Noise Reduction with LMS on Pynq

Zach Neveu

April 24, 2020

Abstract

In this work, Python code for an LMS adaptive filter for audio is developed. Using the Pynq [1] framework and Xilinx HLS tools, the LMS filter is transferred to run in in the programmable logic of a Zynq 7000 FPGA. Several hardware implementations are compared in terms of resources and performance, and an ≈ 600x algorithm speedup is achieved by moving the LMS filter to hardware.

Introduction

The presence of background noise can greatly impact the quality of recorded and transmitted sound, impacting people's ability to communicate in certain environments. Background noise often overlaps greatly with speech in both the spectral and temporal domains, which makes removing noise via static-coefficient filters largely ineffective. In the spatial domain, however, separation between speech and background noise can be achieved.

The primary technique for achieving this involves using two microphones as shown in Figure 1. One microphone picks up a mix of signal and noise, while the other microphone picks up only the noise. Unfortunately, the two noise signals are not identical because they vary in recording location, however it is generally valid to assume that they are linearly related. The job of an adaptive filter is then to estimate the response of this unknown linear system in order to estimate and remove the noise contribution to the mixed signal.

In order to perform this task, adaptive filters update their coefficients in order to minimize an error signal. There are several kinds of adaptive filters which balance trade-offs between computational cost and performance. Foremost among these adaptive filters are the least mean squares (LMS) and recursive least squares (RLS) filters [2]. The LMS filter minimizes the mean squared error (MSE) between the output and desired signals, while the RLS filter minimizes an exponentially-weighted error biased to weight more recent samples more heavily. Because of these design choices, the LMS filter is less sensitive to quantization noise, and requires less computational power than RLS [3]. These properties make it a good fit for real-time designs and hardware such as field programmable gate arrays (FPGAs) which allow for greater efficiency through custom quantization via arbitrary precision data types.

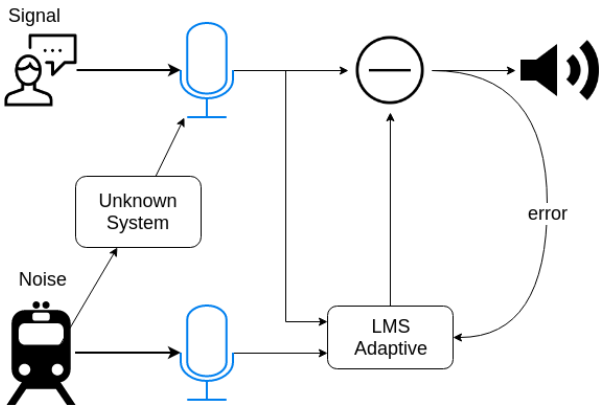

Figure 1: System Diagram for Adaptive Filtering

LMS Algorithm

Given an input signal x and a desired signal y with known autocorrelations Σ_x and Σ_y and cross correlation r_{xy}, a static-coefficient Wiener filter (based on a simple finite impulse response (FIR) filter) will minimize the MSE between the desired and estimated signals [4,] . The optimal weights for such a filter are described by equation 1. In order to implement these weights, both Σ_x and r_{xy} must be known, which in the absence of prior knowledge, involves recording infinitely long seg-

ments of x and y.

$$\boldsymbol{W} = \boldsymbol{\Sigma}_x^{-1} \boldsymbol{r}_{xy} \tag{1}$$

The LMS filter provides a way to approximate optimal weights without requiring an infinitely long signal. The key principle is to use steepest descent to optimize each weight to reduce it's contribution to the error signal. The weight update formula is shown in equation 2. Given the right learning rate μ and stability constant δ, the weights can be proven to converge to optimal [4]. One benefit of the LMS filter over a Wiener filter is that in real world scenarios, signal properties often change slowly over time, and the LMS filter will automatically learn these changes, converging again to optimal performance.

$$\boldsymbol{W}_n = \boldsymbol{W}_{n-1} + \mu e_n \boldsymbol{x}_n \frac{1}{\boldsymbol{x}_n^T \boldsymbol{x} + \delta} \tag{2}$$

Software Implementation

In order to test algorithmic correctness as well as provide a performance reference, a software implementation was created. The first implementation was written in Python for its ease of use for linear algebra thanks to the Numpy package. The prediction and weight update stages of a LMS filter can be written in 4 lines of Python code as shown below.

```
yhat = np.dot(w, X)
e = y-yhat
nu = mu / (delta + np.dot(X,X))
w += nu * e * X
```

Data was generated in Python to test the implementation. The method used to generate data was to use a known-response FIR filter to simulate an "unknown" system. White noise was passed through this filter, then both the original and the filtered noise were passed to the LMS filter. A full diagram of the data generation and testing process is shown in figure 2.

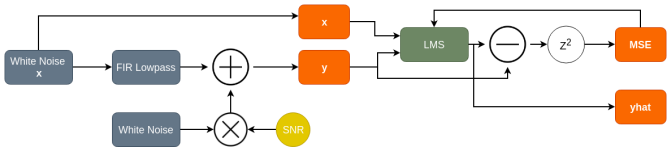

Figure 2: Data Generation and Testing Pipeline

HLS Implementation

As an intermediate step to running the filter in hardware, high level synthesis (HLS) C++ code was written. The primary task for this section of the project was to expand the Numpy linear algebra functions from Python into manual loops in C++. The HLS implementation uses a C++ template class structure in order to allow for easy changes to the number of taps, as well as clean modular code. To verify correctness of the HLS implementation, a C++ test bench was written to follow the data generation and test procedure shown in figure 2. Outputs from this process were captured and analyzed alongside the Python software outputs. When floating point data types were used in the HLS, the losses matched the Python code, showing a correct implementation.

HLS Optimization

In order to optimize the performance of the hardware, several optimizations were applied to the code via HLS `#pragma` statements. The optimizations tried are:

- Data type optimization - `FP32`, `FP16`, `Fixed32`, `Fixed18`, `Int18`, `Int32`
- Loop unrolling
- Loop pipelining
- Loop fission/fusion

Table 1 shows the configurations for all logged experiments with optimization. It is important to note that not all optimizations tried are included in this table. In particular, loop fission appeared to always outperform loop fusion, so only the

loop fission experiments are included. Figure 3 shows a Pareto plot comparing the resources versus the latency for each design in table 1. The HLS tools give the number of each type of resources used (BRAM, DSP48, FF & LUT), as well as the total number available for each category. In order to simplify these numbers into a single "resources" variable, the percentages of available resources were calculated for each category and the mean of all categories was taken to show. The HLS optimization resulted in several surprising results. First, adding extra bits greatly increases costs in latency, which translates into extra resources when loops are unrolled. Second, integer data types offer surprisingly good performance. Finally, the FP32 and FP16 pipelined results offered a surprisingly good trade-off between performance and resources. Because the difference between these two implementations was small in resources, the FP32 pipelined design was chosen for hardware implementation.

Table 1: Configuration of HLS Experiments

Loops	Number Type	Data Width
basic	Floating	32
basic	Fixed	32
basic	Fixed	18
unroll	Fixed	18
unroll	Floating	32
unroll	Int	18
unroll	Floating	16
pipeline	Floating	16
pipeline	Int	18
pipeline	Floating	32
pipeline	Fixed	18
pipeline	Fixed	32

Hardware Implementation

In order to use the design with Pynq overlays, the HLS inputs, outputs, and parameters must be implemented as AXI interfaces. For maximum speed, it is optimal to use AXI-stream interfaces, however Pynq does not support using this type of interface for anything other than `int32` and `int64` data types [5]. When the HLS design is synthesized and run using `int32` data and streaming interfaces, it is able to process 1000 samples in 2.24ms, yielding an $\approx$ 600x speedup over the Python code which took 1.43s. In order to use `int32` data types, it was necessary to modify the HLS code to manually treat integers like fixed point numbers. In order to do this, an integer (2^{14} in this case) was chosen to act as the "identity" value. When multiplications are done in processing, the results are divided by the identity. By dong this, the multiplication results are re-normalized, effectively allowing for multiplying by fractional numbers. The block diagram for the hardware implementation is shown in figures 4 (top level) and 5 (sub-level) in the appendix.

Hardware Debugging

To check the correctness of the design and debug problems, the LMS IP was connected to a Vivado design with a trace analyzer as shown in figure 6 in the appendix. In this configuration, the trace analyzer provides valuable insights into the behavior of the AXI stream interfaces, allowing full visibility of what is happening on each cycle of each interface. When debugging the LMS IP, this feature was useful in debugging misaligned data that would have been difficult to notice otherwise.

Conclusions

FPGAs can provide significant speedups to LMS filtering. The ability to use arbitrary width data types is particularly useful when working with audio, where standard data types are 16 and 24 bits.

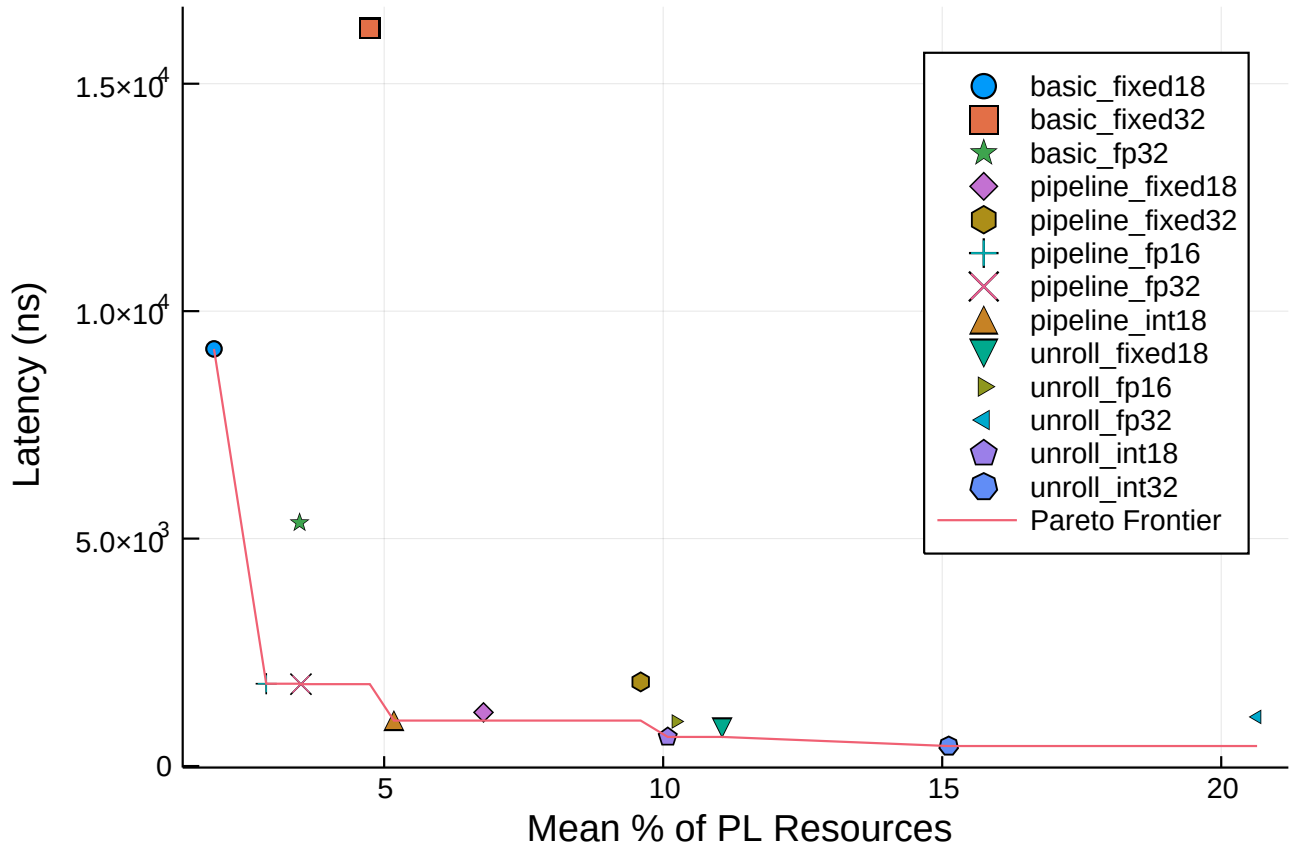

Figure 3: Pareto Plot of Latency vs. Resource Usage

Additionally, the tree of multiply/accumulate operations required for both the weight updates and FIR filtering can be parallelized [6], allowing for further speedups on multicore architectures including FPGAs, especially for filters with many weights. C++ class-based algorithm development in HLS provides a valuable bridge between high level software development and low-level hardware design. When paired with the Pynq development environment, this workflow provides a straightforward way to refactor critical software algorithms into hardware to increase performance.

References

[1] "PYNQ - Python productivity for Zynq," http://www.pynq.io/.

[2] P. S. R. Diniz, *Adaptive Filtering Algorithms and Practical Implementation*, 2020, oCLC: 1129184597.

[3] T. Adali and S. H. Ardalan, "On steady-state performance of the fixed-point RLS algorithm," *Computers and Electrical Engineering*, vol. 25, no. 1, pp. 1–16, 1999.

[4] N. Wiener, *Extrapolation, Interpolation, and Smoothing of Stationary Time Series*. The MIT Press, 1964.

[5] "Xilinx/PYNQ," https://github.com/Xilinx/PYNQ.

[6] A. Gonzalez, M. Ferrer, M. de-Diego, C. Roig, and G. Pinero, "Filtered-X LMS adaptive algorithms on multicore processors systems," in *18th International Congress on Sound and Vibration 2011, ICSV 2011*, vol. 1, Jul. 2011.

Appendix: Block Diagrams

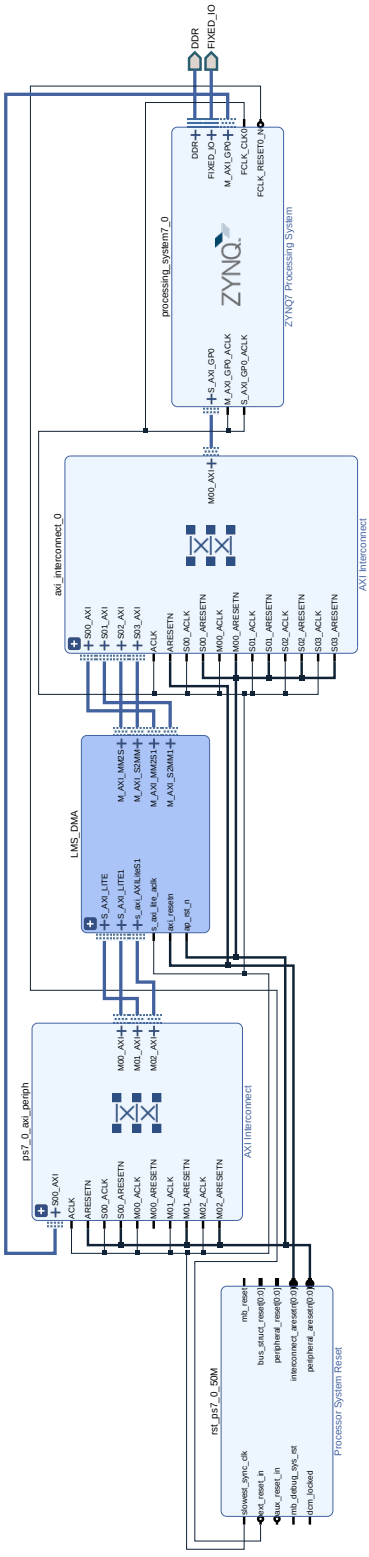

Figure 4: Top Level Block Diagram

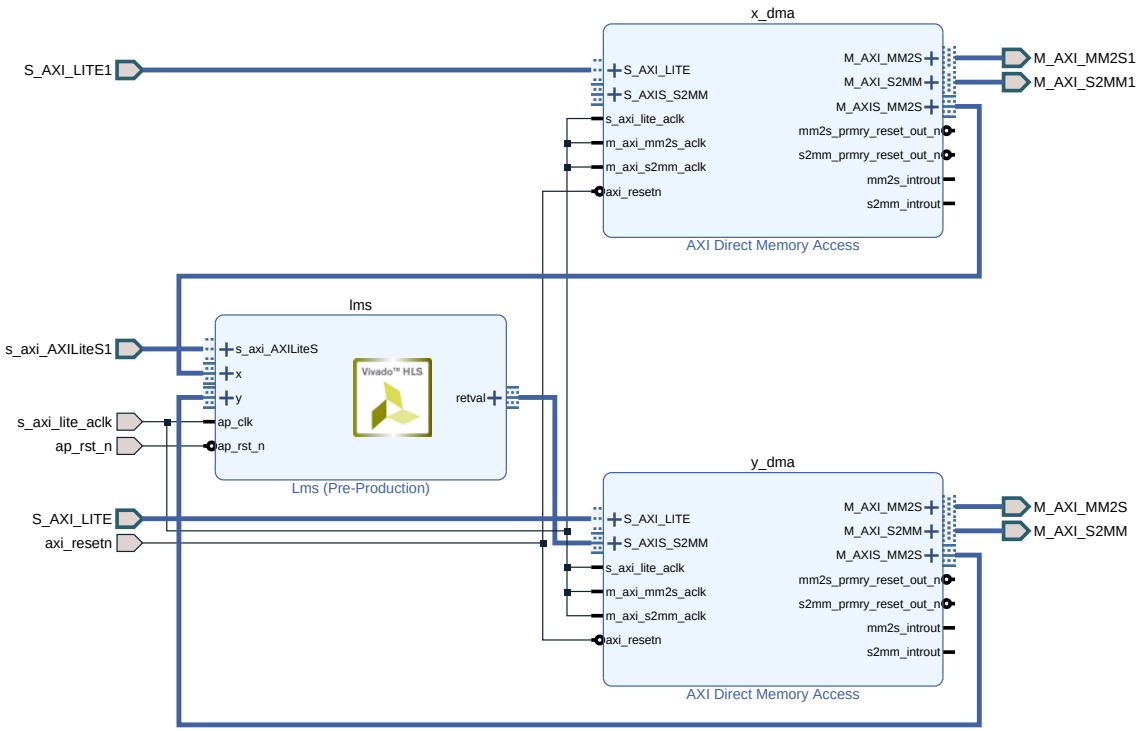

Figure 5: DMA & LMS Sub-block

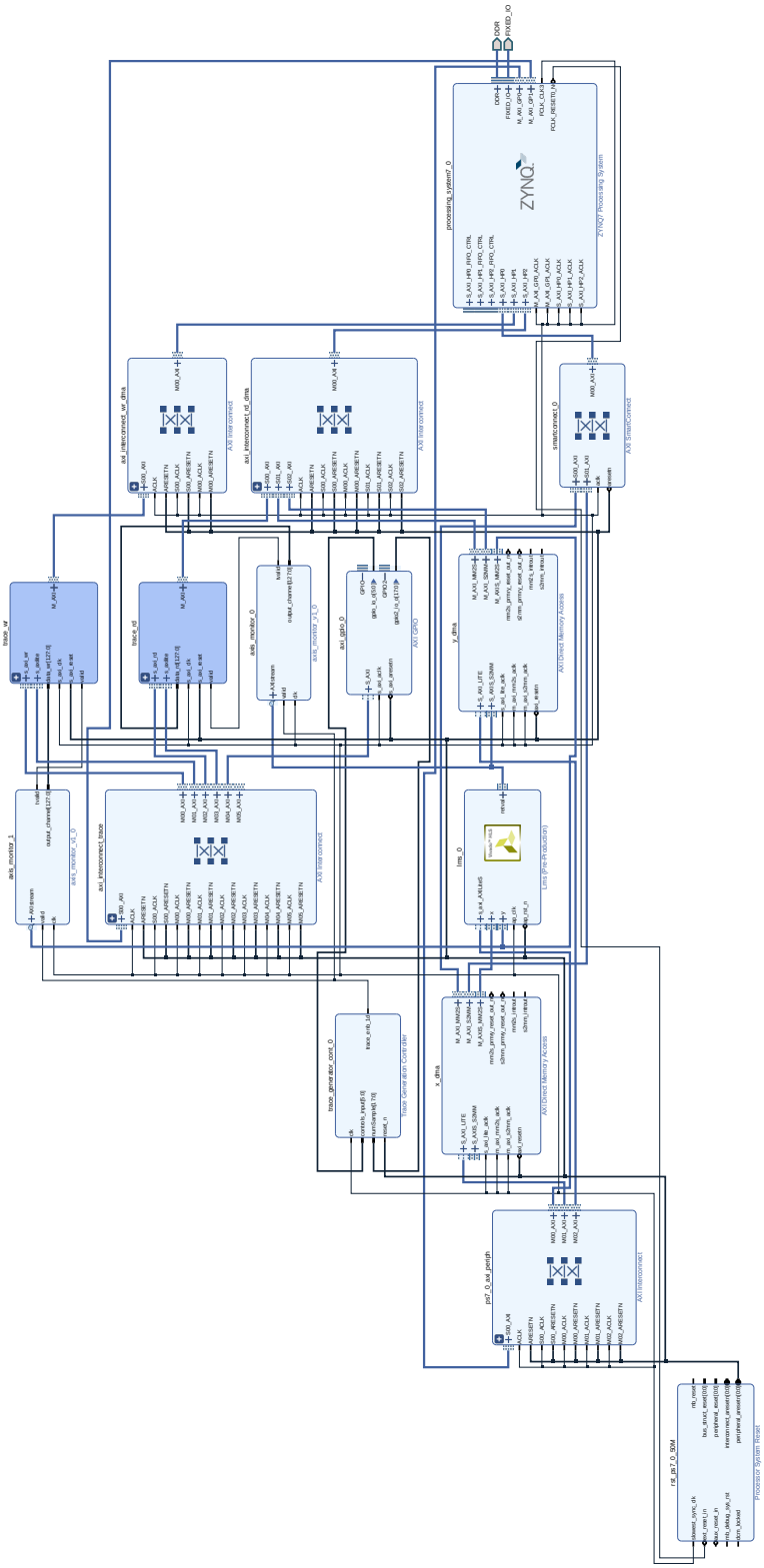

Figure 6: Block Diagram of Trace Analyzer Design