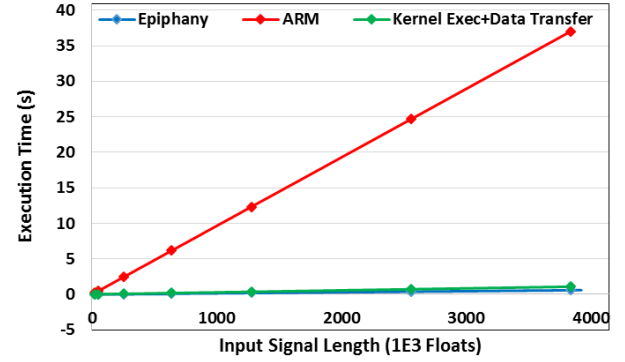


Fig. 9. Convolution Execution Time for Filter Length of 800 taps

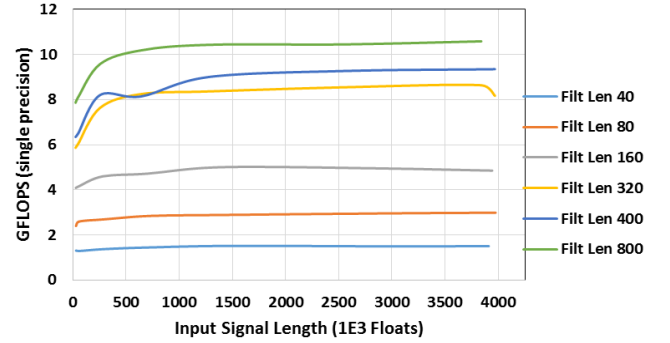


Fig. 10. GFLOPS vs. Signal Length for Filter and Convolution on Epiphany

Epiphany coprocessor in blue. Just as in Fig. 7, the timing for the green line includes reading and writing data from the main chip to the DRAM and the full execution of the kernel program. Comparing the blue and green lines, there is very little overhead from transferring data onto DRAM from the main chip. The increased complexity of the calculation caused the ARM processor's execution time to exceed 37 seconds.

Fig. 10 shows the performance in GFLOPS for the calculations on the Epiphany coprocessor for different filter lengths. The longer filter lengths mean that more time is spent on multiply accumulates per point in the output. The throughput decreases because more time is spent calculating each point in the output. The peak performance topped out at 10.5 GFLOPS. As shown in the graph, the peak performance is reached relatively early for each filter length as the input signal length increases. Beyond 500k floating points most of the peak performance lines level off. This calculation does include the overhead for reading and writing data to the coprocessor core. As the filter length increases the read/write overhead becomes a smaller percentage of the overall time. Reaching the peak performance for a given filter length therefore doesn't require a large buffer size. A large buffer size means more data, and depending on the sampling rate will affect the delay between processing buffers.

VI. DISCUSSION

The goal of this paper was to explore the Epiphany architecture in the context of real-time signal processing. Although the algorithm was simple, if the buffer read/write time and the computational time for this simple application

exceed acceptable standards, then further research should focus on minimizing the data flow overhead.

For real-time signal processing we need to determine the minimum acceptable time delay between receiving a signal, processing it, and sending it to the end device, such as a speaker. According to ITU recommendation G.114 the maximum delay in voice transmission must be kept below 150 ms [11]. An interesting side note is that prior to the recent library update, the initialization time for running a program on the board was 160 ms. This was due to serial transfer of the program kernel from the main processor to each core. The latest update now loads the kernel program to a single core, then transfers the program from core to core via a tree loading process resulting in a speed increase for program initialization of at least 30 fold. Table 1 shows the sample rate for telephone is 8 kHz, and the bit rate is 96 kbps. To set a minimum goal, our code has to process the data at a rate of 12000 bytes/s, with less than 150 ms between output samples. This means the buffer size has to be at least 1,800 bytes long, and return every 150 ms. Our problem sizes were significantly larger, but even the filter length of 800 taps returned data in 3.1 ms with a buffer length of 51,200 bytes. This was the minimum acceptance criteria for applicability and we have exceeded the data rate and return rate for the samples with ample time for further processing. Follow on work will be to implement a more computationally complex DSP task such as Polyphase filtering to see if memory constraints on the Epiphany, computational intensity, and data transfer rates will still allow us to produce data at a rate that exceeds 1,800 bytes every 150 ms.

Looking at an application that involved more complex processing and larger buffer sizes, video processing involves rapidly decoding large amounts of data from a buffer. The computational intensity involved in the video processing exceeds our simple test scenario, but we are interested in the data rates and buffer sizes required to tackle this problem. Video is delivered at 30 frames per second for live action sports. This translates to a buffer delay of 33 ms for higher quality video. The bit rate depends upon the resolution of the video. A resolution of 720p would require between 1.5 to 4 Mbit/s [13]. This would mean that 6,187.5 bytes are processed every 33 ms on the low end of that bit rate. The 800 tap filter processed 51,200 bytes in 3.1 ms, which is approximately one-tenth of the time allotted for video processing. It is difficult to draw any conclusion from this comparison because the computational intensity of decoding the video data could exceed either the memory on the Epiphany or the 27 ms that remain before we exceed the data processing requirement. This comparison only establishes that more research into complex encoding and decoding tasks on the Epiphany is warranted since the limited bandwidth on and

off the Epiphany has not exceeded the allotted processing time.

VII. FUTURE WORK

Other signal processing applications involve the processing of multiple bands concurrently from a single time domain input. In the case of a Polyphase Filter, for example, one wide-band time domain signal containing many channels is processed and separated into individual channels that purposely contain noise. When a specific channel is chosen, all of the channels are recombined so that the noise cancels out everything but the desired channel. The algorithm is more computationally complex than the Hilbert transform, and it is inherently multithreaded. The next step is to push the bounds of performance for the Epiphany chip to see if it can return data fast enough to perform complex real-time analysis for SDR.

Signal encoding and decoding is another computationally intense task in the realm of SDR. Complex signals need to be encoded and decoded in real-time with minimal delay. The Epiphany architecture may be able to perform this complex processing in real-time with minimal delay.

VIII. REFERENCES

- [1] "Epiphany Architecture Reference rev 14.03.11", Adapteva, 2013.
- [2] "Parallella Reference Manual rev 14.09.09", Parallella.
- [3] D. Richie, "Coprth API Reference", Brown Deer Technology, 2014.
- [4] J. Ross, D. Richie, S. Park, D. Shires, "Parallel Programming Model for the Epiphany Many-Core Coprocessor using Threaded MPI", ISCA 15, Third ACM International Workshop on Manycore Embedded Systems, 2015.
- [5] J. Ross, D. Richie, S. Park, "Implementing Image Processing Algorithms for the Epiphany Many-Core Coprocessor with Threaded MPI", HPEC '15, 19th Annual HPEC Conference, 2015.
- [6] J. Mietus, Hilbert Transform Code, [Online], Available: <https://www.physionet.org/physiotools/apdet/apdet-1.0/ht.c>, Accessed March 2016.
- [7] GNU Radio Repository, [Online], Available: <https://github.com/gnuradio>, Accessed March 2016.
- [8] GNU Radio Website, [Online], Available: <http://www.gnuradio.org>, Accessed March 2016.
- [9] M. Chan, "More Advanced Digital Signal Processing Techniques", Queen's University, [Online], Available: <http://www.physics.queensu.ca/~phys352>, Accessed March 2016.
- [10] F. Kschischang, "The Hilbert Transform," University of Toronto, 2006.
- [11] S. Smith Ph.D, "The Scientist and Engineer's Guide to Digital Signal Processing", California Technical Publishing, 2011.
- [12] "ITU Recommendation G. 114: One-way Transmission Time", International Telecommunication Union, 2003.
- [13] "Live encoder settings, bitrates and resolutions", [Online], Available: <https://support.google.com/youtube/answer/2853702?hl=en>, Accessed April 2016.