PROGRAMMING FPGA PLATFORMS FOR REAL-TIME AUDIO SIGNAL PROCESSING IN C++

Pierre COCHARD (pierre.cochard@insa-lyon.fr) [1], **Maxime POPOFF**[1], **Romain MICHON**[1], and **Tanguy RISSET**[1]

[1]**INSA Lyon**, **Inria**, CITI, EA3720, 69621 Villeurbanne, France

ABSTRACT

Syfala is a toolchain aiming to facilitate the development of real-time audio Digital Signal Processing (DSP) applications on Field-Programmable Gate Array (FPGA) platforms. Since its introduction in 2021, the project has been relying on the FAUST programming language as its primary programming interface. The resulting C++ code generated by the FAUST compiler can be easily translated into Hardware Description Language (HDL) code using High Level Synthesis (HLS), thereby facilitating integration into FPGA designs. However, FPGA platforms may in some cases require very specific tuning and optimizations in order to reach their full potential, which FAUST can sometimes fail to provide, because of its platform-agnostic nature.

In this technical paper, we present a new way of using Syfala with C++ and HLS. This potentially allows for a better control over parallelization and pipelining as well as for potentially more optimized and efficient code for AMD/Xilinx FPGA platforms. This method can greatly facilitate the implementation of complex DSP algorithms on FPGAs for C++ programmers. Finally, we highlight the performance gains obtained by using this methodology with respect to the original Syfala toolchain starting from FAUST programs.

1. INTRODUCTION

Field-Programmable Gate Array (FPGA) platforms offer unmatched performances and features in the context of real-time audio Digital Signal Processing (DSP). With the exception of Graphics Processing Units (GPUs) where comparable results can be achieved [1,2], highly parallelizable algorithms can be run more efficiently on an FPGA than on most other computing platforms (i.e., Central Processing Units, etc.). FPGAs don't need buffering and can run audio DSP algorithms at a very high audio sampling rate (in the megahertz range), providing extremely low latency [3]. They can also be interfaced with large numbers of low-level hardware components (such as audio codecs) through their numerous GPIOs.

FPGAs are very difficult to program for a software programmer. However, audio DSP programs can be expressed using dedicated languages (i.e., FAUST [4]) and might be easier to compile on FPGAs than general purpose programs because of their limited scope. This technical paper has been written as a tutorial to help C++ audio programmers to efficiently compile their programs down to FPGA. It takes advantage of the methodology that emerged from the Syfala toolchain which compiles FAUST programs on FPGA [3,5]. Syfala (which is open source [1]) is now able to compile C++ audio programs on FPGA using AMD/Xilinx High Level Synthesis (HLS) tools. In this paper, we explain the concepts that software programmers must know to master HLS in order to compile efficient C++ audio programs on FPGA.

The next section presents a review of the use of FPGAs in the audio industry as well as in academia. It also describes how audio DSP programs should be compiled for FPGA. Section 3 presents the Syfala toolchain and Sections 4-5 explain how to adapt existing C++ programs. Finally, Section 6 presents some performance results obtained by this new FPGA compilation flow.

2. AUDIO ON FPGA: A STATE OF THE ART

FPGAs can be found in a broad range of audio products: Dante audio interfaces,[2] digital mixers (e.g., Midas M32 [3]), audio processing modules (e.g., Antelope Audio Synergy Core [4]), sound synthesizers (e.g., Novation Summit Keybord [5]), etc. Similarly, they've been used in academia for various applications, mostly targeting computational performance [6–8] (to name a few).

2.1 Language and Architecture for audio DSP

The advantages offered by FPGAs come at the cost of ease of programming. Indeed, FPGAs are considered as "hardware" and are low-level in nature, making them extremely hard to program by non-specialized engineers. They require the use of Hardware Description Languages such as VHDL and Verilog, which work in a very different way than programming languages aiming software and that are traditionally used for real-time audio DSP such as C/C++.

[1] https://github.com/inria-emeraude/syfala/ – All URLs in this paper were verified on May 21, 2024.
[2] https://www.audinate.com/
[3] https://www.midasconsoles.com/
[4] https://en.antelopeaudio.com/
[5] https://novationmusic.com/

Most FPGA manufacturers (i.e., Intel/Altera and AMD/Xilinx) now provide High-Level Synthesis (HLS) tools allowing for the programming of FPGAs using C or C++ [9, 10]. Since programs written in these languages are often not optimized to be run on an FPGA, directives and pragmas can be used to provide additional information to help the HLS tool make the right decisions in terms of pipelining, parallelization, etc.

Audio DSP implies a broad range of constraints when it comes to real-time implementation (e.g., buffering, control vs. audio rate, interfacing with audio drivers, etc.). These constraints – despite the potential use of HLS – add up to the aforementioned difficulties related to programming FPGA platforms. As far as we know, very few projects (not to say only one [11]) used HLS on FPGA in the context of real-time audio DSP as it is done in Syfala.

Since the inception of digital audio programs, specialized languages and architectures tailored for audio Digital Signal Processing (DSP) have been proposed. Audio signal processing often demands significant computational power, implementing mathematical concepts like FIR and IIR filters, FFT, etc., at a high rate (i.e., audio sampling rate). It's important to note that a characteristic of audio signal processing is that some computations are performed for each sample, while others, like those used for audio control, can be scheduled at a lower "control rate".

In the early days of digital audio, dedicated DSP units were employed in commercial systems. This trend persists today, exemplified by the continued use of specialized DSP units like Analog Devices SHARC DSPs. However, there is a growing trend in utilizing regular multi-core CPUs for audio purposes, as seen with ARM processors in Teensy [6] microcontrollers.

The selection of hardware for implementing audio signal processing is guided by two key metrics: computational power and computation latency. Computational power depends on the operations performed for each sample at sample rate; the more demanding the computations, the more powerful the hardware needs to be. Dedicated DSPs, GPUs, or FPGAs may be employed if standard CPUs are not sufficiently powerful. As for latency, it is often associated with the size of the sample buffer needed to perform the DSP computation. The nature of the Von Neumann processor architecture (i.e., a CPU connected to memory) necessitates buffering audio samples to circumvent memory bottlenecks. A typical size for an audio buffer in processing systems is 256 samples. At a sampling rate of 48kHz, a 256-sample latency corresponds to approximately 5 ms. FPGA technologies can can help in providing much smaller latency. For instance a latency of $11\mu s$ (analog to analog) was reached in [3].

An important decision in the design of audio signal processing involves selecting the programming language to express these algorithms. Numerous computer languages and environments have been dedicated to audio programs, with today's most popular choices including Max/MSP, PureData, and Faust [4]. Conversely, many audio developments, particularly in the industry, use general-purpose languages, mainly C/C++, but also Python, Matlab, or Java.

The Syfala toolchain allows us to map Faust programs onto FPGAs [3, 5], leveraging C++ code generated by the Faust compiler. It seems that many programmers express a desire to program FPGAs directly in C++ since it serves as their native audio programming language. In the following sections, we present guidelines for writing C++ code in a manner that facilitates efficient mapping onto FPGAs using Syfala.

FPGAs present a programming model distinct from regular CPUs, as they can be regarded as "programmable circuits" in contrast to non-programmable digital circuits like ASICs. An FPGA is composed of many small *configurable* blocks, called Look-Up Tables (LUTs), capable of executing any logic function with 4 to 6 boolean inputs and storing the result in a 1-bit register. These LUTs are utilized to implement diverse digital circuits. Additionally, FPGAs incorporate various components such as embedded multipliers, known as *DSPs*, that facilitate the implementation, or *BlockRAMs* which can store several kilobytes of data.

Over the past decade, FPGAs have incorporated *processing systems*, which are system-on-chip entities featuring complete processors and associated peripheral drivers. For example, the Digilent Zybo Z20 development board includes a dual-core ARM processor capable of operating at 600MHz. This processing system supports running a complete operating system and facilitates communication with the hardware design mapped onto the FPGA LUTs. This processing system is used to offload the hardware component from computationally less time-sensitive tasks.

2.2 Mapping audio DSP on FPGAs

The process of mapping an audio application onto an FPGA involves making crucial design decisions, extensively discussed in [5]. Typically, sample-rate computations are implemented on the FPGA LUTs, while control rate computations are executed on the processing system. Another important choice is the buffering of samples; one can choose to compute one sample at a time (one-sample strategy) or a buffer of samples at a time (multi-sample strategy). Fig. 1 illustrates a generic architecture for an audio application running on an FPGA.

The hardware/software partitioning is shown on Fig. 1. It distinguishes the control-rate and the sample-rate computations. The control part runs on the processing system (refered to as `app.elf` on Fig. 1). It is itself interfaced with the FPGA board DDR memory and also to the host computer (or any computer on the network) in order to provide a way to control the audio DSP computations. The sample-rate computation are implemented on the FPGA (refered to as *DSP IP* on Fig. 1). This represents the core computation of the DSP kernel. It is interfaced with the FPGA board DDR memory, but it can also communicate directly with the `app.elf` application (i.e., the DSP IP is usually seen as a peripheral driver on the processing system). Finally, the IP is interfaced with one or several audio codecs via the I^2S protocol.

The next section explains how the system presented on

[6] `https://www.pjrc.com/`

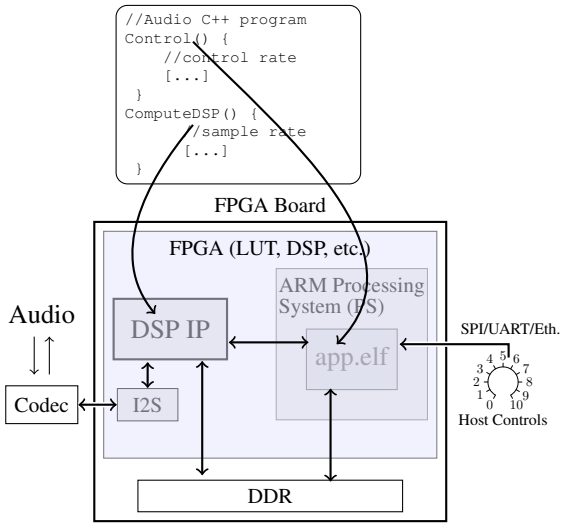

Figure 1. Overview of the deployement of audio programs on FPGAs.

Fig. 1 has been implemented in the Syfala toolchain, which compiles FAUST programs down to FPGA.

3. FROM FAUST TO FPGA: THE SYFALA TOOLCHAIN

Syfala was introduced in [3, 5]. It is a tool to facilitate the development of real-time audio DSP applications on FPGAs using the FAUST programming language [4]. Programs written in FAUST can be exported to AMD/Xilinx FPGAs on Digilent boards such as the Zybo Z7 or the Genesys. [7]

3.1 Using FAUST as an Input to Syfala

Among the features provided by Syfala, one can cite the following ones:

- Many kinds of audio codecs are supported;
- The DSP IP can be used as a hardware accelerator on embedded Linux running on the FPGA board [12];
- SPI [3], MIDI, HTTP, and OSC control are available on the processing system [12];
- Ethernet can stream audio in and out from the FPGA [13];
- It is compatible with dedicated multichannel audio interfaces providing unique specific features such as low latency and low-cost spatial audio.

The "traditional" way of using the Syfala toolchain is to call the FAUST compiler on a `.dsp` file (i.e., a FAUST program) in order to generate its C++ code equivalent, which is then going to be translated into Hardware Description Language (HDL) code using High Level Synthesis tools.

The resulting code describes the DSP computations as a flow (or circuit) of hardware-level logic operations and interconnections, forming an ensemble that we call an IP (for Intellectual Property). In this context, an IP can be viewed as a higher-level 'block' abstraction that can be connected to different other blocks through a variable number of input and output ports, as shown on Fig. 2.

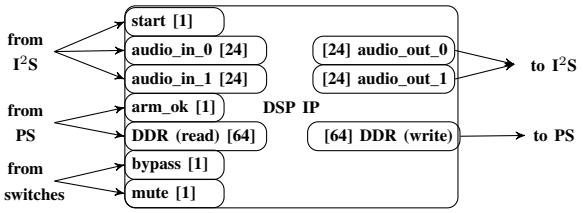

Figure 2. A logic view of the DSP IP as generated in the Syfala compilation toolchain.

When running, the data-flow design can be described as follows: audio data will be sent from the codec(s) to the I^2S at a predetermined rate (called audio-rate or sample-rate), and then transmitted to the DSP IP, alongside a 'start' signal, which will enable the computations. Once these computations are done, the output samples are sent back to the I^2S, and in turn to the audio codec(s), bit-by-bit.

Since this is all done at a hardware level, there is no additional latency introduced in the system (from, for instance, an operating system or any other hardware/software component), which makes real-time sample-by-sample processing possible in this context. The main requirement is that the computations are still done within the time limits set by the audio clock, otherwise making the flow discontinuous and hence degrading the resulting audio signal.

In order to guarantee that the latency stays below this limit, FPGAs can rely on their great parallelization potential, which on the other hand will be limited by the available on-board logic resources. Indeed, since multiple signals are processed simultaneously on parallel branches, with the same chain of operations, it also means that the logic operators used on the FPGA are duplicated instead of being sequentially re-used. This can in some cases lead quite rapidly to resource saturation.

Therefore, when programming an FPGA, having control over this latency/resource trade-off is key. The programming language that is used should in this context provide flexible ways and tools to reach optimal balance for a given program.

3.2 Using C++ as Input to Syfala

While FAUST is undoubtedly a nice and easy way to create complex and fully-controllable audio DSP IPs on FPGA, in some cases – where, for instance, balancing resource utilization and computation latency becomes a critical issue – bypassing FAUST and programming the IP directly in C++ can become a more suitable solution.

The C++ code generated by the FAUST compiler is a translation of the data-flow graph representation of its input program. Due to the compiler inherent structure, the code translation will try to express the maximum level of parallelization that its graph naturally possesses. While the FAUST compiler has a few options that allow us to tune its

[7] `https://digilent.com/reference/programmable-logic/zybo-z7/`

code-generation process, it still in essence stays platform-agnostic, and will not – at least not for the time being – make any FPGA-specific optimizations. Furthermore, since the compiler already carries out several transformation and optimization passes, the resulting code is difficult to apprehend and optimize manually.

On the other hand – since Syfala is already relying on HLS – programming the DSP IP directly in C++ is entirely possible, as it does not introduce any overhead and very little change to the rest of the compilation flow. While being more time consuming than writing the equivalent FAUST program, writing directly in C++ offers a few advantages that are not negligible:

- better control over the generated HDL code output (lower-level language);
- more potential for micro-optimizations and balance control between FPGA resource utilization and computation latency, through the use of various directives and pragmas. [8]

On the other hand, this approach requires to be familiar with the Vitis HLS programming environment and guidelines, which includes, among other things:

- use of a limited version of C++ (up to C++14), as well as a few strict rules on the use of the language's features (such as types, data structures, casts, etc.);
- implement complex HLS interfaces, using pragmas, with a potentially steep learning curve;
- handle manually data exchange with ARM executables through Memory bus interfaces.

4. DSP IP GENERATION FROM C++

In practice, writing a custom C++ DSP IP implies the same steps as writing a regular C++ program: all the code can be written in one or multiple C/C++ source or header files, and it should be C++ 14 compliant (meaning it should also compile using a regular C++ compiler) and have an 'entry point' function (called a 'top-level function' in Vitis HLS), which we will describe in the following sub-sections.

4.1 Top-Level Function Interface

The top-level function of a custom C++ IP can be seen as the equivalent of the 'main' function in a regular C or C++ program, and its arguments should be considered as the declaration of the IP's input and output ports, with:

- non-pointer arguments representing input (read) ports only;
- pointer or array arguments representing input or output ports (read, write, or both).

The listing in Fig. 3 shows a basic example of a DSP IP top-level function, made out of two audio input ports, two audio output ports, two additional '*bypass*' and '*mute*' input ports, as well as a few memory bus interfaces (the *arm_ok*, and *DDR* ports). As one can see, all of these interfaces are specified with a set of HLS-specific *pragmas*, which can themselves be configured with a few additional parameters. These top-level function arguments are explained below.

```
void syfala (
        sy_ap_int audio_in[2],
        sy_ap_int audio_out[2],
           int arm_ok,
        float* DDR,
          bool bypass,
          bool mute
) {
#pragma HLS array_partition variable=audio_in
    type=complete
#pragma HLS array_partition variable=audio_out
    type=complete
#pragma HLS INTERFACE s_axilite port=arm_ok
#pragma HLS INTERFACE m_axi port=DDR
    latency=30
[...]
```

Figure 3. Example of a kernel DSP top-level function used to generate the DSP IP with Vitis HLS.

4.1.1 Audio Input/Output Signal Types

In standard Syfala designs, audio signals are conveyed from and to the audio codecs as streams of 24-bits integers, by default (16 or 32 bits can be used as well). The top-level argument type `sy_ap_int`, which is used for audio input and output ports, is a specialization of a generic fixed-point integer type defined by Vitis HLS in its core libraries. Its width can be changed using the `-sample-width` flag in the syfala command line interface, but it cannot be changed to single or double precision floating point types.

Since audio DSP programs are usually processing float or double-based signals, a few convenience functions (`ioreadf` and `iowritef`) and types for making the proper conversions have been added in the Syfala API headers, which can be directly included in the target C++ file. From these header files, it is also possible to access some useful compile-time data, such as the project's sample rate, sample width, or targeted board model.

4.1.2 PS-PL Memory Interfaces

The two following top-level function arguments (*arm_ok*, *DDR*), and their respective pragma settings, show the use of *Advanced eXtensible Interface* (AXI4) memory buses, in order to create and enable shared memory areas between the DSP IP (which dwells on the Programmable Logic, or PL) and the Processing System (PS). This will allow us, among other things, to share data between the DSP IP and an ARM CPU-based executable, both running simultaneously on the same System on Chip (SoC).

In this context, data can either be shared using the DDR memory controller (through an AXI4 bus), or smaller intermediate PL Block RAMs (BRAM), using a lighter subset of the protocol (AXI4-Lite). In Syfala, these interfaces are implemented by default on standard designs, and may be used for multiple purposes (see Section 5.1). The `arm_ok`

[8] Directives and Pragmas are provided by Vitis HLS to *tune* parallelization and allow for a balance between FPGA resource utilization and computation latency of the DSP IP design.

argument is used in this context as a synchronization point between the DSP IP and the ARM executable while the `DDR` pointer is used to store and retrieve floating-point data in DDR memory.

4.2 DSP Kernel Code

Inside the top-level function, the code dedicated to signal processing can be written like standard C/C++ DSP code. The only requirement is to switch back and forth between *float* and *sy_ap_int* in order to read the input signal arguments, and write back to the outputs, which can be easily done by taking advantage of some of the API functions, as shown below.

```
for (int n = 0; n < 2; ++n) {
    // Read audio inputs
    float s = Syfala::HLS::ioreadf(audio_in[n]);
    // Do something with s
    s *= 0.5s;
    // Write audio outputs
    Syfala::HLS::iowritef(s, audio_out[n]);
}
```

4.2.1 Sample-Block Configuration

By default, Syfala implements the *one-sample* stategy. It is nonetheless possible to configure the DSP IP to process a buffer of samples instead, for potential optimization purposes. Using the multi-sample strategy may succeed in decreasing the *latency per sample* performance of the DSP IP, taking advantage of the pipelining optimizations offered by Vitis HLS, which might be useful in some cases.

The multi-sample strategy is simply activated by adding the *–multisample <N>* flag to a syfala command. However, the DSP IP code also needs to be adapted, since it requires replacing the single-valued input and output ports of the top-level function by *first-in-first-out* (FIFO) arrays, which should be declared as C multidimensional arrays in the HLS context, as shown below.

```
void syfala (
    sy_ap_int audio_in[2][SYFALA_NSAMPLES],
    sy_ap_int audio_out[2][SYFALA_NSAMPLES],
    [...]
){
#pragma HLS INTERFACE ap_fifo port=audio_in
#pragma HLS INTERFACE ap_fifo port=audio_out
#pragma HLS array_partition variable=audio_in
#pragma HLS array_partition variable=audio_out
```

4.2.2 Internal Loops Pragmas

Finally, code optimization in the HLS environment is usually done using various sets of pragmas, which can be viewed as higher-level hints or instructions given to the compiler to process a specific part of the code in a certain way. These pragma declarations usually target an individual variable, function or loop, and can often be tuned with various parameters.

The complete mastering of all Vitis HLS pragmas is out of the scope of this paper, but one needs to know that many adjustments can be done with *loop unrolling* and *loop pipelining*. Loop unrolling (`HLS unroll` pragma as below) is used to to increase parallelism by using additional resources. If a loop is not unrolled, it will be implemented sequentially (i.e., no additional resources) and Vitis HLS will try to pipeline the loop, i.e., overlap successive iterations without using additional resources.

In the code below, 4 iterations of the first loop will be executed in parallel and the second loop will be pipelined. Increasing the unroll factor will reduce the overall latency and increase resource usage:

```
for (int i = 0; i < NMODES; i++){
    #pragma HLS unroll factor=4
    for (int n = 0; n < SYFALA_NSAMPLES; ++n) {
        #pragma HLS pipeline
            [....]
        }
    }
```

4.3 Additional HLS Programming Environment Tools

Depending on the actual code and machine it is running on, using the Syfala toolchain to build a project based on a FAUST or a C++ file can take quite some time: one can expect a total build duration somewhere between 15 and 60 minutes. Hence, before running hardware synthesis, the two following points have to be checked, at least:

1. the DSP IP code is valid (it produces the outputs it is supposed to, from the inputs that it is given);
2. its implementation fits on the targeted platform, both in terms of FPGA resources and computation latency.

Fortunately, Vitis HLS offers tools that can make this kind of verification much faster (i.e., few seconds): C simulation (CSIM) and synthesis reports, which are also both well-integrated to the Syfala toolchain workflow.

4.3.1 Verifying Code With C Simulation (CSIM)

C simulation is an important Vitis HLS feature that allows us to test a C-written custom IP without having to get through the full synthesis process. Vitis HLS guarantees (with a few exceptions) that the outputs produced during simulation are going to be the same as what would happen in a "real" context of execution (implemented as hardware).

In Syfala, this simulation process can be directly ran from the command-line interface, and the Syfala documentation website [9] offers in its tutorials a few templates and guidelines on how to write a custom CSIM program.

4.3.2 Monitoring Latency & Resource Usage

Once it is established that the DSP IP code is valid, the other essential information to obtain is whether the design fits on the targeted board, both in terms of FPGA resource usage and latency. For the latter, it is also important to make sure that the DSP computations are within the time limits set by the FPGA clock rate, as well as the project's sample-rate constant. For instance, the DSP IP computations should last less then 2604 FPGA clock cycles for the computation of a single sample, for a sample rate of 48kHz and a FPGA clock rate of 125MHz.

[9] `https://inria-emeraude.github.io/syfala/`

After running the high-level synthesis step, the Syfala command-line interface automatically displays in the terminal an overview of Vitis HLS resources and latency estimates for the current program. If a more accurate report is needed, the *–accurate-use* flag can also be added to the command, which runs the implementation step of the design on the targeted platform. The *–accurate-use* flag usually adds a few more minutes to the HLS process to complete.

5. ARM EXECUTABLE INTERFACES

In some cases, it may be necessary to provide Syfala with a *custom* ARM executable in order to address specific needs, such as:

1. Using memory buses to share custom data and computations, as mentioned in Section 2.1;
2. Overriding the default configuration of the platform's peripherals (when using new external codecs);
3. Adding external software components, or implementing custom control interfaces, using for instance the Musical Instrument Digital Interface (MIDI) or Open Sound Control (OSC) protocols.

Custom ARM executables can easily be interfaced with the DSP IP and the system peripherals through a set of modules provided by the Syfala ARM API, which have separate implementations in baremetal and embedded-Linux build contexts.

An example of code for a minimal ARM executable, in embedded Linux context, can be seen in Fig. 4, in which a few different API modules are included, configured and initialized, all within the scope of the `main` function. This custom program can be added as a replacement to the default one provided by Syfala using the `-arm-target <file.cpp>` option in the command-line interface.

```
#include <syfala/arm/audio.hpp>
#include <syfala/arm/gpio.hpp>
#include <syfala/arm/uart.hpp>
#include <syfala/arm/dsp.hpp>

using namespace Syfala::ARM;
int main()
{
    XDSP dsp_ip;
    UART::data udata;

    GPIO::initialize();
    UART::initialize(udata);
    Audio::initialize();
    DSP::initialize(dsp_ip);
    DSP::set_arm_ok(&dsp_ip, true);
    Status::ok("Application ready, now running");
    // Main event loop:
    while (...) {
        // ...
    }
    return 0;
}
```

Figure 4. Example of minimal control program running on the ARM processor (i.e., `arm.elf` on Fig. 1).

The Syfala ARM API documentation, which describes in details all of the available modules and their interface, can be browsed on the Syfala documentation website. [10] Since the DSP and Memory modules constitute special cases insofar as they can be used to ensure interoperability between the ARM and the FPGA, we will also describe in the following subsections how they work and the benefits they are able to provide.

5.1 ARM/FPGA Interoperability

Sharing data and processing between PS and PL can prove essential for making the load lighter on the DSP IP. Hence the ARM executable can be used with specific strategies in mind, such as:

- The initialization of data arrays (wavetables) or constant expressions, by the ARM executable;
- Storing long delay-lines in DDR memory, instead of BRAM;
- Computing control-rate signals.

In order to implement these different strategies, sharing data between the DSP IP and the ARM executable is key, and can be easily achieved by leveraging the Advanced Microcontroller Bus Architecture (AMBA) interfaces set up on the System on Chip, specifically the AXI4 bus systems, which are natively supported in Vitis HLS.

5.1.1 AXI4-Lite Interfaces

The AXI4-Lite bus protocol is the quickest and simplest way to setup data exchange between a custom-made IP and an ARM executable. When adding an AXI4-Lite-registered argument to the top-level function of a custom C++ IP, Vitis HLS will automatically generate its matching 'driver' getter and setter functions in a C header/source file during synthesis. An example of the generated `Set/Get` function prototypes for the `arm_ok` argument seen in Fig. 3 is displayed below. The registered argument (`arm_ok` here) can either hold a single value, or an array of multiple values.

```
// xdsp.h
typedef struct {
    u64 Control_baseAddress;
    u32 IsReady;
} XDSP;

void XDSP_Set_arm_ok(XDSP *InstancePtr, u32 Data);
u32 XDSP_Get_arm_ok(XDSP *InstancePtr);
```

From the ARM executable point of view, the generated functions and their common C-struct handle (such as the one above) can then be accessed and called. Syfala will then take care of automatically including and linking the generated C sources when building the executable.

5.1.2 AXI4 Memory Interfaces

In order to read and write through the AXI4 bus, the process is somewhat similar, since it relies on the same Vitis HLS function-generation mechanism, but the data that is passed to the generated functions now consists in pointers:

[10] `https://inria-emeraude.github.io/syfala/`

```
// xdsp.h
u64 XDSP_Get_DDR(XDSP *InstancePtr);
```

For convenience, the Syfala ARM Memory module can be called to take care internally of passing valid and exclusive DDR memory addresses to the DSP IP, which can then be retrieved in the *DDR* pointer argument in the HLS code (as seen in Fig. 3).

```
[...]

#include <syfala/arm/memory.hpp>
using namespace Syfala::ARM;

#define WAVETABLE_LEN 16384

static void
initialize_wavetable(Memory::data& m) {
    int w = WAVETABLE_LEN;
    for (int n = 0; n < w; ++n) {
         mem.f[n] = sin((float)(n)/w * M_PI * 2);
    }
}

int main() {
    [...]
    Memory::data mem;
    Memory::initialize(dsp_ip, mem, 0, WAVETABLE_LEN);
    initialize_wavetable(mem);
    [...]
    Status::ok("Application ready, now running");
    while (...) {}
    return 0;
}
```

The code above shows an example of using the Syfala Memory module in order to setup a basic sinewave table on a custom ARM executable. Since the computations for the initialization of the wavetable are only made once, it is best to let the CPU handle them, instead of using FPGA resources. The resulting data is then shared internally with the DSP IP through the AXI4 bus, and handled in the HLS code using the *DDR* pointer.

5.1.3 AXI4 vs. AXI4-Lite

Although AXI4-Lite is usually faster when sharing a restricted amount of data, due to that fact that it does not implement all of the AXI4 bus features, it can also potentially lead to bottlenecks when this amount significantly increases. Unlike AXI4, AXI4-Lite does not support *burst* reads or writes, which means that multiple data cannot be retrieved in a single transfer. Consequently, and since read and write accesses are sequential, they can, by accumulating, rapidly catch up the latency penalty inherent to AXI4, which, on the other hand, will take advantage from its parallel data access capabilities when a large memory zone is involved.

Conversely, AXI4 will be less efficient at handling smaller data sets, which is amplified by the fact that it does not benefit from a higher-level interface to manage allocation offsets for individual or fragmented data, which should for now be manually handled by the user, as it has not yet been implemented either in the Syfala API.

Finally, AXI4-Lite adapters are always implemented in the Programmable Logic with Vitis HLS, alongside the DSP IP, meaning that it uses a lot more resources than its AXI4 counterpart, for which the actual memory is in the external DDR.

5.2 Adding External Control Interfaces

One additional reason to provide a custom ARM executable would be to bridge the DSP application with the rest of the system, both in terms of hardware and software components. In a baremetal context, this could mean for instance using the Serial or SPI interfaces to control DSP parameters or, on embedded Linux, taking advantage of USB or Ethernet interfaces to implement MIDI and/or OSC control.

```
[...]

static int gain_handler (
                [...]
            lo_arg** argv
               void* user_data
){
    XDSP* dsp = (XDSP*)user_data;
    float gain = argv[0]->f;
    XDSP_Set_gain(dsp, *(u32*)(&gain));
    return 0;
}

int main() {
    [...]
    auto osc = lo_server_thread_new("8888", error_hdl);
    lo_server_thread_add_method(osc, "/gain",
        "f", gain_handler, &dsp
    );
    lo_server_thread_start(osc);
    [...]
    Status::ok("Application ready, now running");
    while (...) {usleep(5000);}
    return 0;
}
```

The code above shows a basic implementation of OSC control in a custom ARM executable, running in an embedded Linux context, using the liblo library. [11] As one can see, the AXI4-Lite data exchange (which can be considered as 'control-rate' here) occurs in this context directly in the OSC callback function `gain_handler`.

6. RESULTS AND PERFORMANCE

We were able to conduct experimentation on a few complex audio programs, which were initially difficult to implement on the Zybo Z20 development board using FAUST. These examples include Wave Field Synthesis [14] algorithms, capable of supporting up to 10 sources for 256 output channels, biquad-based modal Reverb [15] (up to 18,000 modes), FIR filters (10 filters with 4000 coefficients).

It is difficult to assess the performance improvements between the design yield by FAUST and the design obtained from C++ as it heavily depends on the way the Faust and C++ programs have been optimized. In general, the performance can be improved by a factor of 10 (in terms of resource usage or latency).

This significantly helped us with the prototyping of complex spatial audio systems. For this, we took advantage of the FPGAs reprogramming capabilities and large number of GPIOs to interface them with large speaker arrays. In this context, using C++ allowed us to make significant optimizations, which would not have been possible using FAUST, especially when targeting small FPGA boards, such as the Zybo Z7, which have limited logic resources.

[11] `https://liblo.sourceforge.net/`

7. CONCLUSION AND FUTURE WORKS

This paper presents a method for writing C++ audio programs to map them efficiently on Xilinx FPGAs using the Syfala toolchain. It can help C++ programmers to take advantage of FPGA capabilities in terms of performances and latency. It also enables the implementation of costly algorithms such as modal reverbs, which are generally too computationally expensive to be run on a CPU in real time.

Finally, being able to use Syfala with C++ will allow us to interface the toolchain with other C++ based frameworks or programming environments (e.g., JUCE, SuperCollider, PureData with Heavy, Max/MSP with gen or Rnbo, etc.).

One of our research direction is to abstract the way pragmas are used in the C++ program so as to include them in the FAUST compiler.

Acknowledgments

This project has been partially funded by the French ANR (Agence Nationale de la Recherche) through the FAST project [12] (ANR-20-CE38-0001) and the Inria/Stanford Plasma Associate Team. [13]

8. REFERENCES

[1] T. Skare and J. Abel, "Gpu-accelerated modal processors and digital waveguides," in *Proceedings of the 2019 Linux Audio Conference (LAC-19)*, Stanford, USA, 2019.

[2] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. Nielsen, "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms," *Signal Processing: Image Communication*, vol. 68, pp. 101–119, 2018.

[3] M. Popoff, R. Michon, T. Risset, Y. Orlarey, and S. Letz, "Towards an FPGA-Based Compilation Flow for Ultra-Low Latency Audio Signal Processing," in *Proc. Int. Conf. in Sound and Music Computing, SMC-22*, Saint-Étienne, France, Jun. 2022.

[4] Y. Orlarey, S. Letz, and D. Fober, *New Computational Paradigms for Computer Music*. Paris, France: Delatour, 2009, ch. "Faust: an Efficient Functional Approach to DSP Programming".

[5] M. Popoff, R. Michon, T. Risset, P. Cochard, S. Letz, Y. Orlarey, and F. de Dinechin, "Audio DSP to FPGA Compilation: The Syfala Toolchain Approach," Univ Lyon, INSA Lyon, Inria, CITI, Grame, Emeraude, Tech. Rep. RR-9507, May 2023.

[6] C. Wegener, S. Stang, and M. Neupert, "Fpga-accelerated real-time audio in pure data," in *Proceedings of the International Conference in Sound and Music Computing, SMC-22*, 2022.

[7] T. C. Vannoy, "Enabling rapid prototyping of audio signal processing systems using system-on-chip field programmable gate arrays," Master PhD, 2020.

[8] C. Dragoi, C. Anghel, C. Stanciu, and C. Paleologu, "Efficient FPGA Implementation of Classic Audio Effects," in *Proceedings of the 2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. Pitesti, Romania: IEEE, Jul. 2021.

[9] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2018.

[10] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[11] P. P. de Garanderie, "Traitement du signal audionumérique à partir des modulations sigma-delta et pcm," Master's thesis, École Nationale Supérieure Louis-Lumière, 2015.

[12] P. Cochard, M. Popoff, A. Fraboulet, T. Risset, S. Letz, and R. Michon, "A programmable linux-based fpga platform for audio dsp," in *Proceedings of the 2023 Sound and Music Computing Conference (SMC-23)*, 2023, pp. 110–116.

[13] P. Cochard, J. Weber, R. Michon, T. Risset, and S. Letz, "Open Source Ethernet Real-time Audio Transmission to FPGA," INRIA, Tech. Rep. RR-9542, Feb. 2024. [Online]. Available: https://inria.hal.science/hal-04491503

[14] J. Ahrens, *Analytic methods of sound field synthesis*. Springer Science & Business Media, 2012.

[15] J. S. Abel, S. Coffin, and K. Spratt, "A modal architecture for artificial reverberation with application to room acoustics modeling," in *Audio Engineering Society Convention 137*. Audio Engineering Society, 2014.

[12] https://fast.grame.fr
[13] https://team.inria.fr/emeraude/plasma/