Hardware Accelerated HD Radio Decoding

Warren Hu
wsh003@ucsd.edu

***Abstract*—This project aims to accelerate a software HD radio decoder with hardware functions implemented on an FPGA. The workflow for accelerating an embedded C application using Xilinx tooling on a Pynq-Z2 board system is explained. Performance requirements for real time decoding are estimated. The hardware accelerated HD radio decoder is then compared to the non-accelerated version.**

I. Introduction

NRSC-5, also marketed as HD radio, is the standard for digital radio broadcasts in the United States. It allows FM radio stations to broadcast additional OFDM encoded data in a hybrid or all digital format. Most radio stations opt for the hybrid format, where the digital data is broadcast in conjunction with the traditional FM analog station as illustrated in Fig. 1. A security company named Theori has developed a software based NRSC-5 decoder [1] but it is too slow to run in real time on the Pynq-Z2 board.

Decoding a digital signal is computationally intensive since it generally requires digital filters, computing the FFT, and error correction. In addition, an NRSC-5 signal has a bandwidth of approximately $336.8\,\text{kHz}$. At the bare minimum, to fully capture this signal we need to sample at twice the bandwidth or at about $673.6\,\text{kHz}$. Most software decoders tend to sample at even higher rates and decimate afterwards to prevent aliasing. The HD radio decoder we are accelerating samples at 4x the bandwidth or $1.347\,\text{MHz}$. This does not give us very much time to process each incoming IQ sample.

In addition, the 1 GHz Arm-A9 cores that Pynq-Z2 uses do not support vector instructions which implies we need to process each incoming sample with roughly 10 instructions while also running with the rest of the decoding pipeline.

Fortunately, each sample must be processed in the same way with little to no branching and generally consists of many operations on small amounts of data at a time. This matches the performance profile of an FPGA very well and allows us to easily create much faster hardware implementations of various common processing functions.

[1] https://github.com/theori-io/nrsc5

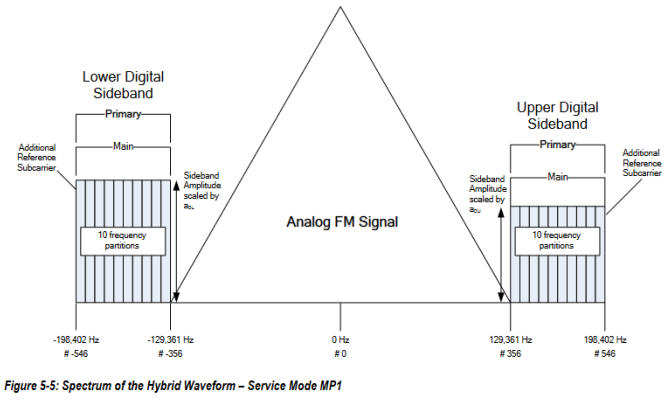

Fig. 1. Hybrid HD radio spectrum [1]

II. Software Architecture and Profiling

The software NRSC-5 decoder is written as a pipeline consisting of discrete stages as outlined in Fig. 2. The throughput of the pipeline is therefore decided by the throughput of the slowest stage. If the throughput of the decoder cannot handle the amount of input samples, then samples will be dropped and lost. However, too many lost samples will cause errors in channel estimation and decoding. Eventually, the receiver will lose synchronization with the broadcasting station and the decoding will stop.

The decoder first reads the raw IQ samples from the RTL-SDR over USB in batches of $524\,288$ bytes or $262\,144$ samples. The samples are fed into a front end decimation filter and a band pass filter. The filtered and decimated samples are then passed to an OFDM synchronizer that performs packet detection and coarse carrier frequency offset correction. The pilot symbols are then extracted and used to do additional channel estimation. The data is then converted to bits in the demodulator.

After error correction using a Viterbi decoder, the bits are passed to the higher layers where the NRSC-5 metadata can be extracted. This metadata contains things like weather information, audio timestamps, traffic data, or the latitude/longitude of the broadcasting station. Finally the audio is extracted and decoded from AAC to raw audio samples.

Perf was used to profile the application while it was running on the Pynq-Z2 board. The front end FIR filters and the Viterbi decoder were found to be consuming the largest percentage of CPU cycles and were therefore likely to be limiting the throughput of the entire system. FIR filters are easy to translate to an FPGA so were choosen to be hardware accelerated first.

III. Xilinx Acceleration Workflow

A. Overview

Xilinx has developed an "Embedded Application Acceleration Workflow" that targets these types of embedded systems. It takes the application code, the hardware kernels, and a model of the target board and automatically generates the interfaces for communicating between the programmable logic

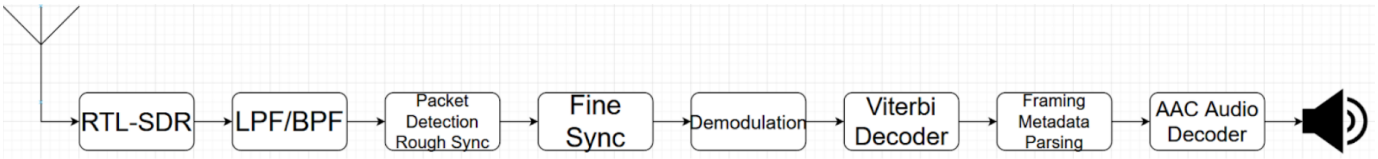

Fig. 2. Software Decoding Pipeline

(PL) and the Arm processor (PS). The output is a Linux image that can be flashed to the target board. The image contains the application and a bitstream that can be loaded into the PL.

B. Creating the Vitis Platform

First we need to model the Pynq-Z2 and set up some basic IP blocks that are required for Linux to run and communicate with the PL. In Vivado we can create a block diagram with the Zynq processing system and an interrupt controller. Then we can connect the Zynq AXI master to DDR and to an AXI peripheral device that the Xilinx tools will use to communicate with the PL. The clocks and any peripheral hardware is also configured in this step. For the Pynq-Z2 board a $100\,\mathrm{MHz}$ PL clock was configured. After running the synthesis step, a XSA file is created that can be used to create a custom application acceleration platform.

The platform files for Vitis 2020.2 I built can be found here [3]. These files can be reused by other applications targeting the Pynq-Z2.

In addition to a board config XSA file, the Linux kernel, boot files, and rootfs also need to be configured and built. I ran the Pynq 2.7 board creation workflow [2] which built a Linux kernel and all the required boot files. The rootfs was created by copying the Pynq rootfs and adding all the required libraries and headers needed to build the software NRSC-5 decoder. These libraries and headers were also copied to the Xilinx common sysroot. Note that this process is only supported on Linux, Windows is not currently supported.

The final rootfs and sysroots I used can be found here [4]. Other application developers may wish to start with the Xilinx common rootfs and sysroot instead.

C. Calling a Hardware Kernel

Vitis provides the ability to write hardware kernels using either high level synthesis or RTL. If using HLS, then Vitis will handle creating the interfaces automatically. It supports either AXI-Lite or streaming interfaces. Interface parameters can either be pointers or scalars. Returning data to the host requires a pointer parameter.

The Vitis application acceleration environment supports all the Vitis HLS synthesis pragmas to support optimizations like pipelining, loop unrolling, and array partitioning. However, I could not figure out how to create separate test benches for each kernel. I eventually resorted to testing each kernel by calling it from the PS and validating the results by hand.

Calling the hardware kernels is done through the Xilinx Runtime Library (XRT) [5]. The XRT library provides a set of functions to interact with the PL by calling into a XRT Linux kernel driver. The basic workflow is to open a compute device, open a shared or exclusive reference to the kernel, and then allocate input/output buffers for the PL to use. One catch I found was that using the XRT library requires root access or properly configured udev rules in order to be able to open the compute device.

In order to actually run the kernel and pass data to it, XRT offers two main methods. The first is to use xrtBOWrite and xrtBORead to copy data to and from the PL buffers. This method has the highest call overhead since the CPU needs to handle copying the memory. The second method is to call xrtBOMap to map a CPU pointer to PL memory. Then any writes to the pointer will update the PL memory. This offers lower overhead since the extra copy is avoided, but this is less flexible and harder to integrate into an existing application. For this project, the overhead of copying buffers using xrtBOWrite and xrtBORead was low enough that the added complexity of mapping buffers was unnecessary.

D. Determining Kernel Performance Requirements

Before kernel development began, the actual performance requirements needed to be estimated. I choose to start with developing two FIR filters.

The first filter is a decimating low pass filter used to prevent aliasing. It processes all incoming IQ samples, applies a low pass filter, and decimates it by two.

The PL is configured to run at $100\,\mathrm{MHz}$ and the RTL-SDR is configured to produce $1.347\,\mathrm{Msamples/s}$. Each loop of the software FIR filter processes two input samples and produces one output sample. Then the hardware kernel needs to process two samples every:

$$\frac{2}{1.347\,\mathrm{MHz}} = 1.485\,\mu\mathrm{s} = 148.5\,\mathrm{cycles}$$

The second filter is a slightly more complicated 32 tap band pass filter. Since it is run after the decimating low pass filter, this filter only needs to process $673.5\,\mathrm{ksamples/s}$ but it only processes one sample per iteration.

$$\frac{1}{673.5\,\mathrm{kHz}} = 1.485\,\mu\mathrm{s} = 148.5\,\mathrm{cycles}$$

With careful pipelining and array partioning, it is relatively easy to get an initiation interval of 1 since FIR filters are pretty straightforward and do not require much memory. The throughput of both of these hardware FIR filters would be orders of magnitude faster than the software version, but the latency would likely be worse. This is because the PL only

runs at 100 MHz but the PS runs at 1 GHz. In addition due to the significant overhead of calling into the PL, we need to process samples in relatively large batches. Therefore the latency would likely be a lot higher unless we spend a lot of area and power to unroll the loop and process more samples in parallel.

For this application however, the latency is not very important as all the samples will be delayed by the same amount so decoding is not affected. The much higher throughput of a hardware implementation is worth the latency tradeoff.

IV. Results

Two hardware FIR filters were implemented in HLS and integrated into the HD radio decoder. `fir_q15_execute_decim` implements the low pass decimating filter and `fir_q15_execute_fm` implements the FM bandpass filter to isolate the digital sidebands.

`fir_q15_execute_decim` takes the number of samples to process as an input and supports a maximum of 262144 samples at once. The main loop is pipelined with an initiation interval of 36 cycles and a iteration latency of 223 cycles. It has a throughput of $\frac{1000}{10\,\mathrm{ns} * 36\,\mathrm{cycles}} = 2.77\,\mathrm{MOps/s}$. It processes 2 samples per operation. The worst case runtime of the kernel is about 11.798 milliseconds. The resource consumption of the filter can be found in table I.

`fir_q15_execute_fm` always processes $140\,400$ samples per execution. It has an initiation interval of 1 cycle and an iteration latency of 170 cycles. It processes 1 sample per loop iteration for a throughput of $100\,\mathrm{MHz}$. The kernel takes 1.406 milliseconds to complete. The resource consumption can be found in table II.

To demo the project download and uncompress the file `sd_card.img.gz` from [4]. Then write this to a large enough SD card and attach an RTL-SDR to the Pynq. To tune to station 89.5, digital channel 1, run the application with the following:

1) `cd /boot`
2) `sudo ./radio 89.5 1 -o out.wav \`
   `-x ./binary_container_1.xclbin`

We can roughly benchmark the performance of these execution time of the accelerated and non accelerated functions by getting the CPU time with `clock()`. This function gets the current CPU time. Note that the application is multithreaded, so the absolute CPU time is likely to be off. However, the ratio of the two elapsed times between the accelerated and non accelerated version should still be approximately correct.

The `input_push_cu8` function runs the decimating low pass filter. The nonaccelerated version takes $167\,\mathrm{ms}$ on average [6] while the accelerated version takes $64\,\mathrm{ms}$. This is an overall speedup of x2.6.

The `acquire_process` functions runs the bandpass filter. The nonaccelerated version takes $177\,\mathrm{ms}$ on average and the accelerated version completes in $57\,\mathrm{ms}$. The overall speed up is about x3.1.

It is likely that the overall runtime of these two functions is now dominated by other parts of the function so the overall performance increase is not as great as what was estimated. But performance has improved and the decoder is now stable enough to decode in real time.

TABLE I
Resource consumption of fir_q15_execute_decim

Name	BRAM_18K	DSP	FF	LUT	URAM
Total	2	8	1584	1789	0
Available	280	220	106400	53200	0
Utilization (%)	$\sim$0	3	1	3	0

TABLE II
Resource consumption of fir_q15_execute_fm

Name	BRAM_18K	DSP	FF	LUT	URAM
Total	4	32	2963	2053	0
Available	280	220	106400	53200	0
Utilization (%)	1	14	2	3	0

V. Conclusion

The use of only two small hardware kernels has improved the performance of the decoder enough to achieve real time decoding. Application specific hardware design can offer significant speed ups, especially on resource constrained systems. The tooling has also improved in that it is much easier to develop and interface with these HLS kernels when compared to RTL kernels. It makes it easy to hardware accelerate small pieces of an application. However, the tooling is still fairly hard to use. It was pretty difficult to setup a working XRT environment and even getting the application to build in the first place was a lot of work. The tooling is not that well documented, especially if you are creating a custom platform for a new board.

If necessary, additional kernels could have been implemented. But these two FIR kernels were enough to decode in real time so implementing additional kernels was unnecessary for the scope of this project. Future work could look into adding additional hardware kernels and attempting to decode multiple streams at once. NRSC-5 allows combining several virtual radio stations at reduced audio quality into the same frequency range. It could be interesting to allow decoding all virtual stations at the same time.

Output is also only written to a file and not played on the boards headphone jack. I ran out of time to implement this but it seems straightforward enough and a good feature to implement in the future.

References

[1] HD Radio™ Air Interface Design Description Layer 1 FM Rev. G, NRSC 1011s, 2011.
[2] https://github.com/xilinx/pynq/tree/image_v2.7
[3] https://github.com/keringar/PynqZ2-Platform
[4] https://drive.google.com/drive/folders/1hytkwT-_TCfe6ojVgehl9g0o_twajMYQ?usp=sharing
[5] https://xilinx.github.io/XRT/2020.2/html/xrt_native_apis.html
[6] B. Jacob and T. N. Mudge,Notes on calculating computer performance.University of Michigan, Computer Science and Engineering Division,1995.