GPU AND FPGA BASED ARCHITECTURE DESIGN FOR REAL-TIME SIGNAL CLASSIFICATION

Nilangshu Bidyanta (nbidyanta@email.arizona.edu)[1], Ali Akoglu (akoglu@email.arizona.edu)[1], Garrett Vanhoy (gvanhoy@email.arizona.edu)[1], Mohammed A. Hirzallah (hirzallah@email.arizona.edu)[1], Tamal Bose (tbose@email.arizona.edu)[1], and Bo Ryu (boryu@episyscience.com)[2]

[1]Electrical and Computer Engineering, University of Arizona, Tucson, AZ
[2]EpiSys Science, Inc., Poway, CA

ABSTRACT

Among the modulation classification methods, spectral correlation density (SCD) is an attractive method as it operates effectively under low SNR conditions. However, computation complexity of the SCD makes it an impractical solution for real-time signal classification. In this study, we map the entire SCD flow onto high-end GPU (Tesla-K20), mobile GPU (Tegra-K1), and hybrid architecture that couples Tegra-K1 with the Zynq-7000 FPGA as a single lane. We discuss our parallelization approach on each platform and present details of our lane-based parallel architecture. We achieve a signal classification throughput of 111 signals/second on Tesla-K20 GPU (390x faster than single-threaded implementation running on a 2.33GHz processor), 9 signals/second on the Tegra-K1, and 19 signals/second on the hybrid architecture. Finally, we present the model of an n-lane architecture with a dual ring bus interconnect whose throughput increases linearly with a better power budget than the Tesla-K20.

1. INTRODUCTION

The task of determining the modulation of an incoming signal has been named modulation classification (MC), recognition, or identification [1], and has a variety of applications in the military and commercial domains [2]. Recent advances in wireless communications across the globe, especially low-cost software defined radios and their wide availability, readily allow adversaries to devise and dynamically change communication schemes and patterns that are inherently much harder to detect. In a non-cooperative communication, the transmitter does not intend for its data to be interpreted by the observing radio. Signal classification involving non-cooperative communications is in general a more challenging scheme [3] than cooperative communications as properties of the incoming signal, such as its coding scheme may not be known a-priori. Particularly computation complexity of signal classification methods pose as the main barrier for real-time signal classification using general purpose processors.

A feature-based classification method extracts a set of descriptive values from the signal that differentiates each signal from each other. These features can include cumulants, statistics, Fourier Transform coefficients, Wavelet Transform coefficients, or a combination of them [4]. Feature extraction methods in MC can be largely separated into two types: likelihood-based and feature-based. Generally likelihood-based methods can be considered optimal in the sense that they achieve the least probability of misclassification [5]. However, this comes at the cost of a computational complexity that eludes real-time implementation in many cases [6]. From algorithmic point of view, because of their computational efficiency, feature-based methods coupled with decision structures such as, support vector machines (SVM) have been preferred over likelihood-based methods. However, such an approach is suboptimal in terms of probability of correct classification. Furthermore, each feature extraction method varies in the amount of data it needs, the percent correct classification, computational complexity, and the amount of modulations it can classify.

Feature based MC can be divided into three stages, each of which with its own challenges. First, a signal goes through a pre-processing stage in which common signal parameters such as center frequency or symbol rate can be estimated and equalization can be done. This is done to maximize the effectiveness of the second stage, the feature extraction stage, in which a set of features are extracted from the signal using a variety of algorithms. Choosing a set of features that will most effectively describe the incoming signal has been the primary subject of research in MC. Lastly the classification stage implements a decision structure to best differentiate the incoming signals based on the extracted features. In this study we focus on the feature extraction stage, and we are particularly interested in the spectral correlation density (SCD) method as it operates effectively under low SNR conditions and is able to extract features for classifying numerous types of signals. The SCD estimation algorithm comes in two varieties: FFT Accumulation method (FAM) [7] and Spectral Strip Correlation Algorithm (SSCA) [8]. In our design, we chose to implement the FAM variety since it is considered to be more computationally efficient than SSCA [9]. However, computation demand of the SCD method (65,536 of 32-point complex FFTs for one signal during one stage of the SCD flow), still makes it an impractical solution for real time

signal classification. A highly parallelized concurrent execution flow is needed to be able classify numerous signals in real-time that shift their center frequency unpredictably, have inherently low signal to noise ratios. To the best of our knowledge there is no prior parallelization work on SCD targeting multicore and many-core architectures. The related work by [9] focuses on implementing a SVM on a FPGA to classify the results obtained from the SCD flow. Furthermore, our work is the first study on mapping SCD onto a FPGA-GPU integrated system.

Our goal is to design a heterogeneous parallel computing platform that couples Field Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU), and achieve real-time classification. We restructure each stage of the SCD program flow, expose fine and coarse grained parallelism, achieve massively concurrent execution, and take advantage of computation power of each device in a complementary manner.

The rest of the paper is organized as follows. In section 2, we present the SCD flow concurrently with our GPU based parallelization approach for each stage. We present benchmarking studies on FFT and data manipulation intensive stages of the SCD flow and justify our workload partitioning between the GPU and FPGA. We then present our parallelization approach for the tasks that are suitable for the FPGA. In section 3, we present a detailed performance comparison between the GPU-only and GPU-FPGA integrated platforms. We discuss our profiling and modeling based partitioning methodology, and lane based architecture development approach that offers massive parallelism for the SCD flow. In section 4, we finally present our conclusions and future work.

2. ALGORITHMIC APPROACH: STEPS OF THE SPECTRAL CORRELATION DENSITY FLOW

Step 1: Framing and Windowing

The signal of an arbitrary length is split up into P parts of length Np each. Each part overlaps with its predecessor. The offset between the beginning of two consecutive parts is set to L where L = Np/4. This process is called framing of the signal. These frames are then arranged column-wise into a matrix of size Np x P as illustrated in Figure 1.

If a signal is sampled using a rectangular filter (effective output of previous stage), the steep cut-offs at both ends cause high frequency components to be artificially introduced into the Fourier transform of the signal. Therefore a windowing function of length Np is applied to the framed signal as shown in Figure 2.

Since both signal and window are 1D vectors, the window is applied by a simple vector multiplication with the framed signal as illustrated in Figure 3. The number of scalar multiplications in this case is Np x P. Factoring in the number of signals to be processed, this value scales to Np x P x signal-count. On the GPU, a single thread will handle a scalar multiplication. Assuming typical values for Np, P and signal-count (256, 32 and 30 respectively), the total number of scalar multiplications is 245,760. The K20 GPU can concurrently run approximately

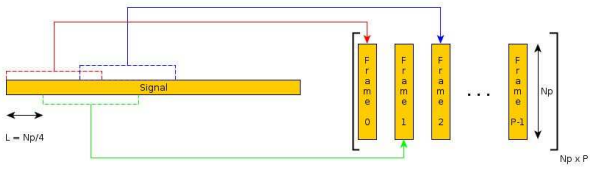

Figure 1: Framing process.

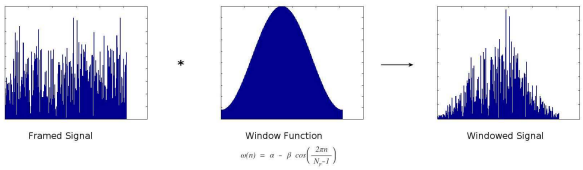

Figure 2: Window function eliminates artificial high frequency components.

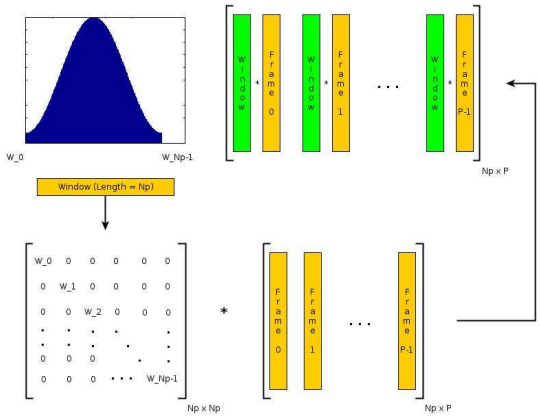

Figure 3: Applying window through vector multiplication.

26K threads (13 SMP x 2048 Threads / SMP) to complete this process in 10 iterations.

On the GPU, each thread block processes one frame and consists of 256 threads. There are 32 such thread blocks since the signal is divided into 32 overlapping parts. This mapping is illustrated in Figure 4. Data from each frame is brought into the shared memory and vector multiplied with the hamming window. The windowed frames are then stored into a matrix in column major order. The matrix is stored as a linear contiguous block in the global memory.

Step 2: Iterative 256-point FFT

Each frame obtained from step-1 goes through the FFT as shown in Figure 5. This stage is executed in batch mode, where 32 columns of 256-point element arrays go through the FFT stage in parallel for each signal.

Step 3: FFT Shift + Down Conversion + Transpose

The third kernel handles three operations - FFT shift (to get the FFT result in 'natural order'), down conversion of the FFT

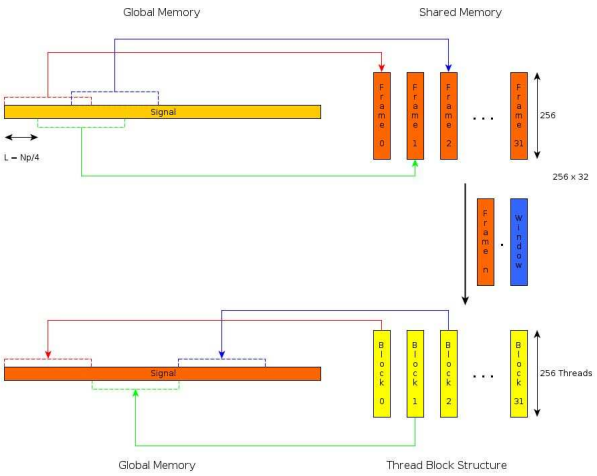

Figure 4: Framing and windowing the signal on the GPU.

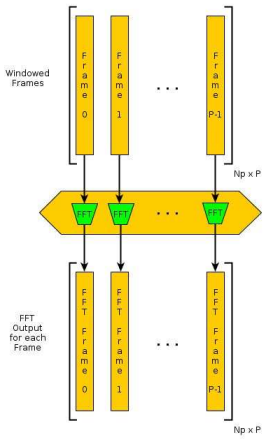

Figure 5: Applying 256-point FFT on Windowed Frames.

shifted signal and transpose of the resulting matrix in order to prepare it for the next set of FFTs.The expression responsible for down conversion is shown in equation 1.

The FFT output is symmetrical around the zero frequency. In accordance with this, axis +/- m represents the frequency axis. Axis k represents the number of parts the signal was split up into (P). Once elements are distributed to threads, each thread (with coordinates (k,m)) performs the computation as illustrated in Figure 6. Correspondingly, threads processing elements that belong to one column (i.e. elements belonging to the same FFT output frame) are grouped into thread blocks.

However, the CUDA / C++ libraries do not support complex exponentials and therefore the equation is converted to a sum of cosine and sine using Euler's formula (equation 2). After this conversion, resulting expression is shown in equation 3. The down conversion step is a purely element-wise operation and therefore can be shifted to the very beginning or to the very end of the group of three operations. This of course requires a mod-

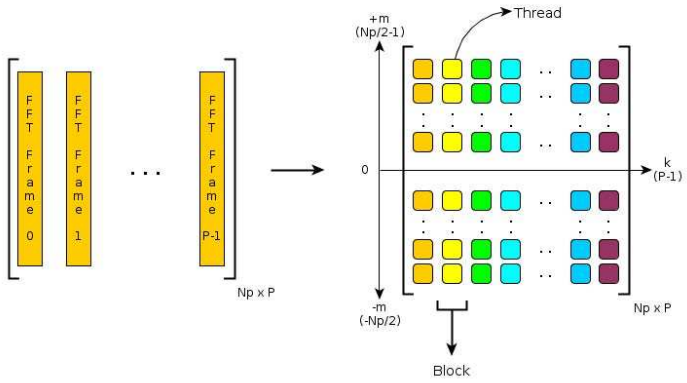

Figure 6: Down Conversion over each element of the 256x32 matrix.

ification in the equation computing this stage. In order to keep recalculation of the equation simple, we have chosen to perform down-conversion first, followed by a movement of elements that achieves FFT shift and transpose in one go. The final expression is shown in equation 4, which is substituted in equation 3.

$$out_{km} = out_{km} \times e^{\frac{-2\pi kmLi}{Np}} \tag{1}$$

$$e^{i\theta} = \cos\theta + i\sin\theta \tag{2}$$

$$out_{km} = out_{km} \times [\cos\frac{2\pi kmLi}{Np} - i\sin\frac{2\pi kmLi}{Np}] \tag{3}$$

$$m = (m + \frac{Np}{2}) mod Np \tag{4}$$

The last step of moving elements around to perform the FFT Shift and transpose is illustrated in Figure 7. The matrix is arranged in column major order. The highlighted regions (red, green, blue and yellow) of the matrix show how the data move before and after the transformation takes place. In addition to moving between quadrants of the matrix, each highlighted region also flips about the main diagonal of that region.

To perform FFTshift + transpose, we have the option of choosing between a 1D kernel and a 2D kernel. Assume we chose a 1D kernel. The minimum length of such a kernel must be 32 in order to maintain coalesced memory access and to ensure there are no idle threads. This is because there are 32 threads in a warp. Such a kernel must read column-wise (since we store in column major format) and write rows. For example, a kernel must read a column of contiguous points (say, column 2 of the input matrix consisting of points 9 to 12 in Figure 7) and write a row in the output matrix as shown by the red box around the elements. Note that the output matrix is also stored in the column major format since the next stage will need access to contiguous sets

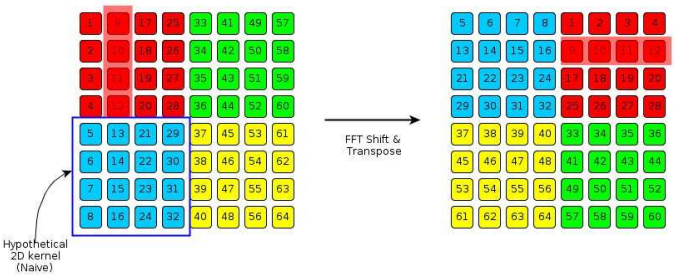

Figure 7: FFT Shift and Transpose.

of rows of the pre-transformation matrix, i.e. points 1, 9, 17, 25 must be contiguous post transformation. Had the output matrix been stored in row-major format, elements of rows from the pre-transformation matrix would still be strided. As is clearly seen, a 1D kernel will be writing rows to the global memory in a strided fashion despite having 32 threads. With a 2D kernel, the threads will read in a 2D block of data from the global memory into the shared memory, column-wise. After performing down conversion on each element, the kernel will then read the shared memory row-wise and write to the output matrix column-wise. Note that warps will read in a contiguous column and write a different set of data contiguously to the output matrix. This is achievable since coalescing rules do not apply to the shared memory. The important thing to remember is every warp's reads are stored in the shared memory and accessible by all warps of that thread block. In contrast to this, the 1D case had 1 warp reading in and storing its data exclusively. It then needed to write the same data back to the output matrix in a strided fashion and hence 2D kernels are better suited for this transformation.

The most naive way of implementing this stage with a 2D kernel is to design a kernel with thread block dimensions equal to one of the highlighted regions of Figure 7, assigning each element of the region to a thread inside the block. Given the input parameters (256 as window size and 32 overlapping parts), such a kernel is impossible on devices with compute capability less than 3.0 as each thread block would have 2048 threads. Devices with compute capability 2.0 have a limit of 1024 threads per thread block. This is much lower than the limit of concurrent blocks that can be launched on the GPU. With fewer blocks, a small amount of shared memory will see numerous concurrent accesses resulting in more bank conflicts and slower operation.

The next approach is to divide the matrix (shaded with regions from Figure 7) among thread blocks numbered 0 to 7 as shown in Figure 8. Since we divide the matrix into 8 thread blocks, shared memory access is more relaxed leading to fewer bank conflicts. The regions of same color in the input and output matrix are handled by the same thread blocks. Each thread block consists of a whole number of warps. The dimensions of a thread block are also representative of the shared memory size allocated within. Consider thread block 7 (lower right corner, in yellow). Data is read column-wise into the shared memory (green band, representing two warps of threads). Next, a half-warp reads the shared memory row-wise. This action is denoted by a red band. Note that this data is different from what was read into the shared memory by the threads earlier. The data is then written to the output matrix at the appropriate location. Access to the output matrix is denoted by the blue band. The most obvious defect of this approach is that coalesced memory accesses take place only in half-warps when writing the output matrix.

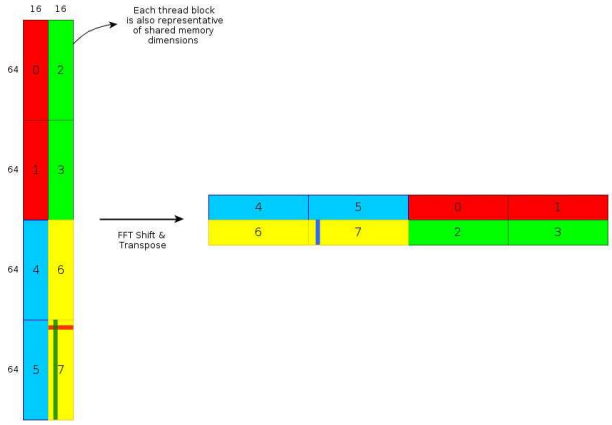

Figure 8: Thread block organization for 2D Kernel.

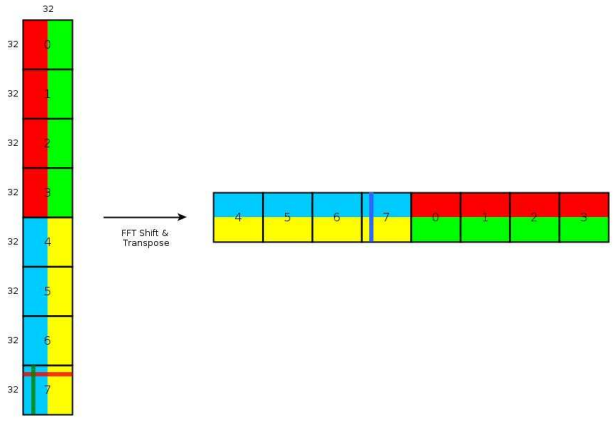

Figure 9: Revised thread block organization for 2D Kernel.

We simply change the dimensions of the thread blocks to 32x32 instead of 64x16 as in the previous approach. Both address 1024 threads each. This is illustrated in Figure 9. Using the same convention as the previous approach, the green band (one warp of threads) in thread block 7 reads data from the global memory into the shared memory. Unlike the previous approach, the red and blue bands also represent complete warps of threads, reading row-wise from the shared memory and writing column-wise to the output matrix, respectively. This approach is the best considering the GPU and program parameters on hand. Furthermore, trying to reduce the block size below this approach's (32

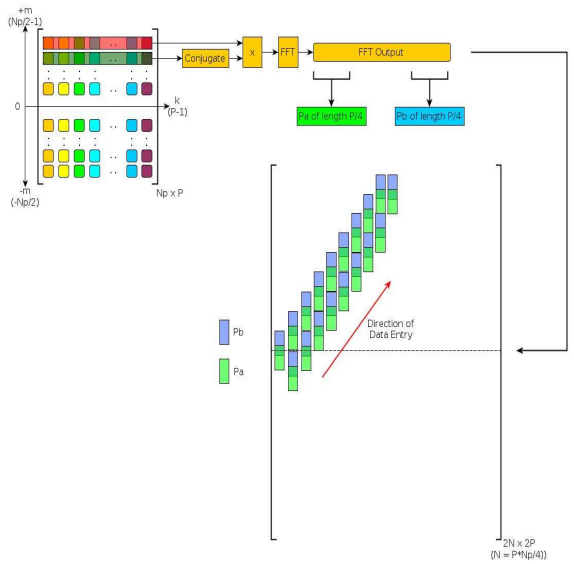

Figure 10: SCD Matrix formation through Conjugate product and 32-point FFT for all pairs (256x256).

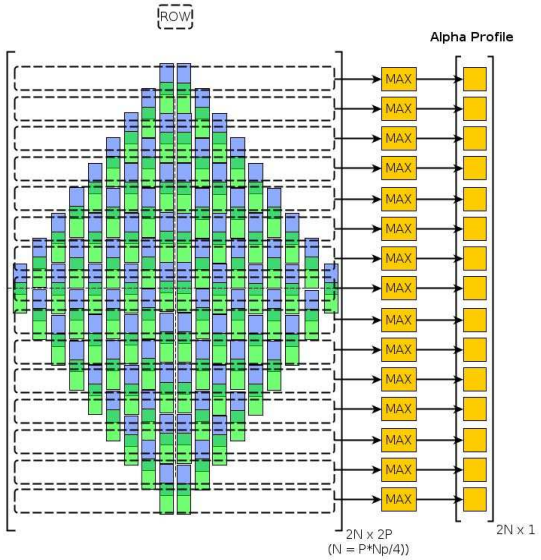

Figure 11: Alpha profile generation for classification.

x 32) in order to increase the number of blocks to reduce shared memory bank conflicts, won't be beneficial because we will begin dealing with incomplete warps, i.e. less than 32 threads in at least one dimension.

Step 4:SCD Matrix and Alpha Profile Calculation

This last stage deals with the formulation of the SCD matrix and calculation of the alpha profile, which is the most compute intensive stage in the entire algorithm. Broadly, this stage can be divided into the following four steps: *Conjugate product calculation, FFT of the conjugate products (FFT shifted to ensure natural order), SCD Matrix formation, and Alpha profile calculation.*

Overview: The output of Step 3 is a 2D array with 256 rows and 32 columns. All pairwise row combinations (256x256 in number) go through the computation flow shown in Figure 10. First row and the conjugate of the second row undergo vector multiplication. The FFT of the product (32-point FFT) is calculated and P/4 sections from each end of the vector are extracted while the middle section is discarded. Let the left end of the vector be called Pa and the right end Pb. After extraction, the "scd" matrix is populated (which is initialized to zero) as shown in the figure. Pb and Pa are aligned one on top of the other while filling up the matrix in the direction pointed by the red arrow. This process is repeated for all possible pairs of rows, including repetitions (i.e. the same row may be counted twice). The result of this stage is a scd matrix with non-zero values (Pb - Pa columns) taking a diamond shape as shown in Figure 11.

Step 4.1: Conjugate product calculation- In the first step, (NpxNpxP) element-wise multiplications take place considering all possible pair-wise combinations of all rows of the down-converted matrix. With the current set of input parameters, this number turns out to be 2,097,152. This far exceeds the number of concurrent threads that can be launched on any GPU. The number of point-wise multiplications for all combinations of one particular row with all rows, inclusive of itself, is Np x P, which turns out to be 8,192 threads. This falls well within the capabilities of most GPUs. Based on the above deduction, this stage is computed by a serialized loop over all rows in the down-converted matrix.

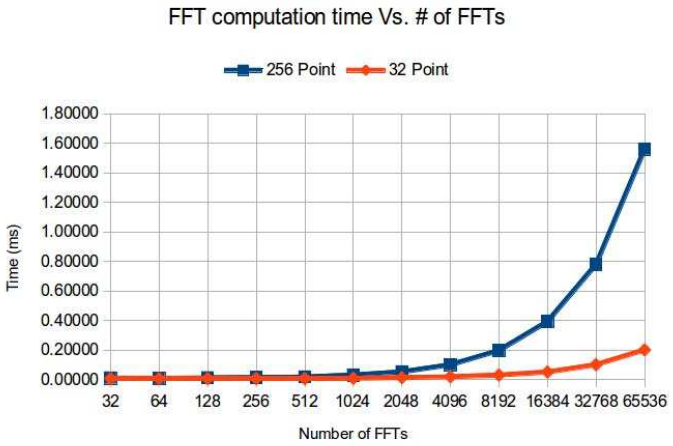

Figure 12: Number of FFTs vs execution time for 256-points and 32-points on K20 GPU.

Step 4.2: 32-point FFT- The output of Step 4.1 is 65,536 32-element vectors for one signal. Each vector goes through 32-point FFT computation in this step. Here we present an analysis of FFT on GPU and FPGA. This analysis will lead to our conclusion on the choice for running FFT on the GPU over FPGA.

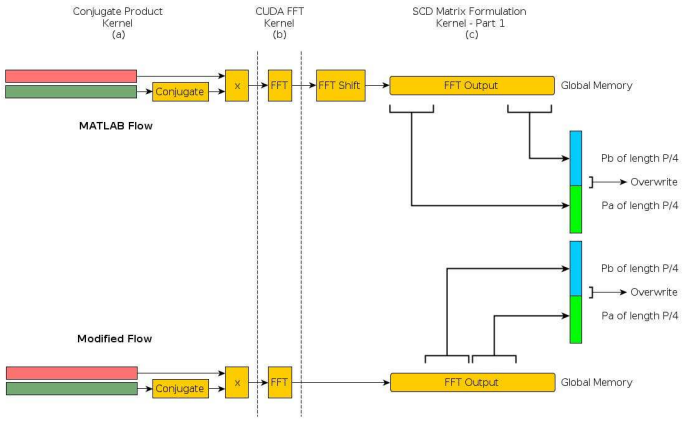

Figure 13: FFT Computation and Shift, Matlab flow (top), GPU flow (bottom).

Figure 12 shows the computation time trends for 256 and 32 point FFTs of various batch sizes. We observe that 2048 and 16,384 are the threading limits for the GPU over 256-point and 32-point FFTs respectively. For example, beyond 2048 FFTs for 256-point FFT, execution time increases linearly for every additional 2048 FFTs. 65K 32-point FFT is completed in 0.2ms on the K20 GPU.

Recently the Tegra K1 GPU (also known as the Jetson K1) targeted for mobile devices has been made available for programming with CUDA. Interestingly, the K20 and the Tegra K1 have the same multiprocessor architecture. The number of concurrent threads per multiprocessor is 2048 for both K20 and Tegra K1. The number of thread blocks that can be launched on a multiprocessor is 16.While K20 houses 13 multiprocessors, Tegra K1 houses only one multiprocessor. Table 1 shows the FFT resource usage analysis on a Virtex7 FPGA. Table 2 compares FPGA with two GPUs in terms of the execution time of a single FFT and 65K FFTs along with the precision of operations, device cost and peak power consumption features. Here we conclude that the Kepler (K20) GPU is a more desirable choice as a platform for the FFT intensive kernels of the SCD flow. Even for a high-end FPGA, such as Virtex7, the total number of FFTs (with the above specification) one can instantiate is 300 FFTs based on the available DSP blocks (Table 1). Using the remaining available soft logic blocks, number of FFTs executed concurrently could be increased (600-800 FFTs), however in such a mapping scenario during the iterative FFT stage, FPGA needs to be configured first to carry out the conjugate products and then reconfigured to execute the FFTs. Reconfiguration time overhead then would become a major concern under real-time requirement.

Step 4.3: SCD Matrix Formation- Following the FFT computation, the rows are then FFT shifted and chopped into smaller segments as shown in the serial flow (implemented in MATLAB) in Figure 13. However, FFT shift is an expensive operation. In the GPU implementation, the FFT shift stage has been

Table 1: Resource usage 32-point FFT on Virtex7 XC7VX690T

Resources	LUT	FF	BRAM	DSP
Available	433200	866400	2940	3600
FFT	2559	3094	8	12

Table 2: Virtex7, K20 and Tegra K1 FFT performance analysis.

	Virtex7	K20	Tegra K1
32-point FFT(ms)	0.002	0.05	0.05
65,536 FFTs in batches	304	4	32
Total time (ms)	0.612	0.201	1.613
Precision	Fixed	Floating	Floating
Cost($)	7,836	3,021	192
Peak Power (W)	5.68	225	5

eliminated. This re-positions Pa and Pb segments to lie in a contiguous manner (Modified flow) around the center of the FFT output vector, as opposed to lying on opposite ends (MATLAB flow). This also takes care of global memory coalescing issues on older GPUs.

The third step forms a partial SCD matrix and computes a local partial alpha profile from this matrix. While the GPU can easily store multiple SCD matrices in the global memory corresponding to multiple signals, forming such a matrix would result in numerous uncoalesced writes owing to the structure of the matrix and thus slow down the overall performance of the program. To overcome this hurdle, the entire matrix is processed by 8 threadblocks each of which brings in a separate portion of the FFT output matrix into the shared memory. The regions marked Pa and Pb (Figure 13) are then extracted from each row present in the various shared memories. To compute a local partial alpha profile, elements joined by a solid line double headed arrow (Figure 14) are replaced by the maximum of the two elements. Since these are operations within the shared memory, they are relatively quick. At the end of the kernel, every block writes its results back to the global memory.

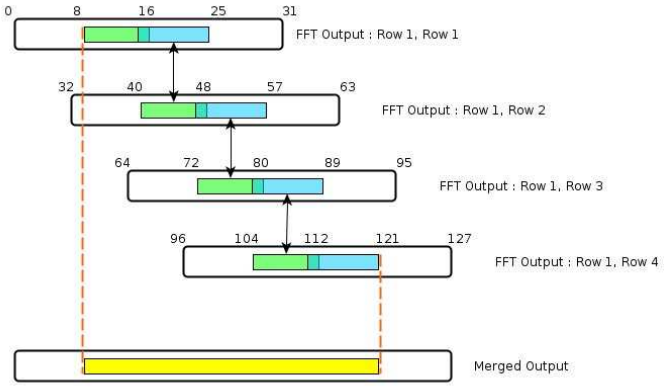

Figure 14: Comparator based partial local alpha profile generation.

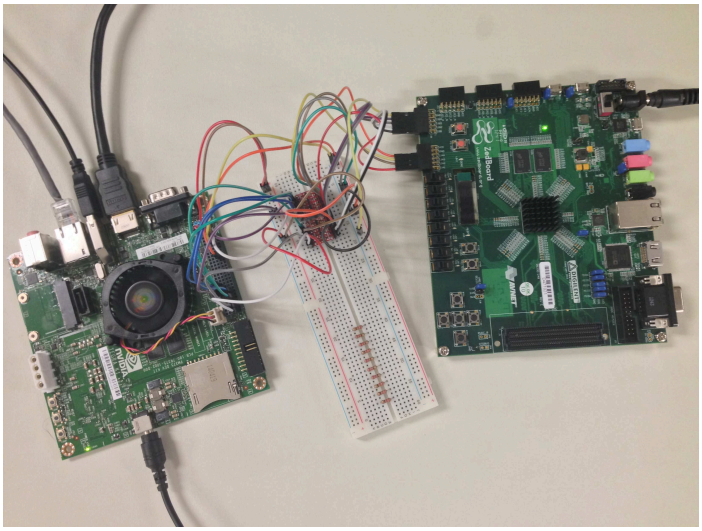

Figure 15: Serial connection with Tegra GPU and Zynq FPGA.

This brings us to the first point of inefficiency in the flow. The length of sections Pa and Pb taken together are smaller than 32, the warp size. Therefore, a fraction of the warp will operate on them to find the maximum value between sections of consecutive rows. Referring Figure 14, once Pa and Pb have been isolated, threads will be mapped to locations 8 to 25, 48 to 57, 80 to 89, 112 to 121 and so on. For a maximum block size of 1024 threads, this amounts to 264 threads (16 non-overlapping elements - 8-15 and 113-121 in Figure 14, plus 8 overlapping elements out of 16, multiplied by the number of comparisons among 32 rows (32-1)). Since this kernel was initially called for 1024 threads to extract Pa and Pb while ensuring coalesced access, 1024-264 = 760 threads go unused. This results in a maximum thread utilization efficiency of 25.78%. Note that actual efficiency will be slightly lower than this since the kernel has conditional statements within warps, not just at warp boundaries, as is necessary for efficient kernel execution. However, with this block structure, threads do write in a coalesced fashion back to the global memory after the operation has been performed.

Step 4.4:Alpha Profile Generation- The fourth step is responsible for merging the local partial alpha profiles from various thread blocks into one complete alpha profile. This kernel's operations are denoted by the bold dashed line double headed arrow in Figure 14. Once multiple local alpha profiles have been merged, its contribution is updated on the main alpha profile. At the end of all iterations, the final alpha profile will be available. Inefficiency concerns are also pertinent to the fourth kernel. Global memory sections to be merged are 8 elements in length each and strided.

Summarizing, calculations by the steps 4.3 and 4.4 are suboptimal primarily because of unused threads within a warp and branching statements within warp boundaries. In our design, we choose to give preference to coalesced global memory access over the above two possibilities

2.1. GPU based Profiling

Earlier, we concluded that based on the amount of FFT calculations required by the SCD flow, GPU was preferable over the FPGA as it offered higher degree of parallelism, even though FPGA was superior over GPU based on single FFT calculation (Table 2). Table 3 presents our profiling analysis for each stage of the SCD flow running on the GPU. This analysis has led us to a critical design decision on which kernels are suitable for the GPU and which kernels are suitable for the FPGA. In Table 3, for the "Warp Execution Efficiency" column, a value closer to 100% is desirable. A warp efficiency of 100% means that all threads in the warp were executed in parallel. Any value less than that indicates some threads were serialized. Not surprisingly, the kernels moving data around are the ones that don't utilize the GPU as efficiently. Therefore, modules that compute the alpha profile (Merge partial local alpha profile and Update main alpha profile) are more suitable for FPGA implementation as they involve memory manipulation intensive operations, which FPGAs are known to be superior for. Based on this workload partitioning, we built a prototype system that integrates Tegra K1 GPU and FPGA, where steps 1 through 4.2 are executed on the GPU, and steps 4.3 and 4.4 are executed on the FPGA.

Table 3: Stages of the SCD flow, GPU resource utilization, Number of blocks (B) and threads per block (T) on the GPU

Kernel name	Warp Efficiency %	T	B
1-Framing and Windowing	100	256	32
2-256 point FFT	100	128	4
3-FFTShift +DC+Transpose	100	1024	8
4.1-Conjugate Products	100	1024	8
4.2-32point FFT	100	128	8
4.3.1-Partial alpha profile	76.32	1024	8
4.3.2-Merge alpha profile	58.33	32	8
4.4-Update alpha profile	98.19	288	8

We consider several design choices when integrating the Tegra K1 GPU and FPGA. The working testbed is shown in Figure 15, which transfers data at 1 bit/cycle (32 cycles/element) between the Tegra GPU and FPGA. In this setup, the execution time for the FPGA tasks including data transfers from and to the FPGA is 278ms. Tegra GPU and Zynq FPGA used in this testbed have 195 and 484 pins available respectively. A design that transfers 1 element/cycle (32 pins for send, 32 pins for receive) between GPU and FPGA reduces the execution time to 45.03ms and requires 69 pins including the 5 bits needed for control signals. When we transfer 2 elements/cycle (101 pin usage), execution time reduces to 37.53ms. We can actually transfer up to 4 elements per cycle and receive 1 element per cycle resulting with a total of 165 pin usage.

Figure 14 shows iterative way of generating the alpha profile, where one pairwise comparison is executed in each cycle. Other design option is to keep increasing the comparator units. Table 4

shows the total execution time for the case of 2 elements/cycle transfer with respect to change in the number of concurrent comparison operations. Step-4 takes 6.59ms on the Tegra GPU. Based on our analysis, beyond 64-way parallelism, the benefit of parallelization is minimal.

Table 4: FPGA execution time for alpha profile generation with respect to number of concurrent comparisons

Tegra K1 (ms)	FPGA(ms) 64-bit data bus			
	1-way	16-way	32-way	64-way
6.59	37.53	6.18	5.00	4.42

Table 5 shows the benefit of transferring 4 elements/cycle over 2 elements/cycle. In the table, bit-serial design refers to sending data 1 bit at a time from Tegra GPU to FPGA. Simple parallel refers to 1 element/cycle transfer. We conclude that the final architecture should support 4 elements/cycle transfer from GPU to FPGA and the FPGA should operate with 64 parallel comparators. This results with a 1.609ms execution time including the data transfer overhead. Fortunately, pin limitation does not change the whole number of signals processed per second for the 64-way configuration. (64 way, 8 elements - 19.6 signals/sec; 64 way, 4 elements - 19.25 signals/sec). Table 6 shows the post placement and routing based timing analysis for the two kernels chosen to be implemented on the FPGA. In this implementation we target the Zynq7000 and the Block RAMs used are 36 kilobit BRAMs. For generating the post-placement and routing timing results, we set a timing constraint that specified the clock to have a rate of 140MHz. We also conclude that for these two kernels a low-end FPGA such as Zynq7000 is sufficient. A Virtex7 FPGA would result with underutilized FPGA , which is not a cost effective choice.

Table 5: FPGA execution time for alpha profile generation with respect to elements/cycle and 16-way and 64-way comparison.

Execution time (ms) for Step4: Alpha profile generation					
TegraK1	Bit serial	Simple parallel	16-way 1-elem	16-way 2-elem	64-way 4-elem
6.59	278.35	45.03	9.92	6.18	1.61
-	0.02x	0.15x	0.66x	1.07x	4.09x

3. PERFORMANCE ANALYSIS AND ARCHITECTURE DEVELOPMENT

Table 7 shows the execution times calculated for a 4096 point signal over 2 different hardware configurations. Configuration 1 (GPU-Only) involves executing SCD entirely on the K20 GPU (706MHz, 2496 Cores) or the Tegra K1 (850 MHz, 192 Cores). Configuration 2 is the GPU/FPGA partitioned SCD flow running in low-power mode (Tegra K1 and Zynq 7000). Serial (Matlab) code is executed on Intel I5 2.33 GHz 64 bit Processor with 8GB of RAM. Execution times include data transfer overhead between the GPU and FPGA. We verified that the GPU and Matlab implementations had the same output alpha profile with a minimum error of 0.0041% and a maximum error of 0.0051%. This error is due to the fact that MATLAB uses 64 bit double variables whereas the GPU implementation uses 32 bit floating point variables. Although not a user defined kernel, there is an overhead incurred for transferring data from the CPU to the GPU. In the current scenario, this is a one time cost. Transferring a 4096 point complex digital signal takes about 0.11ms.

Table 6: Post-routing resource usage and timing analysis for alpha profile generation on Zynq7000.

BUFGs	1 of 32	3%
External IOBs	81 of 220	36%
LOCed IOBs	0 of 81	0%
RAM36_EXPs	4 of 48	8%
Slices	55 of 7200	1%
Slice Registers	153 of 28800	1%
Slice LUTs	69 of 28800	1%
LUT-FF	184 of 28800	1%
Min. period	4.257ns	
Max. frequency	234MHz	

Table 7: Execution time, throughout, power consumption analysis. Serial SCD Flow (Matlab) vs. GPU (K20, Tegra K1) vs. GPU-FPGA based on 4096 points digital signal

	Serial	GPU		Hybrid
Platform	Intel I5	K20	Tegra K1	FPGA + Tegra K1
Total time(ms) (per signal)	3502.28	8.98	111.61	50.95
Speedup	-	390x	31x	58x
Signals/second	<1	111	9	19
Power (W)		51W	3.5W	5W

Running the entire flow on the K20 results with the best throughput at 111 signals per second. However, power consumption of the K20 when running the SCD is measured at 51W. It is clear from the previous discussions that data manipulation intensive kernels (steps 4.3 and 4.4) are not mappable to the GPU efficiently particularly because of incomplete warp executions and idle threads. The hybrid configuration (a single lane that couples FPGA and Tegra K1) shows the trade-off between power consumption and throughput, resulting with 19 signals per second at 5W. Therefore an architecture that houses a number of these lanes poses as a desirable solution for exploiting signal level parallelism. Based on these conclusions, we are now in a position to lay out the details of our proposed n-lane hybrid architecture.

As shown in Figure 16, we introduce a multi-lane architecture, where each lane is composed of a Tegra TK1 GPU coupled with

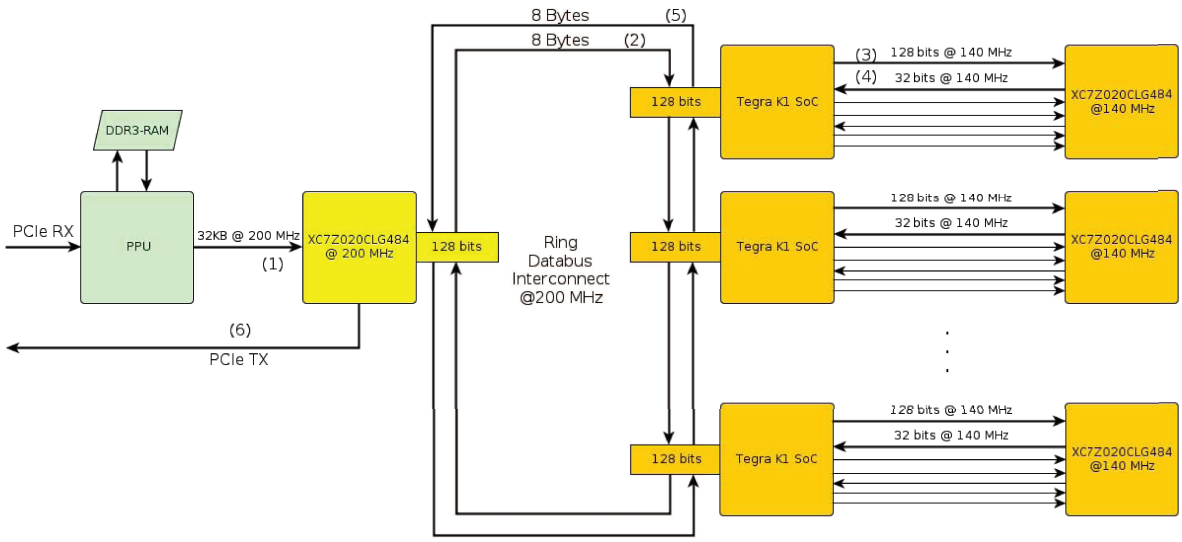

Figure 16: Tegra TK1 GPU coupled with Zyng FPGA as a single lane forming hybrid processing unit .

a Zynq FPGA. We introduce two types of bus structures. The connections between the GPU and FPGA were introduced in the earlier sections with a 4 elements/cycle data transfer from GPU to FPGA and 1 element/cycle data transfer from FPGA to GPU at the rate of 140MHz. We choose 140MHz to be in the same clock domain as the FPGA. The second interconnect structure is between the streaming FPGA and each execution lane. Here natural choice would be a star topology, since the streaming FPGA as a head node sends signal data to its designated lane. However, considering the pin count limitation on the Tegra GPU, this would result with a design that is not scalable. Ring structure is preferable, because the packetizer FPGA pin count is fixed. The overhead of round robin fashion of streaming to individual lanes turns out to be small overhead when the bus clock rate is at 200MHz. Here we summarize the execution flow by referring to the numbers shown in Figure 16.

1. Pre-processing unit (PPU) generates "n" buckets. Each bucket is storing elements of an isolated signal. This data (32KB per signal) is sent to "packetizer FPGA" (Xilinx, Zedboard, SoC with 2 ARM processors and Zynq FPGA).

2. Packetizer FPGA streams data of each signal to the designated Lane (8 Bytes/clock cycle at 200MHz). We use ring bus comprising of two channels (1 clockwise, 1 counter-clock).

3. Tegra K1 on a specific lane snoops the bus, receives the packets designated to that lane, completes GPU tasks and sends data to Zynq FPGA to execute the FPGA tasks

4. FPGA sends the results back to Tegra K1

5. Tegra K1 sends the 4096 point alpha profile back to Packetizer FPGA

Tegra K1 houses 4 ARM Cores. We utilize two of these cores in our design. First ARM Core on the Tegra K1 controls the management of signal coming into the board and its storage in the on-board RAM. This core also controls the data transfer from RAM to the global memory of the GPU and launches the GPU kernels as the host. Second ARM Core, controls the data transfer from the GPU's global memory when GPU tasks are completed to the on-board RAM and from on-board RAM to the FPGA. Since the PCB bus lies between the ARM core and FPGA with the latter running at a lower clock speed, the bus frequency will be at most equal to the FPGA clock speed, i.e. 140 MHz. Also, each transfer between the ARM core 2 and the FPGA involves sending 4 single precision floating point numbers. Therefore 16 bytes of data need to be transferred in the amount of time saved from eliminating the two kernels on the GPU. The transfer overhead for the PCI-E bus between the GPU and the RAM for 16 bytes of data is 16 bytes / 5.0443 GBps (3.17ns). The PCI-E bus speed was obtained from the profiling data on the K20 GPU system. It is likely that this speed is higher on the Tegra K1 since it is a SoC solution. Furthermore the PCI-e transfer overhead is negligible in comparison to the PCB bus speed. This is also true for the system bus between the RAM and the ARM core 2. Theoretical maximum PCB bus speed is then equal to 140 x 10^6 x 16 Bytes, hence 2240 Megabytes per second. Required bus speed based on the savings from kernel elimination is 2.38 Megabytes per second (16 bytes / 0.00671 ms). Therefore we are well within the theoretical PCB bus speed limits. Second ARM core receives the FPGA output and places it into the on-board RAM. This data is then sent to the host device of the GPU-FPGA integrated accelerator board. Considering that each lane consumes around 5W, a 10-lane architecture would consume the same amount of power as the K20 GPU. With 10 lanes, the theoretical throughput would be 190 signals/second.

4. CONCLUSION AND FUTURE WORK

In this study we discuss our parallelization approach on the GPU, present the implementation details for the entire SCD flow in CUDA, and show that on a high-end (single Tesla K20) GPU, execution time is 8.9ms (111 signals/second) with a power consumption of 51W. On the low power (Tegra-K1) GPU, we achieve 9 signals/second with a power consumption of 3.5W. We then couple Tegra K1 with Zynq 7000 FPGA as an end-to-end fully functional platform (single lane). We partition the workload as data parallel coarse-grained calculations with regular data access patterns running on the GPU and fine-grained calculations running on the FPGA. We discuss our profiling and modeling based partitioning methodology, and lane based architecture development approach. Finally we show that on a single lane, we achieve 19 signals/second with an estimated power consumption of 5W. We present the model of a scalable n-lane architecture with a dual ring bus interconnect, and show that the throughput is 190 signals/second with an estimated power consumption that is equivalent to the power consumption of Tesla K20 GPU.

We believe that there are a few avenues for improving the algorithmic efficiency of the SCD. One promising strategy is to focus on specific segments of the diamond structure (Figure 17) and evaluate whether these segments are revealing enough information or not for a successful classification. For example if the region in Figure 17 is good enough for classifying a signal, then the workload size is expected to reduce by 75% resulting with an estimated execution time reduction by a factor of 4.

Another problem to address would be to investigate whether a specific region or pattern reveals just enough information for a successful classification across all types of signals (AM, FM, BPSK, QPSK, etc) consistently or not. We believe that there is a chance of each signal type requiring different pattern. In this case, when processing a signal, our approach would involve a multi-stage evaluation. After processing the first 32KB of a signal, the signal profile analysis stage could look for a patterns or a signature that may lead to data reduction in the subsequent 32KB data for the same signal. This may lead to a solution that involves adaptive filtering by processing region of interest in the previous iteration and converge to the exact match in subsequent iterations.

5. ACKNOWLEDGMENT

This paper was supported in part by the Broadband Wireless Access and Applications Center (BWAC); NSF Award #1265960.

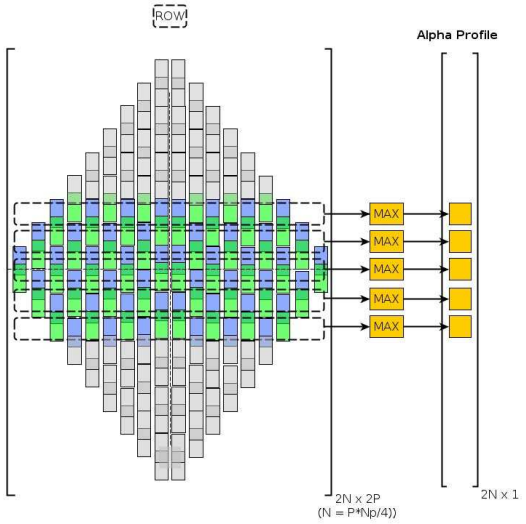

Figure 17: Signal profile generation.

REFERENCES

[1] O. A. Dobre and F. Hameed, "On performance bounds for joint parameter estimation and modulation classification," IEEE Sarnoff Symp., pp. 1-5, Apr. 2007.

[2] W. Su, F. Monmouth, J. A. Kosinski, and M. Yu, "Dual-use of modulation recognition techniques for digital communication signals," IEEE Long Island Systems, Applications and Technology (LISAT) Conference, pp. 1-6, 2006.

[3] D. Jakubisin and R. Buehrer, "Improved modulation classification using a factor-graph-based iterative receiver," Military Communications Conference (MILCOM), pp. 1-6, Oct. 2012.

[4] A. Hazza, M. Shoaib, S. A. Alshebeili, and A. Fahad, "An overview of feature-based methods for digital modulation classification," Int. Conf. on Communications, Signal Processing, and their Applications (ICCSPA), pp. 1-6, Feb. 2013.

[5] F. Hameed, O. A. Dobre, S. Member, and D. C. Popescu, "On the likelihood-based approach to modulation classifications," IEEE Trans. on Wireless Communications, vol. 8, no. 12, pp. 5884-5892, 2009.

[6] O. A. Dobre, A. Abdi, Y. Barness, and W. Su, "A survey of automatic modulation classification techniques : Classical approaches and new trends," IET Communications, vol:1, no. 2, pp. 137-156, April 2007.

[7] S. R. Schnur, "Identification and classification of ofdm based signals using preamble correlation and cyclostationary feature extraction," DTIC Document, Tech. Rep., 2009.

[8] D. Simic and J. Simic, "The strip spectral correlation algorithm for spectral correlation estimation of digitally modulated signals," in *Telecommunications in Modern Satellite, Cable and Broadcasting Services, 1999. 4th International Conference on*, vol. 1, 1999, pp. 277–280 vol.1.

[9] A. R. Castro, L. C. Freitas, C. C. Cardoso, J. C. Costa, and A. B. Klautau, "Modulation classification in cognitive radio."