BLACKLYNX

BlackLynx Software Defined Radio Application Analysis and Acceleration APIs

BlackLynx, March 2019, v1

Overview and Problem Statement

This document describes a methodology for accelerating software defined radio (SDR) GNU Radio processing blocks when mission needs require high performance operation. The BlackLynx acceleration APIs used are also presented.

GNU Radio is a free and open-source SDR toolkit for signal processing. It provides a development environment centered around signal processing "blocks" that may be interconnected in various ways to create or simulate signal processing systems. Xilinx SDAccel is a development and runtime environment for implementing OpenCL applications on Xilinx FPGAs. Utilized in combination, BlackLynx transparently offloads GNU Radio block processing to FPGA implementations, seamlessly integrating inputs and outputs with the rest of a standard CPU-software based GNU Radio waveform. By utilizing an FPGA, GNU Radio blocks that interface with SDAccel can implement high performance signal processing blocks that would not be possible to implement purely in software.

The applicability of low-density parity-check (LDPC) and fast Fourier transform (FFT) operations for acceleration are discussed to highlight the importance of adopting a best-practices-approach of profiling, CPU optimization, and finally FPGA optimization, but always within the confines of a larger SDR framework, based on mission need. Our specific mission objective for purposes of the examples in this document is to achieve consistent steady-state sub-millisecond performance for the identified signal processing blocks using a single 1U Dell R640 COTS server, with a single ½-height, ½-width "low profile" PCIe FPGA acceleration card populated. The acceleration card leverages a Xilinx SDAccel-capable VU9P FPGA, which is also the same class of FPGA that is leveraged in the cloud on AWS F1 instances.

We highlight that LDPC encoding operations can be meaningfully accelerated in CPU fabric alone to sub-millisecond performance with proper software implementations, but FPGA acceleration is required to achieve consistent steady-state sub-millisecond performance for LDPC decoding. As will be shown, the FPGA accelerated implementation is able to be seamlessly integrated into existing GNU Radio waveforms and graphical flowgraphs without end users having to be aware of any of the underlying FPGA implementation details.

We also briefly discuss FFT, to demonstrate that some existing GNU Radio FFT blocks require no further acceleration since the default/stock GNU Radio block performance is already consistently sub-millisecond, as determined from initial best-practices profiling.

BlackLynx Technology Connected to Mission Architecture

Figure 1 below depicts a high-level view of how a BlackLynx-enabled architecture fits into a larger mission environment. An arbitrary external transceiver topology feeds or is fed by our solution using digitized data streams (such as, perhaps, by simple multicast protocols) across a standard 1GbE, 10GbE, 40GbE or 56Gbps FDR InfiniBand network topology. BlackLynx-enabled COTS hardware, such as a variety of popular Dell servers, simultaneously act as sinks or sources as

needed and as defined by mission requirements leveraging a variety of BlackLynx-accelerated SDR tools, such as (but certainly not limited to) GNU Radio.

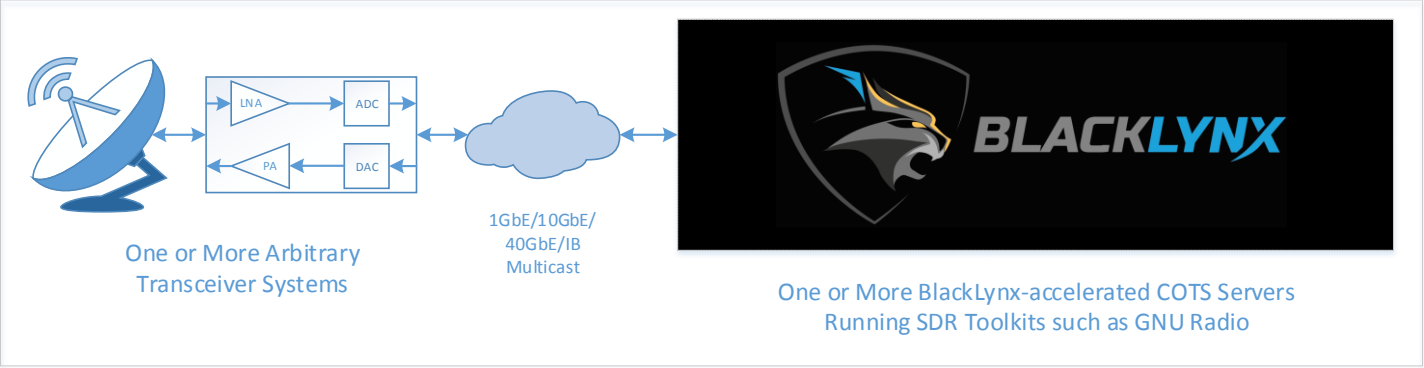

Figure 1: System Architecture of BlackLynx-Enabled Arbitrary Mission Waveform Analysis

BlackLynx Accelerated LDPC Example Analysis

Our first example application is an excellent example of the proper selection and use of both CPU- and FPGA-based GNU Radio acceleration. Figure 2 below shows a GNU Radio flowgraph demonstrating various LDPC encoders and decoders. In the flowgraph a random signal is generated and fed into one the LDPC encoder blocks. Next, Gaussian noise is added to the encoded output to simulate a noisy transmission channel. Then the modified signal is fed into one of the LDPC decoder blocks. Finally, the decoder output is subtracted from the original input signal and can be plotted to demonstrate correctness of the decode operation. The original signal along with the "received errors" (original signal with noise added) can also be plotted for demonstration purposes.

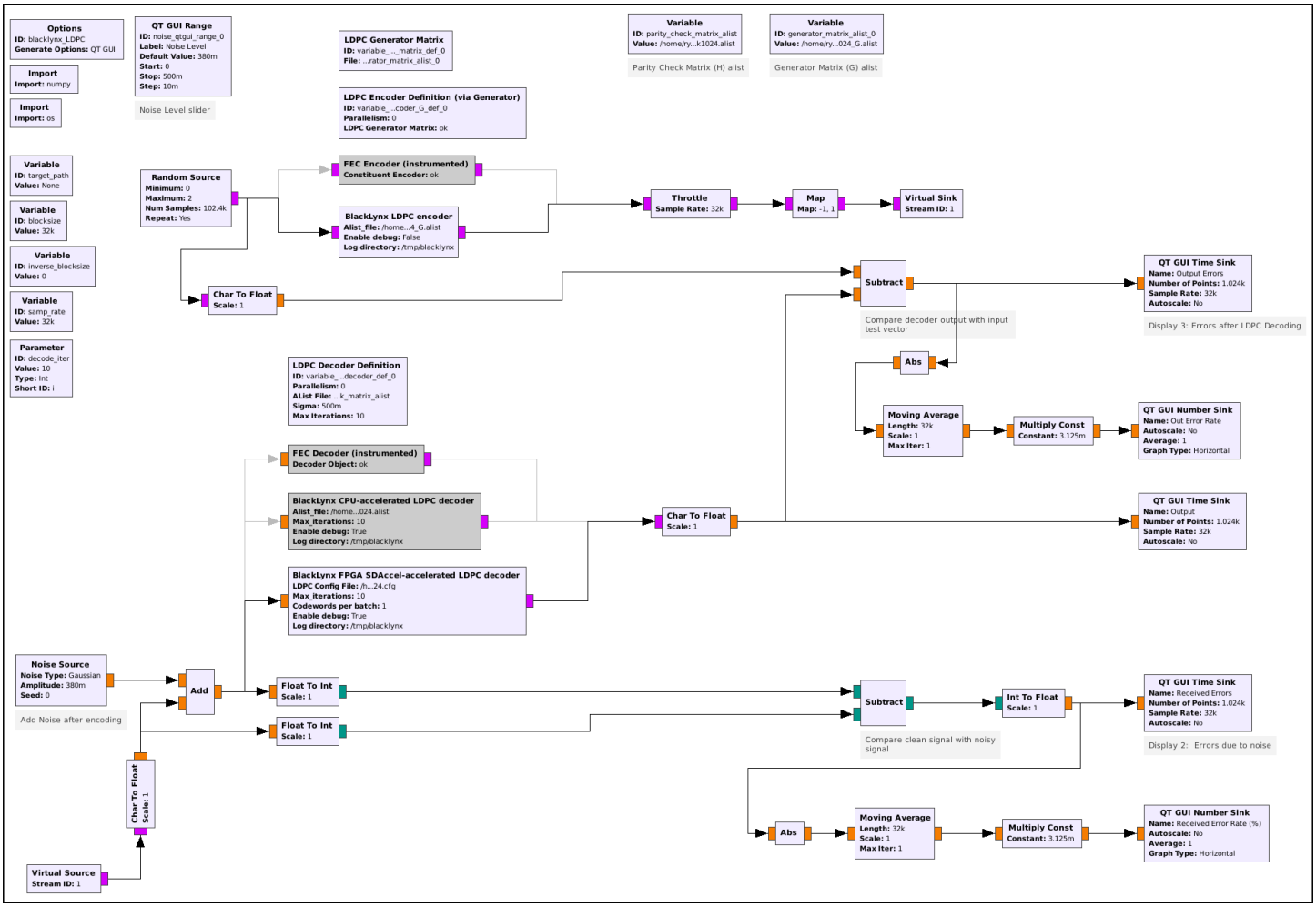

Figure 2: LDPC Encoder/Decoder Example GNU Radio Flowgraph

The LDPC code used in the example is a code specified by The Consultative Committee for Space Data Systems (CCSDS) 131.0-B-2 specification with message size of 1024 bits and codeword size of 2560 bits.

Recall, as noted in the initial problem statement, that our mission objective is to achieve consistent steady-state sub-millisecond performance for both the encode and decode operations running on a single 1U Dell R640 server with one available FPGA acceleration board installed.

Note that the LDPC encoder is implemented with a stock GNU Radio block with a small amount of instrumentation code added to determine the performance of the block. In addition, we implement a BlackLynx CPU-accelerated GNU Radio block. From an implementation perspective, BlackLynx leveraged an appropriate set of vectorized operations internal to the `general_work()` function to implement the CPU-based acceleration, which is a much faster approach than the stock GNU Radio software blocks employ. As we will see, our CPU-accelerated block performance is sufficient to meet the mission requirement in our target server environment, so we do not bother creating a separate FPGA-accelerated block for the encode operation.

The LDPC decoder is implemented in three different ways. First, the stock GNU Radio block is used with a small amount of instrumentation code added to determine the performance of the block. Second, BlackLynx implemented a CPU-only accelerated block using vectorized principles to replace the legacy `general_work()` operation. As we will see, the CPU-accelerated version was not able to achieve consistent sub-millisecond performance, and so we also implemented a proper FPGA-accelerated block leveraging our acceleration API, which is described later in this document.

LDPC Encoder Performance

Both the instrumented default GNU Radio block encoder and the BlackLynx CPU-optimized block are implemented entirely in CPU software running on a Dell R640. A performance comparison is detailed in Table 1 below for (the same) 100 input messages:

	GNU Radio	BlackLynx Optimized
Input Messages:	100	100
Total Duration (ms) :	1540.790	30.960
Average time / msg (ms) :	15.408	**0.310**
Speedup:	--	**49.7**

Table 1: LDPC Encoder Performance Comparison

In this specific example, a nearly 50x performance speedup was realized, as was consistent steady-state sub-millisecond performance. We report the average processing time per message with the BlackLynx CPU-optimized approach as just 0.310 milliseconds for the encode operation. In general, we expect to see well above an order-of-magnitude performance improvement using vectorized CPU-optimized acceleration techniques when compared to non-optimized stock CPU implementations.

LDPC Decoder Performance

Three LDPC decoder blocks are implemented to compare and contrast performance metrics, all running on a Dell R640 with a single low profile FPGA acceleration board supporting Xilinx SDAccel installed. The blocks and types of acceleration employed are summarized below:

- GNU Radio built-in LDPC decoder, instrumented – software-based; run on CPU
- BlackLynx CPU-accelerated LDPC decoder – software-based; runs on CPU
- BlackLynx FPGA-accelerated LDPC decoder – software-based; runs on FPGA

A performance comparison of the blocks is detailed in Table 2 below for (the same) 100 input codewords:

	GNU Radio	BlackLynx CPU	BlackLynx FPGA
Input Codewords:	100	100	100
Total Duration (ms) :	634.936	106.598	28.579
Average time / CW (ms) :	6.349	1.066	**0.286**
Computation Power Consumption (W) :	24	24	**9**
Speedup:	--	**5.96**	**22.20**

Table 2: LDPC Decoder Performance Comparison

The stock GNU Radio block does not meet the mission objective. Its average time per codeword is 6.349ms, yet our requirement was to achieve consistent steady-state sub-millisecond codeword performance. The BlackLynx CPU-accelerated average value shows just slightly higher than 1 millisecond. This means that some codewords must be taking longer than 1ms to process, making the CPU-accelerated code block insufficient to meet mission need. A closer inspection of a sampling of the actual codeword processing results shown in Figure 3 below proves this, detailing significant variance across even well-optimized CPU-based codeword processing. This is not shocking given the nature of CPU processing of LDPC decode operations.

```
BlackLynx CPU LDPC decoder duration: 0.725669 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.711674 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 1.42298 ms (2 input codewords)
BlackLynx CPU LDPC decoder duration: 0.70909 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 1.16074 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.83518 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.758264 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.721496 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.734134 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.722318 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.721588 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 1.13802 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.739528 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 1.70621 ms (2 input codewords)
BlackLynx CPU LDPC decoder duration: 1.01208 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 1.12187 ms (1 input codewords)
BlackLynx CPU LDPC decoder duration: 0.818176 ms (1 input codewords)
```

Figure 3: BlackLynx CPU-accelerated Results Sample

The profiling exercise proves that we must move beyond CPU acceleration to ensure that we meet the mission requirements.

Acceleration Methodology and API

The design of a BlackLynx FPGA-accelerated GNU Radio block is broken roughly in half between the software interface and the underlying FPGA firmware.

GNU Radio blocks are written as classes in either C++ or Python. The Xilinx FPGA SDAccel framework provides a C library interface so blocks that intend to interface with it should be written in C++. SDAccel GNU Radio blocks provide the same software interface as ordinary blocks. All of the SDAccel interface code is encapsulated within the block.

A base class, called `blacklynx_sdaccel`, stores common interface and utility functions needed to create an SDAccel GNU Radio block. Each SDAccel GNU Radio block inherits from both `gr::block` (the standard GNU Radio block base class) and `blacklynx_sdaccel`.

The `blacklynx_sdaccel` class contains the following member functions:

- *`constructor`* – Finds all SDAccel-capable Xilinx devices attached to the system (throws error if none are found). Creates a new OpenCL context. Attempts to import the specified OpenCL kernel image, and errors out if it cannot be found. Next, creates an OpenCL program that ties together the device, context, and kernel binary. Finally, allocates an OpenCL buffer to hold metadata.
- *`destructor`* – This function is unused and therefore empty.
- `sdaccel_start()` – Create an OpenCL kernel using the program instance and the specific name of the kernel. Also creates an OpenCL command queue.
- `sdaccel_stop()` – Destroys and deallocates the OpenCL kernel, command queue, context, binary, and program instances. Also destroys and deallocates all of the OpenCL buffer instances.
- `sdaccel_go()` - This function transfers data and metadata (configuration) to the kernel running in the FPGA and then waits for the output.
- `read_metadata()` – Reads metadata from the kernel and returns it to the caller.
- `read_output()` – Reads raw output from the kernel.
- `write_input()` – Write input data to be transferred to the kernel.
- `write_lbus()` – Write (configuration) data to the local bus to be transferred to the kernel.
- `allocate_input()` – Allocate OpenCL input buffer of the specified size.
- `allocate_lbus()` – Allocate OpenCL local bus buffer of the specified size.
- `allocate_output()` – Allocate OpenCL output buffer of the specified size.
- `add_register_write()` – Add a register write to the local bus buffer.
- `add_register_read()` – Add a register read to the local bus buffer.
- `replace_register_write()` – Replace a register write to the specified address with new data.
- `print_lb_accesses()` – Print (to stdout) the list of local bus accesses.

The standard block interface consists of the following functions:

- *`constructor`* – Calls the `blacklynx_sdaccel` constructor (see above) passing it the specific kernel name, corresponding path to firmware image, and desired buffer sizes. Finally, the constructor allocates the various input and output buffers and performs design specific configuration of the kernel, generally be writing a series of configuration registers via the local bus.
- *`destructor`* – This function is unused and therefore empty.

- `start()` – This function is called when a GNU Radio design containing the block is run. The function calls `sdaccel_start()` (see above) and then performs any additional startup operations needed by the block.
- `stop()` – This function is called when a GNU Radio design containing the block is stopped. The function calls `sdaccel_stop()` (see above) and then performs any additional stop operations needed by the block.
- `general_work()` – This is the "main" function; it is called by the GNU Radio framework to take input, manipulate it, and produce output. This function sets any per call configuration, calls `write_input()` to load the input into the OpenCL input buffer, calls `sdaccel_go()` to send the input and configuration to the kernel and get the output, and finally calls `read_output()` to retrieve the output data.

LDPC FPGA-Accelerated Decoder Implementation, Usage, and Benchmarking

The BlackLynx FPGA-accelerated block was coded in C++ and seamlessly integrates to GNU Radio flowgraphs as a standard signal processing block. Behind the scenes, of course, it leveraged our API as documented above to interface directly to the BlackLynx-enabled FPGA SDAccel-framework to manage data flow to and from the FPGA, and perform the appropriate operations in the FPGA fabric, and the output stream is propagated forward pursuant to GNU Radio block definitions. Everything becomes seamless, and end flowgraph users of the graphical GNU Radio Companion do not have to understand anything about the FPGA fabric, the APIs, or the implementation details. The blocks can be subsequently used natively just like any other GNU Radio block.

Referring again to Figure 3, the BlackLynx FPGA accelerated block performance is outstanding and meets mission need. It outperforms the stock non-optimized GNU Radio implementation by 22x, shows average single-codeword performance at well under 1ms, with Figure 4 below showing a snippet of the actual performance results verification for consistent behavior. Note that this FPGA implementation uses true single codeword operations, for what is often termed in the FPGA space as a "batch size of 1". It is routine for FPGA implementations to require large batch sizes, thereby reducing the real-time performance aspects of a solution. This is not the case for the BlackLynx SDAccel design. True single codeword performance is achieved at consistent sub-millisecond times.

```
BlackLynx FPGA LDPC: 0.214056 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.210057 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.20717 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.205383 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.451225 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.255696 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.231237 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.21363 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.211269 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.211962 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.213319 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.218596 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.210824 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.211567 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.211875 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.208053 ms (1 input codewords)
BlackLynx FPGA LDPC: 0.206311 ms (1 input codewords)
```

Figure 4: BlackLynx FPGA-accelerated Results Sample

FFT Example Analysis

Our next example further highlights the importance of following a best-practices approach of profiling existing SDR blocks for comparison against mission performance requirements. Figure 5 below investigates a GNU Radio flowgraph demonstrating a simple FFT operation. Two sine wave signal sources are summed and passed through the stock GNU Radio FFT block as a continuous streaming waveform.

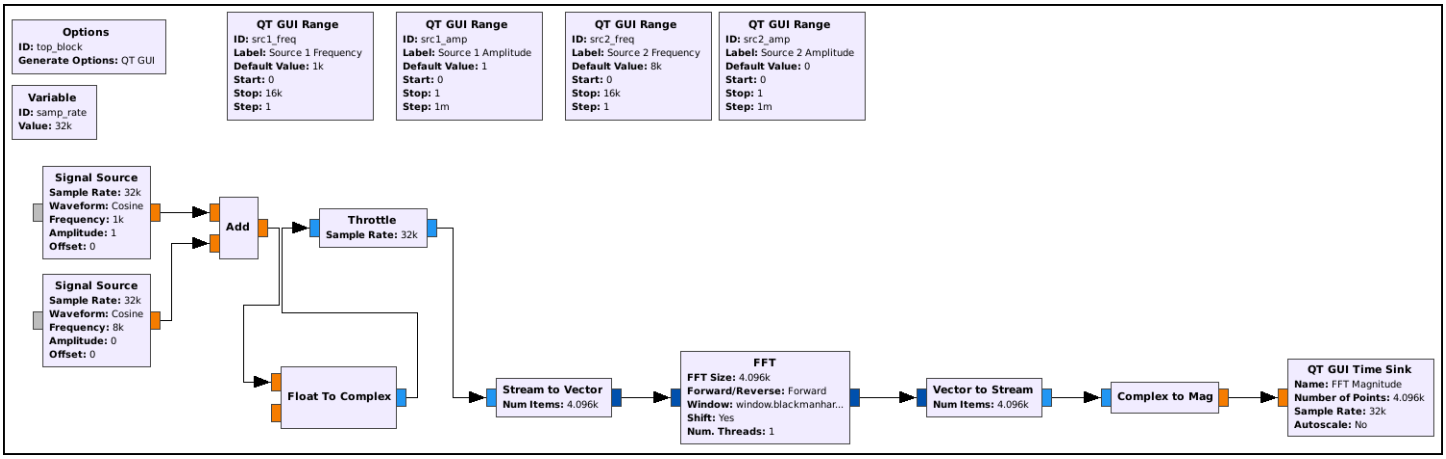

Figure 5: FFT Example GNU Radio Flowgraph

After researching the FFT block's default implementation, it appears to implement a high-performant optimized approach for CPU-based FFT operations, and as such we predict that it will achieve sub-millisecond performance without further optimization. We profile the block to prove our hypothesis, and the results are shown in Table 3 below:

	GNU Radio
Average FFT block time (ms) :	**0.036**

Table 3: Standard GNU Radio FFT Block Performance

The stock GNU Radio block, with an average FFT block time of 0.036ms is well under the required mission performance objective of 1ms. This is fantastic performance, and there is no further need to optimize the block. Purely as an exercise, Figure 6 below shows a results analysis snippet containing individual block times for a portion of a continuous run, where the `fft_vcc_fftw.cc` stock GNU Radio FFT source file's `general_work()` function was instrumented to note the block's performance.

```
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000054 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000021 seconds
CPU fft_vcc_fftw.cc: 0.000035 seconds
CPU fft_vcc_fftw.cc: 0.000033 seconds
CPU fft_vcc_fftw.cc: 0.000035 seconds
CPU fft_vcc_fftw.cc: 0.000022 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
CPU fft_vcc_fftw.cc: 0.000060 seconds
CPU fft_vcc_fftw.cc: 0.000034 seconds
```

Figure 6: GNU Radio Default FFT Block Results Sample

FPGA-Accelerated Size, Weight and Power Advantages

Low size, weight, power and cooling (SWaP-C) has been and continues to be an essential consideration for many constantly evolving signal processing missions. Although in this document we focused on SDR use cases running on a datacenter-class dual socket 1U Dell R640 server, such high-powered CPU systems are not always available to mission. Various platforms for example may have extremely limited CPU compute resources for a variety of operationally relevant reasons, such as power budgets and/or in-theater or on-platform cooling considerations. This means that it is entirely plausible that some of the CPU-only acceleration results that were demonstrated here may not be applicable to mission environments.

Referring once again to Table 2, note that the FPGA-accelerated LDPC decoder ran in just a 9W computational power envelope, while still outperforming the CPU computation power envelope, which was 24W. What was not noted in the table was that the Linux-driven 1U Dell R640 server as configured for the benchmark runs at a quiescent steady-state power of around 226W, even when it is just sitting there! Many non-datacenter mission environments cannot provide hundreds of Watts of power, and so smaller form factor low-power compute is often required. In these environments, FPGA acceleration alongside less-capable lower-power CPUs can be an excellent choice to meet mission needs, yet still offer the benefits of SDR-based signal processing frameworks. The best-practices methodology and analysis described in this document apply equally well to designs destined for low SWaP-C environments.

BlackLynx Deploys Seamlessly to the Cloud

The Xilinx VU9P FPGA that was employed in the descriptions in this document is the same Xilinx FPGA family variant that is used in the Amazon Web Services F1 cloud-based instance type. BlackLynx-accelerated GNU Radio solutions deploy identically, and seamlessly, even into the AWS cloud, with or without FPGA acceleration.

This means that DSP engineers are no longer constrained by on-premise-only or cloud-only deployment strategies, but instead can leverage hybrid deployment strategies, intelligently enabling businesses to choose when and where to solve problems using the identical BlackLynx APIs in all such cases. Since the same SDAccel-capable FPGA family used in these benchmarks is used in the AWS F1, the resulting FPGA-accelerated performance results will be practically identical when leveraging the AWS F1 instance.

The same logic holds true when targeting the latest generation SDAccel-capable Xilinx Alveo Accelerator Cards in the Nimbix Cloud.

Analysis Summary

Three mission-related examples were presented and analyzed in this document, targeting a single 1U COTS Dell R640 server, outfitted with a single ½-height, ½-width low profile PCIe Xilinx SDAccel-capable FPGA acceleration card. The first example demonstrated that we could achieve mission performance requirements for LDPC encoder operations purely with a BlackLynx CPU-optimized approach. The second example demonstrated the powerful ability to seamlessly leverage our BlackLynx optimized FPGA SDAccel-based APIs alongside GNU Radio to achieve mission performance targets for LDPC decoding. The resulting LDPC decoder signal processing block becomes a drop-in to any GNU Radio flowgraph.

We then presented a final example where the stock GNU Radio implementation of an FFT met mission requirements without requiring any extra acceleration.

BlackLynx has adopted a best-practices SDR-based signal processing development strategy starting with performance profiling, followed by CPU optimization, and finally FPGA optimization, but always within the confines of the larger SDR framework and target hardware/server environment, based on analysis and understanding of mission need. This enables signal processing missions to leverage the power of SDR frameworks (development time, deployment time, and time-to-decision advantages) without sacrificing mission performance, whether running at the edge, in a datacenter, in the cloud, or any combination thereof.

When mission requirements outpace what a pure software solution can provide, the BlackLynx SDAccel-based FPGA acceleration API works seamlessly alongside existing SDR GNU Radio blocks to target acceleration only for those portions of the waveform that matter to the mission, allowing the bulk of operations to remain in pure CPU-side software.

Learn More

To learn more about BlackLynx's COTS-deployable accelerated solutions and how they can help your business thrive in the era of always-on analytics, please visit our website at www.blacklynx.tech.