Wideband Channelization for Software-Defined Radio via Mobile Graphics Processors

Vignesh Adhinarayanan† and Wu-chun Feng*†
†Department of Computer Science and *Department of Electrical & Computer Engineering
NSF Center for High-Performance Reconfigurable Computing
Virginia Tech, Blacksburg, Virginia, U.S.A.
{avignesh, wfeng}@vt.edu

***Abstract*—Wideband channelization is a computationally intensive task within software-defined radio (SDR). To support this task, the underlying hardware should provide high performance and allow flexible implementations. Traditional solutions use field-programmable gate arrays (FPGAs) to satisfy these requirements. While FPGAs allow for flexible implementations, realizing a FPGA implementation is a difficult and time-consuming process. On the other hand, multicore processors while more programmable, fail to satisfy performance requirements. Graphics processing units (GPUs) overcome the above limitations. However, traditional GPUs are power-hungry and can consume as much as 350 watts, making them ill-suited for many SDR environments, particularly those that are battery-powered.**

Here we explore the viability of low-power mobile graphics processors to simultaneously overcome the limitations of performance, flexibility, and power. Via execution profiling and performance analysis, we identify major bottlenecks in mapping the wideband channelization algorithm onto these devices and adopt several optimization techniques to achieve multiplicative speed-up over a multithreaded implementation. Overall, our approach delivers a speedup of up to 43-fold on the discrete AMD Radeon HD 6470M GPU and 27-fold on the integrated AMD Radeon HD 6480G GPU, when compared to a vectorized and multithreaded version running on the AMD A4-3300M CPU.

***Index Terms*—polyphase filter banks; mobile GPU; wideband channelization; software-defined radio**

I. Introduction

The wideband channelizer, a critical component within software-defined radio (SDR), extracts individual communication channels from a digitized wideband spectrum. This is useful in many electronic warfare applications, where wideband spread spectrum signals are demodulated in real-time, e.g., SDRs such as communication intelligence (COMINT) and signal intelligence (SIGINT) [1]. Fig. 1 shows an example of a wideband channelizer used in a SDR.

To perform real-time wideband channelization for such SDR applications, the underlying hardware should be capable of providing high performance under strict power budgets without sacrificing on programmability [2]–[5]. The performance demands for wideband channelization arise from its proximity to the analog-to-digital converter, which forces the channelizer to operate at the highest sampling rate among all stages in a SDR pipeline. The channelizer becomes a bottleneck stage as it processes the most data. To overcome this bottleneck, the hardware should provide sufficiently high performance. The power requirements arise from a deployment perspective. With military applications oftentimes being deployed on battery-powered mobile terminals, the hardware should consume minimal power to reduce battery-drain. The strict power constraints, combined with high performance requirements, have led researchers to dub such architectures as *mobile supercomputers*, meaning these devices should be capable of providing supercomputer-like performances at the power budget of a mobile phone processor [5].

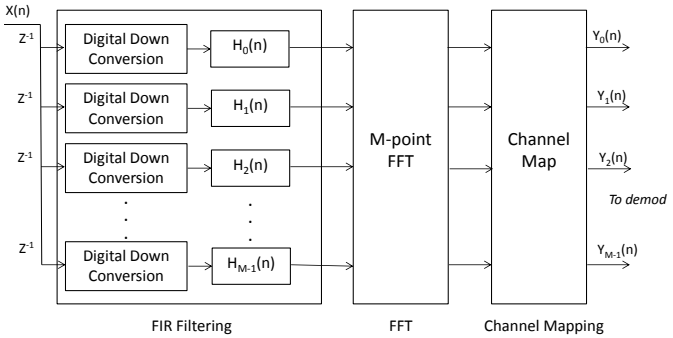


Fig. 1. **Polyphase Filter Bank (PFB) channelizer.** The input RF band signal x(n) is divided into M equally spaced channels. This implementation, based on GNU Radio, is divided into three stages - FIR Filtering, FFT and Channel Mapping.

A software-defined radio (SDR) should also adapt and reconfigure itself to different standards, networks, and bandwidth in which it might operate. While static reconfiguration at the domain site is an option, many applications, particularly in the military domain, need the radio to adapt to different standards instantaneously and in real time. Apart from the three requirements of performance, flexibility, and power, the portability of such applications to different hardware is also highly desired, as highlighted in the study by Ulversøy [2].

The various architectural options have their relative merits and demerits in performing the compute-intensive portions of SDR, such as wideband channelization. While ASICs have previously been used to tackle the performance requirements of wideband channelization, they lack the flexibility necessary for SDRs. In contrast, the more flexible DSPs and and general-purpose multicore CPUs suffer from performance

limitations and are generally incapable of performing real-time channelization. Though FPGAs can provide the necessary performance, realizing a FPGA implementation can be difficult and time consuming. Furthermore, FPGAs are not particularly good at real-time adaptation to different standards as FPGA reconfiguration is not instantaneous. Considering the above, general-purpose graphics processing units (GPGPUs) seem to be an appropriate choice to perform this task. Recent studies have, in fact, shown the viability of accelerating wideband channelization using discrete graphics processing units [6]. However, their deployment in the field, e.g., on battery-powered mobile terminals, presents challenges due to their high power consumption.

In this scenario, low-power mobile GPUs provide a potential alternative to perform this task. These devices form a subclass of graphics processors that consumes only a fraction of the power consumed by traditional desktop and server GPUs. This class of devices include emerging chips that integrate the CPU and GPU onto a single die such as the Qualcomm Snapdragon S4 and AMD APU (accelerated processing unit). The tradeoff is that they are not as computationally capable as their high-powered GPU brethren. Thus, if the performance needs of SDR are to be met, understanding how to make best use of these hardware resources is of paramount importance.

In this work, we explore the viability of using mobile graphics processors to accelerate polyphase filter bank (PFB) channelization, a highly efficient method for wideband channelization [7]. We investigate the application of esoteric application-specific and device-specific optimizations as a way to bridge the performance gap. Our implementations are based in OpenCL in order to ensure functional portability across hardware. Our key contributions include:

- An efficient mapping of the performance-critical stages of the PFB channelizer on GPUs.
- The realization of optimization techniques to improve the performance on such devices [8]–[10] and reduce associated data-transfer overhead [11].
- An evaluation of our implementation on mobile GPUs, including, the first such evaluation on integrated CPU+GPU devices.

The results demonstrate that our proposed mapping and optimization onto a GPU delivers a speedup of up to 43-fold for the discrete mobile GPU (AMD Radeon HD 6470M) and 27-fold for the integrated mobile GPU (AMD Radeon HD 6480G), when compared to an optimized multithreaded CPU implementation running on the CPU of an AMD A4-3300M.

The rest of the paper is organized as follows. Section II covers background material on PFB channelization and GPU architecture. Section III discusses related work. We present our approach and its associated mapping and optimization in Sections IV and V, respectively. Next, we present our results in Section VI and conclude in Section VII.

II. Background

Many techniques have been explored to extract individual channels from a wideband signal. Among these techniques, polyphase filter bank (PFB) channelization is very popular due to its high efficiency and low complexity [7], and therefore, it is the target channelizer for our study. Specifically, our implementation is based on the serial implementation of the PFB channelizer from GNU Radio [12], the architecture and mechanism of which is described by Harris [13]. We briefly discuss the three stages within the PFB channelizer below:

1) *FIR Filtering.* The finite impulse response (FIR) filter is a digital filter commonly used in signal processing applications. In the wideband channelizer, a set of *N* filters is used to isolate and decimate the channels. To perform this task, a series of multiply-accumulate (MAC) operations take place on the input signal (of length T+1) and the filter coefficients (also known as taps). The output of an FIR filter can be mathematically written as

$$y(n) = \sum_{i=0}^{T} t_i x(n-i) \tag{1}$$

$y(n)$ is the output signal at discrete time n.
$x(n)$ is the input signal at discrete time n.
t_i is the filter tap/coefficient.
$T+1$ is the total number of taps in the filter.

2) *FFT.* The fast Fourier transform (FFT) is a computationally efficient way to compute discrete Fourier transform (DFT). This operation transforms the input signal from the time domain to the frequency domain. In this application, this transformation converts each channel into baseband. This stage has the highest time complexity ($O(N\ log_2 N)$) among all stages within the PFB channelizer. Mathematically, the output of an FFT operation can be written as

$$X_k = \sum_{n=0}^{N-1} x(n) e^{-i2\pi k \frac{n}{N}} \tag{2}$$

$x(n)$ is the input to this stage.
X_k is the output from the stage.

3) *Channel Mapping.* In this stage, the output of the polyphase filter is mapped to specific output channel numbers. This allows for channels to be selectively choosen for further baseband processing. If the mapping function is specified by means of a map vector m, then mathematically, the output of this stage can be represented as

$$y(n) = x(m_n) \tag{3}$$

$x(n)$ is the input to the stage.
$y(n)$ is the mapped output.

A. OpenCL Programming Model

OpenCL is an open-standard framework for writing programs on heterogeneous platforms. A program written in OpenCL can execute on a variety of devices such as CPUs, GPUs, APUs, DSPs, and other processors. OpenCL programs are divided into two parts. One part is written in C/C++ and executes on the host (CPU). This part launches jobs onto the GPU and manages the GPU. The other part, known as a kernel, is where the GPU-parallelized portion of the program is executed. A kernel specifies the work done by a single GPU. In AMD's terminology, a thread is referred to as a work-item. A group of work-item makes up a work-group. A work-group exists to allow interaction between work-items within it.

To understand how these work-items are mapped to the GPU, we briefly describe GPU architecture in AMD terminology. A GPU compute device is made up of a collection of compute units (CU). Each compute unit consists of an array of processing elements (PE), and each PE consists of one or more ALU. Each work-item executes on these processing elements. In AMD GPUs, a collection of 64 work-items make up a wavefront. A wavefront is mapped onto a compute unit and the threads within a wavefront operate in a lock-step fashion. There should be at least as many wavefronts as there are compute units to keep all the compute units within the GPU device busy. A compute unit may also have more than one wavefront mapped to it.

To make best use of a GPU, we should exploit its memory hierarchy in addition to keeping all the compute units busy. The GPU's memory hierarchy consists of five different types of memory: private, local, constant, image, and global. Of these, the global memory is off-chip and has a very high-access latency. It can be used by all work-items. The image memory is a special mode of global memory that uses the hardware cache and is used to store images with a unique access pattern. The other memories are on-chip and have a comparatively lower latency. Private memory is allocated in registers and is private to a work-item. It has the fastest access latency among all memory types. Constant memory has slightly higher latency and is used for fast lookup of data. Local memory is accessible to all work-items within a work-group and can be used for communication between work-items within a work-group. It has better access latency than global memory. As on a CPU architecture, an OpenCL application should leverage the GPU memory hierarchy to achieve good performance.

III. Related Work

Many studies have sought to accelerate the compute-intensive wideband channelization. Savir [1] discusses a scalable FPGA implementation of polyphase filter bank channelizer. Fahmy and Doyle [14] present another reconfigurable implementation of the PFB channelizer for spectrum sensing, which only scales up to 1024 channels (for 57 taps) due to block RAM-size limitations. Another 8192-channel, 4-tap FPGA implementation by Monroe et al. [15] suffers from the same limitation. In contrast, our GPU implementation scales well beyond these configurations.

In recent times, researchers have explored accelerators other than the FPGAs to accelerate wideband channelization. Hamilton [16] used the Cell Broadband Engine to accelerate polyphase channelization and obtained a six-fold speedup over a serial CPU implementation. Alas, the achieved performance of 200 MFLOPS is insufficient for our needs and the target SDR application, which requires tens to hundreds of GFLOPS in order to process thousands of channels.

SIMD-based DSP processors have been shown to be suitable for SDR applications. Examples of such architectures include SODA [17], [18] and AnySP [19]. Performance evaluation on such architectures have highlighted the ability of SIMD-based processors to provide the high performance necessary for SDR applications at a reduced power budget. Similar benefits can be expected from the SIMD-based GPU architectures, which can also simultaneously allow programming with high-level languages, thereby delivering better programmability.

Several researchers have studied the GPU acceleration of various stages of SDR pipeline, including wideband channelization. van der Veldt et al. [6], for example, investigated the acceleration of a PFB channelizer using high-powered GPUs and multicore processors for radio astronomy applications. They reported speed-up results for up to 1024 channels. To obtain performance improvements, they applied a limited set of optimizations, in particular, data transfer optimization, where they used page-locked, write-combined, mapped buffers.

Our work goes significantly farther, including a wider range of optimization techniques. First, to tackle the I/O bottleneck, we adopt a *data streaming* approach [11] as we observed better performance for this approach from our experimental results. Second, our PFB channelizer and associated optimizations target the low-power mobile graphics processors, as oftentimes military applications require deployment on battery-powered terminals. Consequently, we focus on such architectures for our evaluation, including a new class of architecture called the accelerated processing unit (APU), where the CPU and GPU are integrated onto a single die.

Apart from the above studies, other studies have been conducted to accelerate individual stages of the channelizer independently. Govindaraju et al. present a study on accelerating FFT on NVIDIA GPUs [20]. del Mundo et al. present a detailed study on optimizing the FFT stage of polyphase channelization for different classes of AMD GPUs [21]. Accelerating the FIR filtering stage has been the focus of several studies [22], [23]. In this work, we design and optimize our GPU implementation of the entire PFB channelizer — FIR filter, FFT, and channel mapper.

IV. Algorithm Mapping

In this section, we describe our approach in mapping the PFB channelizer onto the GPU.

A. Profiling the Serial Implementation of a PFB Channelizer

We profile the serial implementation of a PFB channelizer from GNURadio to identify its performance critical stages. The choice of parameters for the channelizer reflect the typical requirements of the target defense applications, viz. thousands of channels and a dozen of taps per channel. Fig. 2 shows the breakdown of execution time for the three stages (FIR filtering, FFT and channel mapping) when running on the CPU portion of the AMD A4-3300M. We observe that the FIR filtering stage dominates the overall execution time in spite of its modest $O(N)$ time complexity. Therefore, achieving high performance for this stage is critical.

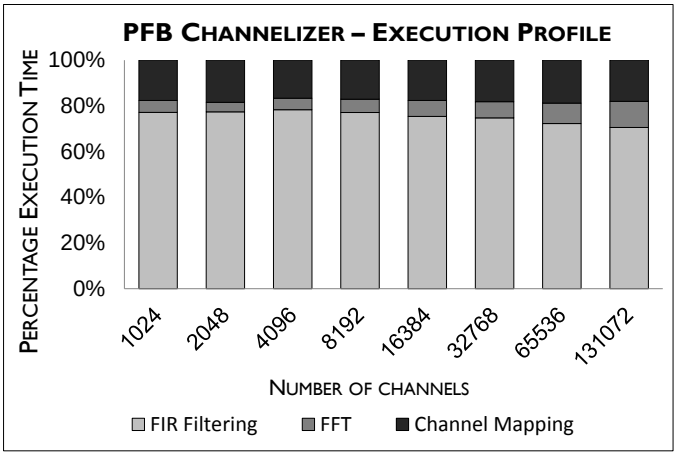


Fig. 2. **Time profile of PFB channelizer.** Percentage of total execution time spent in each stage for different problem sizes. FIR filtering stage dominates the overall execution time.

It is important to note that in order to achieve multifold speedup over the CPU implementation, not only do we need to accelerate the FIR filtering stage, but we must also accelerate the other stages so that they do not become performance bottlenecks. In short, we accelerate all three stages and apply relevant optimization techniques. However, a greater emphasis is laid on the bottleneck stage of FIR filtering.

B. FIR Filtering

GPUs have high memory-access latency to global memory. In order to overcome such high latency, many threads are executed simultaneously so that when one set of threads (referred to as wavefront) waits for data, another set can take its place. Therefore, the FIR filtering stage must run many threads on GPU to hide the high memory-access latency. A simple mapping scheme, where the *N* FIR filters operating in parallel, are mapped to *N* GPU threads does not suffice. To obtain the necessary degree of parallelism, there are two options: (1) exploiting hierarchical parallelism and (2) batching multiple iterations together. In a hierarchical implementation, we can parallelize the dot-product operation for each filter, apart from obtaining parallelism via the *N* filters operating in parallel. However, the reduction step in the dot-product will invariably require atomic operations, which are known to perform poorly in GPU architectures [24]. If multiple iterations are batched together, we can achieve the required parallelism without paying any performance penalty, and hence, this is the approach we chose.

C. FFT

To accelerate the FFT stage, we use AMD's highly optimized FFT library [25]. We make use of the batching support offered by the FFT library to keep the GPU cores busy. Input to and output from this stage are kept device-resident to minimize data-transfer overheads. It is critical to have the intermediate stages be device-resident. In fact, the previous work on accelerating polyphase channelization by van der Veldt et al. reported a nine-fold performance degradation from data transfers [6]. In contrast, when we ensure that the intermediate stages are device-resident and couple it with data streaming, the data transfer overhead goes to virtually zero.

D. Channel Mapping

The channel mapping stage is a memory-intensive operation. In our implementation, a single map operation is performed by a GPU thread so that many threads can run in parallel to make the best use of the bandwidth offered by global memory. The total number of memory transactions depends on the order of elements in the input map vector. Therefore, we rearrange the elements in the map vector, which is a one-time cost, in order to minimize the total number of memory transactions.

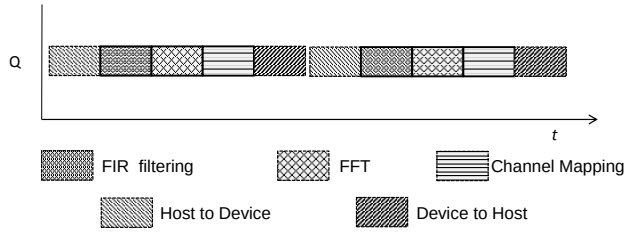
V. Performance Optimization

This section describes the various computation and communication optimizations that we propose and realize in our implementation.

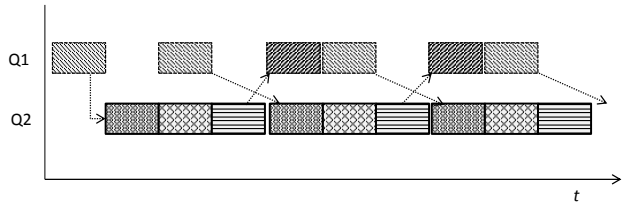
A. Reducing Data-Transfer Overhead

The overhead from data transfer across PCI Express is a significant problem for signal-processing applications like wideband channelization due to the large input sizes associated with such applications. Though the CPU and GPU are integrated on a single APU die, the data still has to pass across PCI Express for the GPU to manipulate it, and therefore, this problem exists in these devices as well [26]. To overcome this problem, techniques such as data streaming and data compression have been developed [11]. For our application, the data compression technique is not applicable so we focus only on the data streaming approach. In this technique, a portion of the data is first transferred to the GPU. As the GPU operates on this partial data, the rest of the data is transferred to the GPU, thereby overlapping computation and communication.

Our approach *eliminates* the double-buffering[1] overhead associated with the data streaming approach. This is important in mobile GPUs, where all resources including memory, are limited. Fig. 3 shows how we avoid using two buffers for the input data. The FIR filtering kernel takes the data in the input buffer, operates upon it, and places the output in an intermediate buffer. Since the GPU can perform some other task, namely FFT, using the intermediate buffer, the input buffer becomes available for data transfer. When the GPU executes the FFT stage by operating upon the intermediate buffer, the next segment of data is transferred to the original input buffer. Similarly, the device-to-host transfer takes place after the channel mapping stage finishes and the FIR filtering stage of the next iteration begins. Through this technique, we can hide nearly all of the data transfer overhead without having to allocate two sets of buffers for the input or output.



(a) Using a single queue for data transfer and kernel execution results in unnecessary serialization.



(b) Using two separate command queues for kernel execution and data transfer allows overlap of computation and communication. Dependencies across stages shown using arrows.

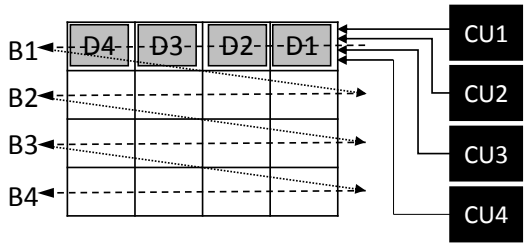
Fig. 3. **Data Streaming.** Reduces data transfer overhead by overlapping computation and communication.

B. GPU Optimizations

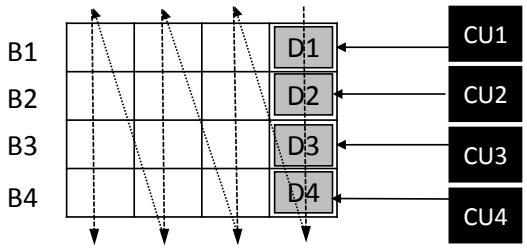
1) Increasing GPU occupancy by batching: Running many threads is necessary to hide the memory latency of GPUs and to keep the GPU cores busy. We do so by batching many iterations together. However, if the register usage per thread is very high, GPU occupancy comes down due to the limited register file size. Only so many threads can be kept on flight as can be supported by the register file. In addition, high register usage can lead to register spills into global memory. So, while running many threads, we must also simultaneously ensure that register usage per thread is kept low so that occupancy is not affected. This is achieved by unrolling loops fewer times.

Batching also results in fewer OpenCL calls. These calls introduce non-negligible overhead, which is greatly reduced when several iterations are batched together into a single call.

2) Data layout transformation: Our baseline implementation does not access memory locations contiguously as it retains GNURadio's data layout. This can lead to bank conflicts for certain problem sizes, resulting in serialized memory accesses and increased kernel execution time. Our analysis revealed that large execution times for the serial implementation of FIR filtering stage (shown in Fig. 2) were due to poor data layout even though the filtering operation by itself was highly optimized. We reordered the input data so that memory access requests go to contiguous locations, as shown in Fig. 4. As a side effect of this transformation, we arrived at a layout that allows data streaming, where partial results can be computed from partial data.



(a) Default layout: Requests from all compute units go to the same bank. Such requests are serialized and therefore extra time is spent accessing the data.



(b) After reorganizing data: Requests go to different banks and therefore can be served in parallel.

Fig. 4. **Data Layout Transformation.** Dotted arrows represent the order in which data is stored. Solid arrows represent data access from the compute units. Reorganizing the data as shown in (b) helps avoiding bank conflicts.

3) Using local memory: Local memory, also known as local data share (LDS), provides much higher bandwidth than global memory. This memory can be shared by work-items within a work-group. Thus, such memory should be used by data that is frequently reused and shared by work-items within a work-group. FIR filtering exhibits a high reuse of data. Our software design, where the work-items within a work-group

[1]With the double-buffering technique, a set of two buffers is used for input data. When the GPU operates on one buffer, data is transferred to the other, and the GPU alternates between the two buffers.

always operate on the same filter, ensures that the work items share and reuse this fast-accessible data.

4) Using constant memory: The access latency of constant memory is several times faster than that of global memory. In our application, where the value of filter taps remains the same across iterations, we store this value in constant memory for fast access. We employ a blocking technique for larger tap sizes in order to fit them into constant memory.

5) Reducing dynamic instruction count: We adopt several techniques to reduce the number of dynamic instructions. We partially unroll loops so that dynamic conditional checks can be reduced. We limit the number of unrolls to four iterations in the FIR kernel so that register spills to the high latency global memory do not occur. We also eliminate the common sub-expressions involved in calculating the array indexes in the FIR kernel to reduce the dynamic instruction count.

6) Vectorization: Loading memory in vector types such as `float2` and `float4` is faster than loading floats. Also, using vector types in performing arithmetic operation ensures high packing density and resource utilization in VLIW architectures. In our implementation, we use the `float2` data type to represent the real and imaginary parts of the complex input signals. We use `float8` values in the FIR kernel to perform the mathematical operations.

7) Branching: GPUs perform poorly when it comes to branching operations. Though our application does not require any significant branching, a poorly designed solution can waste significant time performing branches. We carefully designed our implementation ensuring that there are *no* branches in our kernel implementations.

8) Fused Multiply-Add: We use fused multiply-add (FMA) operations to perform the FIR filtering so that architectures having special units for such operations can make use of them. Studies have shown that better performance can be achieved at a lower power budget by using such units, as the control overhead is greatly reduced [18]. However, we do not evaluate power benefits from optimization in this work.

VI. Performance Evaluation

A. Experimental Testbed

We evaluate the performance of our OpenCL implementation of a PFB channelizer on an AMD Radeon HD 6480G (integrated GPU) and AMD Radeon HD 6470M (discrete mobile GPU) and compare it to the CPU of the AMD A4-3300M APU (Llano). Table I shows the details of the hardware platforms. For convenience, we refer to HD 6480G, HD 6470M, and the CPU portion of the A4-3300M as *integrated GPU*, *discrete GPU*, and *multicore CPU*, respectively.

The details of software platform are as follows. We use OpenCL v1.1 and AMD APP v2.8 SDK for our channelizer implementation. For FFT, we use the accelerated parallel math library for FFT, clAmdFft-1.10.274, from AMD. All our implementations run on 64-bit Windows 7 OS.

TABLE I
Hardware platform characteristics

H/W Type	Multicore CPU	Integrated GPU	Discrete GPU
Platform	A4-3300M	HD 6480G	HD 6470M
Compute Units	2	3	2
Total Cores	2	240	160
Core clock	1900 MHz	444 MHz	750 MHz
Memory size	4096 MB	512 MB (shared)	1024 MB
Memory clock	675 MHz	675 MHz	800 MHz
Memory type	DDR3	DDR3	DDR3
TDP	35 W (combined)		9 W
	26 W	9 W	

Note: The individual TDP of CPU and GPU portions of the llano APU and the memory clock of APU were obtained from *CPU-Z* and *GPU-Z* utility tools. Other values are obtained from the specification sheet.

The discrete GPU is connected to the host via second-generation PCIe x16. We disable the power-saving option for all experimental runs, meaning the devices operate at the clock settings shown in Table I for the entire duration of execution. All performance results reported in this section are median values of 25 runs.

B. Performance Results

In this section, we report the performance obtained from the integrated and discrete GPUs and compare it to the performance obtained from the multicore CPU for our optimized OpenCL implementation.

Fig. 5 shows the execution time for 1024- to 8192-channel PFB channelizers with 12 taps per filter. The numbers are reported for the multicore CPU and the two types of GPU. This implementation of the channelizer takes complex signals as input and uses floating-point taps. Each FIR filter within the channelizer operates on 1K sample points for a total of 1M-8M sample points for the respective channelizers. Fig. 5 also shows the speedup results for the GPUs on the secondary axis. Our baseline implementation is an optimized multithreaded implementation making use of vector instructions running on the CPU of the AMD A4-3300M. Our optimized GPU implementation outperforms the baseline implementation by a factor of 18.3-43.0 for the discrete GPU and 18.4-27.2 for the integrated GPU for the different problem sizes. The discrete GPU, which has fewer cores but operates at a higher clock rate, was able to outperform the integrated GPU for up to 2048 channels. Beyond 2048 channels, the discrete GPU showed diminishing returns with performance marginally poorer than the integrated GPU. An analysis of the performance counters from the AMD APP Profiler [27] showed an increase in `LDSBankConflict` and a decrease in `CacheHit` counters

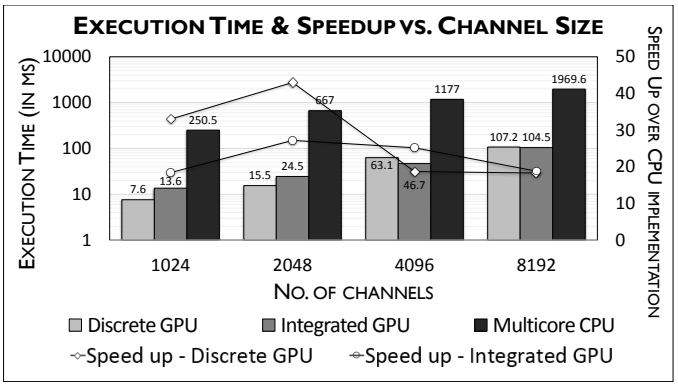


Fig. 5. **Performance and Scalability Results.** Execution time on multicore A4-3300M CPU, integrated HD 6480G GPU and discrete HD 6470M GPU. Speedup of GPU implementation over optimized CPU implementation shown in secondary axis.

for the discrete GPU for the larger problem sizes. Thus, the relatively poor performance can be attributed to the inability of discrete GPU to cache and reuse large data sizes effectively.

Fig. 6 shows the performance obtained for the PFB channelizer (excluding the post-processing stage of channel mapping) on the integrated GPU and discrete GPU, in terms of GFLOPS. In the best case, we achieved a performance of 26.14 GFLOPS on the discrete GPU and 17.48 GFLOPS on the integrated GPU. This corresponds to an energy efficiency of 2.9 GFLOPS/watt and 1.94 GFLOPS/watt for the discrete and integrated GPUs, respectively, in comparison to the 0.19 GFLOPS/watt for the high-power GPUs reported in earlier work [6].

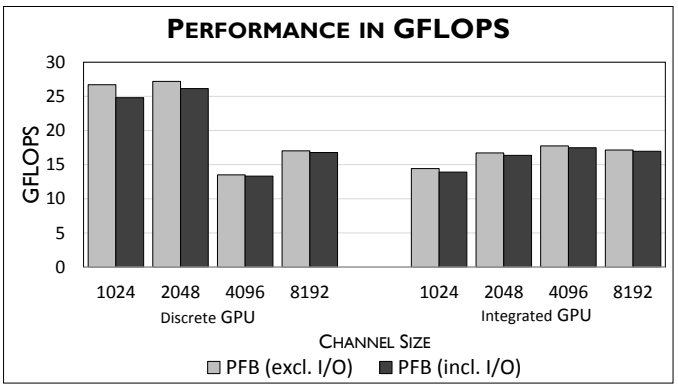


Fig. 6. **Performance of PFB channelizer in GFLOPS.** The peak performance of the discrete and integrated GPU are 240 GFLOPS and 213.1 GFLOPS, respectively.

C. Impact of Data Transfer Optimization and GPU Optimizations

Here we evaluate the performance impact of our optimization techniques described in Section V. We broadly classify the optimizations into data transfer optimizations and GPU performance optimizations.

Fig. 7 shows the contribution of each class of optimizations in improving the performance of PFB channelization for the discrete GPU and integrated GPU. The execution times are shown for a 1024-channel PFB channelizer with 12 taps per filter. On the discrete GPU, the unoptimized implementation takes 102.5 ms to finish execution. Upon applying the GPU optimizations discussed in Section V, this comes down to 14.1 ms, giving a speedup of 7.27. By efficiently hiding data transfer overheads, we achieve an *additional* speedup of 1.85, resulting in an overall execution time of 7.6 ms. Similarly, for the integrated GPU, the GPU optimizations and data transfer optimizations result in a speedup of 4.82 and 1.43, respectively, and together reduce the execution time from 93.5 ms to 13.6 ms. It is important to note that data transfer optimizations helped in improving the performance of both discrete and integrated GPUs equally even though in terms of percentage improvement, the values stand at 85% and 43%, respectively. In terms of absolute magnitude, the performance improvement is comparable and the seemingly lower percentage for integrated GPU is due to the relatively longer time spent in computation.

Overall, the optimizations result in a speedup of 13.4 and 6.9 for the discrete GPU and integrated GPU, respectively, thereby contributing significantly to the multi-fold speedup over the CPU implementation.

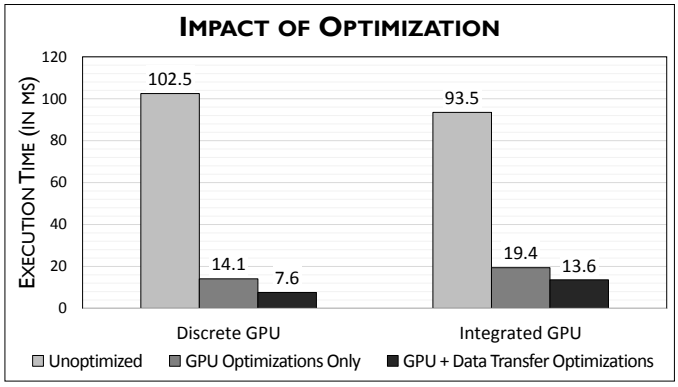


Fig. 7. Performance improvement from optimization techniques.

VII. CONCLUSIONS AND FUTURE WORK

Performing wideband channelization in real time for software-defined radio (SDR) is a challenging problem due to the high computational requirements and generally low-power constraints. In this work, we explored the viability of using mobile graphics processors to perform this task in place of the traditionally used FPGAs and high-power GPUs.

In this work, we identified the stages within a polyphase filter bank (PFB) channelizer that are sources of performance bottlenecks and presented an efficient mapping scheme that makes efficient use of the underlying hardware. We overcame the I/O bottlenecks associated with data transfers over PCI

Express by adopting a data streaming approach. We make use of several optimization techniques identified in the literature to improve the performance of our unoptimized GPU implementation by factors of 13.4 and 6.9 for our discrete and integrated GPUs, respectively. As a consequence, we showed that it is possible to achieve several-fold speedup (of up to 43-fold) even for mobile GPUs, thereby providing a realistic alternative platform to perform wideband channelization.

In the future, we will explore the possibility of using the CPU and GPU portions of a fused device (e.g., APU) simultaneously with each type of processor performing computations that best suits it. We will also compare and contrast the performance and power consumption of GPUs with other accelerator platforms such as the FPGA.

ACKNOWLEDGMENT

This work was supported in part by NSF I/UCRC IIP-0804155 via the NSF Center for High-Performance Reconfigurable Computing. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] G. Savir, "Scalable and Reconfigurable Digital Front-End for SDR Wideband Channelizer," Master's thesis, Delft University of Technology, 2006.

[2] T. Ulversøy, "Software Defined Radio: Challenges and Opportunities," *IEEE Communication Surveys and Tutorials*, vol. 12, no. 4, pp. 531–550, Oct. 2010.

[3] M. Woh, S. Seo, H. Lee, Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "The Next Generation Challenge for Software Defined Radio," in *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2007 (SAMOS '07)*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4599, pp. 343–354.

[4] K. C. Zangi and R. D. Koilpillai, "Software Radio Issues in Cellular Base Stations," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 4, pp. 561 –573, Apr 1999.

[5] M. Woh, S. Mahlke, T. Mudge, and C. Chakrabarti, "Mobile Supercomputers for the Next-Generation Cell Phone," *Computer*, vol. 43, no. 1, pp. 81–85, 2010.

[6] K. van der Veldt, R. van Nieuwpoort, A. L. Varbanescu, and C. Jesshope, "A Polyphase Filter for GPUs and Multi-Core Processors," in *Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy Date (Astro-HPC '12)*. ACM, Jun 2012, pp. 33–40.

[7] L. Pucker, "Channelization Techniques for Software Defined Radio," in *Proceedings of SDR Forum Conference*, Jun 2003, pp. 1–6.

[8] "AMD Accelerated Parallel Processing OpenCL Programming Guide," Advanced Micro Devices, Dec 2012. [Online]. Available: http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[9] M. Daga, T. Scogland, and W. Feng, "Architecture-Aware Mapping and Optimization on a 1600-Core GPU," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS '11)*. IEEE Computer Society, Dec 2011, pp. 316–323.

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)*. ACM, Feb 2008, pp. 73–82.

[11] C. Lee, S. Lo, N. Chen, Y. Chung, and I. Chung, "GPU Performance Enhancement via Communication Cost Reduction: Case Studies of Radix Sort and WSN Relay Node Placement Problem," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '12)*. IEEE Computer Society, May 2012, pp. 132 –139.

[12] GNU Radio Website, accessed December 2012. [Online]. Available: http://www.gnuradio.org

[13] F. J. Harris, *Multirate Signal Processing for Communication Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[14] S. A. Fahmy and L. Doyle, "Reconfigurable Polyphase Filter Bank Architecture for Spectrum Sensing," in *Proceedings of the 6th International Conference on Reconfigurable Computing: architectures, Tools and Applications (ARC '10)*. Berlin, Heidelberg: Springer-Verlag, Mar 2010, pp. 343–350.

[15] R. M. Monroe and R. Jarnot, "Broad-Bandwidth FPGA-Based Digital Polyphase Spectrometer," Tech Brief, Jet Propulsion Laboratory, Aug 2011.

[16] B. K. Hamilton, "Implementation and Performance Evaluation of Polyphase Filter Banks on the Cell Broadband Engine Architecture," Undergraduate Thesis, University of Cape Town, Oct 2007.

[17] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A Low-power Architecture For Software Radio," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 89–101.

[18] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, , C. Chakrabarti, and K. Flautner, "SODA: A High-Performance DSP Architecture for Software-Defined Radio," *IEEE Micro*, vol. 27, no. 1, pp. 114–123, Jan. 2007.

[19] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "AnySP: Anytime Anywhere Anyway Signal Processing," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. New York, NY, USA: ACM, 2009, pp. 128–139.

[20] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High Performance Discrete Fourier Transforms on Graphics Processors," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*. IEEE Press, Nov 2008, pp. 2:1–2:12.

[21] C. del Mundo, V. Adhinarayanan, and W. Feng, "Accelerating Fast Fourier Transform for Wideband Channelization," in *2013 IEEE Conference on Communications (ICC '13)*. IEEE Press, Sep 2013.

[22] P. Goorts, S. Rogmans, and P. Bekaert, "Optimal Data Distribution for Versatile Finite Impulse Response Filtering on Next-Generation Graphics Hardware Using CUDA," in *2009 15th International Conference on Parallel and Distributed Systems (ICPADS '09)*. IEEE Computer Society, Dec 2009, pp. 300 –307.

[23] D. Llamocca, C. Carranza, and M. Pattichis, "Separable FIR Filtering in FPGA and GPU Implementations: Energy, Performance, and Accuracy Considerations," in *2011 International Conference on Field Programmable Logic and Applications (FPL '11)*. IEEE Computer Society, Sep 2011, pp. 363 –368.

[24] M. Elteir, H. Lin, and W. Feng, "Performance Characterization and Optimization of Atomic Operations on AMD GPUs," in *2011 IEEE International Conference on Cluster Computing (CLUSTER '11)*. IEEE Computer Society, Sep 2011, pp. 234 –243.

[25] AMD Accelerated Parallel Processing Math Libraries (APPML). AMD. [Online]. Available: http://developer.amd.com/tools/hc/appmathlibs/Pages/default.aspx

[26] C. Gregg and K. Hazelwood, "Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without The Answer," in *Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE Computer Society, Apr 2011, pp. 134–144.

[27] AMD APP Profiler, accessed July 2013. [Online]. Available: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-app-profiler/