Audio Engineering Society

Convention Paper 10250

Presented at the 147th Convention
2019 October 16 – 19, New York

An Open Audio Processing Platform using SoC FPGAs and Model-Based Development

Trevor Vannoy[1,2], Tyler B. Davis[2], Connor Dack[2], Dustin Sobrero[2], and Ross K. Snider[1,2]

[1]*Electrical & Computer Engineering, Montana State University, Bozeman, MT USA 59717*
[2]*Flat Earth Inc, Bozeman, MT USA 59718*

Correspondence should be addressed to Trevor Vannoy (trevorvannoy@montana.edu)

ABSTRACT

The development cycle for high performance audio applications using System-on-Chip (SoC) Field Programmable Gate Arrays (FPGAs) is long and complex. To address these challenges, an open source audio processing platform based on SoC FPGAs is presented. Due to their inherently parallel nature, SoC FPGAs are ideal for low latency, high performance signal processing. However, these devices require a complex development process. To reduce this difficulty, we deploy a model-based hardware/software co-design methodology that increases productivity and accessibility for non-experts. A modular multi-effects processor was developed and demonstrated on our hardware platform. This demonstration shows how a design can be constructed and provides a framework for developing more complex audio designs that can be used on our platform.

1 Introduction

As audio applications and algorithms become more demanding in terms of performance and latency, application-specific high performance hardware is becoming necessary. Due to their inherently parallel nature, SoC FPGAs are the ideal target architecture for high performance, low latency digital signal processing (DSP), and are the only platform that provides high performance with deterministic latencies. However, development and testing of audio signal processing algorithms for SoC FPGAs is a lengthy process requiring both hardware and software skills.

Fortunately, advances in computer-aided design software are making the hardware side of SoC FPGA development less cumbersome. By utilizing Simulink and automatic code generation, we are able to eliminate the lengthy process of implementing algorithms in a low-level hardware description language (HDL), making development more accessible for people who aren't familiar with HDLs such as VHDL or Verilog. In addition to simplifying hardware development, we also seek to streamline embedded software development for SoC FPGAs by automatically creating device drivers.

Utilizing off-the-shelf Intel SoC FPGA development boards and custom audio daughter boards, we are developing an open source audio processing platform[1]. The intent of this platform is to allow students and

[1]https://github.com/fpga-open-speech-tools

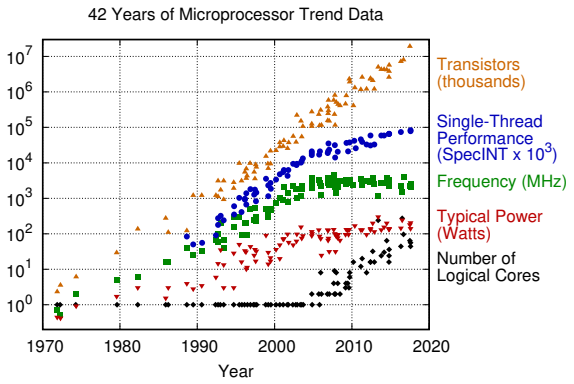

Fig. 1: Microprocessor trends [1]. CPU performance is no longer increasing linearly with increased number of transistors. Original data up to the year 2010 collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. New data collected by K. Rupp.

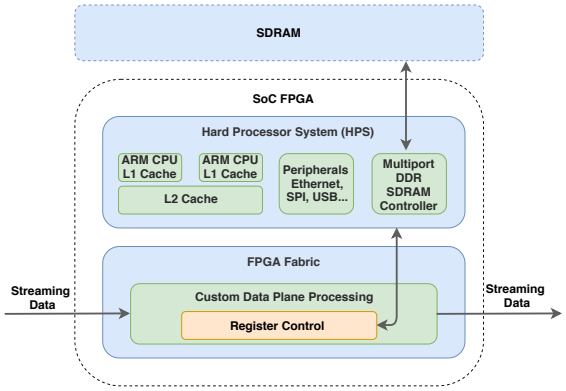

Fig. 2: SoC FPGAs contain a complete computer system known as the Hard Processor System (HPS) as well as the FPGA "fabric" that is composed of various types of logic blocks. The FPGA fabric allows the creation of custom hardware.

researchers in speech, hearing, and audio fields to prototype and deploy custom audio processing systems.

The rest of the paper is as follows. Section 2 presents an overview of SoC FPGAs and model-based design. Section 3 presents related work using model-based development and FPGAs in audio processing. We then detail the hardware and software architecture of our open source SoC FPGA audio processing platform in Section 4. Finally, Section 5 discusses our development process and efforts to make SoC FPGA development more accessible. Example applications are presented in Section 6, followed by conclusions and future work in Section 7.

2 Background

2.1 SoC FPGAs

Given that traditional CPUs are no longer getting significantly faster as more transistors are added, as shown in Fig. 1, the way to increase performance is through parallel computer architectures.

FPGAs are integrated circuits that contain a matrix of logic blocks with programmable interconnects, which is referred to as the FPGA "fabric". Computation in the FPGA fabric is inherently parallel, so extra performance can be gained by creating a larger FPGA with more logic blocks. The most numerous and primitive of these logic blocks are lookup tables that can implement arbitrary logic functions. Other logic blocks commonly used in DSP applications include memory blocks, multipliers, and adders. VHDL or Verilog is used to describe and abstract the logic functions and to create the signals that are routed between the logic functions. The logic described by the HDL is then converted to a bit stream by the vendor's synthesis tools. This bit stream file, when loaded into the FPGA, configures the FPGA with the desired logic functionality.

SoC FPGAs combine the flexibility of the FPGA fabric with a full computer system, as shown in Fig. 2, providing a full embedded system. The computer system inside the FPGA is referred to as the Hard Processor System (HPS) because it is "hardened" in the integrated circuit, rather than built with logic blocks in the FPGA fabric. Linux is commonly used as the operating system for the HPS because it is modular, flexible, and has many programs and libraries already built for it. While Linux is not a real-time operating system with deterministic latency, this is not a problem because real-time functionality is put in the FPGA fabric where latencies are deterministic.

2.1.1 Latency

FPGAs provide the lowest possible latency of any general purpose computational device because data interfaces can be connected directly to the FPGA fabric through the FPGA's I/O pins. Furthermore, the timing

of the computational output is known to the clock cycle, which never changes. This provides low latency that is deterministic, which is not possible with traditional CPU architectures containing caches. In CPUs, data access times can vary greatly due to data not being in the cache. Other indeterminate timing effects include waiting for DRAM refresh cycles and waiting for the operating system to perform housekeeping tasks.

2.1.2 High Performance DSP

FPGAs provide the highest performance DSP by providing thousands of multipliers, adders, and memory blocks that run in parallel. Intel provides up to 8,736 floating-point DSP blocks [2] in one FPGA; one DSP block [3] contains one single-precision floating-point multiplier and one adder, which provides 11.8 trillion floating point operations per second (FLOPS). In comparison, an Intel i7-8700K CPU is capable of 355.2 billion double-precision FLOPS or 710.4 billion single-precision FLOPS [4]. Such processing power may seem excessive for simple audio processing operations such as filtering, however this hardware provides an avenue to perform real-time audio processing for complex applications such as beamforming with many microphones.

2.2 Model-based Design

Model-based design allows developers to design at the system-level across multiple domains (mechanical, electrical, etc.). Operating at this abstraction level helps increase productivity by masking low-level details. Modeling and simulation is typically done with a graphical programming language, such as Simulink. When the system is ready to deploy, automatic code generation is utilized, reducing development time.

3 Related Work

Model-based design has seen great success in the automotive industry, with General Motors saving an estimated 24 months of development time on a hybrid powertrain [5]. In the audio processing world, model-based design is gaining traction as a way to reduce development time and complexity [6, 7, 8]. This is of particular interest for FPGA development, which has traditionally been difficult and time consuming. FPGA-based audio applications that have taken advantage of model-based design include beamforming [9] and virtual reality [10] applications.

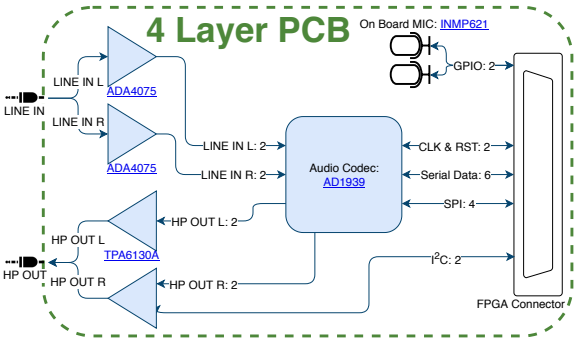

Fig. 3: Audio Mini hardware block diagram.

4 System Architecture

The following section describes the hardware and software architecture of our SoC FPGA audio processing platform.

4.1 Hardware

The hardware system comprises Terasic's DE10-Nano FPGA development kit and Flat Earth's Audio Mini daughter card that plugs into the DE10-Nano. A block diagram and 3D rendering of the daughter card are shown in Fig. 3 and Fig. 4, respectively. The Audio Mini has a 3.5 mm line input, a 3.5 mm headphone output, and two on-board digital microphones.

Audio signals are passed into the board through the 3.5 mm line in audio jack. The right and left channels are split and passed through low noise amplifiers, converting the single-ended audio signals into differential audio signals. The differential signals are then digitized by the audio codec (AD1939) and passed to the FPGA through the board-to-board 40-pin headers.

After being processed by the FPGA, the digital audio is passed back to the codec and converted to differential analog audio. The audio is then passed to the low noise headphone amplifier and converted to single-ended signals. Furthermore, the audio traces are length-matched to ensure that the signals arrive at the codec simultaneously, preventing delays.

The frequency response of a passthrough system using the Audio Mini is shown in Fig. 5, exhibiting a flat response above 400 Hz.

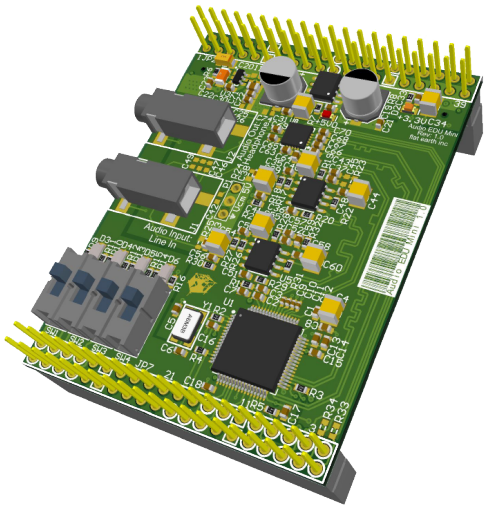

Fig. 4: Audio Mini 3D rendering.

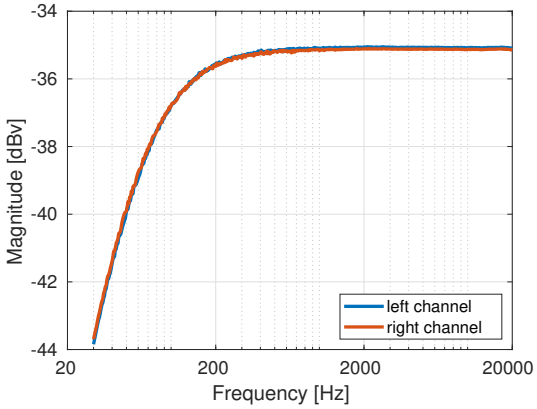

Fig. 5: Frequency response of the Audio Mini.

4.1.1 Latency

In 2010, Wang et al. [11] tested audio latency on all major desktop operating systems; the lowest latency achieved was 1.68 ms on Linux, followed by 3.54 ms on Mac OSX. For Android-based systems, Zaporowski et al. [12] found a best-case latency of 8.06 ms. Using a buffer size of 2 on a BeagleBone Black, McPherson and Zappi [13] measured a latency of 1.02 ms. On the Audio Mini with the codec running at 192 kHz, a total system latency was measured at 180 μs—almost an order of magnitude less than the Linux system and less than one fifth the latency of the BeagleBone Black system. The codec itself contributed practically all of the latency.

4.2 Software

The software architecture for our open audio processing platform, shown in Fig. 6, comprises three main components:

1. A remote client application that provides a user interface to control algorithm and device parameters
2. A server that communicates with the client and abstractly controls devices and algorithm parameters via device drivers
3. Device drivers that control devices, which can be either external hardware or components in the FPGA fabric

The client application is responsible for providing a means to remotely control the hardware and algorithms through a graphical user interface (GUI), which is programmed using C# and the Unity game engine. Unity allows the creation of cross-platform applications that can be run on Windows, Linux, macOS, Android, and iOS. The client and server communicate over TCP/IP, which allows multiple clients to be connected the same server.

When a client first connects to a server device, the server sends version information back to the client. The client then checks this version information to ensure compatibility. This version handshaking is important because the SoC FPGA could be programmed to run a different algorithm than what the client expects. To control hardware and algorithm parameters, the client sends messages that contain the device name, register name, and value.

When the server receives a register write command from the client, it parses the message and passes the value to the appropriate device driver. These drivers are in essence programs that perform all the operations required to communicate with the hardware. Device drivers implement the specific interfaces to communicate with the various devices, masking these details from the user; this in turn creates a more generic interface between software and hardware, which makes the software easier to write. Finally, when a register

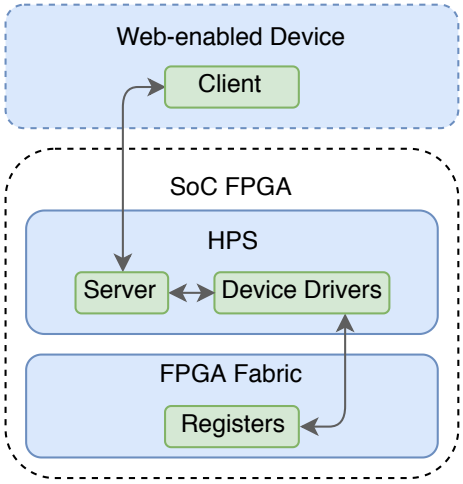

Fig. 6: Software architecture block diagram. The client application sends and receives commands to and from the server running on the HPS. The server runs commands and interacts with device registers via device drivers.

value changes, the server publishes this change to all connected clients.

5 Development Process

Although there are many advantages to using SoC FPGAs, as detailed in Section 2, these advantages come at the cost of increased development complexity. To make development easier, we are implementing a model-based hardware/software co-design methodology that will enable rapid prototyping and implementation of embedded DSP algorithms using SoC FPGAs.

A typical development workflow for signal processing applications starts with high-level models and simulations in MATLAB or Simulink. Once the developer is satisfied with the simulation results, they implement the algorithm for the target hardware using a low-level language like C or VHDL/Verilog. After implementation, they verify the algorithm. At any of these steps, they may have to go back to one of the previous steps to modify or correct the algorithm, then propagate the change, which is time consuming.

The development workflow presented here seeks to eliminate as much of the hardware-specific implementation as possible by using model-based development coupled with code autogeneration. This development process comprises the following steps, which will be detailed later:

1. Develop and verify the algorithm in Simulink
2. Autogenerate VHDL code and Linux device drivers from the Simulink model
3. Integrate the autogenerated code into a Quartus project and create the FPGA programming files
4. Create a client GUI to control the algorithm, if desired

While this may seem like many steps, much of it is automated, and more of it will be automated in the future. The primary advantage of this workflow is that the majority of the hardware implementation is automatically generated; consequently, when changes to the algorithm are made, the rest of the implementation is automatically updated.

5.1 Simulink Modeling

Development starts with creating a Simulink model, an example of which is shown in Fig. 7. At the top level of the model, algorithm parameters and simulation stimuli are defined outside of the algorithm and are passed into the main subsystem, which is synthesized into VHDL.

In order to handle streaming data, like audio, the Simulink model implements the data, channel, valid, and error signals in Intel's Avalon Streaming interface. For stereo audio applications, the channel signal specifies whether the incoming data is the right or left channel. The valid signal indicates that the incoming data is valid.

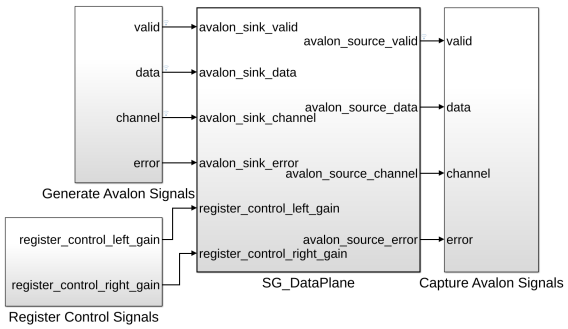

Fig. 7: The top level of an example Simulink model.

5.2 Code Autogeneration

By using HDL-compatible Simulink blocks, MATLAB's HDL Coder toolbox can generate VHDL or Verilog code. In the workflow presented here, the HDL Coder toolbox is used to generate the majority of the code, but additional VHDL code must be generated to make the Simulink model compatible with Intel's Avalon Streaming interface and development tools. Using information from the Simulink model, we generate this additional VDHL code and the C code for the device driver that controls the algorithm parameters.

5.3 Project Creation

To program and use the FPGA, Intel's FPGA design software, Quartus, must be used. Once the code has been generated from the Simulink model, a Quartus project needs to be created so the model can be put on the SoC FPGA. The new project is based off a reference design, so developers only need to add their custom components (i.e. audio processing algorithms) to the project. When this is done, the project can be programmed into the FPGA.

5.4 GUI Creation

If desired, a GUI can be created to control algorithm and hardware parameters. Depending on the application, the GUI can be manually created or autogenerated. To facilitate GUI autogeneration, developers put desired GUI information in the Simulink model; the client uses this information to autogenerate the GUI when connecting to a server.

If the application doesn't require a GUI, the algorithm can be controlled using the Linux command line on the SoC FPGA, or it can run without user interaction.

6 Applications

To demonstrate the utility of the hardware platform and development process, this section highlights some sample applications that have been developed for audio and speech processing.

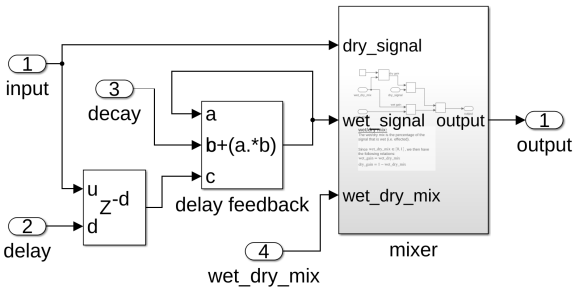

Fig. 8: Simulink model of a simple echo.

6.1 Multi-effects Processor

A multi-effects processor that allows users to create a custom signal chain in the FPGA fabric and control effect parameters from a web application has been developed to illustrate the flexibility of using SoC FPGAs for audio processing. This multi-effects processor is fully programmable and gives users full control over the effects chain.

Following the development process in Section 5, audio effects, such as the echo model shown in Fig. 8, are created in Simulink. In the Quartus project, users can put any combination of audio effects in the FPGA fabric.

The multi-effects processor GUI shown in Fig. 9 is autogenerated based on parameters provided in the Simulink model. These parameters include the names, minimum and maximum values, and UI element types (e.g. dial or slider) for each effect parameter. This information is sent to the client, and then the client autogenerates the GUI. To control effect parameters, the client sends messages containing the effect name, parameter name, and value. The server then interprets these messages and updates the effect parameters. Effects can be bypassed by turning them off in the GUI, and they can be rearranged by dragging them.

Eventually, a community library of audio effect blocks and preconfigured effects chains can be made, allowing users to collaborate and share designs. While interesting on its own, this application can be used as a reference design for more complex audio processing applications.

6.2 Linux Sound Card

FPGAs can be difficult to develop on, and many current audio signal processing groups use programs developed

Fig. 9: An example of the multi-effects processor GUI. This example shows an effects chain with a bitcrusher, flanger, ehco, lowpass filter, and gain control. The flanger and echo are bypassed. The plots in the bitcrusher and lowpass sections show a preview of the effects.

in languages such as C or MATLAB; therefore, in an effort to make our open audio processing platform more flexible, we developed software that allows users to access the audio codec as though it were a Linux sound card [14].

Once implemented, the AD1939 on the Audio Mini appears in Linux as a standard sound card. To emphasize this, we ported the open Master Hearing Aid application which was originally designed to operate on a BeagleBone Black [15]. This program runs real-time audio signal processing algorithms that allow a digital hearing aid to be tuned by the user.

6.3 Other Applications

Applications we are developing for the speech and hearing community include a simple hearing aid model [16, 17] and a real-time computational model of the auditory nerve that includes the synapse between the inner hair cell and auditory nerve [18].

7 Conclusion

The open source SoC FPGA audio processing platform presented here is a powerful development platform for both students and professionals. Moreover, the hardware platform has a latency of 180 μs, which is significantly faster than the latencies of other computing systems. Using model-based design and automatic code generation simplifies the complex SoC FPGA development process, enabling rapid prototyping of standalone audio processing applications.

We are currently directing development effort towards automating more of the development process. Future plans also include supporting hardware-in-the-loop simulation. In addition to the hardware presented here, we are also developing a higher performance prototyping platform as well as hardware for beamforming applications.

Acknowledgments

The authors would like to thank Aaron Koenigsberg, Alex Salois, Bailey Galacci, James Eaton, Justin

Williams, and Matthew Blunt for their help in testing and developing applications on our audio processing platform. This work was supported by the National Institutes of Health (Grant No: 4R44DC015443-02).

References

[1] K.Rupp, "42 Years of Microprocessor Trend Data," 2019, https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[2] Intel, "Intel Agilex F-Series FPGA and SoC FPGA Product Table," 2019, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/intel-agilex-f-series-product-table.pdf.

[3] Intel, "Intel Agilex Variable Precision DSP Blocks User Guide," 2019, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/agilex/ug-ag-dsp.pdf.

[4] Intel, "Export Compliance Metrics," 2018, https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf.

[5] EDN, "General Motors Develops Two-Mode Hybrid Powertrain With Model-Based Design," 2009, https://www.edn.com/electronics-products/electronic-product-releases/other/4367477/General-Motors-Develops-Two-Mode-Hybrid-Powertrain-With-Model-Based-Design.

[6] Ananthan, A., "Rapidly Prototyping and Implementing Audio Algorithms on DSPs Using Model-Based Design and Automatic Code Generation," in *Audio Engineering Society Convention 123*, 2007.

[7] Bentall, N., "Audio and Control: Simulation to Embedded in Seconds," in *Audio Engineering Society Conference: 43rd International Conference: Audio for Wirelessly Networked Personal Devices*, 2011.

[8] Tradowsky, C., Figuli, P., Seidenspinner, E., Held, F., and Becker, J., "A New Approach to Model-Based Development for Audio Signal Processing," in *Audio Engineering Society Convention 134*, 2013.

[9] Salom, I., Celebic, V., Milanovic, M., Todorovic, D., and Prezelj, J., "An Implementation of Beamforming Algorithm on FPGA Platform with Digital Microphone Array," in *Audio Engineering Society Convention 138*, 2015.

[10] Fohl, W. and Hemmer, D., "An FPGA-Based Virtual Reality Audio System," in *Audio Engineering Society Convention 138*, 2015.

[11] Wang, Y., Stables, R., and Reiss, J., "Audio Latency Measurement for Desktop Operating Systems with Onboard Soundcards," in *Audio Engineering Society Convention 128*, 2010.

[12] Zaporowski, S., Blaszke, M., and Weber, D., "Measurement of Latency in the Android Audio Path," in *Audio Engineering Society Convention 144*, 2018.

[13] McPherson, A. and Zappi, V., "An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black," in *Audio Engineering Society Convention 138*, 2015.

[14] Steinsbo, B., "DE1-SoC Alsa Audio," 2015, https://rocketboards.org/foswiki/Projects/DE1SoCALSAAudio.

[15] Herzke, T., Kayser, H., Loshaj, F., Grimm, G., and Hohmann, V., "Open signal processing software platform for hearing aid research (openMHA)," in *Proceedings of the Linux Audio Conference*, pp. 35–42, 2017.

[16] Snider, R., Casebeer, C., and Weber, R., "An open computational platform for low-latency real-time audio signal processing using field programmable gate arrays," *J. Acoust. Soc. Am.*, 143, 2018.

[17] Casebeer, C., Weber, R., and Snider, R., "An Open FPGA-based Computational Platform for Developing Real-Time Cochlear and Hearing-Aid Processors," 2018, poster presented at the Association for Research in Otolaryngology (ARO) 41st Annual MidWinter Meeting, San Diego CA.

[18] Bruce, I. C., Erfani, Y., and Zilany, M. S., "A phenomenological model of the synapse between the inner hair cell and auditory nerve: Implications of limited neurotransmitter release sites," *Hearing research*, 360, pp. 40–54, 2018.