High-Performance Wideband SDR Channelizers

Islam Alyafawi[1], Arnaud Durand[2], and Torsten Braun[1(✉)]

[1] University of Bern, Bern, Switzerland
{alyafawi,braun}@inf.unibe.ch
[2] University of Fribourg, Fribourg, Switzerland
arnaud.durand@unifr.ch

Abstract. The essential process to analyze signals from multicarrier communication systems is to isolate independent communication channels using a channelizer. To implement a channelizer in software-defined radio systems, the Polyphase Filterbank (PFB) is commonly used. For real-time applications, the PFB has to process the digitized signal faster or equal to its sampling rate. Depending on the underlying hardware, PFB can run on a CPU, a Graphical Processing Unit (GPU), or even a Field-Programmable Gate Arrays (FPGA). CPUs and GPUs are more reconfigurable and scalable platforms than FPGAs. In this paper, we optimize an existing implementation of a CPU-based channelizer and implement a novel GPU-based channelizer. Our proposed solutions deliver an overall improvement of 30 % for the CPU optimization on Intel Core i7-4790 @ 3.60 GHz, and a 3.2-fold improvement for the GPU implementation on AMD R9 290, when compared to the original CPU-based implementation.

Keywords: SDR · CPU · GPU · Channelizer

1 Introduction

Different wired/wireless systems, such as the Global System for Mobile Communications (GSM), use frequency-division multiplexing (FDM) techniques [12]. In FDM-based systems, the signal is divided into multiple channels without cross-synchronization among themselves. In [3], we developed a single-band passive receiver that can capture, decode, and parse uplink messages of different GSM Mobile Devices (MDs). The proposed receiver is implemented as a GNURadio (GR) module with definite input and output interfaces. The output from multiple receivers (MD identity and received signal power) was sent to a centralized system to perform the localization process as described in [4]. However, since GSM MDs in a certain area may connect to multiple GSM Base Transceiver Stations (BTSs) operating at different frequencies, it is required to support the proposed Software-Defined Radio (SDR) receiver with wideband capturing capabilities. To do so, we use a channelizer, which splits the wideband spectrum into a set of independent channels fed to different instances of the single-band receiver as shown in Fig. 1.

L. Mamatas et al. (Eds.): WWIC 2016, LNCS 9674, pp. 3–14, 2016.
DOI: 10.1007/978-3-319-33936-8_1

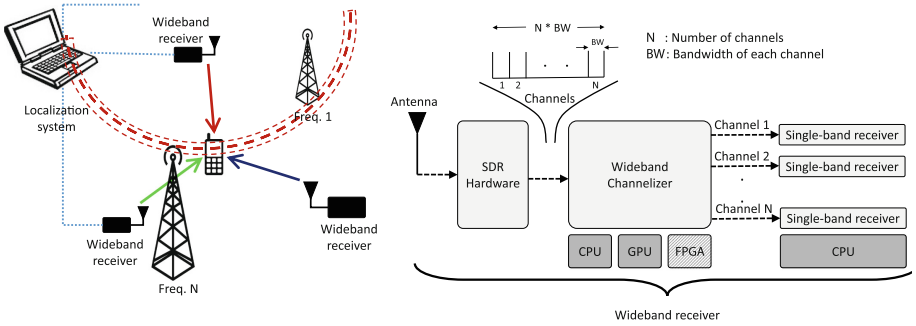

Fig. 1. Architecture of a wireless localization system using the wideband GSM receiver.

Polyphase Filterbanks (PFB) channelizers are powerful tools that (i) split the wideband signal into equispaced channels and (ii) filter the split channels with a given filter. The channelizer performance varies based on the underlying processing hardware architecture. While FPGA-based approaches can provide the required processing performance, they lack the run-time flexibility of GPP-based approaches [5]. The authors in [1,2,8,14] show the possibility to perform a PFB channelizer with a GPU underlying processing unit. However, these solutions are (i) not implemented as GR modules, (ii) not provided as an open-source and (iii) not optimized with respect to throughput and latency for data transfer between system memory (RAM) and GPU memory.

The contribution of this paper is optimizing an existing GR CPU-based channelizer using advanced CPU instructions and implementation of a new high-performance GPU-based channelizer suited for GR (c.f., Sects. 3 and 4). Our proposed solutions are provided as GR open-source modules [6] with optimized data transfer between RAM and GPU memory. An evaluation of the proposed solutions is presented in Sect. 5.

2 Problem Formulation

Our focus is to improve the PFB channelizer performance as a GR module. The channelized streams can then be sent to a centralized or a distributed system for signal processing and data acquisition. Our solution starts by understanding the signal processing and conversions between the antenna and the processing machine holding the channelizer.

2.1 SDR Hardware

Our proposed solution is hardware agnostic and, hence, it is possible to use any SDR equipment from any vendor that meets the minimum performance requirements. The Universal Software Radio Peripheral (USRP) is a series of SDR-enabled devices [19]. We use a set of USRP N210 devices (we call them

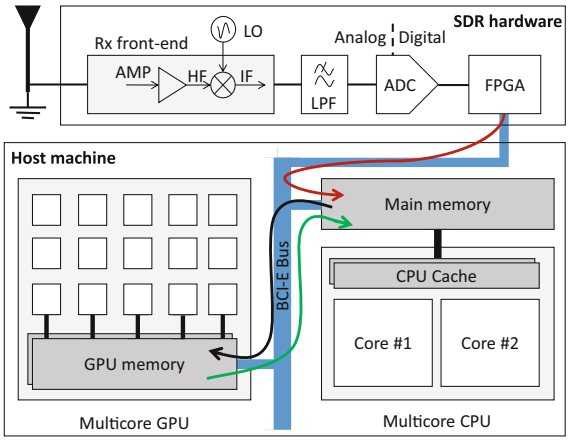

Fig. 2. SDR system architecture.

N210 for simplicity). A simplified architecture of the N210 is illustrated in Fig. 2. The received radio signal will be amplified before being converted from the High-Frequency (HF) to the Intermediate-Frequency (IF) band. Then, it will be filtered using a Low-Pass Filter (LPF) with a bandwidth up to 20 MHz. The filtered signal is digitized inside the Analog-to-Digital Converter (ADC), which has 14-bit precision for each In-phase and Quadratic (I/Q) samples. The ADC digitizes the signal with a fixed sampling rate equal to 100 Msps (Mega samples per second). The FPGA changes the data to 32-bit samples when configured in 16-bit mode (16 bit I and 16 bit Q). The UHD driver is used to control I/O communication between the USRP and the host computer [22]. It is configured to transfer up to 25 Msps in 16-bit mode over a Gigabit Ethernet (GbE) interface. The UHD driver converts sampled signals obtained from the USRP to IEEE 754 floating point numbers and store them in the host memory.

2.2 Wideband Channelizer

GR is an open-source framework to build SDR systems. Radio samples are typed *gr_complex* in GR, but are in fact only a redefinition of the C/C++ `std::complex` data type. Complex numbers are represented as two consecutive single-precision Floating-Point Units (FPUs), thus taking 64 bits. The default GR PFB channelizer makes use of the host CPU to process the entire bandwidth signals, which can take advantage of CPU-specific SIMD (Single Instruction Multiple Data) operations. Without any up-/down sampling operations, the overall output sample rate of a PFB channelizer is equal input sample rate. Nevertheless, the available processing resources will reach their limits with the increasing width of the channelized spectrum. Hence, there is a need for alternative optimized (or newly implemented) PFB solutions within the same processing platform [9].

GPUs are fine grain SIMD processing units suited for general-purpose processing with a relatively low cost compared to other specialized

architectures [18]. GPUs contain a large number of computing units. Hence, they can be used to implement a high-performance channelizer. There are several technologies available to realize GPUs, such as the CUDA programming model or the OpenCL framework [24]. However, to integrate such solutions with the GR framework, it is mandatory to implement the channelizer as a GR module. An essential requirement is the ability to transfer data between CPU (where GR works) and GPU (where the channelizer can be implemented) at high bandwidth and low latency. As shown in Fig. 2, existing communication methods that transfer data through CPU memory to the GPU might be used. Additional to the original data transfer between the FPGA and system memory (RAM), data must cross the PCI Express (PCIe) switch twice between the RAM and GPU memory.

2.3 Polyphase FilterBank Channelizer

A PFB channelizer is an efficient technique for splitting a signal into equally-spaced channels [12]. A PFB is mainly composed of two main modules: M FIR (Finite Impulse Response) filters (the filterbank) and an M-point FFT (Fast Fourier Transform). A basic implementation of the PFB channelizer is illustrated in Fig. 3, which represents the input with M FDM channels that exist in a single data stream and the resulting M Time Division Multiplexed (TDM) output channels. The fundamental theory of polyphase filters is the polyphase decomposition [16]. We discuss here the two principal components of the decomposition process.

FIR filters are digital filters with a finite impulse response. The transfer function of FIR filter samples is expressed in Eq. 1.

$$\mathrm{f}[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{1}$$

N is the number of FIR filter coefficients (known as filter taps), $x[k]$ are FIR filter input samples, $h[k]$ are FIR filter coefficients and $\mathrm{f}[n]$ are FIR filtered samples.

A **Fast Fourier Transform (FFT)** is a fast way to calculate the Discrete Fourier Transform (DFT) of a vector x. The FFT operation transforms the input signal from the time domain into the frequency domain. The M-point FFT transfer function is expressed in Eq. 2.

$$y[k] = \sum_{m=0}^{M-1} \mathrm{f}[m]e^{-2\pi jmk/M} \tag{2}$$

$\mathrm{f}[n]$ is the input and $y[k]$ is the output. The time it takes to evaluate an FFT on a computer depends mainly on the number of multiplications involved. FFT only needs $Mlog_2(M)$ multiplications.

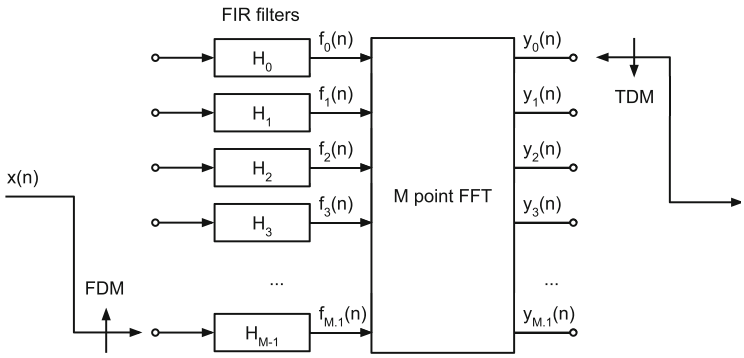

Fig. 3. PFB channelizer.

3 Enhanced CPU-Based PFB

GR takes advantage of architecture-specific operations through the Vector Optimized Library of Kernels (VOLK) machine [23]. The VOLK machine is a collection of GR libraries for arithmetic-intensive computations. Most of the provided VOLK implementations are using x86 SIMD instructions, which enable the same instruction to be performed on multiple input data and produce several results (vector output) in one step. The FFT operations are computed through the fftw3 library [10]. This open source library takes advantage of modern CPU architectures by making use of specific SIMD operations. As we consider this library to be optimal for CPU usage when used correctly, we will not consider further FFT optimizations. FIR filters are accumulators that perform a lot of multiply-accumulate operations. Thus, the PFB channelizer relies heavily on the VOLK multiply-and-add routine and most of the CPU cycles are spent inside this function [20]. The fastest multiply-and-add routine available in the VOLK machine for our CPU (Intel Haswell) is `volk_32fc_32f_dot_prod_32fc_a_avx` using the Advanced Vector Extensions (AVX) instruction set [15]. In the following, we present a set of optimizations by the available AVX PFB implementation.

Memory Alignment: CPUs access memory in chunks. Depending on the memory access granularity, a CPU may retrieve data in 2, 4, 8, 16, 32-bytes chunks or even more. Depending on the particular instruction, it may or may not be allowed to load data from memory into registers using addresses not being multiple of the memory access granularity [7]. Indeed, if allowed, loading unaligned data requires more operations as the register must execute shifting and merging operations of different memory chunks. AVX are extensions to the x86 instruction set. They allow operations on 256-bit registers, enabling eight single-precision floating point numbers to be stored side-by-side in the same register and processed simultaneously. AVX uses relaxed memory alignment requirements. It means that using unaligned data is allowed for most instructions, but this comes with a performance penalty. When allocating memory using `malloc()`, compilers are in charge of the alignment, and we cannot assume that the returned

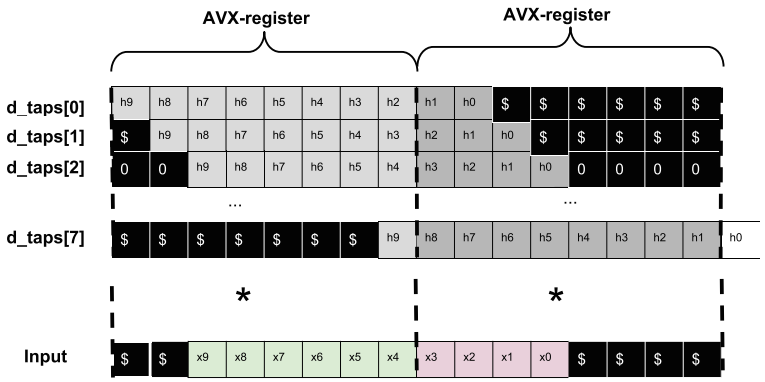

Fig. 4. GR FIR taps alignment and AVX dot product.

address is aligned. However, it is possible to force alignment when desirable. GR provides such facility through the `volk_malloc()` function [11]. Such a function allocates a slightly bigger amount of memory (desired size + granularity - 1) and moves the pointer to the next aligned address. However, the FIR filters from the PFB channelizer cannot use this function because GR buffers are not necessarily aligned. Instead, GR generates a set of aligned taps d_taps for each possible architecture alignment as shown in Fig. 4. In the illustrated example of Fig. 4, we have a FIR filter with a history of 10 samples $h[0]$, ..., $h[9]$. By measuring the alignment of the input samples, we can select the correctly aligned taps. For example, if the input register is aligned with two floats, GR will choose to multiply with d_taps[2] (a set of taps aligned with two floats). $ indicates unaligned data and $x[0]$, ..., $x[9]$ are the input samples. GR performs SIMD operations on aligned data and non-SIMD operations on the remaining input samples. To avoid non-SIMD operations, one solution is to make the number of filter taps as a multiple of 8. Then, we can fill taps corresponding to unaligned data with zeros as shown in Fig. 4 (**d_taps[2]**). We then consider the data aligned in GR buffers because zeros will discard unaligned input values.

Fused Multiply-Add (FMA) Operations: FMA operations are AVX2 instructions to 128 and 256-bit Streaming SIMD Extensions (SSE) instructions [13] (AVX2 is an expansion of the AVX instruction set). AVX2 FMA operations are floating point operations similar to c ← (a * b) + c, which performs multiply and add in one step (c.f. Fig. 5). These operations are compatible with the GNU compiler collection. However, AVX performs (a * b) + c in two steps (first a * b, then + c). Benchmarks of pure multiply and accumulate operations on Intel Haswell CPUs show double performance using AVX2 FMA compared to AVX. The PFB implementation uses the product of complex and float vectors. This implementation requires some further operations like deinterleaving an input block of complex samples into M outputs.

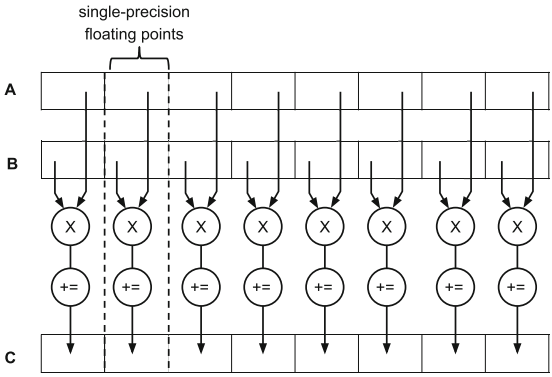

Fig. 5. FMA SIMD operation.

Better Pipelining: complex numbers are just two consecutive floats as defined in `complex.h` of the C standard library. From the previous example c ← c + (a * b), a is a complex number representing an input sample and b is an integer representing a FIR filter coefficient. Consider that a is a tuple of (r, qi): r is the in-phase part of a, q is the quadrature part of a, and i is the complex number $\sqrt{-1}$. Complex arithmetic rules define a*b = (r + qi) * b = (r + qi) * (b + 0i) = (rb + qbi). As the GR FIR filters are not specifically designed for AVX operations, the coefficients are just stored as consecutive floats. Unfortunately, feeding 256-bit registers with consecutive floats is a costly process because it requires a for-loop operating over all input sample to perform the (r + qi) * b instructions. The best we can do is to create a more compatible layout of filter taps to benefit from AVX2-FMA operations. To do this we load (`_mm256_loadu_ps`), unpack (`_mm256_unpackXY_ps`) and then permute (`_mm256_permute2f128`) FIR coefficients before operations. The behavior of unpack does not allow to permute coefficients across 128-bit boundaries. This behavior is inherited directly from the legacy SSE unpack operation. However, by re-shuffling the taps during the creation of the PFB as illustrated in Fig. 6, we can create a layout that enables the taps to be in the right order when unpacked (without permutation). Then, we can perform four (r + qi) * b instructions in one step.

4 GPU-Based PFB

We can make PFB processing faster by taking advantage of massively parallel architectures, such as GPUs [18]. There are several technologies available to perform GPU processing. OpenCL is an open framework for heterogeneous computing [21]. It is open by design and, therefore, best suited to the philosophy of GR. For this reason, we will implement a PFB channelizer using OpenCL. In our test setup, we use a discrete graphics card, meaning that the CPU and GPU do not share the same memory (c.f. Fig. 2). There are several techniques to transfer data between the CPU and GPU, with various transfer speeds [17]. However, when using physically shared memory between the GPU and CPU using

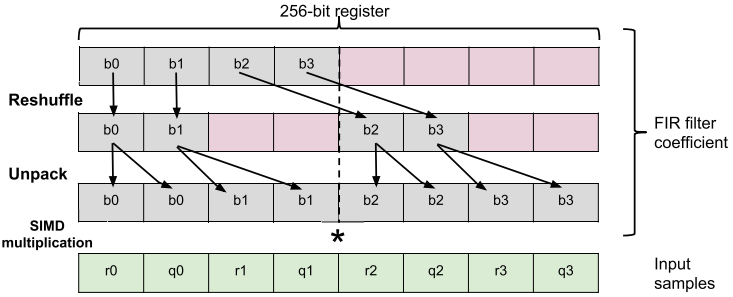

Fig. 6. AVX unpack operation and multiplications.

fused architectures, e.g., using Intel Integrated Graphics Processors (IGP), we are not required to transfer data. Data could be copied using the CPU or GPU direct memory access (DMA). However, IGPs are generally less powerful than discrete GPUs and thus, not used in our research. To process data in real-time, we have strict requirements for transfer speeds. Since GR has no built-in support for coprocessors, buffers could only be transferred to/from the GPU inside the `work()` function (where we want our code to be implemented). Every time `work()` is called by the GR scheduler, we should (i) transfer a portion of input samples (decided by the GR scheduler) from the main memory to the GPU, (ii) perform the PFB channelizer on the GPU and (iii) transfer the channelizer output from the GPU to the main memory. This situation is not optimal as the (i) and (iii) lead to a low throughput and high latency data transmission. For real-time processing, data transfer between CPU and GPU is a time critical action. The time spent transferring data is time lost for processing. It is important to obtain a balance between transfer delay and processing delay. As a solution to this situation, we proposed the following: First, to avoid inefficient data transfer to a single stream, we perform the FDM operation of Fig. 3 inside `work()`. This is simply done using the `stream_to_streams` (s2ss) function, which converts a stream of M items into M streams of 1 item. Hence, we get the inputs of the FIR filters on different streams with different buffers and transfer data more efficiently. Second, to avoid inefficient data transfer (small amount of data with high transferring latency), we batch the s2ss transferring buffers using `clEnqueueWriteBuffer()` with non-blocking write transfer. With the non-blocking write transfer, control is returned immediately to the host thread, allowing operations to occur concurrently in the host thread during the transfer between host and device proceeds. On the GPU device, as illustrated in Fig. 7, our filter taps are copied to the GPU memory once at the startup of our PFB channelizer. Each GPU FIR filter (inside the PFB channelizer) runs in parallel inside an independent GPU computing unit. Once we get the output from the FIR filters, we pass the output of the FIR filters to the input sequencer such that we can perform the M-point FFT. There are several OpenCL FFT libraries available like AMD clFFT (an open source library). clFFT usage is

straightforward. Here, we take advantage of the functions `bakePlan()` to apply device optimizations and `setPlanBatchSize()` to handle several clFFT operations concurrently.

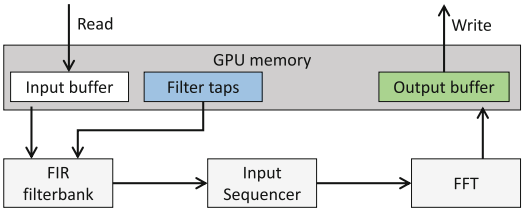

Fig. 7. PFB Channelizer on GPU.

5 Experimental Results

In this section, we consider the GR CPU-based implementation of the PFB channelizer as a reference. We are using GR version 3.7.5. All experiments are performed on a Linux-based machine with Intel Core i7-4790 @ 3.60 GHz CPU, AMD R9 290 (Tahiti) GPU (4 GB video memory), and 32 GB RAM. We use a simple benchmark experiment illustrated in Fig. 8. The signal source is a GR block used to generate continuously a certain type of a signal, such as sine, cosine or constant. In this experiment, we used *gr_complex* defined in GR as an output to emulate the behavior of a USRP N210. The head block stops the signal source when N items of the gr_complex type are processed. Hence, we can configure experiments' duration through N. The s2ss block performs the FDM operation in Fig. 3. We used three implementations for the PFB channelizer: (i) original GR with AVX instructions, (ii) our optimized GR implementation with AVX2-FMA instructions, and (iii) our GPU implementation. The channelized streams are sent to a null sink because our target is to evaluate the performance of the PFB channelizer only.

5.1 A Running Example

To measure the performance of our solutions proposed in Sects. 3 and 4, we will use a running example based on our passive GSM receiver [3]. Each GSM receiver requires a precise sample rate R for each GSM channel: $R = 1625000/6 = 270833$ complex samples per second (sps), which is equivalent to the data rate of the GSM signal itself. From these numbers, we can calculate a few elements for a wideband GSM receiver:

- We need a wideband sampling rate $R_b = R * M_{\text{ch}}$, where M_{ch} is the number of GSM channels.
- We can capture up to $M_{\text{ch}} = 25 \times 10^6/R = 92$ channels using a N210 USRP device with 16 bit precision.

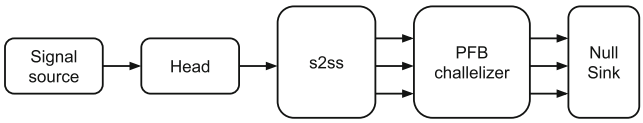

Fig. 8. PFB channelizer benchmark overview.

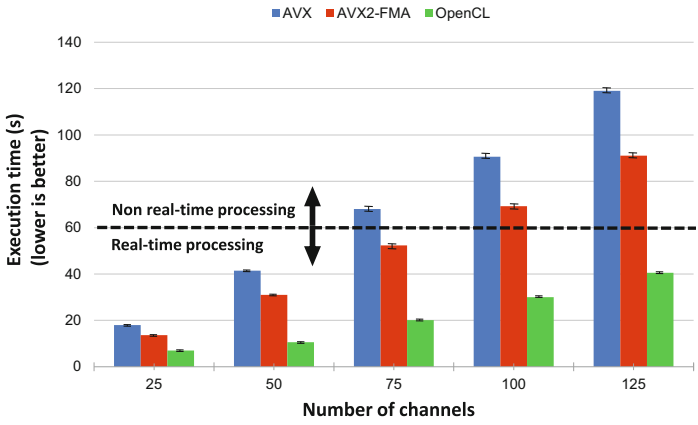

Fig. 9. Real time processing of PFB channelizer.

5.2 AVX vs AVX2-FMA vs GPU

In these experiments, we want to measure the improvement provided by the optimized AVX2-FMA PFB (presented in Sect. 3) and the proposed GPU-based PFB (shown in Sect. 4) compared to the default GR AVX implementation. We use a pre-designed filter of 55 taps, and we want to test the performance for different numbers of channels. The channelizer performance is agnostic to the data content, i.e., feeding a generated signal to the channelizer is the same as feeding a USRP input stream. For each number of channels shown in Fig. 9, we generate a signal equivalent to 60 s of data. For example, in experiments with 50 channels, we use a signal source block (a producer of data) with sampling rate $R_b = 50R$ and a number of samples $X = 60R_b$. The benchmark output is the total runtime, and the presented results are the arithmetic mean of 5 runs. If the processing time exceeds the 60 s, it means that the PFB channelizer cannot process data in real-time. As illustrated in Fig. 9, the improvement using our optimized AVX2-FMA ranges from 30 % to 33 %. Our optimized filter can process in real-time up to 76 channels. It is, however still, lower than what a USRP N210 device can support (c.f., Sect. 5.1: 92 channels). Compared to the reference channelizer, there is an average of 3.2-fold improvement by the GPU-based PFB implementation. That means we can channelize more than 125 GSM channels, more than what the USRP N210 can capture.

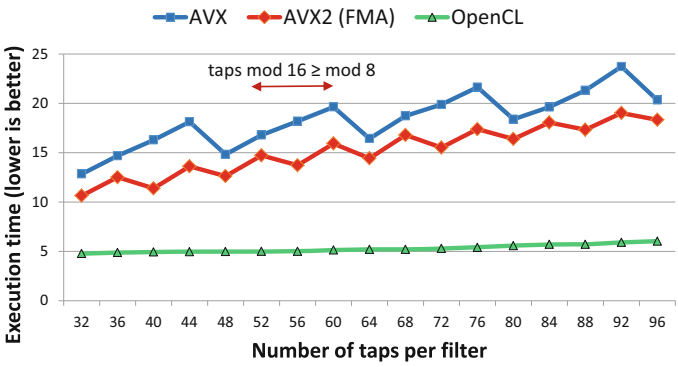

Fig. 10. Taps impact on PFB channelizer.

5.3 Taps per Filter

In this experiment, we run the same benchmark as in Sect. 5.2 with a constant number of channels equal to 25, but varying the number of taps per FIR filter (channel). Again, the results present the arithmetic mean of 5 runs. As illustrated in Fig. 10, the execution time increases with the increasing number of FIR taps. The different zig-zag behavior for AVX and AVX2 implementations is because AVX2 registers have the double size of AVX registers. Extra samples in both implementations are processed using standard non-SIMD operations. This behavior is not noticed in the OpenCL implementation due to the different mechanisms for data transfer from/to GPU. Note that it is important to use a multiple of 2^N so that the compiler can optimize the costly division operation to a bit shifting operation.

6 Conclusions

GR is the framework of choice for many SDR projects. We demonstrated that it was possible to create high-performance GSM channel processing by reusable components for wideband applications. The channelizer technique is applicable for any distributed system with a set of independent channels. The polyphase filterbank channelizer is not only useful for FDM-based technologies but is a critical component in many SDR systems. Our AVX2 optimizations improve the performance by up to 30 % using recent CPU instructions, namely fused multiply-add operations, compared to the AVX implementation. Moreover, the GPU implementation opens up opportunities for applications, where expensive specialized hardware was required previously. The GPU-based implementation is a GR-like module with 3.2-fold improvement when compared to the original GR implementation on CPU only. Our proposed solutions are available as independent GR modules [6].

References

1. Adhinarayanan, V., Feng, W.C.: Wideband channelization for software-defined radio via mobile graphics processors. In: International Conference on Parallel and Distributed Systems (ICPADS) (2013)
2. Adhinarayanan, V., Koehn, T., Kepa, K., Feng, W.C., Athanas, P.: On the performance and energy efficiency of FPGAs and GPUs for polyphase channelization. In: International Conference on ReConFigurable Computing and FPGAs (2014)
3. Alyafawi, I., Dimitrova, D., Braun, T.: Real-time passive capturing of the GSM radio. In: ICC (2014)
4. Alyafawi, I., Schiller, E., Braun, T., Dimitrova, D., Gomes, A., Nikaein, N.: Critical issues of centralized and cloudified LTE-FDD radio access networks. In: ICC (2015)
5. Awan, M., Koch, P., Dick, C., Harris, F.: FPGA implementation analysis of polyphase channelizer performing sample rate change required for both matched filtering and channel frequency spacing. In: Asilomar Conference on Signals, Systems, and Computers (2010)
6. Channelizer. http://cds.unibe.ch/research/hp_sdr_channelizer.html
7. Intel Corperation: Intel C++ compiler intrinsics reference (2006)
8. del Mundo, C., Adhinarayanan, V., Feng, W.C.: Accelerating fast fourier transform for wideband channelization. In: ICC (2013)
9. Durand, A. (2015). http://cds.unibe.ch/research/pub_files/internal/du15.pdf
10. FFTW. www.fftw.org
11. GNURadio. https://gnuradio.org
12. Harris, F.J.: Multirate Signal Processing for Communication Systems. Prentice Hall PTR, Upper Saddle River (2004)
13. Jain, T., Agrawal, T.: The haswell microarchitecture - 4th generation processor. IJCSIT **4**(3), 477–480 (2013)
14. Kim, S.C., Bhattacharyya, S.S.: Implementation of a high-throughput low-latency polyphase channelizer on GPUs. EURASIP J. Adv. Signal Process. **1**, 1–10 (2014)
15. Lomont, C.: Introduction to intel advanced vector extensions. Intel Corperation (2011)
16. Madisetti, V.K.: The Digital Signal Processing Handbook, 2nd edn. CRC Press, Boca Raton (2009)
17. Munshi, A.: The opencl specification, version 1.1. Khronos OpenCL Working Group (2010)
18. Plishker, W., Zaki, G.F., Bhattacharyya, S.S., Clancy, C., Kuykendall, J.: Applying graphics processor acceleration in a software defined radio prototyping environment. In: International Symposium on Rapid System Prototyping (2011)
19. Ettus Research. http://www.ettus.com/
20. Rondeau, T.W., Shelburne, V.T., O'Shea, V.T.: Designing anaylsis and synthesis filterbanks in GNU radio. In: Karlsruhe Workshop on Software Radios (2014)
21. Tsuchiyama, R., Nakamura, T., Iizuka, T., Asahara, A., Son, J., Miki, S.: The OpenCL Programming Book. Fixstars, Tokyo (2010)
22. UHD. http://code.ettus.com/redmine/ettus/projects/uhd/wiki
23. VOLK. https://gnuradio.org/redmine/projects/gnuradio/wiki/volk
24. Wilt, N.: The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley Professional, Boston (2012)