Accelerating Fast Fourier Transform for Wideband Channelization

Carlo del Mundo*, Vignesh Adhinarayanan†, Wu-chun Feng*†
*Department of Electrical & Computer Engineering
†Department of Computer Science
Virginia Tech
{cdel, avignesh, wfeng}@vt.edu

***Abstract*—Wideband channelization is a compute-intensive task with performance requirements that are arguably greater than what current multi-core CPUs can provide. To date, researchers have used dedicated hardware such as field programmable gate arrays (FPGAs) to address the performance-critical aspects of the channelizer. In this work, we assess the viability of the graphics processing unit (GPU) to achieve the necessary performance. In particular, we focus on the fast Fourier Transform (FFT) stage of a wideband channelizer. While there exists previous work for FFT on a NVIDIA GPU, the substantially higher peak floating-point performance of an AMD GPU has been less explored. Thus, we consider three generations of AMD GPUs and provide insight into the optimization of FFT on these platforms. Our architecture-aware approach across three different generations of AMD GPUs outperforms a multithreaded Intel Sandy Bridge CPU with vector extensions by factors of 4.3, 4.9, and 6.6 on the Radeon HD 5870, 6970, and 7970, respectively.**

I. Introduction

A wideband channelizer divides a given input RF band into multiple output channels for further baseband processing. It represents a compute-intensive task within the wideband receiver [1]. The channelizer is particularly useful in military applications where wideband spread spectrum signals must be demodulated in real time, e.g., software-defined radio (SDR).

Figure 1 shows an example of a wideband channelizer in support of SDR. To be effective, these receivers must fulfill performance and flexibility requirements. Given the number of channels in a wideband channelizer, multithreaded CPU implementations, while adaptable to a wide range of radios, generally cannot satisfy the necessary performance requirements. Special-purpose ASICs can satisfy performance requirements but are inflexible to adaptation. FPGAs simultaneously try to address performance and flexibility requirements, but their programmability remains a pervasive issue. GPUs address these shortcomings by delivering higher performance, better programmability, and low cost.

To motivate the need to accelerate FFT in wideband channelization, we profiled a polyphase filter bank (PFB) channelizer from GNURadio to identify its computational requirements. FIR filtering, FFT, and channel mapping encompass the stages of the PFB channelizer. Figure 2 depicts the relative execution time for each stage. FIR filtering and channel mapping grow in $O(n)$ time while FFT grows as $O(n\ log\ n)$. Our empirical data shows that as the channel size increases, the FFT dominates the overall execution. Thus, we focus on accelerating the FFT stage of the wideband channelizer.

This work was supported in part by an AMD Research Faculty Fellowship.

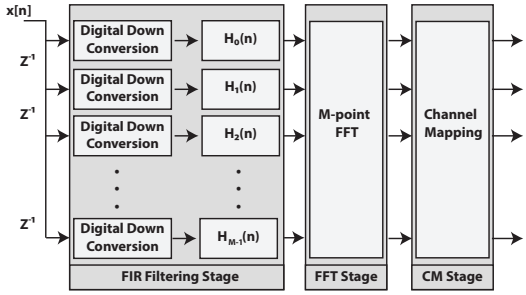

Fig. 1. **Polyphase FFT channelizer.** The input RF band signal x(n) is decomposed into M channels.

FFT is central across a wide range of fields including cognitive radio, image and signal processing, and encryption [2]–[4]. Hence, accelerating FFT is a critical aspect for many applications, including wideband channelization. To this end, we make the following contributions:

- A portable building block for FFT, which can be leveraged for actualizing GPU-based radio systems.
- Architecture-aware insights for mapping and optimizing FFT across three different generations of AMD GPUs.

II. Background

Traditionally, DSPs and FPGAs have been used to handle large inputs and the high computational requirements of the wideband channelizer [1]. With the advent of GPUs and supporting programming models such as CUDA and OpenCL, it is now possible to meet the performance requirements of wideband channelization with better productivity. Several works have used GPUs to accelerate channelization and other aspects of SDR. Kim et al. discuss a procedure for implementing SDR modems using GPUs [5]. They implemented an entire mobile WiMAX terminal on a GPU platform. Horrien et al. study the integration of GPUs in a SDR environment [6]. They explored the GPU space for SDRs with implementations of FFT, QPSK demapper, and IIR filtering in OpenCL. They evaluated their implementations on an Intel CPU and an NVIDIA

978-1-4673-3122-7/13/$31.00 ©2013 IEEE

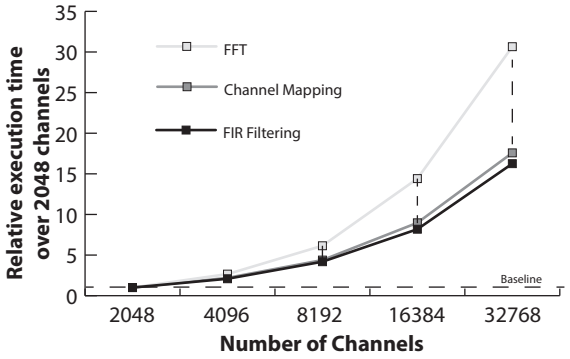

Fig. 2. **Relative execution time over 2048 channels for each stage in a PFB channelizer.** This empirical data shows the relative growths of each stage in the channelizer. The FFT stage grows in order $O(n\ log\ n)$, while FIR filtering and channel mapping grow in order $O(n)$. Each data point is normalized relative to the execution time of 2048 channels.

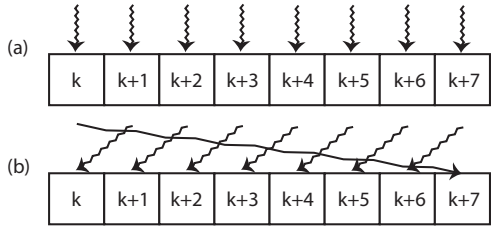

Fig. 3. **Memory access patterns.** (a) and (b) represents an optimal (fully coalesced) and suboptimal access pattern, respectively. In (a), the kth thread accesses memory element k. In (b), the kth thread accesses memory element $k-1$. The first thread, however, accesses memory element $k+7$.

GPU. However, their discussion of optimization was limited to higher-level design decisions such as kernel granularity and algorithmic decisions. Plishker et al. also discuss a design flow for using GPU-accelerated SDR development [7].

Using GPUs to accelerate polyphase channelization is the focus in several studies. Van der Veldt et al. discuss an optimized GPU implementation of polyphase filterbanks [8]. They evaluated their implementations on an Intel CPU, a NVIDIA GPU, and an AMD GPU. They used vendor libraries such as CUFFT and AppleFFT for the GPU. Harrison et al. explore the use of GPUs for polyphase channelization [9]. Their implementations used CUFFT and was evaluated on an older NVIDIA GPU. Other works focused on accelerating other aspects of SDR such as MIMO detection, LDPC decoders, and spectrum sensing using GPUs [10]–[13].

In general, while FFT implementations have been studied widely, the vast majority of the literature focuses on NVIDIA GPUs [14]. In this work, we provide more general insight into the mapping and optimization of FFT across three different generations of AMD GPUs, while also providing the basis for a vendor-agnostic FFT implementation that can be run on CPUs, GPUs, and accelerated processing units (APUs, the fusion of CPU and GPU cores on a die).

III. Approach

Our work seeks to create a portable building block for the FFT to support radio systems in parallel computer architectures such as GPUs, multicore CPUs, and FPGAs. Towards this goal, we present our set of architecture-aware optimizations on AMD GPU platforms as a case study. Specifically, we target the Radeon HD 5870, 6970, and 7970 GPU cards from AMD.

Daga et al. applied architecture-aware optimizations for n-body particle methods [15]. We extend their work by applying a similar set to FFT, a well-known spectral method.

To evaluate the efficacy of our optimizations, we map, optimize, and compare the FFT performance of three generations of AMD GPUs against a state-of-the-art, multi-core Intel Sandy Bridge CPU. The CPU runs an optimized multithreaded FFTW code with AVX extensions. We implemented a batched 1D 16-point, complex-to-complex, single-precision FFT. All GPU codes were validated with a double-precision, serial CPU code [16].

The rest of this section is summarized as follows. Sections III-A through III-F discusses system-level optimizations. Sections III-G to III-I discusses algorithm-level optimizations relevant to the FFT.

A. Run Many Threads

The GPU is a high latency, high throughput, multithreaded device. To effectively utilize this device, many threads must be launched. Failure to do so results in low functional utilization of the hardware. We apply this optimization by processing as many transforms as device memory will allow. Our baseline kernel processes 128 MB or 1,048,576 16-pt FFT batches.

B. Use On-chip Memory

The access time for on-chip memory, such as registers or local memory, is significantly faster than device global memory. On AMD GPUs, the latency of the register file is an order of magnitude faster than its global memory counterpart. Additional on-chip memories such as local memory provide limited capacity but fast access for scratchpad computation and communication. Our optimized versions make use of the register file (RP) and local memory (LM) for computation, communication, or both.

C. Organize Data In Memory

Efficient data layouts are essential for optimized memory transactions. Figure 3 depicts optimal and unoptimal access patterns. In our unoptimized implementation, sets of four threads access memory contiguously. However, the stride between sets of threads is 16. Our optimized version (CGAP) adheres to perfect coalescing rules where 64 threads access memory in a contiguous fashion.

D. Reduce Dynamic Instruction Count

While it is traditionally important to reduce dynamic instruction count, there is no guarantee that the GPU compiler will automatically apply these optimizations. Some examples include common subexpression elimination, loop unrolling,

and function inlining. We take advantage of these optimizations by explicitly programming them in our kernels. We performed loop unrolling (LU) for memory loads and stores, reordered computation to eliminate common subexpressions (CSE), and inlined (IL) kernel helper functions.

E. Use Image and Constant Memory

Image and constant memory provide cached performance and lower latency relative to device global memory. For commonly accessed data elements, the cached constant memory is an excellent candidate for performance improvement. Constant memory is limited in capacity, so it is best suited for a small set of frequently accessed data. We implemented both kernel argument (CM-K) and kernel literal (CM-L) optimizations for constant memory [17]. Our image memory implementations (IM) use a four-channel (RGBA) format where each channel is a float.

F. Use Vector Types

Using vector types is especially important in the context of the Very Large Instruction Word (VLIW) pipeline present in Radeon HD 5870 and 6970. Our baseline implementation uses a float2 vector in interleaved format to represent the real and imaginary components of a complex number. We also explore the use of a planar format where real and imaginary components are allocated in separate vector arrays. We vary the vector size in powers of two. In our optimized implementations, the interleaved format is represented as vector access/scalar math (VASM), while the planar format represents vector access/vector math (VAVM).

G. Algorithmic Mapping

Our mapping strategy is based on the Cooley-Tukey framework, where an N-pt FFT is represented as a 2D N_1N_2-pt FFT. A 16-pt FFT is broken down into 4 radix-4 FFTs. Each thread handles one radix-4 FFT or 32 bytes per thread. In our implementations, a group of 64 threads handles 16 batches of 16-pt FFTs. Sets of four threads work on one batch.

H. Twiddle Multiplication Stage

Twiddle multiplication is characterized by multiplying all elements in the array by a predefined constant or twiddle factor. The accuracy of these multiplications are governed by double-precision transcendental operations. Because each 16-pt FFT uses the same twiddle factors, the use of constant memory is an excellent candidate for optimization. The baseline kernel computes these values on-the-fly while optimized versions precompute on the CPU and store results in constant memory.

I. Transpose Operation Mapping

The FFT requires an explicit transpose step where each thread must communicate its intermediate results to other threads. When threads need to communicate, they write to and read from a scratchpad memory such as local memory. Our local memory optimizations are split into three distinct implementations: (1) communication only (LM-CM), (2) computation and communication (LM-CC), and (3) computation, no transpose (LM-CT). The first implementation performs computation on global memory and local memory for communication. The second performs all computation and communication operations in local memory. The third implements a technique mentioned in the work of Volkov and Kazian where the authors exploit a property of the Cooley-Tukey framework [18]. The communication or transpose step is completely eliminated by performing radix-4 FFTs on the columns, twiddle multiplication, and finally radix-4 FFTs on the rows. The transformed output, in column-major order, is then re-indexed to row-major order from local memory to global memory.

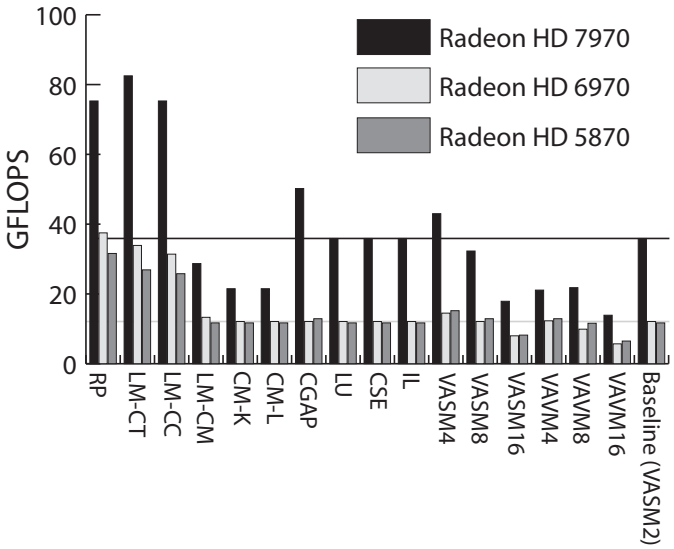

Fig. 4. **Optimizations applied in isolation.** Comparisons should only be made per architecture. RP = register preloading, LM-{CT,CC,CM} = local memory-{computation no transpose, computation and communication, communication only}, CM-{K,L} = constant memory-{kernel argument, literal argument}, CGAP = coalesced global access pattern, LU = loop unrolling, CSE = common subexpression elimination, IL = inlining, VASM{4,8,16} = vector access scalar math{4,8,16}, VAVM{4,8,16} = vector access vector math{4,8,16}.

IV. Results & Discussion

Figure 4 depicts the efficacy of various optimizations in isolation. Figure 5 represents optimizations applied in concert. Sections IV-B through IV-E discuss individual optimizations, section IV-F discusses optimizations applied in combination, and section IV-G discusses our accelerated FFT compared to multithreaded CPU FFTW.

A. Experimental Testbed

For GPU hardware, we used the AMD Radeon HD 5870, 6970, and 7970 cards. For the software environment, we used the AMD APP v2.6 SDK on Ubuntu 10.04 with kernel version 2.6.32-5-amd64 and fglrx driver module version 8.95.3. FFTW version 3.3.2 was configured to utilize four threads on OpenMP with explicit AVX extensions on an Intel i5-2400.

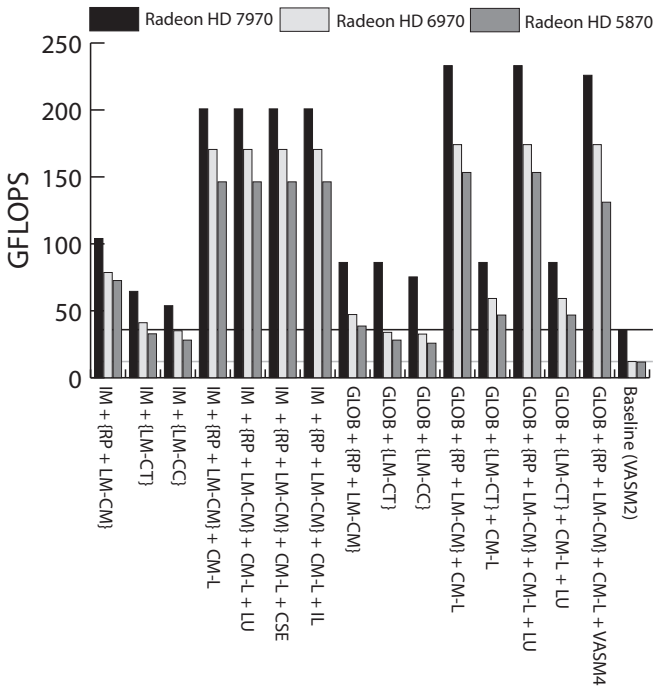

Fig. 5. **Optimizations applied in concert.** Comparisons are to be only considered per architecture. IM = image memory. For clarification, GLOB here refers to global memory as the input buffer. Unless otherwise noted, all implementations make use of VASM2 and CGAP.

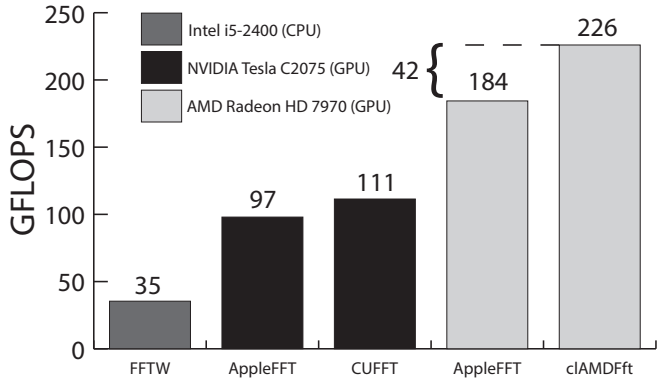

Fig. 6. **Survey of FFT libraries for state of the art CPU and GPU hardware for a single-precision batched 1D 16-pt FFT.** CUFFT and AMD APPML are vendor provided libraries for NVIDIA and AMD GPUs, respectively. AppleFFT is an open source FFT library.

To provide context on accelerated FFT libraries, Figure 6 surveys vendor and open source FFT libraries on state of the art hardware. AppleFFT v1.6, CUFFT v4.1, and clAMDFft v1.8.239 was used. This survey suggests that (1) CPUs provide significantly lower performance than GPU counterparts for FFT and (2) vendor libraries (CUFFT and clAMDFft) are better tuned than existing open source libraries.

B. On-chip Memory (RP, LM-CT, LM-CC, LM-CM)

RP represents a substantial optimization for all platforms yielding a 2.7x, 3.1x, and 2.1x improvement over baseline for the Radeon HD 5870, 6970, and 7970 cards respectively.

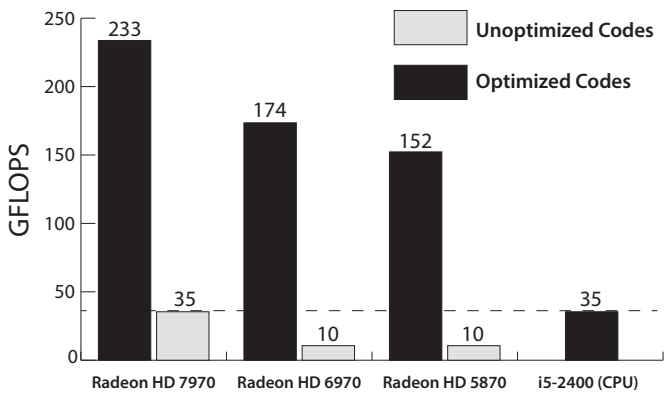

Fig. 7. **Summary of experiment compared with a multithreaded CPU with AVX extensions.** Unoptimized kernels represent our baseline implementations, while optimized kernels represent the best set of optimizations applied to a particular architecture. The Intel i5-2400 CPU shown here uses FFTW 3.3.2.

Three of our local memory implementations exhibit varying performance increases suggesting that even when an optimization is applied, significant effort is still necessitated. LM-CM performs the worst as it only uses local memory for scratchpad communication, while LM-CT provides the best performance. LM-CT performs computation directly on local memory and avoids a communication step by aptly reorganizing data from global memory. We have also found that performing computation with unit-stride (row-major) is better than stride-4 (column-major) computations. This optimization is not shown for brevity.

C. Coalesced Access Patterns (CGAP)

Surprisingly, a perfectly coalesced global access pattern does not improve performance in isolation for all architectures. Our analysis of optimizations in isolation reveal that implementations other than on-chip optimizations (RP, LM) have significant increases in global memory bus traffic. This increase could be as much as 16 times higher than optimal bus traffic. The value of coalesced access does not manifest until combined with on-chip memory optimizations.

D. Reduce Dynamic Instruction Count (LU, CSE, IL)

No improvement was shown for LU, CSE, or IL. These results occur both in isolation and in combination with several optimizations.

E. Vector Types (VASM-{2,4,8,16}, VAVM-{4,8,16})

16-byte vector access/scalar math (VASM4) provides the best performance across all GPUs yielding a 1.2x performance increase. We hit a point of diminishing returns when using vector types greater than 16-bytes. A 16-byte vector is the length suggested by the OpenCL Programming Guide [17].

F. Optimizations in Concert (IM, CM-K, CM-L)

We combined optimizations in a greedy manner with the notion of *stages*. Each stage represents a category of optimizations, and an optimization within a stage is selected

based on its performance. We stacked optimizations until all options were exhausted. For example, of three possible on-chip memory optimizations, {RP+LM-CM}, {LM-CC}, and {LM-CN}, RP+LM-CM performed the best and is thus selected for subsequent implementations.

In Figure 5, implementations are divided between image and global memory. Unless otherwise noted, each implementation is vector access/scalar math (VASM2). Using a regular global memory buffer is better than using image memory (IM). Constant memory provides a significant improvement in combination, which goes counter to the results in Figure 4. There are no differences between embedding constant memory directly as a kernel argument (CM-K) or kernel literal (CM-L). On all platforms, the best set of optimizations is *GLOB + {RP + LM-CM} + CM-L.*

G. Comparison Against CPU Code

Figure 7 shows the relative performance of our optimized and unoptimized codes against multithreaded FFTW with vector extensions using an Intel i5-2400. Our GPU baseline performs poorly compared to the CPU. Applying architecture-aware optimizations provide a significant performance increase relative to the unoptimized versions.

V. Conclusions and Future Work

Applying optimizations on a GPU is unintuitive and effectiveness is primarily driven by the underlying architecture. Furthermore, significant effort is required to find optimal combinations. The degree of programmer effort is met with significant performance improvement compared to naivé versions. Our accelerated versions outperform a multithreaded Intel Sandy Bridge CPU with AVX extensions by factors of 4.3, 4.9, and 6.6 on the Radeon HD 5870, 6970, and 7970, respectively.

In our future work, we will address the following items:

- *Fine-grain implementation of the PFB channelizer.* We will accelerate the FIR filtering and channel mapping stages to realize the channelizer fully on the GPU.
- *Increase FFT generality.* We focused exlusively on accelerating a 16-pt FFT, but radio applications use varying sample sizes. An increase in sample size could potentially impact the efficacy of our optimizations.
- *Evaluate alternative parallel architectures.* This work focuses exclusively on GPUs as a case study. The techniques in this work can be applied to FPGAs, multi-core CPUs, and the fused CPU/GPU platform (APU) as supporting computer architectures for radio systems.

Acknowledgements

The authors wish to acknowledge and thank Thomas Scogland for his extensive feedback on this work.

References

[1] G. Savir, "Scalable and Reconfigurable Digital Front-End for SDR Wideband Channelizer," Master's thesis, Delft University of Technology, 2006. [Online]. Available: http://ce-publications.et.tudelft.nl/publications/704_scalable_and_reconfigurable_digital_frontend_for_sdr_wideba.pdf

[2] I. Uzun *et al.*, "FPGA Implementations of Fast Fourier Transforms for Real-Time Signal and Image Processing," *IEEE Proceedings of Vision, Image and Signal Processing*, vol. 152, no. 3, pp. 283–296, Jun. 2005.

[3] Q. Zhang *et al.*, "An Efficient FFT for OFDM based Cognitive Radio on a Reconfigurable Architecture," in *Proceedings of the IEEE International Conference on Communications (ICC)*, Jun. 2007, pp. 6522–6526.

[4] V. Lyubashevsky *et al.*, "SWIFFT: A Modest Proposal for FFT Hashing," in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, K. Nyberg, Ed. Springer Berlin / Heidelberg, 2008, vol. 5086, pp. 54–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71039-4_4

[5] J. Kim *et al.*, "Implementation of an SDR System using Graphics Processing Unit," *Communications Magazine, IEEE*, vol. 48, no. 3, pp. 156–162, 2010.

[6] P.-H. Horrein *et al.*, "Integration of GPU Computing in a Software Radio Environment," *Journal of Signal Processing Systems*, vol. 69, no. 1, pp. 55–65, Oct. 2012. [Online]. Available: http://dx.doi.org/10.1007/s11265-011-0639-1

[7] W. Plishker *et al.*, "Applying Graphics Processor Acceleration in a Software Defined Radio Prototyping Environment," in *Proceedings of the IEEE International Symposium on Rapid System Prototyping (RSP)*, May 2011, pp. 67–73.

[8] K. van der Veldt *et al.*, "A Polyphase Filter for GPUs and Multi-core Processors," in *Proceedings of the Workshop on High-Performance Computing for Astronomy*, ser. Astro-HPC 2012. New York, NY, USA: ACM, 2012, pp. 33–40. [Online]. Available: http://doi.acm.org/10.1145/2286976.2286986

[9] G. Harrison *et al.*, "Channelization Techniques for Software Defined Radio," in *Proceedings of SDR Forum Conference*, 2008.

[10] H. Yang *et al.*, "Implementation of Parallel Lattice Reduction-aided MIMO Detector using Graphics Processing Unit," *Analog Integrated Circuits and Signal Processing*, pp. 1–9, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10470-012-9870-3

[11] M. Wu *et al.*, "Implementation of a High Throughput 3GPP Turbo Decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, no. 2, pp. 171–183, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11265-011-0617-7

[12] C.-H. Lee *et al.*, "Parallelization of Spectrum Sensing Algorithms using Graphic Processing Units," in *Cross Strait Quad-Regional Radio Science and Wireless Technology Conference (CSQRWC)*, Jul. 2012, pp. 35 –39.

[13] S. Kang and J. Moon, "Parallel LDPC Decoder Implementation on GPU based on Unbalanced Memory Coalescing," in *Proceedings of the IEEE International Conference on Communications (ICC)*, 2012.

[14] N. K. Govindaraju *et al.*, "High Performance Discrete Fourier Transforms on Graphics Processors," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, ser. Supercomputing 2008. Piscataway, NJ, USA: IEEE Press, 2008, pp. 2:1–2:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413373

[15] M. Daga *et al.*, "Architecture-Aware Mapping and Optimization on a 1600-Core GPU," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems*, ser. ICPADS 2011. Washington, DC, USA: IEEE Computer Society, 2011, pp. 316–323. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2011.29

[16] D. H. Bailey, "A High-Performance FFT Algorithm for Vector Supercomputers," *International Journal of Supercomputer Applications 2*, pp. 82–87, 1988.

[17] "AMD Accelerated Parallel Processing OpenCL Programming Guide," Advanced Micro Devices, p. 286, Jul. 2012. [Online]. Available: http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[18] V. Volkov and B. Kazian, "Fitting FFT Onto the G80 Architecture," May 2008. [Online]. Available: http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf