A Programmable Linux-Based FPGA Platform for Audio DSP

Pierre Cochard, Maxime Popoff, Antoine Fraboulet, Tanguy Risset, Stéphane Letz, Romain Michon

A Programmable Linux-Based FPGA Platform for Audio DSP

Pierre Cochard,[b] **Maxime Popoff,**[a] **Antoine Fraboulet,**[b] **Tanguy Risset,**[a] **Stéphane Letz,**[c] **Romain Michon**[b]
[a]Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
[b]Univ Lyon, Inria, INSA Lyon, CITI, EA3720, 69621 Villeurbanne, France
[c]Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
tanguy.risset@insa-lyon.fr

ABSTRACT

Recent projects have been proposing the use of FPGAs (Field Programmable Gate Array) as hardware accelerators for high computing power real-time audio Digital Signal Processing (DSP). Most of them imply specific developments which cannot be re-used between different applications. In this paper, we present an accessible FPGA-based platform optimized for audio applications programmable with the FAUST language and offering advanced control capabilities. Our system allows fast and simple deployment of DSP hardware accelerators for any Linux audio application on Xilinx FPGA platforms. It combines the Syfala compiler – which can be used to generate FPGA bitstreams directly from a FAUST program – with a ready-made embedded Linux distribution running on the Xilinx Zynq SoC. It enables the *compilation* of complete audio applications involving various control protocols and approaches such as OSC (Open Sound Control) through Ethernet or Wi-Fi, MIDI, web interfaces running on an HTTPD server, etc. This work opens the door to the integration of hardware accelerators in high-level computer music programming environments such as Pure Data, SuperCollider, etc.

1. INTRODUCTION

Whether we're talking about physical modeling, spatial audio, virtual analog, etc., real-time audio Digital Signal Processing (DSP) is always begging for more computational power. Different techniques have been used to increase the performances of real-time audio DSP: optimizing compilers, using dedicated DSP chips, and even hardware accelerators. However, that last approach requires highly specialized skills that most audio DSP engineers don't have. Moreover, audio processing languages such as CSound [1], FAUST [2], Pure Data [3], etc. have been mostly optimized for CPU-based targets and not for hardware accelerators.

FPGAs [1] have been used in the past couple of years as potential hardware accelerators for audio programs [4–6]. [4] introduced Syfala [2] [4, 7], the first compilation tool-chain from FAUST to FPGA which automatically maps the kernel DSP computation of a FAUST program on a hardware IP. [3] In Syfala, control rate computations (user input, wave table initializations, etc.) are executed on a regular CPU closely connected to the FPGA. The control portion of audio DSP algorithms was not integrated in an Operating System (OS) making the use of software components such as TCP/IP stack, HTTP protocol, OSC or MIDI interfaces, WAV of MP3 codec libraries, etc., inaccessible.

These elements are naturally present on all major operating systems. This is why audio applications are often executed within such an environment (as standalone or hosted processes). When hardware acceleration is needed, design is significantly more complex. First, the accelerated part must be handled separately using a specific technology (e.g., GPU, FPGA, dedicated accelerator, DSP chip, etc.). Then, it must be interfaced with the OS through a dedicated driver (i.e., a kernel driver on Linux). Providing this driver and handling concurrent access to a memory shared by the operating system and the accelerator is in general a complicated task as well.

In this paper we propose a platform which significantly simplifies the hardware/software integration task. It targets Xilinx FPGA-based boards and it provides a ready-to-use system that can automatically implement any hardware audio DSP accelerator controlled by an application running on a Linux operating system embedded on the Xilinx Zynq SoC. The Alpine Linux distribution that we provide can be easily tuned for different needs. Any audio DSP application can be compiled from a FAUST code down-to an FPGA IP and interfaced with the application.

Section 2 presents more precisely the goals of our work and also reviews existing comparable approaches. Section 3 presents the Syfala compiler that we rely on for FPGA IP generation and describes how Linux integration is performed in the Syfala toolchain. Section 4 provides some insights on the building of an embedded Linux distribution for Xilinx SoCs. Section 5 presents the specific software components that we added to our Linux distribution to handle the various needs of audio applications. Finally, Section 6 presents how we could, with very few additional efforts, implement a Minimoog emulation (al-

[1] *Field Programmable Gate Array*

[2] https://github.com/inria-emeraude/syfala – All URLs presented in this paper were verified on Feb. 6, 2023.

[3] Throughout this paper, *IP* stands for *Intellectual Property*, i.e., a circuit component.

ready implemented in the FAUST libraries [8]) on the Xilinx Zybo board.

2. CONTEXT AND STATE OF THE ART

Since the end of Moore's law, hardware acceleration is seen as a major mean for increasing the performances of digital systems. The vast ecosystem of dedicated accelerators is difficult to handle by regular software programmers though. This is very pregnant in computer vision and image processing [9] and it is also a new tendency in audio signal processing.

FPGA-based platforms have been used a lot for dedicated audio processing during the past 20 years [10, 11]. More recently, some projects have proposed the use of embedded Linux in this context [5, 6, 12, 13]. However, none of them focus on real *compilation* of audio programs: they all propose a dedicated hardware IP design for each implemented audio application.

Recent technical developments of FPGA-based platforms make them more suitable for the acceleration of audio applications. Modern FPGA chips include a powerful CPU integrated in a SoC tightly connected to the FPGA fabric. The integrated SoCs (Zynq on Xilinx/AMD platforms, Intel SoC for Altera/Intel platforms) are powerful enough to run an operating system. OS integration is generally facilitated by FPGA vendor tools that can be used to generate drivers easing the interfacing of the IP synthesized on the FPGA with the OS present on integrated SoCs, etc. This whole process remains complex to settle and is very dependent on the quality of the tools provided by the vendors.

For instance, Xilinx/AMD proposes Petalinux, which is a Linux distribution dedicated to the Zynq SoC family. Petalinux is built using the Yocto open-source toolchain which is widely used but quite complex. We could not effectively configure and build Xilinx's Petalinux on desktop computers using recent OS, hence we decided to build our distribution from scratch. Using a custom Linux distribution has many advantages: precise control of the content of the distribution, facilitated re-using of existing software components, etc.

In this paper, we describe a generic way of building a Linux system compatible with Xilinx Zynq platforms. We use the Linux kernel provided by Xilinx but we do not use Petalinux. This has the major advantage of being much simpler as this process doesn't rely on Yocto. This also gives us a more direct control on the deployed components.

As far as we know, Syfala is the only compilation flow for audio applications on FPGA-based boards. The work presented here enriches the Syfala compilation flow with the simple deployment of a lightweight embedded Linux distribution.

3. NEW DEVELOPMENTS IN THE SYFALA COMPILER

The Syfala compiler has been developed for Zynq-based Xilinx FPGA-based boards. [4] These include a CPU (dual core Cortex-A9 ARM) running the control part of the compiled audio application while core DSP computations are executed on the FPGA. This initial flow is presented in Fig. 1 where the whole compilation process starts from a single FAUST file named `audio.dsp`.

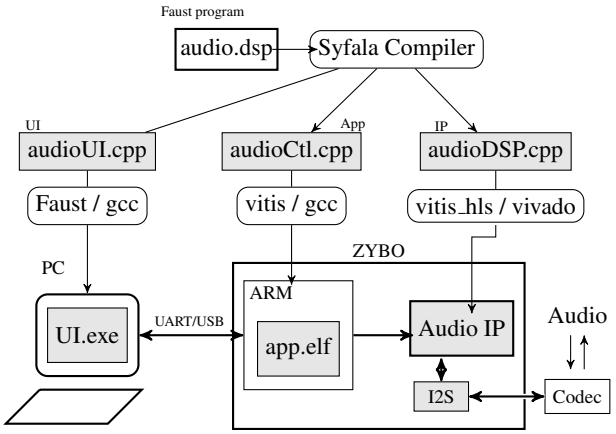

Figure 1. Initial FAUST to FPGA-based boards compilation flow as proposed in [4], gray boxes are generated during the compilation flow. The whole process is automatic, the user only has to write the `audio.dsp` file. The `audio IP` hardware component is referred as the *Syfala IP* in the rest of the paper.

The focus of the initial version of Syfala was on the compilation of FAUST programs down-to an ultra-low latency FPGA IP. Hence, less efforts went on the control/interface of the system. The control of the application (`audioCtl.cpp` on Fig. 1) is carried out through a single `main()` program running bare-metal (i.e., without any operating system) on the ARM processor. This `main()` program communicates with the host computer in a very primitive way by sending characters on a UART port which is embedded in the USB connection between the host and the FPGA board. The user interface of the audio application (`audioUI.cpp` on Fig. 1) re-uses a FAUST-generated GUI and sends in a infinite loop the controller values to the ARM via the USB/UART connection.

The improvements presented in this paper concern the way the user interface is handled on the ARM processor. As mentioned in the previous section, using an operating system on the ARM allows the user to re-use many software components that are already compiled for Linux.

The new compilation flow is presented in Fig. 2. An embedded Linux distribution is built once (see Section 4). It includes libraries for managing control protocols such as OSC and MIDI as well as any other standards supported by the FAUST ecosystem. Once the FPGA platform boots on the SD card containing this Linux distribution, the user has access to the Linux shell via the dedicated USB/UART port. It is also possible to connect to the board via Ethernet (DHCP protocol used by default) and possibly add new packages to the Linux distribution on the SD card. After this, the Linux system doesn't have to be rebuilt: any other Syfala audio application will run with the same Linux SD card. Compiling a new applica-

[4] `https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`

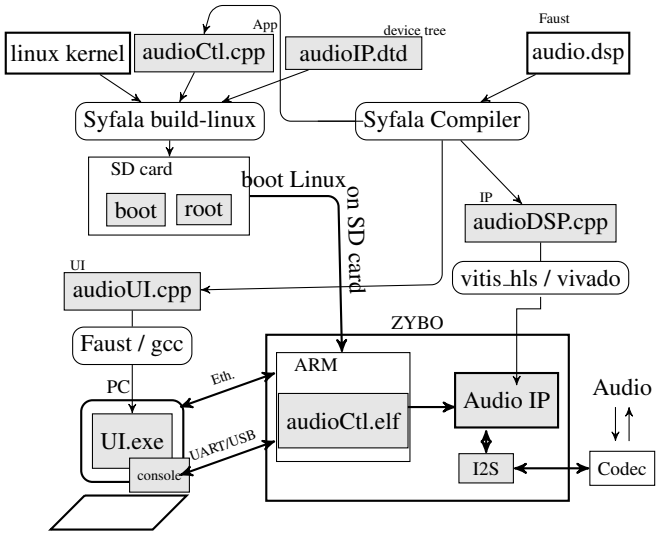

Figure 2. The extension to the original Syfala tool-chain (shown in Fig. 1) proposed in this paper with an embedded Linux OS booting from an SD-card. The user interface can now use many common communication protocols (e.g., OSC, MIDI, etc.) to control the audio DSP.

tion "`audio2.dsp`" on the FPGA board only requires to copy the new "`audio2Ctl.elf`" on the SD card using an `ssh` connection.

4. AD-HOC EMBEDDED LINUX BUILDING

This section describes the building process of the Linux distribution. It insists on the specificity related to our platform, in particular the fact that the FPGA IP driver must be handled properly. The complete procedure is described in more details on the Syfala GitHub [5] It is a standard Linux building procedure with a focus on driver construction for the Syfala IP.

The procedure is split in two stages which build the `boot` and `root` partitions present on the SD card. The `boot` partition contains the first and second stage bootloader (`U-Boot`), the Linux kernel image and modules as well as the device-tree describing the peripherals used on the platform. The `root` partition contains the complete Linux filesystem that will be mounted after booting. It includes in particular the `audioCtl.elf` application as well as the FPGA bitstream, both built by the Syfala compiler.

Boot Partition: The `U-Boot` binary and the Kernel sources are provided by Xilinx [6] (we used the version 2022.2 of Xilinx tools). We added a custom configuration file activating the userspace I/O system (for the Syfala IP), as well as various drivers (i.e., ALSA, Wi-Fi, etc.).

The device-tree is a set of tools and a data structure for describing the hardware. It is read by the operating system at boot time to avoid hard coding information in the system.

[5] `https://github.com/inria-emeraude/syfala/tree/dev/linux`

[6] `https://github.com/Xilinx/u-boot-xlnx` and `https://github.com/Xilinx/linux-xlnx`

In general, the complete device-tree is described in the text file: `system.dts`. Due to the difficulties encountered when generating a device-tree suiting our very specific needs with the Xilinx tools, we added a static device-tree source file that has been modified by hand in order to facilitate the procedure. The following presents some of the changes and additions that we made to the standard Zybo Z7 device-tree:

- The boot arguments were modified in the following way:

```
bootargs = "earlycon uio_pdrv_genirq.of_id=generic-uio
     root=/dev/mmcblk0p2 rootfstype=ext4 rw rootwait";
```

- A section of 128 MB of reserved memory used by both the Syfala IP and the Linux applications was reserved. This section is not managed by the memory management unit (MMU):

```
reserved-memory {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    reserved: buffer@35000000 {
        no-map;
        reg = <0x35000000 0x08000000>;
    };
};
reserved-driver@0 {
    compatible = "xlnx,reserved-memory";
    memory-region = <&reserved>;
};
```

- Our Syfala IP was added as a `generic-uio` device. It is accessible through an `axilite` bus as `/dev/uio0`:

```
 syfala: syfala@40010000 {
          clock-names = "ap_clk";
          clocks = <&misc_clk_0>;
          compatible = "generic-uio";
          status = "okay";
          reg = <0x40010000 0x10000>;
          xlnx,s-axi-control-addr-width = <0x7>;
          xlnx,s-axi-control-data-width = <0x20>;
};
```

Root Partition: We chose an Alpine Linux distribution in order to build the root filesystem. It is known to be lightweight, not relying on `glibc` and `systemd`, but instead on `musl`, `busybox`, and `OpenRC`. The required files, such as the u-boot, apk-tools or linux-firmware archives, can be directly downloaded from the official Alpine Linux repositories. We then used the QEMU ARM CPU emulator and a `chroot` operation to install and configure our file system (Zynq SoCs include an ARM CPU).

The procedure for the creation of the `rootfs` structure and contents is standard (see on GitHub for details). We selected a certain number of packages to be pre-installed, activated the required drivers to configure the board's peripherals, and installed the additional files needed for the control of the Syfala IP (i.e., the FPGA bitstream and its associated Linux control application).

For the compilation of the control application `audioCtl.cpp` in Fig. 2, the Xilinx driver library generated by `vitis_hls` needs to be included. Its API contains for instance all the `UIO` system calls required for the communication between the application and the Syfala

IP. For instance, below is the `initialize()` function that initializes the IP after booting:

```
void initialize(XSyfala& x) {
      [...]
      XSyfala_Initialize(&x, "syfala");
      [...]
```

The shared-memory that we explicitly reserved in our device-tree also needs to be mapped in a certain way in order to be properly accessed from our Linux control application (it has to be coherent with the kernel):

```
#define MEM_ADDR 0x35000000
#define MEM_LEN  0x08000000
int fd = open("/dev/mem", O_RDRW | O_SYNC);
void* mem = mmap(NULL, MEM_LEN, PROT_READ |
     PROT_WRITE, MAP_SHARED | MAP_FILE, fd, off);
if (mem == MAP_FAILED) {
    perror("Can't map reserved memory space");
    exit(1);
}
```

Once the `boot` and `root` partition are flashed on the SD card, and the FPGA board is booted, the Linux shell can be accessed through the USB serial port connection as shown in Fig. 3.DHCP is used for IP address attribution.

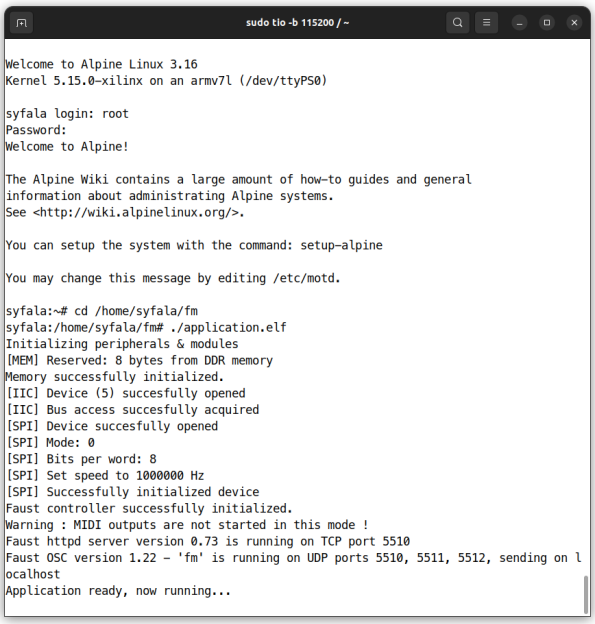

Figure 3. Linux boot console received by the host computer on serial port `/dev/ttyUSB1` from the Zybo board booting on our Linux distribution (simple sinewave audio DSP program).

5. SOFTWARE CONTROL

While MIDI is still widely-used as a protocol for controlling generic audio interfaces, other standards have also made their way into the audio software ecosystem. Examples are Open Sound Control (mainly over UDP), OSCQuery (using a HTTP/WebSocket layer on top of the traditional UDP OSC [7]), or even just plain HTTP. The main advantage of these protocols reside in their flexibility over the type of value transmitted. [8] These protocols are also supported by the large diversity of devices that can be used to control an audio process remotely: computers, smartphones or tablets, with plethora of dedicated third-party applications and interfaces.

[7] `https://github.com/Vidvox/OSCQueryProposal`

[8] MIDI, with the exception of the 14-bits pitchbend control values, only allows for 7-bits-based integer values.

```
import("stdfaust.lib");
declare options "[midi:on]";
declare options "[osc:on]";
// sliders
oscFreq = hslider("oscFreq [midi:ctrl 13]",
                  80,50,500,0.01);
lfoFreq = hslider("lfoFreq [midi:ctrl 14]",
                  1,0.01,8,0.01);
lfoRange = hslider("lfoRange [midi:ctrl 15]",
                   1000,10,5000,0.01) : si.smoo;
noiseGain = hslider("noiseGain [midi:ctrl 16]",
                   0,0,1,0.01) <: _*_;
masterVol = hslider("masterVol [midi:ctrl 17]",
[...]
```

Figure 4. Example of a FAUST program with metadata indicating which FAUST slider will be associated with which MIDI controller. OSC control can be used too.

The FAUST ecosystem has been supporting these protocols for many years, facilitating their use on a wide range of devices. The link between the FAUST program and the controllers is established by writing metadata in the FAUST DSP code itself. An example of such metadata is shown in the program depicted in Fig. 4. The FAUST compiler and the various tools associated with the FAUST ecosystem generate the code needed to communicate with the chosen protocol on the targeted architecture.

The Linux distribution that we propose natively supports Xilinx Zybo and Genesys FPGA boards. It also contains the FAUST architecture files, meaning that its MIDI, OSC, and HTTP features and interfaces are already enabled and ready-to-use on the board. This is of course much harder to implement in a bare-metal configuration, due to the multitude of drivers and additional libraries that would have been required.

After Linux booting, MIDI control can be accessed through its USB "On The Go" port (configured as "host" in our device-tree), by plugging-in a compliant controller. This requires the board to be powered-up by an external source (unless the controller itself is already powered up), and its jumper configuration to be arranged accordingly. The `ALSA` kernel modules, with the addition of a few `RtMidi` and FAUST utilities, then take care of properly mapping the received control values to the DSP code.

The implementation of the OSC and HTTP-based interfaces consists of a set of FAUST wrappers around the `liblo` and `libmicrohttpd` libraries, which are both available as `dev` packages in the Alpine Linux repositories. These interfaces are launched at the initialization of the control application, and have the advantage of being synchronized with one another, meaning that playing with a slider on a MIDI hardware controller effectively updates the same (virtual) slider on both the HTTP and OSC interfaces. An excerpt from the control application code, showing the initialization of the MIDI, OSC and HTTP interfaces in the `audioCtl.cpp` on the ARM, can be seen in Fig. 5.

By default, the OSC Uniform Resources Identifiers (URI) of these control parameters are automatically attributed by

```
int main() {
    [...]
    /* Initialize the Faust control handler
     * and the different control interfaces */
    Faust::Control::data ctrl;
    Faust::Control::initialize(
        ctrl, mem.i_zone, mem.f_zone);
    /* Initialize MIDI, HTTP, OSC and
     * Avahi interfaces */
    rt_midi rt("MIDI");
    MidiUI midi_ui(&rt);
    httpdUI http("http",
        ctrl.dsp.getNumInputs(),
        ctrl.dsp.getNumOutputs(), 0, 0);
    OSCUI osc("osc", 0, 0);
    avahi::service avahi_svc;
    /* Link the control interfaces: */
    ctrl.dsp.buildUserInterface(&osc);
    ctrl.dsp.buildUserInterface(&http);
    ctrl.dsp.buildUserInterface(&midi_ui);
    /* Start the FPGA-ARM communication and
    * the control interfaces: */
    control(ctrl, x, spi, mem);
    midi_ui.run();
    http.run();
    osc.run();
    avahi::initialize_run(avahi_svc);
    [...]
}
```

Figure 5. Excerpt from the `audioCtl.cpp` main initialization function, showing how the control interfaces are instantiated, linked with one another, and started.

FAUST in a very specific way, following its own GUI hierarchical tree and format (`DSP/group/slider`). For more flexibility, it is also possible to add aliases directly in the FAUST code metadata, in the case where the client device has a fixed OSC URI format. [9] On the other hand, the HTTP interface does not require any configuration from the user: it uses a dedicated JSON-based protocol, which describes the application's control interface set in the DSP code, and is used internally by the underlying JavaScript engine to build the graphical user interface. [10]

Finally, in order to make the connection process from the client devices a bit less cumbersome on DHCP-configured networks, an `Avahi` service is also started whenever the control application is initialized. Given that the client device is on the same network as the FPGA board, one can conveniently discover all the registered Syfala devices (with the help of the `avahi-browse` command, for instance), pick up their IP address and UDP/TCP ports, etc. Once this information has been retrieved, the user can finally connect to the OSC/HTTP servers from various applications, such as a browser or an OSC-compliant interface.

6. APPLICATION: MINIMOOG ON FPGA

As part of the developments made on our new Linux-based toolchain, we set up a standalone synthesizer, which is largely based on the emblematic virtual-analog "Minimoog" (Model D) emulation written in the FAUST language by Julius O. Smith. [11]

[9] See `https://faustdoc.grame.fr/manual/osc/` for more information.

[10] `https://faustdoc.grame.fr/manual/http/`

[11] `https://github.com/grame-cncm/faust/tree/master-dev/examples/SAM/virtualAnalog`

This monophonic synthesizer has 1 external audio input, 2 audio outputs, and is composed of 3 oscillators (with, for each one of them, a set of 6 different waveforms to choose from, in real time), as well as a pink/white noise-generator, an amplitude envelope, and, of course, a Moog Ladder 4-pole virtual-analog filter.

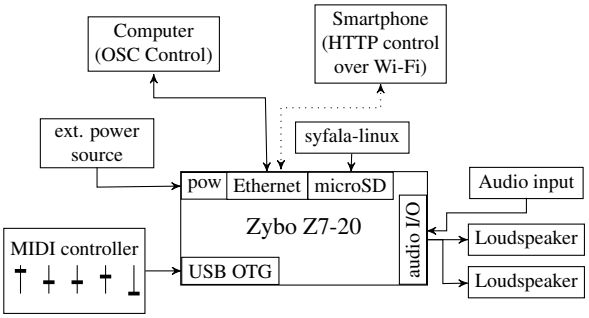

Figure 6. Zybo-based Minimoog (Model D) standalone synthesizer emulation (hardware view of the setup).

Figure 7. The Zybo Minimoog synthesizer set-up.

In this specific configuration, the Zybo board has its own external power source and is (i) configured to boot automatically on a microSD card containing our syfala-linux build and (ii) connected to a MIDI-USB controller, a network router, and a pair of speakers with 3.5mm jack inputs (see Fig. 6 and 7).

When powered up, the Linux Kernel image is booted up, and its drivers and peripherals are initialized. Once the process is complete, the root filesystem is then mounted, and, with the help of an autologin and initialization script, the Minimoog FPGA bitstream and its associated Linux control application are immediately loaded. The system can still be accessed through SSH, allowing the user to load other bitstreams and control-applications remotely.

It takes between 25 and 30 seconds for the whole boot process to complete on average, from the moment the board is powered-up to the moment the user is able to play a note on the MIDI controller and hear the resulting sound. Given the time needed to load the FPGA bitstream (between 50 and 100 milliseconds) and to initialize the control application (between 2 to 5 seconds), the boot time could be improved a lot.

For MIDI control, we managed to map a widely-used controller, the Novation LaunchControl XL, to some of the FAUST DSP control parameters, using the metadata system. These parameters include: the oscillators' tuning,

amplitude and waveform selection, the Moog Ladder corner frequency and "Q" factor, the pink/white noise levels, etc. We have also set up the controller's pads to be used as "keys" in order to control the main pitch frequency and the triggering of the main amplitude envelope, which also forced us to introduce some minor changes to the original Minimoog FAUST DSP code.

The hardware/software split is handled as usual by Syfala: all audio-rate computations are directly handled on the FPGA while all control-rate and initialization computations are carried out in the ARM.

For anyone knowing FAUST Minimoog implementation, this experiment is not *really* unique as the Minimoog FAUST program can actually run on a regular PC. Indeed, the Syfala tool-chain has been optimized for ultra-low latency. The resource used by this design, detailed hereafter, could be improved a lot in many ways: allow for longer latency, use fixed point computations, etc. We are actively working in this direction. However, what is unique is the speed with which the whole hardware/software design was realized (a few hours), compared to a traditional VHDL/Verilog IP manual implementation.

The resources used by the Minimoog IP are shown in Figure 8. The Zybo Z7-10 board model was insufficient to implement the Minimoog (164% of the number of available lookup tables were needed), but the Zybo Z7-20 model was large enough: bringing down LUT resources usage to only 46%. Concerning the computation time for each sample, it takes approximately 1899 cycles for the FPGA to perform the computation of a single sample (15.601 µs). If the sampling frequency is 48KHz, the time between two sample is approximately 22.385 µs. Therefore, while resource usage is far below its maximum in our case (46% on a Zybo-Z7-20), we are approaching more rapidly (at 69%) the maximum allowed latency for the computation of the Syfala IP. The total allocated DDR memory (mainly used for the oscillator wavetables) for this project is 50 kBytes. DDR is accessed 18 times per sample and that is where the latency comes from.

	BRAM	DSP	FF	LUT	cycles
Used	5	74	25917	24767	1899
Avail.	280	220	106400	53200	2736
%	3%	33%	24%	46%	69%

Figure 8. Post-synthesis resource/latency usage report for our Minimoog implementation on the Zybo Z7-20.

Concerning control signals which are flowing between the ARM and the FPGA, the FPGA code accesses and copies 80 integer/floating point values at every sample. These values are the result of the control-rate expression computations made by the ARM and shared with the Syfala IP through a memory-mapped AXILITE bus (while DDR accesses use the M_AXI bus). The GUI interface is able to tune 60 input parameters which are control values coming from the FAUST virtual controllers (e.g., sliders, knobs, buttons, etc.). These values are continuously retrieved and updated by the OSC, HTTP, and MIDI interfaces, the HTTP interface generated for the Minimoog is shown in Fig. 9. It takes on average 110 to 115 µs for the ARM to compute all the control-rate expressions and write the result in memory. As one can see, this does not fit in a one sample time interval but it does not need to as we are in the control rate space.

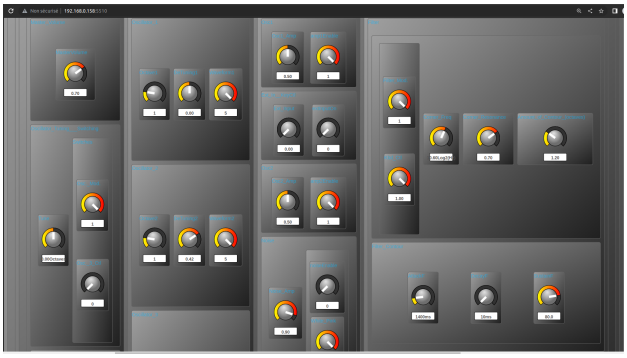

Figure 9. HTTP interface example, generated for the Minimoog application.

Regarding build and compilation times: it takes for this project 15 seconds, for the FAUST compiler to generate both FPGA and ARM C++ code. To give an order of magnitude, the C++ code generated for the audio-rate computation function is 180 lines long, 80 lines for the control-rate function. The High Level Synthesis (HLS) process takes approximately 2 minutes to compile the FAUST-generated code down-to FPGA Hardware Description Language (HDL). The full project synthesis, made with Vivado, remains the toolchain's longest process, taking about 30 to 40 minutes to complete for this specific DSP target. Finally, given that the Linux root partition has already been previously built, only 1 to 2 minutes are usually required in order to build the control application, either directly on the board, or within a `chroot` environment.

7. CONCLUSION

In this paper, an extension of the Syfala tool-chain providing a Linux-based hardware accelerator for audio applications has been presented. This work is freely accessible (open source) on the Syfala Github[5] (except for Xilinx synthesis tools of course). While it is currently only targeting Xilinx Zynq-based platforms, it could easily be modified to support other SoC-based FPGAs.

This work should open the door to other kinds of applications relying on Linux. For example, embedding FAUST generated IP as externals in Pure Data, CSound or SuperCollider which would run the ARM processor. This would be an easy way to extend these tools with hardware acceleration. This would however require to work on the interface between these tools with the Syfala IP, this is something that we would like to explore in the future. As mentioned before, the current performances of the Syfala compiler have to be improved as it has been designed for ultra-low latency. In a future version of Syfala, the balance between performance and latency will be a parameter of the compilation flow.

More generally, we believe that FPGAs have an important role to play in the field of real-time audio DSP by providing more computational power, but also unparalleled performances in terms of latency, number of channels that can be processed in parallel, ultra-high audio sampling rate, etc. We're currently exploring all these new domains of applications.

Acknowledgments

This work has been carried out in the context of the FAST ANR project [12] (ANR-20-CE38-0001) funded by the French ANR (Agence National de la Recherche).

8. REFERENCES

[1] V. Lazzarini, S. Yi, J. Heintz, Ø. Brandtsegg, I. McCurdy *et al.*, *Csound: a sound and music computing system*. Springer, 2016.

[2] 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".

[3] M. Puckette, "Pure data: Another integrated computer music environment," *Proceedings of the Second Intercollege Computer Music Concerts*, 1996.

[4] 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.

[5] C. Wegener, S. Stang, and M. Neupert, "Fpga-accelerated real-time audio in pure data," in *Proc. Int. Conf. in Sound and Music Computing, SMC-22*, 2022.

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

[7] T. Risset, R. Michon, Y. Orlarey, S. Letz, G. Müller, and A. Gbadamosi, "Faust2FPGA for Ultra-Low Audio Latency: Preliminary work in the Syfala project," in *IFC 2020 - Second International Faust Conference*, Paris, France, Dec. 2020.

[8] J. O. Smith, "Signal processing libraries for faust," in *Linux Audio Conference - LAC-12*, Stanford, USA, 2012.

[9] 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.

[10] M. Pfaff, D. Malzner, J. Seifert, J. Traxler, H. Weber, and G. Wiendl, "Implementing digital audio effects using a hardware/software co-design approach," in *10th International Conference on Digital Audio Effects*, 2007.

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

[12] T. Vannoy, T. Davis, C. Dack, D. Sobrero, and R. Snider, "An open audio processing platform using soc FPGAs and model-based development," in *Audio Engineering Society Convention 147*. Audio Engineering Society, 2019.

[13] R. Scheibler, J. Azcarreta, R. Beuchat, and C. Ferry, "Pyramic: Full stack open microphone array architecture and dataset," in *16th International Workshop on Acoustic Signal Enhancement, IWAENC 2018, Tokyo, Japan, September 17-20, 2018*. IEEE, 2018, pp. 226–230.

[12] `https://fast.grame.fr`