Low-Latency Sound Source Localization: Accelerating the Generalized Cross Correlation through Vitis HLS

Malina, Kacper

Summer 2025 (AOHW25_447)

1 Introduction

Algorithms that are used in the location of the source of some traveling waves, such as sound or EM, usually involve splitting the problem into two: first finding the delays between the same wave arriving at different sensors, then using mathematical transformations to calculate (or guess) the position of the source based on those delays[2]. This project is concerned with accelerating the first step, that is finding the delays, or time differences of arrivals, TDOA for short.

One of the methods (and a pretty common one) is the Generalized Cross Correlation, hereafter called GCC[1]. The specific method that will be considered here is called GCC-PHAT, where PHAT means Phase Transform. It is a type of normalization applied in the calculation. The IP core described here could easily be adapted for other, similar, types of normalization, or it could be turned off completely.

This paper focuses on the implementation of GCC-PHAT on an FPGA. To enable easy integration with other cores, all the interfaces used are AXI4-Stream[7], shortened to AXIS. To ease the prototyping, a UART-AXIS bridge is also provided in the project.

2 Method

The core supports multiple channels (default 4) to yield multiple delays (number of channels less 1). Each channel is has its delay calculated w.r.t. the reference channel 0. It involves performing the GCC 3 times, one for each non-reference channel.

Lets assume that there are k channels x_k, and w.l.o.g. channel x_0 is the reference channel. Fourier transform (FT) is $\mathfrak{F}$ and inverse FT is $\mathfrak{F}^{-1}$. A channel is fixed-length time-domain signal, with the default length being 1024 samples in the core. This is also denoted here with τ.

Firstly, do FT on all channels

$$x_k \xrightarrow{\mathfrak{F}} X_k$$

Note that X_k are complex signals. Then the cross-correlation is performed

$$Y_k = X_k \cdot X_0^*, k > 0$$

Then the values are normalized (the -PHAT part)

$$Y'_k = Y_k / |Y_k|, k > 0$$

Lastly, perform inverse FT,

$$Y'_k \xrightarrow{\mathfrak{F}^{-1}} y'_k, k > 0$$

The time coordinate t_k of the maximum $\max(y'_k), k > 0$ is the delay between channels 0 (reference) and k. Note that FT works on periodic signals, thus t_k is periodic, with period τ, and $t_k = t_k + n\tau, n \in \mathbb{Z}$. More obvious result of that is that, if $t_k > \tau/2$, it means that t_k might actually be negative, with value $t_k = t_k - \tau$.

The Python script, which implements this directly, as a reference, is provided in `/py/golden_gcc.py` in the repository.

3 Implementation

For the implementation of that method, Vitis HLS was chosen, as it is relatively easy and quick to get all the math operations synthesized.

The method has been directly translated from the above description into C++. Please see

`/hls_gcc/gcc_phat/gcc_phat.cpp` for more details. The sample width on the input was selected to be fixed-point 16-bit, although it does get expanded in the calculations to 32-bit fixed-point and 32-bit float. This is done to conserve the BRAM used for storing the results in between. The output has been selected as 16-bit signed integer, which fully contains the possible output of GCC PHAT, given that the FFT Logicore IP[5] allows a maximum of 2^{16} for the FFT length.

The core is implemented with AXIS input and output streams and no further control, that is this is a data-driven IP core.

The reported latency, with the default settings, post-synthesis in Vitis is 350 μs. The part selected for implementation is XC7S50CSGA324-1, available on the RealDigital Urbana devboard. This is a single-clock design, with the clock being 100MHz.

Stage	BRAM	DSP	FF	LUT
Vitis	23	24	26.5k	22.5k
Synth.	11.5	72	17k	14k
Impl.	11.5	72	16.5k	12.5k
Use%	15%	60%	25%	39%

Note that the values reported for Implementation include UART-AXIS bridge and 2 Data Width converters, each of which reports around 40 LUT and 60 FF post-synthesis. The reported power is 422mW. Last row *Use%*, reports the % use w.r.t. total available resources on the FPGA, post-implementation.

All the strategies and settings not mentioned above are set as default. The design was connected with Block Design tool.

4 UART-AXIS bridge

To allow writing (and reading) data to (and from) the core, special UART-AXIS bridge was prepared. The implementation is available in the repository, with the filename `uart_axis.v`. It directly divides the 100MHz clock by 33 (on default) to get 3.03MHz and allow for 3Mbaud UART communication. It is within 1% of the required frequency.

Since the specific use-case is that the UART communication takes place on a short on-board trace, that is, between the FTDI chip on Urbana and the FPGA chip, 3Mbaud communication works correctly. Due to that, and the demo conditions, no checksum checking core has been devised here. That is, in this configuration, all UART data is directly injected into the AXIS interface, and vice-versa on the reading side. There is no FIFOs implemented, and failure to read the data results in its loss. However, since UART communication is *much* slower (300kB/s) than the FPGA and all the interfaces, this is unlikely to happen here.

The USB part of the USB-UART FTDI chip unfortunately has latency that does not allow for low-latency ping-pongs between the core and the computer. It has not been implemented here, as this only serves as a demo, but additional FIFOs would be required to actually stream data through this interface. It works good enough for debugging and demo purposes, though.

In place of the UART-AXIS bridge, Ethernet-AXIS bridge was considered. Especially that the most basic Ethernet interface supports much higher data rates than UART does. If the whole setup would be used in production, then definitely UART is not a good fit, as even 300kB/s is too little to keep up with live sound streams. For this setup, $2 \cdot 44100 \cdot 4 = 352.8$kB/s would be required at minimum to consistently stream the live audio data into the chip.

5 Results

For the purposes of testing the core, it was placed on the FPGA with the UART-AXIS bridge. Data was sent through the on-board FTDI chip with the Python script `/py/connect.py`. The results were automatically compared with the reference implementation.

The Python script loads the samples from a file and then offsets them according to the requested delay. Random variable with range $[-0.1, 0.1]$ is added to each sample to simulate noise. The design has its reset deasserted just before running the script. Since there is no checksum implemented, any kind of inconsistency on the UART lines can make the results fail. This especially happens when the FTDI chip deasserts the lines for a longer period.

A run is considered inconsistent if any of the 3 delays does not equal with the reference result *and* the reference is equal to the theoretical result. The results are consistent with the reference around 99.5% of the time if the above conditions are used. The

noise does not matter w.r.t. consistency.

It also became apparent that a proper clock source is required, so MCMM was used through the Clocking Wizard. Otherwise there are some hold slack issues in the FFT core. The design could also be further optimized by removing some of the FIFOs depth, especially on the stream_in port. Unfortunately, cosimulation with generated RTL has (documented) issues when using the selected Data-Driven Control scheme, so the iteration time for simulating the RTL is much longer and it was simply not feasible to experiment with that further. Using a more programmatic approach to simulating the resultant IP core could be something to consider in the future – perhaps using Verilator to mock the core post-synthesis in Vitis would be a good option.

6 Conclusions

The core for accelerating the calculation of the sound source location was created. It was tested in demo conditions with results, that are consistent with the "perfect" algorithm running on a PC.

Even though systems for High Level Synthesis can be difficult to start working with, compared with regular writing of RTL, they speed up the work a lot. This is especially true when one performs a lot of math computations and data moving.

The project result is not a production-grade core, but rather a demo that serves as an example. It could obviously be integrated with more cores to create a complete pipeline, but that might require optimizing the core a bit.

References

[1] Knapp, C. H. and Carter, G.C., *The Generalized Correlation Method for Estimation of Time Delay.* IEEE Transactions on Acoustics, Speech and Signal Processing

[2] Schau, H. and Robinson, A., *Passive source localization employing intersecting spherical surfaces from time-of-arrival differences* IEEE Transactions on Acoustics, Speech, and Signal Processing

[3] RealDigital *Urbana: Reference and Schematic*

[4] AMD *Vitis High-Level Synthesis User Guide* UG1399

[5] AMD *Fast Fourier Transform LogiCORE IP Product Guide* PG109

[6] AMD *Vivado Design Suite Tcl Command Reference Guide* UG835

[7] ARM *AMBA AXI-Stream Protocol Specification*