IMPLEMENTING A LOW-LATENCY PARALLEL GRAPHIC EQUALIZER WITH HETEROGENEOUS COMPUTING

Vesa Norilo, *

Centre for Music & Technology
University of Arts
Helsinki, Finland
vesa-petri.norilo@uniarts.fi

Math Verstraelen

Computer Architecture for Embedded Systems
University of Twente
The Netherlands
m.j.w.verstraelen@utwente.nl

Vesa Välimäki

Department of Signal Processing and Acoustics
Aalto University
Espoo, Finland
vesa.valimaki@aalto.fi

ABSTRACT

This paper describes the implementation of a recently introduced parallel graphic equalizer (PGE) in a heterogeneous way. The control and audio signal processing parts of the PGE are distributed to a PC and to a signal processor, of *WaveCore* architecture, respectively. This arrangement is particularly suited to the algorithm in question, benefiting from the low-latency characteristics of the audio signal processor as well as general purpose computing power for the more demanding filter coefficient computation. The design is achieved cleanly in a high-level language called *Kronos*, which we have adapted for the purposes of heterogeneous code generation from a uniform program source.

1. INTRODUCTION

The graphic equalizer is a signal processor with a variety of purposes, ranging from tonal correction of rooms and transducers to sound design and mastering [1]. One of the key benefits of the design is the immediate correspondence of user interface and frequency response. The quality and precision of this correspondence is key to a successful graphic equalizer design. In addition, many of its applications require low-latency processing—such as tonal correction of a loudspeaker system in a live situation.

In the subsequent sections, we present a recent parallel graphic equalizer design and show how to adapt it to a heterogeneous system of a PC and a WaveCore signal processor, utilizing a high-level programming language called Kronos. Kronos and WaveCore share a declarative programming methodology, greatly facilitating code generation. In addition, the signal rate factorization capabilities of Kronos are directly applicable to the problem of heterogeneous code generation.

This paper is organized as follows. In Section 2, *Background*, we discuss the history and state of art of the three central themes of this research: graphic equalization, dedicated signal processors, and musical programming languages. Section 3, *Methodology*, presents our equalization algorithm as well as describes the dedicated signal processor hardware, *WaveCore*, this research is founded on. The high-level programming method, *Kronos*, is also presented. Section 4 details the results of this work, discussing the aspects specific to heterogeneous computing and the adaptation of the equalizer algorithm to *WaveCore*, as well as avenues for future research. Section 5 concludes the article.

* This work was supported by the Emil Aaltonen Foundation.

2. BACKGROUND

2.1. Graphic Equalization

There are two types of equalizers, parametric and graphic ones [1]. In a parametric equalizer the user has access to the gain, center frequency, and bandwidth of the filter. A graphic equalizer can be composed of several parametric equalizers or other similar filters, but their center frequencies and bandwidths are fixed. In the graphic equalizer, the user only controls each band gain by using a set of sliders, which form approximately the desired magnitude response [1, 2, 3, 4]. The band gains are usually called 'command gains'. The main challenge in designing a graphic equalizer is the interaction of neighboring band filters. Problems arise when to adjacent command gains have a large difference.

In this work we implement one of the recent graphic equalizers, which solves the interaction problem by optimizing the parameters with a least-squares technique [4]. For processing the input signal, this graphic equalizer uses the parallel IIR filter structure, which is well suited to parallel computing [5].

2.2. Musical DSP Chips

Digital Signal Processing for music applications is usually based on a sound production model. In general, the nature and complexity of the application of such a model is directly related to the available computational capacity within the constrained boundaries of cost and technological state of the art. The nature of a sound production model can roughly be divided into three areas [6][7]. (1) Digitized analog models. The origin of these models are usually analog electronics which are mimicked by discrete-time models, e.g. discrete time models of analog stomp-box gui-

tar effects [8]. The required computational capacity for this class of production models is usually modest. The sampling rates are usually relatively low, and depend on the amount of non-linearity in the applied models (e.g. amp models) (2) Digital WaveGuide (DWG). DWG models are often abstract representations of physical/acoustical phenomena. An examples is DWG based reverberation [9].(3) Physical Modeling. This class of modeling is based on detailed aspects (e.g. geometry, material, etc.) of physical sound production devices [6]. The associated modeling is often based on multi-dimensional wave equations and often requires a vast amount of computational capacity.

2.2.1. *Relevant processor technologies*

DSP in embedded musical applications is predominantly implemented with dedicated DSP processor chips. These chips are optimized to cost and energy effectiveness. These DSP chips are usually programmed within a C-based development methodology. However, the inherent parallelism within these processors can often only be addressed with processor dependent compiler "intrinsics" which makes it often difficult to port existing application code to different chips. Likewise, the learning curve to program these devices is often steep. DSP processor chips are widely applied for digitized analog modeling and/or digital waveguide. Unlike the often data-flow oriented nature of the application, the programming methodology is merely imperatively, which conceptually adds an extra complexity dimension. Digital Audio Workstation (DAW) are software applications, mostly running on PC platforms. A wide variety of sound production models are implemented with this technology. Development environments for DAW are merely C-based. However, also declarative programming methodologies, like Faust [10], are applied within this domain. Typically, keeping the processing latency within acceptable limits is challenging within PC based real-time audio processing systems.

We mentioned that detailed physical modeling (i.e. based on finite difference modeling) requires a vast amount of computational capacity. Both General Purpose Processors (GPP) and DSP chips cannot deliver the required computational capacity necessary for detailed real-time physical modeling. Moreover, a large physical model may need to be partitioned over several GPP cores when the application is required to be mapped on a PC based computer platform. Given these problems, different processor technologies such as GPGPU (General-Purpose computing on Graphics Processing Units) or FPGA (Field Programmable Gate Array) are applied. The associated programming methodologies are usually data-flow oriented and hence link more naturally to a declarative or functional programming style rather than an imperative C-based programming environment.

2.2.2. *Implications of the multi-core era*

The advance of semiconductor technologies, related to Moore's law, has a huge impact on the evolution of processor technologies (GPP, GPU, FPGA, DSP). Most dominantly, the energy- ILP- (Instruction Level Parallelism) and memory walls have driven the processor evolution into the "multi-core" era [11]. Multi-core has a huge impact on programming methodologies, linked to scalability (number of processor cores). In particular, the imperatively based programming methodologies and associated multi-core processors (e.g. DSP, GPP) are faced with a big challenge. Parallelism needs to be extracted from the imperatively described algorithm and subsequently partitioned and mapped on a multi-core processor. On the contrary data-flow oriented declarative programming styles, associated to GPGPU or FPGA are naturally scalable and are inherently "Moore-proof" to a larger extend. This is exactly the reason why GPUs and FPGAs are gaining attention as processing platforms for scalable and computational intensive algorithms like detailed physical modeling. An FPGA is basically a fabric of primitive programmable logic functions and embedded memories which can be connected in an almost arbitrary way. This means that a non-configured FPGA can be seen as an undefined chip. Therefore, designing an FPGA based application implies that the configurable circuitry of the FPGA (i.e. the "softcore") needs to be developed in a Hardware Description Language (HDL). This softcore development in itself often is a costly process which is partially caused by the abstraction level of commonly applied HDL methodologies (e.g. VHDL). Emerging methodologies, based on mathematical modeling using functional languages [12] are intended to raise the abstraction level and hence shorten the development cycle of softcores. On top of softcore development, the HW/SW interface and the software part needs to be developed. This makes FPGA design a multi-disciplinary task which requires both HW and SW engineering skills.

2.3. Programming Signal Processors

2.3.1. *Programming for DSP Chips*

Perhaps the most utilized method of programming signal processors is via low-level languages such as C or the native assembly language of the target chip. Libraries of typical signal processing primitives (such as digital filters, delay lines and signal transforms) are offered by chip vendors and third parties alike. This approach is relatively straightforward and tends to result in efficient utilization of the hardware, as long as the algorithms being implemented can be expressed in terms of typical primitives.

When the algorithm in question is more complicated, the traditional method of low level programming is no longer quite as attractive. Implementation of novel processing primitives typically requires a high level of expertise on both the algorithm and the hardware in question, and the resulting work is tightly coupled with a specific architecture. These factors make signal processing code averse to being portable.

2.3.2. *High-Level DSP Languages*

Several approaches exist for programming signal processors without a high level of C or Assembly expertise. National Instruments LabVIEW [13] is a graphical interface to the *G language*, which is based on data flow, similar to our methodology. However, the LabVIEW system is proprietary and opaque, so utilization of custom hardware like the *WaveCore* is not easily achievable.

Another commercial DSP design methodology is based on MathWorks' MatLab and SimuLink (e.g. [14]). SimuLink supports generation of C code or direct synthesis of signal processing cores via a hardware description language. These approaches are focused on multi-domain simulation and are likely too elaborate for design and application of musical signal processors.

The advantage of our proposed method is in its domain-specificity; as both *Kronos* and *WaveCore* are specifically designed for musical signal processing, several assumptions can greatly simplify the task for automatic heterogeneous code generation. In the present

study, we utilize a straightforward clock domain mapping for code factorization – discussed further in Section 3.3.2.

2.3.3. Musical Programming Languages

Musical Programming Languages are a topic of active research. The present study is based on our *Kronos* programming language [15]. This section offers a brief overview of influential design paradigms and language implementations specific to musical signal processing.

The most widely adopted paradigm for musical signal processing is the unit generator concept [16, pp. 787–810]. The ugen concept was solidified by the MUSICn family, of which CSound [17] is the contemporary example. Pure Data [18] and SuperCollider [19] are important developments of the concept. Ugen programming is declarative by nature. The programming task focuses on signal flow and topology, rather than the chronological ordering of program statements.

Ugen languages can be criticized for their lack of higher level program constructs. Especially graphical front ends such as Pure Data [18] – an example of a very successful implementation – make it quite difficult to express data types, control flow or program composition, all of which enhance programmer productivity.

Several attempts have been made to address this problem. CLM [20] attempts to merge the ugen concept with Common Lisp [21], an acclaimed high-level language. Since making high-level signal processing programs efficient is difficult, CLM delimits signal processors to a certain subset of Common Lisp and transcompiles that to C. The resulting programming paradigm is a mixture of styles, featuring a C-like programming style with Common Lisp syntax.

SuperCollider [19] is a more actively developed idea in the same vein. SuperCollider is influenced by SmallTalk (e.g. [22]), and integrates ugens built in the more performant C language. As a result, SuperCollider programs are tied with the implementation of the run time library, and the actual signal processors are opaque to the user.

Faust [23] attempts to address signal processing in a functional high-level idiom. Faust features an expressive block diagram composition system capable of succinctly describing many typical signal flows. Faust transcompiles to C, with recent work aimed towards direct compilation using the LLVM [24] framework.

Kronos [15] is the language used and adapted for the present study. Influenced by Faust, it offers a functional signal processing paradigm, enhancing it with advanced metaprogramming capabilities, automatic factorization and an advanced compiler pipeline [25]. Kronos makes use of LLVM [24] for native code generation. For the purposes of the present study, an experimental WaveCore code generator has been developed, along with facilities for heterogeneous compilation of a uniform source program for several distinct hardware targets.

3. METHODOLOGY

3.1. Graphic EQ Algorithm

In this work we implement a recently developed parallel graphic equalizer [4]. The filter itself is a parallel IIR structure in which each filter block has a special second-order transfer function: a second-order denominator transfer function but a first-order numerator transfer function. Additionally, there is a direct path with a real weight from the input to the output.

The poles of the graphic equalizer are set in advance at predesigned frequencies determined by the frequency resolution of the graphic equalizer. For example, when a third-octave graphic equalizer is designed, the poles go at 31 standard frequencies between 20 Hz and 20 kHz (20 Hz, 25 Hz, 31.5 Hz, 40 Hz etc.). To obtain high accuracy, additional poles are assigned at 10 Hz and between each standard center frequency, so that there will be altogether 62 poles. The pole radii as chosen so that the magnitude responses associated with neighboring poles meet at their -3 dB points. All of this is done off-line before running the filter. The mathematics related to this design are detailed in [4].

During real-time operation, the user can adjust the command gains of the graphic EQ. To achieve a great accuracy, this EQ design uses least-squares optimization to adjust the numerator (feedforward) coefficients, two per pole. This is similar to FIR filter design and only requires a matrix operation. However, the nonnegative weighting function is needed to ensure that attenuation is implemented correctly. As a result, every time a command gain is changed, the matrix inversion needs to be executed and all feedforward coefficients of the graphic EQ updated [4].

3.2. WaveCore – a High Performance Audio DSP Core

WaveCore is a programmable many-core processor which is optimized to real-time acoustical and physical modeling. This processor concept aims to address the scalability problem which is described in section 2.2.2. Target applications are all the sound-production models which are mentioned in section 2.2. Classical "digitized analog", and digital waveguide modeling with ultra-low latency using WaveCore has been published in [26]. A recently carried out feasibility analysis on the usability of WaveCore for real-time physical modeling of musical instruments using finite-difference time-domain techniques has yielded promising results [27].

3.2.1. Programming Model

The WaveCore processor can be programmed by means of a description of a data-flow graph. Such a graph consists of one or more processes, that are interconnected by means of edges. The data-elements (i.e. tokens) that are carried over the edges consist of one or more Primitive Token Element (PTE, a floating point number). All processes in the graph are periodically executed (i.e. fired) explicitly by a centralized scheduler, where multi-rate execution is fully supported. When a process is fired, it consumes one token per inbound edge and produces one token per outbound edge. A process may be composed by one or more process partitions (WPP). Ultimately each process consists of a number of interconnected Primitive Actors (PA). The PA has at most two inbound edges and as such consumes at most two PTEs x_1 and x_2 when the PA is fired. The PA produces one PTE $y^{n+\lambda}$ when it is fired through an optional delay-line.

The programming model supports a limited set of different PAs. A few examples are: (1) C-type PA: $y^{n+1} = p$, (2) MAD-type PA: $y^{n+\lambda} = p \cdot x_1^n + x_2^n$, (3) ADD-type PA: $y^{n+\lambda} = x_1^n + x_2^n$ (4) MUL-type PA: $y^{n+\lambda} = x_1^n.x_2^n$, and (5) AMP-type PA: $y^{n+\lambda} = p.x_1^n$.

This data-flow oriented programming model enables a declarative way of implementing a wide variety of signal processing algorithms, like flanger, wah-wah, reverberation, EQ, etc. [26]. The declarative nature of the programming methodology enables

a straightforward mapping to functional languages like Kronos, as we will outline in Section 3.3.

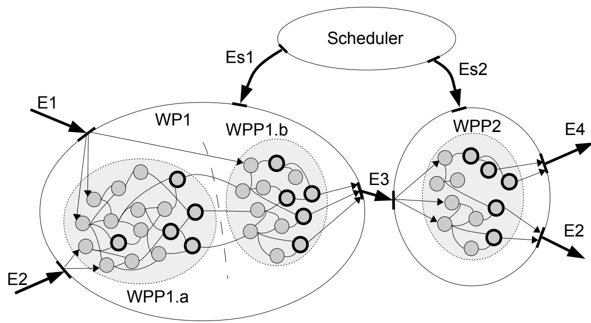

Figure 1: Data-flow oriented WaveCore programming model.

3.2.2. Processor Architecture

The WaveCore processor consists of a scalable cluster of Processing Units (PU). Each PU embodies a small Reduced Instruction-Set Computer (RISC). The instruction set of the PU is fully optimized to the execution of a group of PAs (WPP) where each PA is mapped on a single instruction. The PU can be classified as a pipelined Harvard processor (instructions and operands are located in separate memories). This implies that the PU is capable of executing one PA per clock cycle. The heart of the PU is a single precision floating-point ALU which supports basic arithmetic operations such as add, subtract, multiply, multiply/add, divide, compare, etc. Next to the instruction pipeline the PU is equipped with a DMA controller, called Load/Store Unit (LSU). This LSU is loosely coupled to the processor pipeline and is responsible for moving token data and/or delay-line data between external memory and the processor pipeline. An on-chip network, called Graph Partition Network (GPN) connects all PUs.

The WaveCore compiler automatically partitions and maps a WaveCore process onto the processor hardware. Each WPP is scheduled and mapped on a PU, and the connections between the WPPs are mapped on the GPN. The compiler also takes care of external memory allocation and as such maps all tokens and delay-lines on the LSU parts of the associated PUs.

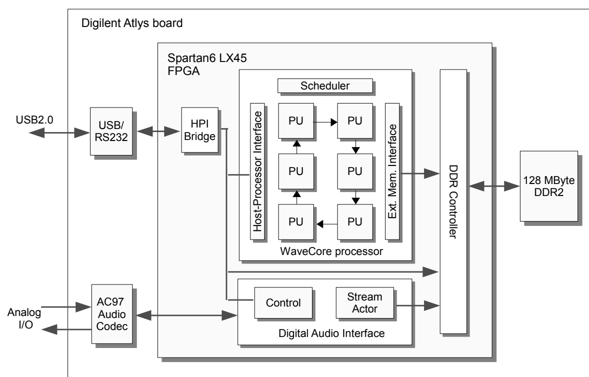

Figure 2: WaveCore processor on Atlys FPGA board.

The WaveCore processor is implemented as a soft-core in a Hardware Description Language (HDL). We have integrated a core instance with 6 PUs in System-on-Chip which subsequently is mapped on a Xilinx Spartan6 LX45 device on the Digilent Atlys development board. The block diagram of this development board and FPGA is depicted in fig. 2. The PU cluster can be initialized (i.e. loading a compiled WaveCore program to the embedded instruction memories within the PU cluster) by means of en externally connected host computer through the USB interface. Runtime control (i.e. run-time modification of control-tokens) is possible through the same USB interface, which also has full access to the external DDR2 memory on the board. The board contains an AC97 compliant audio codec chip (stereo audio DAC and ADC). This codec is enabled to stream autonomously into/from the DDR memory. The WaveCore processor cluster itself, which is capable of executing a process graph with up to 12288 PAs at 44.1kHz audio rate, is also capable of autonomously accessing the DDR memory.

3.2.3. The Graphic Equalizer as a WaveCore process graph

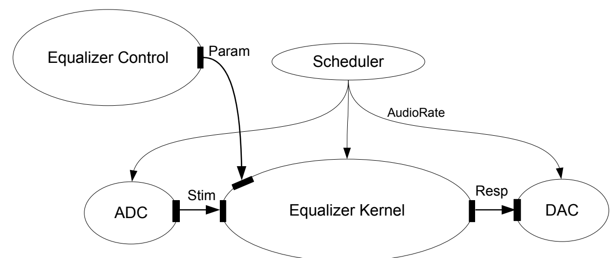

Figure 3: Process graph of equalizer

The equalizer application as a WaveCore process graph is depicted in fig.3. The ”Equalizer Control” process is mapped on the externally connected computer and the ”Equalizer Kernel” process runs on the WaveCore processor. This kernel process consists of one WPP and hence utilizes only one out of the 6 PUs. The ”param” edge is mapped on the USB interface and hence implements the run-time control of the kernel process. The associated control tokens are allocated in DDR memory. The ADC and DAC processes are mapped on the AC97 codec chip and the associated audio edges ”Stim” and ”Resp” are mapped on DDR memory. The tokens which are moved over the ”Stim” edge are produced by the AC97 codec and consumed by the kernel process which is mapped on the WaveCore. Similarly, the tokens which are moved over the ”Resp” edge are produced by the kernel process which is mapped on the WaveCore, and consumed by the DAC process on the AC97 chip. The ”Stream Actor” block within the ”Digital Audio Interface” takes care of moving tokens between the AC97 chip and DDR memory. The scheduler periodically generates fire tokens to the real-time kernel, ADC and DAC processes at audio rate. Note that the ”Equalizer Control” process is not linked to the scheduler, and responds to human interaction. The processing latency is short because there is almost no buffering between the ADC/DAC processes and the equalizer kernel (one token per edge, and hence 2 T_s streaming latency).

3.3. Kronos – a High Level Music DSP Language

Kronos [25] is a high level functional signal processing language designed for musical applications. The declarative dataflow principle is common to Kronos and WaveCore, making WaveCore a natural compile target for the Kronos compiler. An experimental Kronos code generator for WaveCore was implemented for the purposes of this research.

The user-facing aspect of the Kronos language resembles a high level functional programming language with very little data type notation, along with features designed for musical signal processing. The most significant of these include language level memory operators, such as unit delays and recursions, that exhibit pure functional semantics[28] and are reified into stateful constructs by the compiler. In addition, the compiler performs full-program type derivation and dataflow analysis to generate a highly performant statically typed run time representation that can be automatically factored into several clock domains. The factorization process results in a set of driver routines that share and mutate a state buffer that represents the signal memories of the running application as well as the requisite state for transferring signals between clock regions. For a more detailed discussion, the reader is referred to prior work[15].

3.3.1. WaveCore Code Generation

The lowered runtime program representation generated by Kronos is typically compiled into native code with LLVM[24], an open source compiler back end framework. Generating code for the WaveCore compiler is much simpler, as the program format of the WaveCore compiler is declarative, exactly like the intermediate representation produced by the Kronos compiler. The WaveCore backend is a simple idiom translator implemented as a pattern matcher, in which a group of N Kronos primitives is mapped into a group of M WaveCore Primitive Actors. As WaveCore is much more streamlined than a typical CPU, not all Kronos operations map efficiently or even at all onto WaveCore programs, but the overlap is considerable, especially considering that the two designs were not coordinated initially. As of this writing, most Kronos programs dealing with single precision floating point DSP can be mapped to WaveCore, including the equalizer design discussed in this paper.

3.3.2. Heterogeneous Code Generation

Since Kronos already does signal clock factorization, heterogeneous code generation simplifies to the problem of adopting different compile targets for different signal clocks, and generating a transport layer to enable the clock regions to communicate.

Our heterogeneous code generation technique utilizes the Kronos program representation after the global type derivation and data flow analysis passes have completed. At this point, we have a static signal flow graph annotated with clock regions. Normally, the compiler would generate driver routines for each clock region, but for heterogeneous compiling, we have added an option to filter the set of clock regions included in the current compilation unit.

The equalizer design features two main clock regions: one is the audio clock region, including the parallel biquad filter bank and its summation. The other is the control clock region, which is driven by the user interface and includes the coefficient generation code. Because an update to any command gain causes recomputation of all the feed forward coefficients, it is best to use a single clock region for all the command gains.

Table 1: Equalizer error maxima, decibels

Setting	RM	EQ4	PGE
All up	9.8	3.6	0.00
Zigzag	6.3	2.8	0.75
Every 3rd up	4.1	1.5	0.32

Compiling the equalizer program source once for WaveCore, including only the audio clock region, and once for PC, including the control region, results in the requisite program objects for the two architectures. These program objects correspond to the *Equalizer Control* and *Equalizer Kernel* shown in Figure 3. Next, we address the transfer of filter coefficients from the PC component to the WaveCore component.

Since the Kronos data flow analysis detects clock region boundaries, communication can be enabled by special handling of the boundaries that involve a transition between compile targets. Since version 2.0, WaveCore has a well-defined protocol for external control. A ForeignProcess description is generated by the Kronos/WaveCore compiler, including tokens for each signal graph edge that crosses from the PC to the DSP. In the case of the graphic equalizer, these edges represent the feed forward coefficients of the biquad filter bank. On the PC side, such edges are represented by a section of the state buffer generated by the compiler for the Kronos program.

To facilitate transport, an option to invoke a user-defined callback function was added to the compiler whenever a particular boundary is updated. A special compiler driver uses the Kronos JIT Compiler to generate the PC-specific section of the signal processing system, and hooks into the callback mechanism to propagate boundary edge updates to the WaveCore submodule. These updates are sent over the serial port to the WaveCore board as per the control protocol.

4. DISCUSSION

4.1. Equalizer Response

The presented equalizer algorithm offers a highly precise frequency response. Three command gain settings were used to find the maximum error in the actual magnitude response of the equalizer in comparison to the command gains.

Table 1 lists the results of our algorithm (PGE) contrasted to the second order Regalia-Mitra EQ (MR) [29] and a higher order design (EQ4) [2]. The *All up* settings features all the command gains set to $+12$ dB. In *Zigzag*, the command gains alternate between $+12$ dB and unity. The final setting of *Every 3rd up* features one band at $+12$ dB followed by two bands at unity.

The proposed algorithm features the smallest error maxima, staying within a decibel of the target curve on all settings. For a more detailed evaluation and comparison of the PGE algorithm, the reader is referred to [4].

4.2. Performance

As the described implementation is heterogeneous, the performance characteristics that concern us are also varied. As the audio processing on WaveCore obeys hard real time constraints, we are mostly concerned about chip utilization.

Table 2: Equalizer performance summary

Equalizer Kernel	PAs	of PU	% of chip
per channel @ 44.1kHz	740	81%	13.5%
Equalizer Control	avg	max	min
time per update	32ms	51ms	31ms

On the PC side, the computational complexity manifests as control latency – the delay between user interaction and the corresponding change in the equalizer response. This latency is dominated by the time required for coefficient computation.

Table 2 summarizes the central performance characteristics of our solution. Our current WaveCore chip is a cluster of six Processing Units running at 86MHz, achievable on the Digilent Atlys board wiath a Xilinx Spartan6 FPGA. The equalizer control timings were measured on a Windows PC with an Intel Core i7 CPU running at 2.8GHz. Minimum, maximum and average control latency were collected from 1000 simulated coefficient updates. As shown, one audio channel can be equalized by one PU in hard real time. This results in a throughput latency of two sample periods, excluding the latency of A/D/A conversion. Our WaveCore cluster is computationally capable of 6 channels of real time equalization, although the Atlys board is limited to stereo audio I/O.

4.3. Future Work

In the future, the design could be adapted to an embedded setting with, for example, a low power CPU combined with a WaveCore chip and a dedicated control surface. Such a setup would function well as a dedicated hardware equalizer. Alternatively, a room correction module could be developed based on the technology, with a PC-based analysis and filter design solution combined with hardware equalization.

The support of the Kronos language on WaveCore could also be further developed. The areas of interest range from the emulation of double precision floating point arithmetic to automatic factorization of programs to several concurrent WaveCore process partitions to better utilize parallelism.

5. CONCLUSION

In this paper, we presented an implementation of a recent graphic equalization algorithm on a heterogeneous computing platform consisting of a commodity PC and a WaveCore-based signal processing board. The system was shown to exhibit excellent latency characteristics due to the use of dedicated hardware, as well as excellent precision due to the advanced coefficient computation technique made possible by a powerful CPU. The graphic equalizer serves as an example of our proposed heterogeneous signal processor development workflow, which enables automatic factorization of a single source program to two distinct program objects.

The Kronos compiler suite is available in binary and source form at `https://bitbucket.org/vnorilo/k3`. WaveCore is available for the published Digilent Atlys FPGA development board.

6. ACKNOWLEDGMENTS

Vesa Norilo's work has been supported by the Emil Aaltonen foundation.

7. REFERENCES

[1] D. A. Bohn, "Operator adjustable equalizers: An overview," in *Proc. AES 6th Int. Conf.*, Nashville, TN, May 1988, pp. 369–381.

[2] M. Holters and U. Zölzer, "Graphic equalizer design using higher-order recursive filters," in *Proc. Int. Conf. Digital Audio Effects*, Montreal, Canada, Sept. 2006, pp. 37–40.

[3] J. Rämö and V. Välimäki, "Optimizing a high-order graphic equalizer for audio processing," *IEEE Signal Process. Lett.*, vol. 21, no. 3, pp. 301–305, Mar. 2014.

[4] J. Rämö, V. Välimäki, and B. Bank, "High-precision parallel graphic equalizer," *IEEE/ACM Trans. Audio Speech Language Processing*, vol. 22, no. 12, pp. 1894–1904, Dec. 2014.

[5] J. A. Belloch, B. Bank, L Savioja, A. Gonzalez, and V. Välimäki, "Multi-channel IIR filtering of audio signals using a GPU," in *Proc. IEEE Int. Conf. Acoust. Speech and Signal Processing (ICASSP-2014)*, Florence, Italy, May 2014, p. 6692–6696.

[6] S. Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*, Wiley, 2009.

[7] U. Zölzer (ed.), *DAFX: Digital Audio Effects*, Wiley, second edition, 2011.

[8] V. Välimäki, S. Bilbao, J. O. Smith, J. S. Abel, J. Pakarinen, and D. Berners, *DAFX: Digital Audio Effects*, chapter 'Virtual analog effects', pp. 473–522, U. Zölzer (ed.), Wiley, second edition, 2011.

[9] J. O. Smith, "A new approach to digital reverberation using closed waveguide networks," in *Proc. Int. Computer Music Conf.*, Vancouver, Canada, Sept. 1985, pp. 47–53.

[10] Y. Orlarey, D. Fober, and S. Letz, "Faust (programming language)," Available at `https://faust.grame.fr`, accessed Dec. 8, 2013.

[11] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[12] J. Kuper et al., "Cλash, from haskell to hardware," Available at `https://http://www.clash-lang.org/`, accessed June 8, 2015.

[13] J. Travis and J. Kring, *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (National Instruments Virtual Instrumentation Series)*, Prentice Hall PTR, 2006.

[14] A. Krukowski and I. Kale, "Simulink/Matlab-to-VHDL route for full-custom/FPGA rapid prototyping of DSP algorithms," in *Proc. Matlab DSP Conf. (DSP'99)*, 1999, pp. 1–10.

[15] V. Norilo, "Introducing Kronos—A novel approach to signal processing languages," in *Proc. Linux Audio Conf.*, Maynooth, Ireland, May 2011, pp. 9–16.

[16] C. Roads, *The Computer Music Tutorial*, MIT Press, Cambridge, MA, 1996.

[17] R. Boulanger, *The Csound Book*, MIT Press, 2000.

[18] M. Puckette, "Pure data: another integrated computer music environment," in *Proc. 1996 Int. Computer Music Conf.*, 1996, pp. 269–272.

[19] J. McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[20] B. Schottstaedt, "Machine tongues XVII: CLM: Music V meets Common Lisp," *Computer Music Journal*, pp. 30–37, 1994.

[21] G. L. Steele, *Common Lisp*, vol. 2, Digital Press, 1984.

[22] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the future: The story of Squeak, a practical Smalltalk written in itself," in *ACM SIGPLAN Notices*. ACM, 1997, vol. 32, pp. 318–326.

[23] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of Faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[24] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization (CGO 2004)*, 2004, pp. 75–86.

[25] V. Norilo, "Recent developments in the Kronos programming language," in *Proc. Int. Computer Music Conf.*, Perth, Australia, 2013.

[26] M. Verstraelen, G.J.M. Smit, and J. Kuper, "Declaratively programmable ultra low-latency audio effects processing on FPGA," in *Proc. 17th Int. Conf. Digital Audio Effects*, Erlangen, Germany, Sept. 2014, pp. 263–270.

[27] M. Verstraelen, F. Pfeifle, and R. Bader, "Feasibility analysis of real-time physical modeling using WaveCore processor technology on FPGA," in *Proc. 3rd Vienna Talk on Music Acoustics*, Vienna, Austria, 2015.

[28] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, 1989.

[29] P. A. Regalia and S. K. Mitra, "Tunable digital frequency response equalization filters," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 35, no. 1, pp. 118–120, Jan. 1987.