HAL open science

Audio DSP to FPGA Compilation: The Syfala Toolchain Approach

Maxime Popoff, Romain Michon, Tanguy Risset, Pierre Cochard, Stephane Letz, Yann Orlarey, Florent de Dinechin

To cite this version:

Maxime Popoff, Romain Michon, Tanguy Risset, Pierre Cochard, Stephane Letz, et al.. Audio DSP to FPGA Compilation: The Syfala Toolchain Approach. RR-9507, Univ Lyon, INSA Lyon, Inria, CITI, Grame, Emeraude. 2023. hal-04099135

HAL Id: hal-04099135

https://inria.hal.science/hal-04099135

Submitted on 23 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Audio DSP to FPGA Compilation: The Syfala Toolchain Approach

Maxime Popoff , Romain Michon, Tanguy Risset, Pierre Cochard, Stephane Letz, Yann Orlarey, Florent de Dinechin

RESEARCH REPORT

N° 9507

may 2023

Project-Teams Emeraude

ISSN 0249-6399 ISRN INRIA/RR--9507--FR+ENG

Audio DSP to FPGA Compilation: The Syfala Toolchain Approach

Maxime Popoff *, Romain Michon†, Tanguy Risset*, Pierre Cochard†, Stephane Letz‡, Yann Orlarey‡, Florent de Dinechin*

Project-Teams Emeraude

Research Report n° 9507 — may 2023 — 18 pages

Abstract: The implementation of real-time audio Digital Signal Processing (DSP) on FPGA has been extensively studied in the past. Up to now, Audio IPs were designed either "by hand" in VHDL or using predefined IPs in block synthesis environments. The advent of High Level Synthesis (HLS) allows for a real compilation flow from high-level audio DSP specifications down to FPGA bit-streams. This paper presents the principles and the implementation of the first "audio DSP compiler" targeting FPGAs. Our fully open-source system compiles audio DSP programs down to FPGA hardware and up to actual sound production. Many parameters such as the number of output channels, sampling rate, etc. are adjusted automatically by the compiler. Software interfaces can be generated to control the system in real-time. This compilation flow presents two important technological breakthroughs for audio programmers: achieving ultra-low latency real-time audio DSP (few micro-seconds) and the possibility of easily deploying systems with a large number of audio channels.

Key-words: HLS, Compilation on FPGA, Audio DSP, Faust

* *Univ Lyon, INSA Lyon, Inria, CITI, EA3720* 69621 Villeurbanne, France *firstname.lastname@insa-lyon.fr*
† *Univ Lyon, Inria, INSA Lyon, CITI, EA3720 firstname.lastname@inria.fr*
‡ *Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720* 69621 Villeurbanne, France *{sletz, orlarey}@grame.fr*

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

La Chaîne de Compilation Syfala pour le Traitement du Signal Audio sur FPGA

Résumé : La mise en œuvre du traitement du signal numérique (DSP) audio en temps réel sur FPGA a été largement étudiée dans le passé. Jusqu'à présent, les IPs audio étaient conçues soit "à la main" en VHDL, soit en assemblant des IPs prédéfinies dans des environnements dédiés. L'avènement de la synthèse de haut niveau (HLS) permet un véritable flux de compilation depuis les spécifications DSP audio de haut niveau jusqu'aux flux binaires FPGA. Cet article présente les principes et l'implémentation du premier "compilateur DSP audio" ciblant les FPGA. Notre système compile les programmes DSP audio exprimés dans le langage Faust jusqu'au matériel FPGA et jusqu'à la production sonore réelle. De nombreux paramètres tels que le nombre de canaux de sortie, le taux d'échantillonnage, etc. sont ajustés automatiquement par le compilateur. Des interfaces logicielles peuvent être générées pour contrôler le système en temps réel. Ce flot de compilation présente deux avancées technologiques importantes pour les programmeurs audio : l'implémentation de programmes DSP audio en temps réel à très faible latence (quelques microsecondes) et la possibilité de déployer facilement des systèmes avec un grand nombre de canaux audio.

Mots-clés : HLS, Compilation on FPGA, Audio DSP, Faust

Contents

1 Introduction **4**

2 State of the Art **5**

3 FPGA Audio Compilation Scheme **6**

4 Audio-Specific Compilation Optimizations **8**
4.1 Hardware/Software Split: Sample Rate vs. Control Rate 8
4.2 Memory Access Optimization 10
4.3 Control and Human-Machine Interface 10

5 Performance Results **11**
5.1 Audio IP Complexity 12
5.2 Latency Performance 13
5.3 Memory Performance 14
5.4 Multichannel Performance 15

6 Conclusion **15**

1 Introduction

Audio Digital Signal Processing (DSP) is used on a wide range of devices. Some audio or vibratory applications require high throughput, which can be obtained using hardware accelerators such as GPUs, ASICS, or FPGAs. In some specific cases (i.e., dynamic acoustic control used in artificial reverberation, noise cancellation , etc.), "ultra low latency" (in the order of $10\mu s$) is required, implying the use of dedicated architectures such as FPGAs or ASICS. Hence, FPGAs are a particularly good fit for audio programmers developping DSP applications with low latency and/or a large number of audio channels (32 or more).

As ultra low latency can only be achieved with FPGAs or ASICS, some industrial audio devices have been relying on them for a couple of years now. Until recently, the development of such systems required very advanced skills in micro-electronics and was not accessible to audio DSP engineers/programmers. The compilation flow presented in this paper should ease the prototyping of systems (*i*) for active acoustic control used in artificial reverberation, noise cancelation, etc., and (*ii*) involving a large number of audio channels (e.g., spatial audio, ambisonics microphones, etc.).

The use of FPGAs for audio DSP has been studied for a long time (see § 2). However, DSP algorithms are usually expressed at a high-level (i.e., Matlab, Python, C/C++, etc.). Implementing them on an FPGA implies a "translation" step into a hardware description language (i.e., VHDL or Verilog), which can be long and tedious. New hardware[1] design methodologies progressively allowed designers to reduce hardware design time. These new methodologies use either IP-based[2] design or High Level Synthesis (HLS).

IP-based design assembles hardware versions of high-level IP blocks. For instance, *System Generator* uses *MATLAB* libraries of AMD/Xilinx blocks, *HDL Coder* uses *Simulink* or *Stateflow* diagrams to target Intel/Altera FPGAs, etc. Compared to this, High Level Synthesis offers a more direct control on IP design. Algorithms are written in a sequential programming language – C/C++ usually – and mapped directly to a hardware IP. Pragmas or directives can be used by the designer to guide the HLS tool during the parallelization process.

This HLS approach is much more flexible and powerful than IP-based design. Moreover, it offers many optimization opportunities because (*i*) tools (i.e., audio compiler and HLS compiler) have been optimized for years and (*ii*) audio programs are quite specific and many optimization choices can be guided by the fact that the compiled application is an audio program.

In this paper, we show that using a Domain Specific Language (DSL) targeting audio applications can contribute to reducing the exploration space and help with the automation of hardware compilation. Hard real-time constraints occur in audio programs when computing each sample. The human ear is very sensitive and can notice any error even when happening on only one sample. These constraints do not occur when carrying out control operations (e.g., volume change, etc.). Another specificity of audio applications is that audio input and output channels use a standardized interface for audio codecs,[3] i.e., the I2S[4] protocol. Hence, the interface of our compiled IP can be easily specified for any audio applications.

Developers in the computer music community have been using domain specific languages such as CSound [9], Faust [12], PureData [17], etc., for a long time. Indeed, in most cases only a small portion of the extent of C++ is used when expressing audio DSP algorithms. DSLs have

[1] Here, *hardware* refers to elements implemented on the FPGA (as opposed to elements that could be implemented on a CPU). This terminology is used throughout this paper.

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

[3] In this paper, "audio codec" always refers to a hardware component providing analog outputs and inputs (audio ADC/DAC): not an audio compression algorithm.

[4] *I2S* (Inter-IC Sound) is a serial communication protocol used to transmit digital audio signals between IC components.

been developed with a much more intuitive syntax for audio programming, using a data-flow programming models in many cases.

In this paper, we demonstrate that combining HLS with an audio DSL offers many advantages such as: (i) versatility of the DSP algorithm (any audio DSP program allowed by the DSL), (ii) flexibility on the number of input and output audio channels, (iii) reduced design space exploration time and good resulting IP performance (throughput and latency mostly).

The contributions of this paper are the following:

- We introduce the first open-source automatic compilation flow of audio programs on FPGA. This compiler provides many tunable parameters (i.e., number of output channels, hardware or software control interface, audio codec used, etc.) and predictive performance: thanks to a precise performance analysis on Xilinx platforms, performance can be predicted without running complete synthesis.

- Performance of the resulting designs in terms of throughput and latency are evaluated for various audio applications compiled on various Xilinx-based FPGA evaluation boards. Ultra low audio Latency is achieved (i.e., analog to analog) and it is one order of magnitude better that previous works.

Next Section presents the state of the art, Section 3 presents our audio to FPGA compilation flow in more details. Section 4 focuses on audio-specific optimizations that we have implemented (and should be useful to anyone trying to build another audio to FPGA design flow). Sections 5 presents various performance results of the compiler.

2 State of the Art

Audio on FPGA has been studied since the beginning of the 2000s [14, 7, 2]. The general methodology for these works is based on "manual design" (in VHDL or using IP-based design). Recent works usually target SoC FPGAs present in most recent FPGA evaluation boards [21, 6, 27, 20, 10, 8, 19].

Real FPGA *compilation* tool-chains for audio DSP are rare. In 2014, Verstraelen proposed a programmable parallel machine implemented on FPGA [25] which did not lead to further implementation. Vannoy et al. ([22, 23]) proposed an open-source IP-based system (using MathWork *HDL coder*). The GAUT HLS tool [5] was dedicated to signal processing applications but did not use a dedicated audio DSP DSL as input and is not available anymore.

The idea of coupling an FPGA fabric with a CPU has been widely studied. With the advent of SoC integrated in FPGAs and HLS tools, some works proposed to compile high-level languages to FPGA + CPU platforms [11, 13, 4]. Attempts are made to propose tools that perform "real" high-level compilation of complete application for SoC FPGA, such as late *SDSoC* from Xilinx or Intel *SoC FPGA*, but these tools have not been adopted yet by the FPGA designer community.

However, HLS has been recently used to target domain-specific applications such as convolution neural networks [24, 3, 4] or IoT [29, 28]. Taking a similar approach, we are proposing in this paper to specialize/adapt HLS tools to the field of real-time audio DSP. To our knowledge, this work is the first one to propose the compilation of audio DSP applications to FPGA.

Concerning audio latency, the computer music community has been studying that topic for a long time (see Wang's PhD thesis [26] for a complete review). Among the aforementioned works, the shortest announced latency on FPGA processing was 180 µs [22]. Our compiler is able to reduce this latency by a factor of 10.

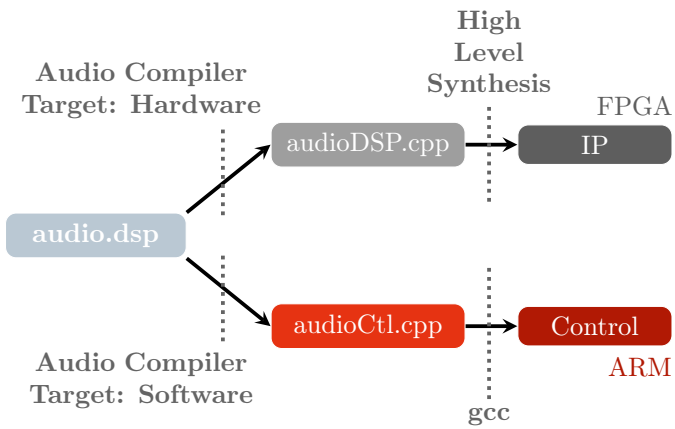

Figure 1: Audio to FPGA compiler global view. We call *hardware part* the section of the program that is mapped on the FPGA and *software part* the section of the program that is compiled to the associated CPU.

Finally, FPGAs are increasingly used in the audio industry. Novation,[5] Antelope Audio,[6] UDO audio,[7] and futur3soundz[8] all offer products integrating FPGAs. Although it is difficult to reverse engineer these products, it seems that FPGAs are mostly used in this context for approximating/emulating analog audio synthesis by running audio algorithms at a high sampling rate. Hardware implementing the Dante protocol (which is widely used in concert halls, stages, etc., for transmitting audio over Ethernet on local networks) is typically based on FPGAs for handling a high number of channels in parallel.[9]

3 FPGA Audio Compilation Scheme

An *audio DSP to FPGA compilation flow* is not limited to the single compilation of signal processing algorithm to FPGA. Fig. 1 presents the global view of the building blocks of an "audio DSP to FPGA compiler." A single source (named `audio.dsp` in Fig. 1) generates both hardware (i.e., audio IP) and software (i.e., control program). This is an important point because most existing audio compilation flow have *already* implemented this hardware/software split when compiling for CPUs.

Most modern FPGA architectures include a powerful general-purpose CPU on the FPGA chip. For instance, a dual-core ARM Cortex-A9 is present on the Zynq 7000 (which is used in many Xilinx FPGA chips). This processor can manage many non-critical operations in software while critical elements are mapped on the programmable logic of the FPGA.

For expressing our audio programs, we chose the Faust programming language [12]. Faust is a functional programming language for real-time audio processing which is widely used in the field of music technology. The main strength of the Faust environment is its optimizing compiler targeting many languages such as C++, C, LLVM bitcode, WebAssembly, Rust, etc. We have extended the targets by including Xilinx hardware boards, using the Xilinx HLS tool *Vitis HLS*.

[5] `https://novationmusic.com/en/synths/summit` – All URLs in this paper were verified on Feb. 15, 2023.
[6] `https://en.antelopeaudio.com/`
[7] `https://www.udo-audio.com/#introduction`
[8] `https://www.futur3soundz.com/`
[9] `https://www.audinate.com/products/manufacturer-products/dante-ip-core`

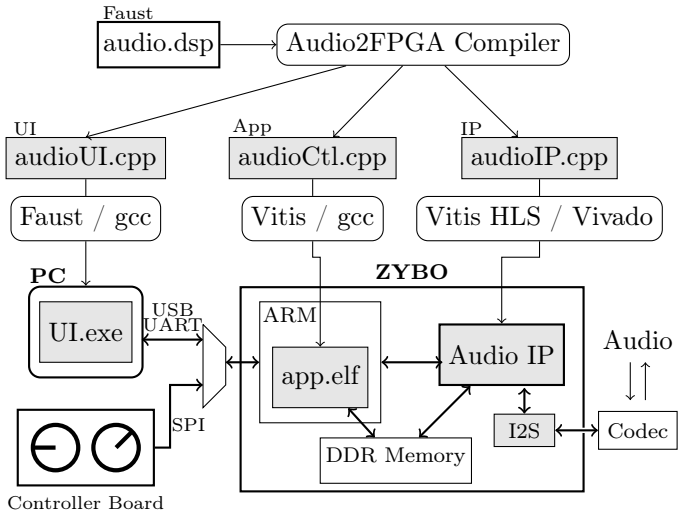

Figure 2: Audio to FPGA compiler complete architecture using Faust and Xilinx tools (grey boxes are generated during the compilation flow). C++ files are generated by the Faust compiler tuned with adequate parameters. The core DSP computation (`audioIP.cpp`) is mapped on the FPGA, the control part (`audioCtl.cpp`) is mapped on the computing system and the user interface (`audioUI.cpp`) is mapped on the host processor or carried out in hardware. Standard interfaces allow the user to use the same compilation commands for all audio programs.

The audio system resulting from the compilation of an audio program contains many interfaces that can act as bottlenecks if they are not optimized properly. The two most important interfaces are the hardware-software interface and the memory interface. Two other physical interfaces must be handled: the audio interface – usually a connection with a hardware audio codec – and the user control interface, which can take many forms. Finally, the interface between the FPGA IPs and the rest of the system must be handled with care, too.

Our implementation of the conceptual view of Fig. 1 is presented in Fig. 2. It uses a Faust program as an input. Our compilation flow to VHDL uses a front-end compiler (Faust) and a back-end compiler (*Vitis HLS*/*Vivado*). The hardware/software partitioning is performed automatically by Faust (see § 4.1) which is able to detect computations that should be executed at each sample (i.e., sample-rate computations) and computations that do not have such strong real-time constraints (i.e., control-rate computations).

Another important component represented in Fig. 2 is the external DDR memory. Indeed, some audio programs use delay lines (e.g., echoes) that might need large chunks of memory. As in the case of general purpose CPU targets, memory access is a potential bottleneck but on the FPGA, the available Block RAMs (BRAMs) allow for important optimizations in memory access performance. These optimizations are explained in § 4.2.

The interfacing of the external audio codec (on the right hand side of Fig. 2) is carried out through a dedicated I2S VHDL component. User interface can be either implemented in hardware using a dedicated board or in software by re-using built-in Faust tools (see Fig. 6).

As we aimed for ultra-low latency, the HLS process shown on Fig. 2 has been tuned to generate audio IPs with a latency of only 1 sample. However, the real latency for acoustic engineers is the analog to analog latency i.e., the latency at the I/O of the audio codec. This analog to analog latency is analyzed in § 5.2.

```
import("stdfaust.lib");
import("hoa.lib");

N = 32; // N speakers
azimuthDelta = 360/N;
speakersPos = par(i, 32, i*phase);
sourcePos = hslider("source_pos",0,0,360,1);

fm(a,fc,fm0,fm1,z0,z1) = car
with{
    mod0 = fm1 + os.osc(z0*fm0,fm0);
    mod1 = fc + os.osc(z1*mod0,mod0);
    car = os.osc(a,mod1);
};

process = fm(1,440,440,440,3,2) :
    circularScaledVBAP(speakersPos,sourcePos);
```

Figure 3: Audio example (`vbap32.dsp`): vector-based amplitude panning for 32 loudspeakers expressed in the FAUST language. Most of the DSP algorithm is hidden in the `circularScaledVBAP` function.

```
void controlmydsp(mydsp* dsp, float* RESTRICT fControl, int* RESTRICT iZone, float* RESTRICT fZone) {
    fControl[0] = (float)(dsp->fHslider0);
    fControl[1] = sinf(0.0174532924f * (31.0f - fControl[0]));
    fControl[2] = sinf(0.0174532924f * fControl[0]);
    fControl[3] = sqrtf(mydsp_faustpower2_f(fControl[1]) + mydsp_faustpower2_f(fControl[2]));
    [...]
    fControl[221] = (fControl[212] * fControl[213]) [...] / fControl[218];
    fControl[222] = (fControl[219] * fControl[220]) [...] / fControl[3];
}
void sendControlToFPGA() {
   XaudioIP_Write_ARM_fControl_Words(&xaudioIP, 0,(u32*)fControl, 223);
}
```

Figure 4: Excerpt of the control code (file `audioCtl.cpp` in Fig. 2) to be run on the ARM processor of the FPGA board generated from `vbap32.dsp` (see Fig. 3) by the FAUST compiler.

4 Audio-Specific Compilation Optimizations

To illustrate our compiler, we chose a non-trivial audio DSP program implementing vector-based amplitude panning (depicted in Fig. 3 for 32 loudspeakers). The goal here is not to understand the panning application (the interested reader will refer to [18]) but to get a sense of how this code is compiled to hardware and software.

4.1 Hardware/Software Split: Sample Rate vs. Control Rate

The panning algorithm carries out a series of `cos` and `sin` operations when the source is moving (within the `circularScaledVBAP` function). These computations are only executed when needed, i.e., when the position of the source – which is controlled by the (`source_pos` slider on Fig. 3) – is modified by the user. Hence, these computations belong to the *control part* implementing the *software portion* running on the ARM processor.

Since its launch in 2002, the FAUST compiler has been optimized for audio programs. In particular, it integrates a memory optimization scheme (sharing delay lines) as well as a precise classification of each computed signal as *control rate* or *sample rate*. We used this classification to generate two files: `audioIP.cpp` and `audioCtl.cpp`. As the FAUST compiler is open-source, we were able to extend it with a dedicated option: `-os` (*one sample*). Activating this option

```
[...]
void computemydsp([...]) {
    float fTemp0 = dsp->fRec0[1];
    float fTemp1 = dsp->fRec1[1];
    dsp->fRec1[1] = dsp->fConst2 + fTemp1 - floorf(fTemp1);
    [...]
    outputs[0] = (FAUSTFLOAT)(fControl[11] * fTemp4);
    [...]
    outputs[31] = (FAUSTFLOAT)(fControl[222] * fTemp4);
}

void audioIP(sy_ap_int in_ch0_V, sy_ap_int* out_ch0_V,
             sy_ap_int in_ch1_V, sy_ap_int* out_ch1_V,
             int ARM_fControl[223],
             FAUSTFLOAT *ram,
             int ramBaseAddr,
             int ramDepth)
{
#pragma HLS INTERFACE s_axilite port=ARM_fControl
#pragma HLS INTERFACE s_axilite port=ramBaseAddr
#pragma HLS INTERFACE s_axilite port=ramDepth
#pragma HLS INTERFACE m_axi port=ram latency=30
    [...]
    if (first_iteration) {
      /* Constant initialization */
      /* from values initialised in DDR */
      instanceConstantsFromMemmydsp([...]);
    } else {
    /* all other iterations: compute one sample */
        computemydsp([...]);
    }
}
```

Figure 5: Excerpt of the audio IP C++ code (file `audioIP.cpp` in Fig. 2) generated from `vbap.dsp` (see Fig. 3) by the FAUST compiler and mapped on the FPGA with *vitis HLS*. The IP root function is `audioIP()`.

generates the three C++ files mentioned in Fig. 2 and restricts the generated code to compute a single sample (while many audio compiler generate code for a *buffer* of samples). This allows the IP to only have a one sample delay.

An excerpt of the code of the software portion generated by our compiler from `vbap32.dsp` is presented in Fig. 4. The `controlmydsp` function computes 223 control signals calculated from the position of the source. Then, the `sendControlToFPGA` function uses the API provided by Xilinx tools (`XaudioIP_Write_ARM_fControl_Words()` function) to communicate the control signals (array `fcontrol`) to the IP on the FPGA. It is important to note that a double buffering mechanism must be set for these external control signals arriving on the FPGA IP in order to have a simultaneous update of all the 223 control signals from one sample to the next.

The *hardware portion* of the code generated by our compiler is presented in Fig. 5. The interface of the IP (the `audioIP()` function) is shown with the pragmas indicating to *vitis HLS* (*i*) which port of the IP are controlled by the ARM processor (with the `s_axilite` protocol), (*ii*) which port is used for the external DDR memory (with the `m_axi` protocol), and (*iii*) which port are connected to other IPs. At boot time, local values are initialized from the ARM, then the `compute` function is executed infinitely, i.e., at each sample. This particular example has two audio input channels and two audio output channels (`in/out_ch0/1_V`) which are explicitly connected to the I2S IP shown on Fig. 2.

This example illustrates the power of the automatic compilation process provided by our compilation flow. The code presented in Fig. 3 is concise and easy to understand by an audio DSP engineer who can take advantage of the whole Faust DSP libraries ecosystem. The C++ code generated for this particular application was composed of 643 lines of code for the IP and 2110 lines of codes for the *control part*. The compilation time for 32 speakers on the Zybo boards was about 15 minutes on a 8-core Intel core i5 laptop at 2.6GHz, mainly spent in *Vivado* synthesis (about 30 minutes when targeting Genesys boards).

4.2 Memory Access Optimization

As for general purpose CPUs, access to external DDR is a potential important performance bottleneck. A memory access can take between 10 and 200 FPGA clock cycles depending on the amount of burst accesses. Conversely, block RAM accesses can be performed in one cycle so the variables and arrays of the audio program should be mapped as much as possible on FPGA block RAMs during the compilation process.

Some audio programs use delay lines that can use megabytes of memory and cannot always be fitted on the FPGA block RAMs. These delay lines are usually accessed only twice at each sample computation (one read and one write access). Our current strategy for optimizing memory accesses is the following: a greedy algorithm first maps small arrays on the FPGA, and switches to DDR mapping when the size of FPGA block RAM is exceeded.

Vitis HLS accepts a pragma for the *expected* latency of each memory access (30 clock cycles in Fig. 5) and uses this information for scheduling the IP. It is of course very difficult to *guess* what the real memory access time is in practice. In our experiments in § 5.3, we show that for a cycle duration of 8ns (125MHz), 30 cycles is usually a good value for the latency pragma, so this is the value currently used for our audio IP memory interface pragma.

4.3 Control and Human-Machine Interface

Both a hardware/physical and a software control interface were developed for the system presented in this paper. The hardware interface takes the form of a PCB board with physical controllers (i.e., buttons and potentiometers) interfaced to the ARM processor via an SPI ADC

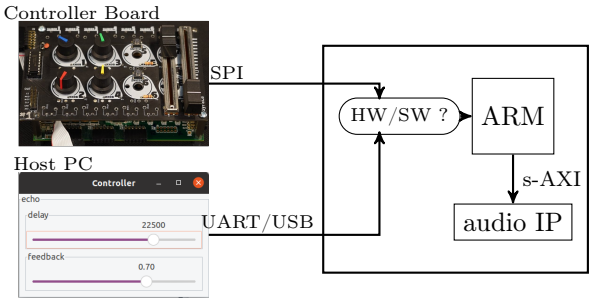

Figure 6: Hardware (knobs) and software (GTK app) interfaces for the FAUST IP (only one of them can be used at a time).

chip (the hardware interface has been more precisely described in [16] and is available on GitHub as open design). This board was designed for demonstration purposes and can only be used for FAUST programs with less than 8 controllers.

A more flexible software interface can be used on the host computer thanks to the USB/serial connection with the FPGA board. It uses the GTK library and it is inherited from the FAUST ecosystem.[10] This interface is automatically generated from the audio DSP program. It is possible to dynamically switch between the software and hardware interface if a compatible hardware board is available. These control mechanisms are depicted in Fig. 6.

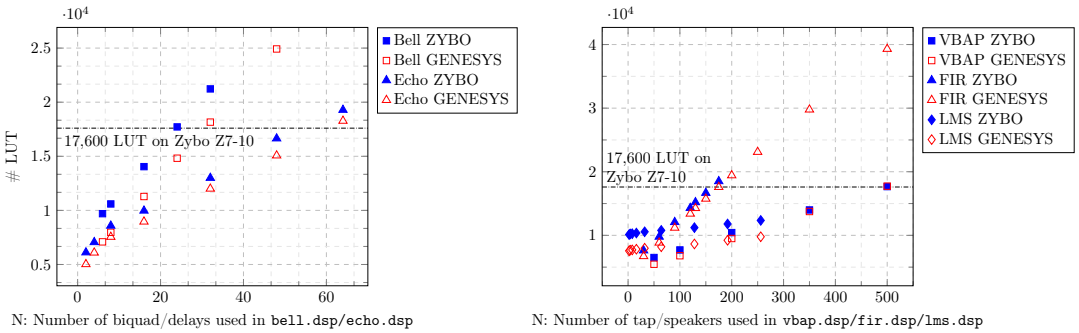

Figure 7: Size of various audio applications in terms of FPGA LUT on the Zybo Z7-10 and Genesys ZU-3EG (70,560 LUTs available). Similar results are available for FF, BRAM, or DSP. Usually, LUT usage is the first to reach 100% among other FPGA resources but latency, i.e., memory access time (Fig. 8) is usually the most important bottleneck.

5 Performance Results

The main contribution of this work is an open-source flow to compile audio DSP programs down to an FPGA with many tunable parameters (i.e., number of input/output audio channels, different audio codec used for each channel, sample rate up to 768 kHz, target FPGA board, etc.) and achieving ultra-low latency.

[10] `https://faust.grame.fr/`

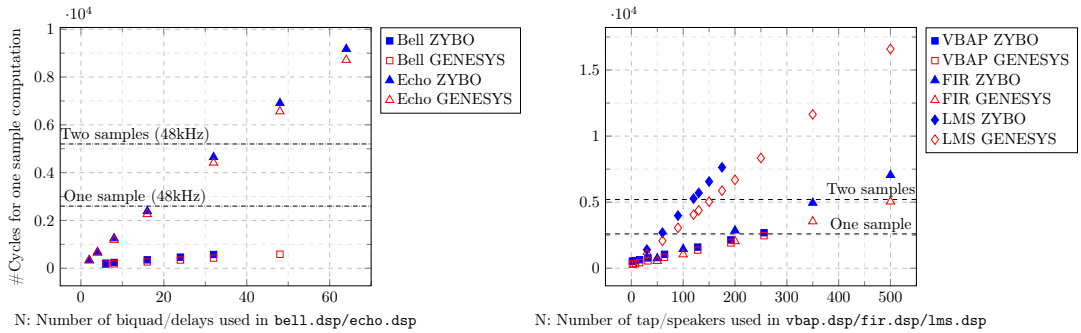

Figure 8: Number of cycles for one sample computations on various applications with an FPGA clock frequency of 125MHz. The number of cycles available between two samples at 48kHz audio sample rate is indicated as a guide.

In this section, we show some performance results related to latency, throughput, and design complexity. We have chosen a set of audio programs from the FAUST libraries[11] that can be parameterized by a size parameter `N`. These applications are: (i) a bell modal model implemented with `N` biquad filters, (ii) an echo with `N` different delays, (iii) a `N`-taps FIR filter, (iv) a `N`-taps fxLMS filter, and (v) the already mentioned `vbap.dsp` panning application with `N` loudspeakers.

5.1 Audio IP Complexity

Fig. 7 shows the size – in terms of FPGA look-up tables (LUTs) – of the design generated by our compiler for the aforementioned audio programs for various values of `N`. We did not show the numbers of Flip-flops, BRAMs, or DSPs as, on all DSP applications that we have tested, they are usually comparable with the number of LUTs (which is usually the most used resource). Fig 8 shows the latency of the computation of one sample expressed as a number of FPGA clock cycles. This computational latency must be inferior to the sample rate (i.e., approximately $20.8\mu s$ at 48kHz) since we do not pipeline computations of samples.[12] Synthesis has been done for the Digilent Zybo Z7-10 and Genesys ZU-3EG boards. Some applications – such as `echo.dsp` – are limited by their computational latency, because they heavily use DDR memory accesses, while some others – such as `bell.dsp` – are limited by the FPGA LUT numbers (at least on Zybo Z7-10).

These complexity results have been obtained without adding any *vitis HLS* pragmas to the ones mentioned before. Again, The computation of the samples are not pipelined, the objective was to optimize the analog to analog latency: all these objects perform their computations during just one sample. The resources used by the audio IP or its latency can be significantly impacted using *vitis HLS* pragmas in the IP source code. In addition, the audio latency constraints could be relaxed and the computation of each sample could last more than one sample if sample computations are pipelined. These results are shown here to demonstrate that low audio latency FPGA designs can be rapidly achieved from various high-level audio programs and that some applications are limited by FPGA resources while others are limited by latency constraints. Further dedicated optimization will allow more efficient implementations but will imply additional design time.

[11] `https://faustlibraries.grame.fr/`

[12] Computational latency and audio latency should not be confused in this section: the later designates the analog in to analog out latency and the other the time it takes to compute one audio sample.

5.2 Latency Performance

We obtained very good latency results from analog input to analog output with our system. They are presented in Fig. 9 for various codecs and various sampling rates.

The analog to analog audio latency depends on many parameters. Of course, the latency needed for the computation of each sample by the FAUST IP has a significant impact. The latency implied by digital audio parameters (bit-width of each samples, sampling rate, etc.) also plays an important role. For instance, at a sampling rate f_s^{I2S} of 48 kHz, one sample corresponds to a latency of 20.833 µs. On the other hand, if a sampling rate of 768 kHz is used, one sample only corresponds to a latency of 1.3 µs. Hence, increasing f_s^{I2S} lowers the latency but it also decreases the available time for the FAUST IP to compute each sample, potentially reducing computational power.

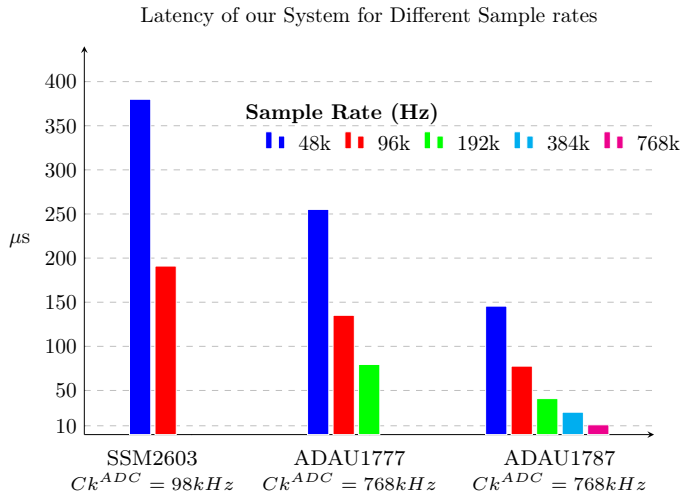

Figure 9: Latency measurements results. Latency was measured from analog input to analog output using square signals in different configurations.

Additionally, audio codecs introduce latency known as *group delay* corresponding to the delay added by the ADC/DAC. This delay depends on many parameters, including the codec internal clock rate that we note Ck^{ADC}, the sampling rate, and the complexity of signal reconstruction filters. Hence it is very important to evaluate experimentally the effective latency.

Three different audio codecs were used for our experiments (very few codecs on the market are optimized for latency and the ADAU1777 and the ADAU1787 are the ones providing the best performance that we could find):

- The Analog Devices SSM2603 (the built-in codec of the Zybo board), which is not optimized for low latency. ($\max f_s^{I2S} = \max Ck^{ADC} = 96$ kHz)
- The Analog Devices ADAU1777, whith better latency performace. ($\max f_s^{I2S} = 192$ kHz, $\max Ck^{ADC} = 768$ kHz)
- The Analog Devices ADAU1787, which provides the best latency performance. ($\max f_s^{I2S} = \max Ck^{ADC} = 768$ kHz)

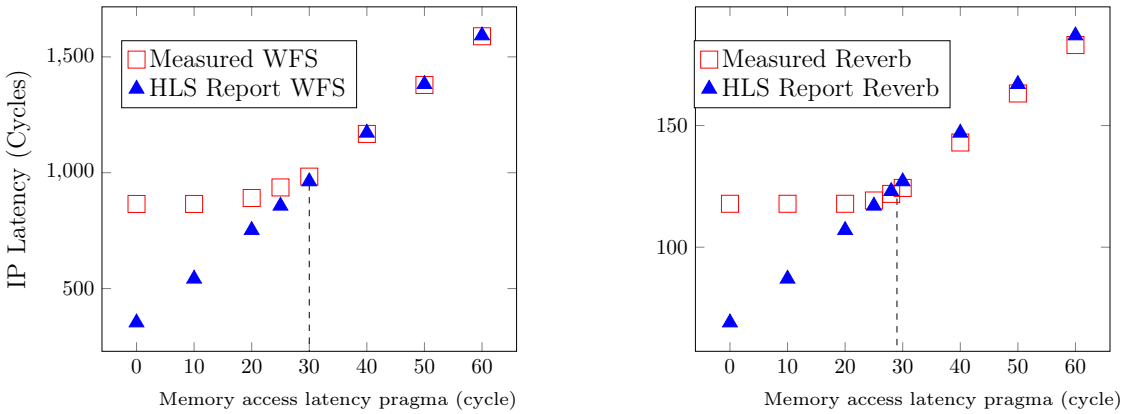

Figure 10: Estimated vs. measured latency of the IP for different values of the *memory latency pragma* for two different audio DSP algorithms (WFS and Reverb) on Zybo board.

The best latency (11 µs) is obtained with the ADAU1787 codec with a sampling rate of 768 kHz. This particular performance has been described in more details in a separate publication [16] and was issued before the complete compilation flow was released. This latency is one order of magnitude less that previously announced works. As mentioned before, audio latency has been widely studied ([26, 15, 22, 23]) and the best latency previously obtained, 180 µs, was announced in [22] with a codec clock at 768 kHz.

5.3 Memory Performance

As explained in §4.2, the performance of memory accesses is very difficult to predict and there are very few official performance metrics provided by Xilinx for external DDR access in HLS designs for the boards used (Zybo and Genesys). During the HLS process, the parallelization of the C++ input code is performed. The latency of the resulting schedule – i.e., number of cycles to compute one sample – is reported at the end of HLS processing. The precision of that report is important because it indicates if computation can be performed before the arrival of the next sample. If it is not the case, the sample is lost[13] and the produced sound will be distorted.

We needed to measure the effective number of cycles taken by the computation of one sample by the IP generated by *Vitis HLS*. For that, we connected the `ap_start` signal sent to the IP to trigger computation, and the `ap_done` signal sent from the IP to indicate the end of the computation of the current sample to an oscilloscope through the board's GPIOs. This allowed us to precisely measure the effective IP latency as a number of cycles.

From our experiments, if no access to DDR memory is carried out in the code – e.g., as in the `bell.dsp` example where all the memory of the program can be stored on FPGA block RAMs – the evaluation reported by the HLS tool (*Vitis HLS*) is very precise, if not perfect. But if reads or writes occur in DDR memory, the precision of the report relies on the *latency* given as a pragma to *Vitis HLS* (e.g., 30 cycles in Fig. 5).

Our measurements are presented in Fig. 10 for two audio DSP programs implementing a Wave Field Synthesis and an artificial reverberation algorithm that have been synthesized and executed for different values of the *latency* pragma. These measures show that, in these two

[13] In the first version of the compiler, successive sample computations do not overlap because we are targeting ultra-low latency. In future work, we will introduce pipelining between successive samples for long sequential sample treatments.

Number of channels	4	8	16	32	256
LUT usage for standard I2S	222	274	386	606	3601
Register usage for standard I2S	300	496	892	1697	12697
GPIO pin usage for standard I2S	4	6	10	18	130
LUT usage for TDM I2S	159	208	271	358	2878
Register usage for TDM I2S	267	501	697	1227	9546
GPIO pin usage for TDM I2S	4	4	4	6	34

Table 1: Size and pins usage for standard I2S and our optimized TDM I2S implementation for various numbers of I/O channels.

cases, the *optimal* value for the latency pragma is 30 cycles. Below 30 cycles, the prediction is not realistic, and above 30 cycles, scheduling is not optimal. This optimal value could be very different for other applications, depending on their memory access patterns. However, for all the audio programs that we tested on both Zybo and Genesys FPGA boards, this 30 cycles value was approximately the best value to use for a 125 MHz clock. Hence, we believe that the restricted nature of audio applications facilitates this memory performance prediction problem.

5.4 Multichannel Performance

Having an audio system with many (i.e., more than 32) audio inputs and outputs at a low cost is one of the challenges that FPGA can help solving. Increasing the number of audio channel connections can be done either by duplicating I2S transceiver modules or by using Time Division Multiplexing (TDM) in the I2S protocol (or the combination of the two). For instance, with the Zybo codec (SSM2603 codec by default), a single conventional I2S module is used providing two input and two output channels as shown in Fig. 2. A conventional I2S transceiver uses 4 pins: bit clock, word select (`ws`), and transmitted and received data (`sd_tx` and `sd_rx`). If conventional I2S IPs are duplicated, they will share the clock and `ws` signals.

When a large number of channels is used, the number of available GPIO pins on the board might become a limitation (32 GPIOs for the ZYBO for instance). In TDM I2S, several channels can be grouped on a single (`sd_tx`, `sd_rx`) wire pair (16 channels for the Analog Devices ADAU1787 codec for instance). Hence, TDM I2S configuration should be preferred for a large number of I/O channels. The size complexity and GPIO used are presented in Table 1 for a standard I2S IPs and for our optimized TDM I2S implementation. First, it can be seen that TDM I2S uses far less GPIOs, then is also illustrates the fact that a large number of channel can be handled by our system.

6 Conclusion

In this paper, we presented the first audio DSP to FPGA compiler offering a high level of flexibility, ultra-low latency, and the possibility to manage a large number of audio channels at a reduced cost. This compilation flow associates a traditional compiler (the FAUST compiler) with a high-level synthesis tool (*Vitis HLS*) which acts as a middle-end by optimizing the resulting implementation for FPGA targets. Our flow uses the standard Xilinx *Vivado* synthesis tool as a back-end. Apart from proprietary Xilinx tools, the platform is open-source and available

on GitHub.[14]We hope that this flow will be useful to audio developers, but also as a way for experimenting the combination of a compilation tool with an HLS tool.

The preliminary performance presented here for various applications could be improved by better tuning the level of parallelism used in the IP. Currently the FAUST compiler flattens the loops in the audio program to exhibit maximum possible parallelism. In some cases, when latency is not the main objective, it might be more interesting to slow down samples computation (i.e., increase latency) in order to increase throughput. This is currently under investigation.

We are also currently studying the application of the compilation flow in the context of active acoustic control [1] as well as for 3D audio techniques (Wave Field Synthesis and ambisonics).

Acknowledgment

This work has been carried out in the context of the FAST ANR project[15] (ANR-20-CE38-0001) funded by the French ANR (Agence National de la Recherche).

References

[1] L. Alexandre, P. Lecomte, M.-A. Galland, and M. Popoff. Feeedback Acoustic Noise Control with Faust on FPGA: Application to Noise Reduction in Headphones. IFC 2022, June 2022.

[2] D. Cannon, T. Fang, and J. Saniie. Modular delay audio effect system on FPGA. In *2022 IEEE International Conference on Electro Information Technology, EIT 2022, Mankato, MN, USA, May 19-21, 2022*. IEEE, 2022.

[3] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen. Cloud-DNN: An open framework for mapping DNN models to cloud fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 73–82, New York, NY, USA, 2019. ACM.

[4] S.-H. Chung and T. S. Abdelrahman. A compilation flow for the generation of CNN inference accelerators on FPGAs. arXiv, 2022.

[5] P. Coussy. GAUT: an open-source HLS tool. In *Free Silicon Conference (FSiC)*, Paris, France, Mar. 2019.

[6] A. S. Deulkar and N. R. Kolhare. Fpga implementation of audio and video processing based on zedboard. In *2020 International Conference on Smart Innovations in Design, Environment, Management, Planning and Computing (ICSIDEMPC)*, pages 305–310, 2020.

[7] 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, July 2021. IEEE.

[8] Y. E. Esen and I. San. Low-Latency SoC Design with High-Level Accelerators Specific to Sound Effects. *International Journal of Advances in Engineering and Pure Sciences*, 33, dec 2021.

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

[14] `https://github.com/inria-emeraude/syfala`
[15] `https://fast.grame.fr`

[10] L. Merah, P. Lorenz, A. Ali-Pacha, and N. Hadj-Said. A Guide on Using Xilinx System Generator to Design and Implement Real-Time Audio Effects on FPGA. *International Journal of Future Computer and Communication*, Sept. 2021.

[11] J. R. G. Ordaz and D. Koch. HLS compilation for CPU interlays. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART2017, New York, NY, USA, 2017. Association for Computing Machinery.

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

[13] B. Pauget, D. J. Pearce, and A. Potanin. Towards compilation of an imperative language for FPGAs. VMIL 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[14] 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.

[15] E. Pongratz and R. C. John. Performance Evaluation of MathWorks HDL Coder as a Vendor Independent DFE Generation, 2019. Master PhD.

[16] 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 *SMC-22 - Sound and Music Computing*, Saint-Étienne, France, June 2022.

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

[18] V. Pulkki. Generic panning tools for MAX/MSP. In *Proceedings of International Computer Music Conference*, 2000.

[19] A. Singhani and A. Morrow. Real-Time Spatial 3D Audio Synthesis on FPGAs for Blind Sailing. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside CA USA, Feb. 2020. ACM.

[20] R. K. Snider. *Advanced Digital System Design using SoC FPGAs*. Springer Cham, 01 2023.

[21] K. Vaca, M. M. Jefferies, and X. Yang. An Open Audio Processing Platform with Zync FPGA. In *2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR)*, Houston, TX, USA, Sept. 2019. IEEE.

[22] 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.

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

[24] S. I. Venieris and C.-S. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[25] M. Verstraelen, J. Kuper, and G. J. Smit. Declaratively Programmable Ultra Low-Latency Audio Effects Processing on FPGA. In *DAFx*, 2014.

[26] Y. Wang. *Low latency audio processing.* PhD thesis, Queen Mary University of London, 2018.

[27] 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.

[28] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen. Machine learning on fpgas to face the iot revolution. In *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017*, pages 894–901. IEEE, Dec. 2017.

[29] Z. Zhang, D. Chen, S. Dai, and K. Campbell. High-level synthesis for low-power design. *IPSJ Transactions on System LSI Design Methodology*, 8:12–25, 02 2015.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399