From WiFi to WiMAX: Efficient GPU-based Parameterized Transceiver across Different OFDM Protocols

Rongchun Li, Yong Dou, Jie Zhou, Baofeng Li and Jinbo Xu
National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology, Changsha 410073 - China
[e-mail: {rongchunli,yongdou,zhoujie,baofengli,jinboxu}@nudt.edu.cn]
*Corresponding author: Rongchun Li

Received May 24, 2013; revised July 14, 2013; accepted August 11, 2013; published August 30, 2013

Abstract

Orthogonal frequency-division multiplexing (OFDM) has become a popular modulation scheme for wireless protocols because of its spectral efficiency and robustness against multipath interference. Although the components of various OFDM protocols are functionally similar, they remain distinct because of the characteristics of the environment. Recently, graphics processing units (GPUs) have been used to accelerate the signal processing of the physical layer (PHY) because of their great computational power, high development efficiency, and flexibility. In this paper, we describe the implementation of parameterized baseband modules using GPUs for two different OFDM protocols, namely, 802.11a and 802.16. First, we introduce various modules in the modulator/demodulator parts of the transmitter and receiver and analyze the computational complexity of each module. We then describe the integration of the GPU-based baseband modules of the two protocols using the parameterized method. GPU-based implementations are addressed to explain how to accelerate the baseband processing to archive real-time throughput. Finally, the performance results of each signal processing module are evaluated and analyzed. The experiments show that the GPU-based 802.11a and 802.16 PHY meet the real-time requirement and demonstrate good bit error ratio (BER) performance. The performance comparison indicates that our GPU-based implemented modules have better flexibility and throughput to the current ones.

Keywords: GPU, SDR, OFDM, WiFi, WiMAX.

This research is supported by National Science Foundation of China (61125201 and 61202126).

http://dx.doi.org/10.3837/tiis.2013.08.010

1. Introduction

Software-defined radio (SDR) technology is designed to support various communication standards through software configuration without altering hardware platforms. SDR technology, which is exploited in wireless communication, has two important characteristics. First, terminals should have high throughput and high computing ability to support high-rate real-time communication. Second, terminals should be flexible enough to support various communication standards. However, the realization of SDR platforms is hindered by the problem of tradeoff between performance and efficiency.

Many SDR platforms are currently based on digital signal processors (DSPs) and field programmable gate arrays (FPGAs). Although DSPs have good code flexibility, their arithmetic operation capability is not sufficient to meet the demand of real time communication. The required rate for next generation network continues to increase as wireless communication continues to undergo radio development. For example, the long-term evolution (LTE) standard in 4G communication requires the rate of several hundred Mbps. However, DSPs cannot meet the increasing speed requirement. By contrast, FPGAs provide high computation power that is required in wireless communication. However, developing FPGAs is highly complicated; developers must learn hardware description languages and gain familiarity with the development of programming and debugging tools. Although Verilog or VHDL supports parameterized designs in adherence to multiple standards, the standard practice is at the RTL level, which cannot be configured in real-time manner to apply for the SDR technology.

The SDR platform based on GPUs can overcome the problem of tradeoff between performance and efficiency. GPUs are extensively used in the fields of image processing, numeric computing, signal processing, and others. The development trend of GPUs matches Moore's Law, and peak performance can reach up to 4.58 tera floating operations per second. GPUs are inherently integrated with large on-chip memory that can run up to 6 GB, which differs from the memory capacity of commodity PCs. In signal processing, SDR algorithms require math-intensive vector operations, which are appropriate for the parallel execution mode of single instruction multiple data (SIMD) on the GPU platform. GPUs are integrated with numerous floating arithmetic units that converting floating point algorithms into corresponding fixed-point ones become unnecessary for real-time communication. Many GPU manufacturers have proposed their own programming models. For instance, the NVIDIA Corporation presents compute unified device architecture (CUDA) [1], which provides a software environment that enables developers to use C as a high-level programming language that facilitates the development of high-performance applications. SDR applications implemented on GPUs can be easily configured in real time using the parameterized method.

Although GPUs are widely used in many fields [2]-[4], such as in numeric computation, graphic processing, and data mining, their adoption in wireless communication is still at the infancy stage. Recently, studies [5]-[8] on GPU-based wireless communication systems have increased. These works have revealed that wireless communication systems based on GPUs can meet the real-time requirement of corresponding protocols and eliminate the difficulty in development using traditional hardware platforms, such as FPGAs and DSPs, because of the application of C-like language. GPUs can serve as an alternative to traditional hardware platforms based on many aspects. Nowadays, several protocols exploit OFDM as their modulation schemes, including wireless fidelity (WiFi), worldwide interoperability for

microwave access (WiMAX), and LTE. However, no study has focused on how to systematically develop the transceiver blocks in OFDM protocols on the GPU platform. In this paper, we present the generic strategy of mapping OFDM modules to GPU platforms. This strategy can be used to develop OFDM protocols on GPU-based wireless communication systems.

However, presenting this mapping strategy is challenging. First, different OFDM protocols have different physical layers. Thus, finding the common characteristic of the PHY modules of different OFDM protocols can be difficult. Second, utilizing the discovered characteristic to present the strategy is also a challenge. Finally, the throughput of these PHY modules should meet the real-time requirement of OFDM protocols. Creating a throughput for all PHY modules that exceeds real-time performance can be complicated.

This paper aims to present an efficient parallel strategy to map the PHY modules of OFDM protocols to the GPU platform. In applying this strategy, the GPU-based PHY transceiver can fulfill the following requirements. First, all OFDM-based protocols can exploit the strategy to generate the corresponding efficient GPU-based PHY modules. Second, the generated PHY modules can meet the required real-time performance.

The main contributions of this paper are as follows:

- After analyzing the complexity and commonality of the generic OFDM baseband transceiver, we introduce the parameterized implementation to allow for the reuse and customization of the baseband module across protocols, namely, IEEE 802.11a and 802.16. The proposed OFDM transceiver can be configured to realize the corresponding PHY of various protocols by altering the parameters.
- According to the dependence of multiple OFDM symbols, the modules in the OFDM baseband transceiver are originally divided into two categories: the blocked modules and the consecutive modules. Considering the memory resources on the GPU, we set the grid configuration on the GPU through the parameters and hardware resource of the specified GPU platform. When the modules in an OFDM baseband transceiver are mapped to GPUs, the developer simply needs to take the grid configuration by applying the corresponding parameters in the OFDM protocol and the corresponding GPU resource configuration.
- We present corresponding parallel strategies for the two types of modules (i.e., blocked and consecutive modules) to efficiently map the modules to the GPU platform with grid configuration. In doing so, the throughput of all modules in the OFDM protocols (i.e., IEEE 802.11a and 802.16) meets the required real-time performance.

The rest of the paper is organized as follows. Section 2 describes related works, and Sections 3 and 4 provide a background on CUDA and the generic OFDM baseband transceiver. Section 5 presents the parameterized method to integrate the baseband modules in the 802.11a and 802.16 protocols. Section 6 describes the GPU-based implementation of the OFDM transceiver. Section 7 discusses the performance evaluation of the processing modules. Section 8 concludes the paper.

2. Related Work

SDR applications, particularly wireless communication, are usually implemented on DSPs [9]-[11] or FPGAs [12]-[14]. Few studies have been devoted to SDR applications implemented on GPUs [5]-[8], [15]-[21], which can be classified into two categories.

In the first category, studies merely implement part of the PHY, which presents low practical value because a complete communication system is not achieved. For example,

Nylanden et al. [15] and Michael et al. [16] implemented MIMO detectors on a GPU. In [17]-[19] LDPC decoders were realized on GPUs. Michael et al. [20] presented a GPU-based turbo decoder and Lin et al. [21] proposed a parallel GPU-based Viterbi decoding algorithm.

In the second category, studies present an SDR communication system based on the GPU platform. In [5]-[7] real-time communication with several standards, such as DVB-T2 [5] and WiMAX [6]-[7] , was realized. Ahn et al. [7] implemented a 2x2 MIMO WiMAX system on a GPU-based platform. Recently, Ahn et al. proposed a wireless system for WiMAX and LTE standards using a GPU cluster with three nodes, archiving an average speedup of 2.6x compared with the system with only one GPU node [8]. However, the modules in each protocol are realized independently and cannot be reused in other standards, thereby extending the development cycle. The performance of these works can be improved. For example, throughput of the Viterbi decoder in one node of GPU is only 16 Mbps, which does not match the increasing speed requirements of wireless communication.

In the present study, we propose a GPU-based full configuration for wireless communication system based on two OFDM protocols, namely, 802.11a and 802.16. The system can be configured by only altering the parameters. In addition, we propose a new parallel strategy to achieve high throughput and short frame duration.

3. CUDA

The framework of CUDA comprises the logical hierarchy and the physical hierarchy, which are depicted in **Fig. 1**. In the logical hierarchy, the CUDA structure consists of a kernel, a grid, blocks, and threads. The CUDA kernel is a device program to be executed on the GPU, and it is hierarchically governed by the thread-block-grid step. Once a kernel is called by the C program on the CPU, a grid is generated on the GPU, which is concurrently executed by thread blocks with multiple parallel threads. GPU is a SIMD architecture in which multiple threads can perform a single instruction with an independent set of data. The GPU provides an identifier to distinguish each thread that should be precisely controlled by developers in SIMD processing.

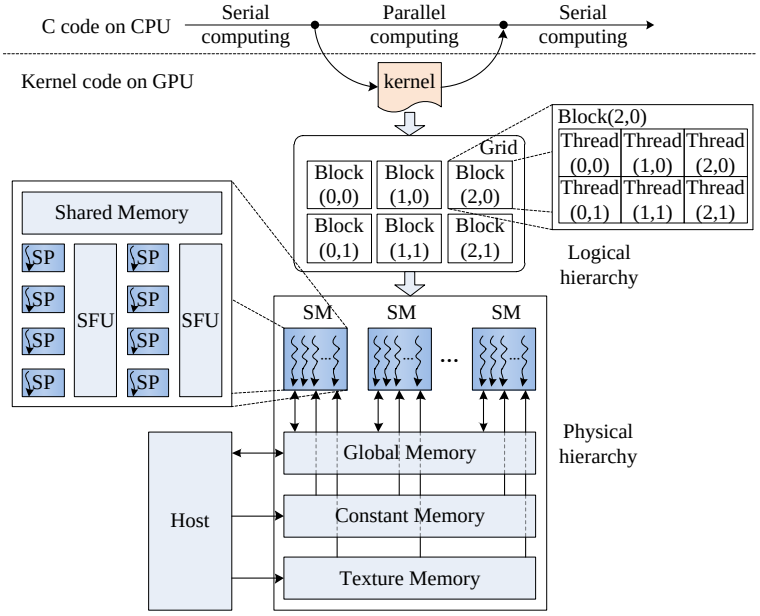


Fig. 1. CUDA Architecture

In the physical hierarchy, the CUDA structure consists of several types of memory as well as multiple stream multiprocessors (SM) with several integrated stream processors (SPs) and a few special-function units (SFUs). Four types of memory are found in the GPU as described in **Fig. 1**: global memory, constant memory, texture memory, and shared memory. Each kind of memory has its own hardware, access speed, and programming scopes. Global memory, which connects the host computer and the GPU accelerator, can be read or written by all blocks with an access latency of more than 400 cycles. Constant and texture memories are read-only memories with no latency. In a block, the most essential component is shared memory, the latency of which is four cycles. Shared memory enables threads to communicate with one another, thereby reducing the overhead incurred by accessing global memory because the registers are instantaneously used by only one thread. CUDA provides a programming interface to exploit the high parallelism of GPUs and to eliminate the complexity of controlling GPUs. CUDA bridges the logical and physical hierarchies in the manner by which the thread blocks are automatically mapped to the idle SMs.

4. Generic OFDM Baseband Transceiver

With the development of wireless communication, protocols will require high data rate. OFDM [22] offers a solution to spread data to the overlapping sub-carriers. The sub-carriers are placed orthogonal to one another, thereby producing zero cross-talk among them. Hence, OFDM provides high data rate as well as low self-symbol and inter symbol interference because of its high spectral efficiency and robustness against multipath interference. Many wireless protocols, such as WiFi and WiMAX, exploit OFDM as their modulation mode. The structure of a generic OFDM baseband transceiver is depicted in **Fig. 2**. All modules in the PHY are sequentially executed, with a buffer between each module. In the transmitter, the data received from the medium access control (MAC) layer are processed by baseband modules and then converted to analog signal by the digital-to-analog (D/A) converter; this signal is transmitted over air. In the receiver, the OFDM signal is formed through the analog-to-digital (A/D) converter and processed by the various processing modules. Finally, the data are sent to the MAC. In this study, we only focus on the parameterized implementation of the OFDM baseband processing modules on GPUs. The radio-front (RF), MAC and A/D or D/A will not be discussed due to their different issues.

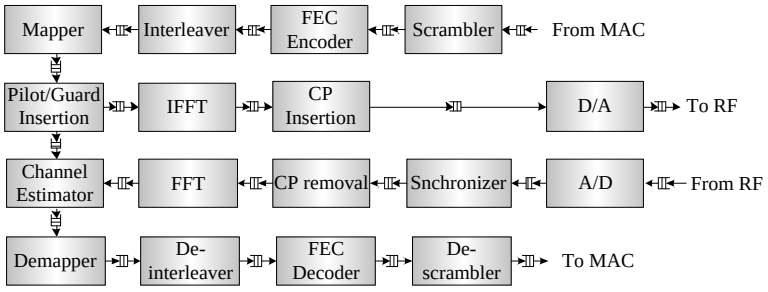


Fig. 2. Block diagram of generic OFDM baseband transceiver modules

The fundamental processing unit in the OFDM modulation is an OFDM symbol, which consists of multiple bits or samples. The samples are represented as complex numbers. In various OFDM protocols, the size of the OFDM symbol is determined by the number of sub-carriers defined in each protocol. Aside from data sub-carriers, pilot sub-carriers are used

by the estimator to achieve the frequency fading in the symbols, whereas guard sub-carriers are used to avoid interference with other carriers. In the following sections, we describe the function modules in two specific OFDM protocols, namely, 802.11a [23] and 802.16 [24]. The functions of the receiver modules are exact opposite of that of the corresponding modules of the transmitter, excluding the frequency domain modules. Thus, we mainly focus on the modules in the transmitter, except for the synchronizer, channel estimator and the forward error correction (FEC) decoder. **Table 1** shows the meaning of the clipped words used in this paper.

Table 1. Meaning of various clipped words in this article

Abbreviation	Meaning	Abbreviation	Meaning
S	number of data bits per subcarrier	N_B	number of blocks
N	number of the OFDM symbols	IS	input signal
R_{CC}	code rate of convolutional code	OS	output signal
R_{RS}	code rate of RS code	SC	scrambler
N_T	number of threads per block	CE	convolutional encoder
N_{SM}	number of SMs in the GPU	SY	synchronizer
N_{TR}	N_T limited by register count	VD	Viterbi decoder
N_{TS}	N_T limited by shared memory	DS	descrambler
N_{TM}	maximum threads per block	XOR	exclusive or
M	number of coded bits in each sub-carrier		

Scrambler: The scrambler randomizes the data bit stream to avoid long sequences of zeros and ones, thus ensuring superior results for FEC.

FEC encoder: The FEC encoder encodes the bit stream by adding redundancy to enable the decoder to detect and correct errors. In the 802.11a protocol, the coding scheme is the convolutional code. The protocol exploits the puncturing technique to archive at a high transmission rate. The 802.16 protocol has an additional FEC encoder called, the Reed-Solomon (RS) encoder, which is processed before the convolutional encoder.

Interleaver: The interleaver permutes the blocks of bits by mapping the adjacent bits into non-adjacent sub-carriers to resist burst errors. The size of the interleaving block is $S \times M$.

Mapper: The mapper maps bits to separate sub-carriers and encodes each sample by frequency, amplitude, and phase. The size of the mapper block is S; one mapper can produce S samples. The values of S in 802.11a and 802.16 are 48 and 192, respectively.

Pilot/guard (P/G) insertion: The P/G insertion inserts the pilot and guard sub-carriers. The 802.11a protocol uses a scrambler to generate a value for the pilots and a null value for the guard sample.

IFFT: The IFFT converts the OFDM symbols from the frequency domain to the time domain. The processing size of IFFT is $4/3 \times S$. The processing sizes in 802.11a and 802.16 are 64 and 256, respectively.

Cyclic prefix (**CP) addition**: The CP addition copy several samples from the end of the symbol to the front. The copied samples are called as CP. The copying process avoids inter-symbol interference caused by the multipath propagation. In this paper, the processing size of the CP addition is set to $5/3 \times S$. The size in 802.11a and 802.16 are 80 and 320, respectively. The size can be altered through parameter configuration.

Synchronizer: The synchronizer detects the beginning position of a packet based on preambles and corrects carrier frequency offset caused by the Doppler Effect. The auto-correlation algorithm is adopted to perform timing and frequency synchronization.

Channel Estimator: The channel estimator uses the pilots to estimate and compensate for

the frequency-dependent signal degradation. Among the several algorithms used for channel estimation, the two-dimensional linear interpolation was adopted.

FEC Decoder: The FEC decoder is used in decoding FEC codes. Both the 802.11a and 802.16 protocols exploit the Viterbi algorithm to decode the convolutional code. The 802.16 protocol also uses the RS decoder to decode the RS code.

In order to analyze the computational complexity of each module, we estimate the number of arithmetic logic unit (ALU) operations in each function module in the OFDM transceiver, which is shown in **Table 2**. Note that n and k are the encoding profile symbols in RS(n,k) and $Poly_i$ is the polynomial in the Galois Field. The table shows that a large percentage of the ALU operations are centralized on the modules of the synchronizer and Viterbi decoder.

Table 2. The estimated number of ALU operations in each module of 802.11a and 802.16 protocol

ALU operation	Sum/Subtract	Multiplication/Division	logic/others
Scrambler	0	0	$2\times S\times M\times R_{RS}\times R_{CC}\times N$
RS Encoder	$S\times M\times R_{CC}\times N\times (n\text{-}k)/8+(n\text{-}k)$	0	$S\times M\times R_{CC}\times N/8\times (n\text{-}k+1)\times (2+4\times \log_2(\Sigma Poly_i))$
Convolutional Encoder	0	0	$4\times S\times M\times N$
Interleaver	$4\times S\times M\times N$	$7\times S\times M\times N$	$2\times S\times M\times N$
Mapper	$S\times M\times N$	$S\times M\times N$	0
P/G Insertion	0	$14/3\times S\times N$	$1/3\times S\times N$
IFFT	$4\times S\times N\times \log_2 N$	$20/3\times S\times N\times \log_2 N$	$4/3\times S\times N\times \log_2 N$
CP Insertion	0	0	0
Synchronizer	$1285/3\times S\times N$	$5\times S\times N$	$\log_2(80\times N)+5/3\times S\times N$
CP Removal	0	0	0
FFT	$4\times S\times N\times \log_2 N$	$20/3\times S\times N\times \log_2 N$	$4/3\times S\times N\times \log_2 N$
Channel Estimator	$12\times S\times N$	$80/3\times S\times N$	$4\times S\times N$
Demapper	$8\times S\times M\times N$	$8\times S\times M\times N$	$9\times S\times M\times N$
Deinterleaver	$3\times S\times M\times N$	$8\times S\times M\times N$	$S\times M\times N$
Viterbi Decoder	$128\times S\times M\times N$	$128\times S\times M\times N$	$192\times S\times M\times N$
RS Decoder	$S\times M\times R_{CC}\times N\times (n\text{-}k)/8+(n\text{-}k)$	0	$S\times M\times R_{CC}\times N/8\times (n\text{-}k+1)\times (2+4\times \log_2(\Sigma Poly_i))$
Descrambler	0	0	$2\times S\times M\times R_{RS}\times R_{CC}\times N$

5. Parameterized Implementations on GPUs

In this section, we introduce parameterized implementation to allow for the reuse and customization of the baseband module across different protocols. The baseband modules are implemented as kernels, which are serially processed on GPUs. The buffer between modules is realized by the global memory on GPUs; the transmitter or receiver only allows two data transmissions between the host computer and the GPUs. The first is the data transmission between the MAC and the PHY. The second is the data transmission between the PHY and A/D or D/A. Each module is mapped on the GPUs to perform SIMD parallel computing, the critical issue of which is mapping the algorithm to multiple threads. The programmer should therefore clarify N_T and N_B on GPUs. The reused modules should consider the parameters of each module to support different protocols. In the following subsections, we introduce the grid configuration and the parameter setup for each module.

5.1 Grid Configuration

For most of PHY modules in OFDM protocols, the N OFDM symbols have no data dependence and can be parallel computed; examples of N OFDM symbols include the RS encoder / decoder, interleaver / deinterleaver, mapper / demapper, P/G insertion, IFFT / FFT, CP insertion / removal, and channel estimator. We call this type of module the ***blocked*** module. The parallelism potential of GPUs can be fully utilized by setting the N_B blocked modules to N; thus, each thread-block on the GPU processes an OFDM symbol. The reason is described as follows. In block-wise modules, dependence exists on the bits or samples within an OFDM symbol, thereby enabling memory access between these bits or samples. If the multiple bits or samples in a symbol are mapped to several thread-blocks on GPUs, global memory must be used to realize the crossing access between two thread-blocks. As mentioned above, access latency to global memory is considerably larger than that to shared memory or registers within an SM. The shared memory and registers in an SM can be adopted to enhance overall performance. Thus, reducing access latency necessitates distribution of multiple bits or samples to a thread-block. Furthermore, computations in an OFDM symbol require synchronization to ensure that all the bits or samples have been processed. However, the synchronization between multiple thread-blocks costs several hundred cycles. By contrast, the synchronization among threads in one thread-block is a lightweight operation that does not influence performance. Based on the above reasons, setting N_B to N is suitable regardless of the GPU resource, such as the registers or shared memory. Such setting is deemed appropriate because the two operations require more GPU cycles than the shortage of memory resource.

As for the left modules in the baseband transceiver, data dependence exists among the N symbols so that it is not suitable to directly map the algorithms on the GPU. We call this type of module the ***consecutive*** module. In the baseband modules, the scrambler / descrambler, convolutional encoder / decoder and synchronizer are the consecutive modules. In the next section, we propose a fully truncated parallel strategy to eliminate the dependence among the OFDM symbols in consecutive modules. As for the consecutive modules, grid configuration must be performed with consideration of the hardware resources on the GPU, including the registers, shared memory in each SM, maximum active threads per block and feasible SMs in GPUs. N_{TR} and N_{TS} can be obtained as:

$$N_{TS} = \frac{\text{shared memory per SM}}{\text{used shared memory per thread}} \tag{1}$$

$$N_{TS} = \frac{\text{number of registers per SM}}{\text{number of used registers per thread}} \tag{2}$$

After calculating N_{TR}, N_{TS}, and N_{TM}, N_T can be obtained by the following formula:

$$N_T = min\left(N_{TR}, N_{TS}, N_{TM}\right) \tag{3}$$

However, it is critical to ensure that the grid configuration can fully utilize all feasible SMs in the GPU. Thus, N_B can be obtained as:

$$N_T = min\left(N_{TS}, \frac{\text{number of processed bits or samples}}{N_T}\right) \tag{4}$$

If the N_B calculated by N_T is smaller than N_{SM}, N_B must be set to N_{SM}, and N_T should be computed by N_B.

Table 3 provides a list of the grid configuration of each PHY module in the 802.11a and 802.16 protocols. The N_T and N_B of blocked modules are clearly listed. The independent N OFDM symbols can be mapped to N thread-blocks. N_B of several modules is set to N+6

because of the inserted short and long preambles used for synchronization and channel estimation. By contrast, consecutive modules should consider parallel granularity and GPU resource for the reason mentioned above.

Table 3. Grid configuration of OFDM baseband modules on GPUs

Module	N_T	N_T	**Module**	N_T	N_T
Scrambler	$N_T(SC)$	$N_B(SC)$	Synchronizer	$N_T(SY)$	$N_B(SY)$
RS Encoder	$S\times M\times R_{CC}$	N	CP Removal	$4/3\times S$	N+6
Convolutional Encoder	$N_T(CE)$	$N_B(CE)$	FFT	$4/3\times S$	N+6
Interleaver	$S\times M$	N	Channel Estimator	S	N+6
Mapper	S	N	Demapper	S	N
P/G Insertion	$4/3\times S$	N	Deinterleaver	$S\times M$	N
IFFT	$4/3\times S$	N	Viterbi Decoder	$N_T(VD)$	$N_B(VD)$
CP Insertion	$5/3\times S$	N+6	RS Decoder	$S\times M\times R_{CC}$	N
			Descrambler	$N_T(DS)$	$N_B(DS)$

Table 4. Parameters setup of OFDM baseband modules on GPUs

Module	**parameter**	**802.11a**	**802.16**
Scrambler	shift register size	7	15
	linear function	x^7+x^4+1	$X^{15}+x^{14}+1$
RS Encoder	encoder profile	N/A	(12,12,0), (32,24,4), (40,36,2), (64,48,8), (80,72,4), (108,96,6), (120,108,6), (255,239,8)
Convolutional Encoder	constraint length	7	
	code rate R_{CC}	1/2, 2/3, 3/4	1/2, 2/3, 3/4, 5/6
	polynomials	(133,171)	(171,133)
Interleaver	block size	48, 96, 192, 288	192, 384, 768, 1152
Mapper	modulation type	BPSK, QPSK, 16QAM, 64QAM	
P/G Insertion	pilot indices	-21, -7, 7, 21	-88, -63, -38, 13, 13, 38, 63, 88
	guard indices	-32 to -27, 0, 27 to 31	-128 to -101, 0, 101 to 127
IFFT	size	64	256
CP Insertion	CP size	16	64
	short preamble	4 16-sample symbols	4 64-sample symbols
	long preamble	2 64-sample symbols	2 128-sample symbols

5.2 Module Parameters Setup

To allow for the reuse and customization of the baseband module across different protocols, the modules should be parameterized to support various protocols. For the 802.11a and 802.16 protocols, each baseband module should have its own parameters. **Table 4** shows the parameters for the configuration of each module in the OFDM PHY transmitter. As the function of the receiver modules is exact opposite of that of the corresponding modules of the transmitter, the parameter setting of the modules in the receiver is the same as that of the

transmitter. In each module, most of the input parameters, such as the S, the FFT size, CP size, and R_{CC}, can be identified by the threads through grid configuration. The configuration of some parameters, including those mentioned preciously, requires minor operation. For example, in the convolutional encoder, the polynomial determines the positions of the XOR registers. Note that the polynomials in the 802.11a protocol are reverse to those in the 802.16 protocol. Therefore, configuring the parameter of the polynomial only requires the interchanging operation at the end of the encoder. Finally, the remaining parameters in the specific modules can be configured using the if-else clauses of the C language. This method is convenient for the programmer. Using the parameter configuration, each module implemented in CUDA can easily support both the 802.11a and 802.16 protocols.

6. Efficient GPU-based Parallel Strategy

As discussed above, data dependence exists among the N OFDM symbols of the consecutive modules. These symbols cannot be parallel computed by the multiple thread-blocks in GPUs. In this section, we propose a fully truncated parallel strategy to excavate the parallelism potential of the consecutive modules. The fully truncated parallel strategy divides the signal sequence into several chunks, each of which can be processed independently. In the following subsections, we expound on several modules, such as the scrambler, convolutional encoder, synchronizer and the Viterbi decoder to illustrate the proposed strategy. In our implementation, the GPU type is GTX580, which comprises 16 SMs, 512 CUDA cores, 49kB shared memory, and 32 k registers. The maximum number of threads per block is 1024.

6.1 Scrambler

Both the scrambler and descrambler XOR each input bit with a pseudo-random binary sequence to randomize the input bit stream. Linear feedback shift registers (LFSR) defined by the linear function serially generates the pseudo-random binary sequence (**Table 4**). For example, in the 802.11a protocol, the scrambler XORs the 1st input bit with the 4th and 7th bits of the LFSR to obtain the output bit. Subsequently, the new LFSR is computed by shifting the feedback bit of the LFSR in the right direction. The next output bit of the new LFSR is calculated by XORing the 2nd bit and the 4th and 7th bits. Thus, the scrambler / descrambler cannot be parallel computed directly due to the data dependence of LFSR. For the purpose to map the (de)scrambler to the GPUs, we present truncated (de)scrambler to touch the aim. The input bit stream of scrambler is divided into multiple chunks, each of which can process a short scramble operation. With regard to the descrambler, the same partition is applied to recover the bit stream. The length of each chunk is set as the length of LFSR, and labeled as L. Each thread requires L registers. Based on **Equations 3** and **4**, N_B is set to 16.

6.2 Convolutional Encoder

Both protocols exploit convolutional coding as the code scheme, which has a constraint of 7 and a code rate of 1/2. The encoded code words are punctured into a higher code rate to achieve a higher data rate. Each input bit produces two output bits through a modulo-2 operation that involves six shift registers, which depend on the input bit and consequently affect the output bits. Every coded bit output is relevant to seven bits, six of which are shift registers and the remaining one is the input bit. The input signal sequence is parallel organized into multiple chunks. Each chunk, comprising seven bits, is assigned to a thread that functions as a sub-encoder of the seven bits and produces two coded bits. The threads can be parallel

executed because of data independence. Finally, the coded bits produced by the threads are stored in the correct addresses in the global memory, which is fed for the next module in the PHY. Algorithm 1 provides the parallel computation of the convolutional encoder on the GPU. In this algorithm, each thread does not require shared memory or registers. Thus, $N_T(CE)$ can be set directly to $N_T(TM)$.

Algorithm 1 GPU-based convolutional encoding algorithm

Initialization:
IS is parallel organized into $S \times M \times N$ chunks,
each chunk comprises 7 bits, having 6 bits overlapped with the next one
Iteration:
for thread-block b 0 to $S \times M \times N / N_T(TM) - 1$ **parallel do**
  for thread t 0 to $N_T(TM) - 1$ **parallel do**
    $i = b \times N + t$;
    Load $IS_{i,0}$ to $IS_{i,6}$ from the global memory;
    $OS_{i,0} = IS_{i,0} \wedge IS_{i,2} \wedge IS_{i,3} \wedge IS_{i,5} \wedge IS_{i,6}$;
    $OS_{i,1} = IS_{i,0} \wedge IS_{i,1} \wedge IS_{i,2} \wedge IS_{i,3} \wedge IS_{i,6}$;
    if WiMAX **then**
      Exchange($OS_{i,0}$,$OS_{i,1}$);
    end if
    Store the final result $OS_{i,0}$ and $OS_{i,1}$ in the global memory;
  end for
end for

6.3 Synchronizer

Synchronization pertains to the detection of the initial frame to acquire the start point and frequency offset of the initial frame. The initial frame is used to maintain the time and frequency synchronization for every frame. In the initial frame detection, the autocorrelation algorithm is used to calculate the frequency offset. **Fig. 3** shows the course of the autocorrelation algorithm. On the one hand, a window with a length that corresponds to the number of FFT points passes through the input signal. On the other hand, a similar window with a fixed distance to the first window moves forward through the input signal. The correlation between the two windows is calculated in each step. The correlation achieves it's the maximum value when the two windows move in the entire frame interval. With the maximum value, the start point and the frequency offset can be obtained.

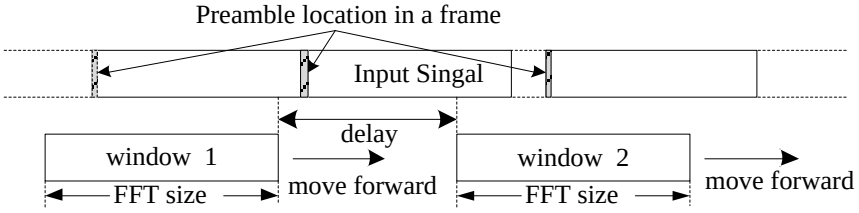


Fig. 3. The autocorrelation algorithm used to frame detection

The primary purpose of the autocorrelation algorithm is to compute the correlation between the two windows at each sample time interval as well as to extract the maximum value for the frequency offset calculation. The input signal sequence is parallel organized into multiple chunks, each of which comprises bits in two windows. Each chunk is assigned to a thread that

calculated the correlation at a sample time interval. The threads can be parallel executed because of data independence. After computing correlation for each thread, a parallel reduction mode of an inverse binary tree is employed to calculate the maximum value as described in **Fig. 4**. At each comparison step, lightweight synchronization occurs among the threads, which does not influence performance. The temporary results are stored in the threads with sequential identifiers. To avoid the bank conflict in the shared memory in each SM, parallel reduction with sequential addresses is performed. Parallel reduction minimizes the complexity of the maximum calculation from O(n) to O($\log_2 n$). After the maximum value of each thread-block is obtained, the values are stored in sequential addresses in the global memory. Subsequently, the same parallel reduction is performed to compute the global maximum value, which can generate the frequency offset for the next module in the PHY. To perform the parallel reduction mode of the inverse binary tree, each thread requires 64 bits of shared memory and three registers. Based on **Equation 3**, $N_T(SY)$ can be set to 512.

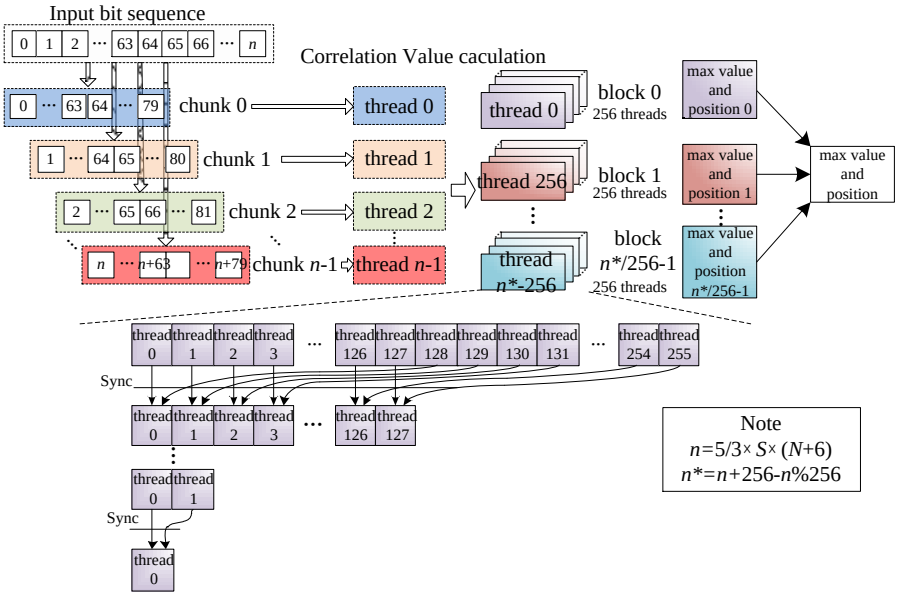
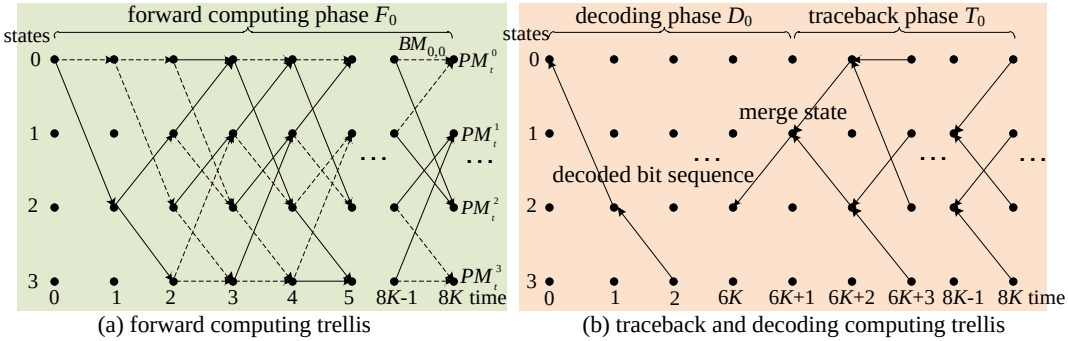


Fig. 4. Conceptual diagram of parallelizing synchronization algorithm



(a) forward computing trellis (b) traceback and decoding computing trellis

Fig. 5. Trellis diagram of Viterbi decoder

6.4 Viterbi Decoder

The Viterbi decoding process can be expressed as a trellis diagram (**Fig. 5**). The state transition diagram is repeated from left to right for a number of time stages equal to the

number of information bits. The procedure of the Viterbi algorithm can be classified into two directions: the forward procedure and the trace-back procedure. The trace-back procedure is used to identify the maximum likelihood path through the trellis. For this purpose, a metric called path metric (PM) is maintained for each possible path through the trellis. For each time stage, at each state, only the path with the minimum PM is retained and transferred to its consumer in the next stage. This action requires an add-compare-select (ACS) operation for each state in the trellis. The survived bits (SBs) are determined for the trace-back procedure using the ACS operation. After the forward creation of the trellis, the algorithm begins to trace backward to identify the maximum likelihood path using SBs. The SBs along this path are given as decoded bits (DBs).

In forward computing in the trellis depicted in **Fig. 5(a)**, the ACS operations among the states exhibit no dependence along the state-axis; hence, these operations can be parallel processed. However, along the time axis, the PM at time t depends on the PM at time t-1. Fortunately, all survivor paths merge into one state after a suitable truncation length. We can then trace back from the merging state to obtain the decoded sequences. **Fig. 5(b)** illustrates the procedure in the trellis. This procedure is knows as three-point Viterbi algorithm [25]. The Viterbi decoder can thus be implemented through the truncation method, which partitions the trellis along the time axis. Each sub-trellis overlaps with the adjacent sub-trellis for the L_t symbols, which are the tail sequence used to perform the track-back phase, in which the merging state is produced. Each sub-trellis can be divided into two phases, namely, the forward computing phase and the trace-back and decoding (TD) phase

Fig. 6 depicts the conceptual diagram of parallelizing the Viterbi algorithm. F_i denotes the forward computing phase of the i^{th} truncation sub-trellis. TD_i represents the trace-back and decoding phase of the i^{th} truncation sub-trellis. To decode the bits precisely, the partition length of F_i is set to $2(L_d+L_t)$, where L_d and L_t are the lengths of the decoding phase and the trace-back phase, respectively.

In the forward computing phase, each ACS operation is assigned to each thread, where the values of BM, PM, and SB are calculated. The network between successive time stages is achieved in the shared memory in each thread-block. Each thread writes the PM and SB values into the corresponding address in the shared memory. Subsequently, lightweight synchronization is carried out among the threads to ensure that all the threads complete the computing and writing operations in a time stage. The correct input PM value can be fetched from the shared memory after synchronization.

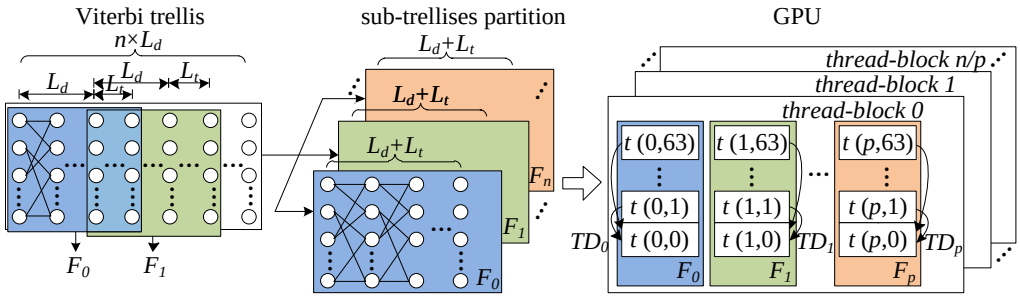


Fig. 6. Conceptual diagram of parallelizing Viterbi algorithm

After the completion of the forward computing phase, the trace-back and decoding phase of the i^{th} truncation sub-trellis is conducted in the first thread of TD_i. A divide-and-conquer method is employed to select the minimum PM value in the final states in each F_i as the trace-back state. After tracing L_t bits, the merging state is determined. Finally, the left L_d bits

are the decoded bits of the i^{th} truncation sub-trellis. The input signal is partitioned into $S \times M \times N / L_d$ chunks, each of which has $2(L_d+L_t)$ bits. Each thread requires 64 bits of shared memory and 8 registers. Based on Equation 3, N_T is set to 512 in the GTX580 implementation, wherein 8-way parallel F_i computing can be performed.

7. Performance Evaluation

7.1 Experiment Setup

We adopt four modulation types for the 802.11a and 802.16 protocols. For each modulation type, we use high code rates as examples. The R_{RS}, R_{CC}, M and S of each modulation are listed in **Table 5**. The maximum frame length is 12694 bits, which comprise 12672 data bits, 16 service parameter bits, and 6 encoding tail bits. The transmitter and receiver of the OFDM PHY are implemented in two segregated GPUs. The GPU type used is GTX580. A total of 16 SMs, each with 32 SPs, form the architecture of 512 CUDA cores. The GTX580 has a frequency of 1.54 GHz and a global memory size of 1.5 GB. Each thread-block can support 1024 threads for parallel execution. After the signal processing of the transmitter, the signal is sent back to the host computer and then sent to the simulated additive white Gaussian noise (AWGN) channel. The signal added noise is transmitted to the GPU in the receiver for the baseband processing.

Table 5. Parameters of modulation and coding in 802.11a and 802.16 standard

Protocol	Modulation	R_{RS}	R_{CC}	M	S
802.11a	BPSK	N/A	3/4	1	48
	QPSK	N/A	3/4	2	96
	16QAM	N/A	3/4	4	192
	64QAM	N/A	3/4	6	288
802.16	BPSK	24/32	3/4	1	192
	QPSK	48/64	3/4	2	384
	16QAM	96/108	3/4	4	768
	64QAM	239/255	5/6	6	1152

7.2 Data Throughput

To measure the processing time of the PHY algorithms on GPUs, we use the profiler provided by NVIDIA Corporation. We decompose the algorithm modules in both the transmitter and the receiver and measure the time cost by each module. **Table 6** shows the processing times and throughput of the modules in the 802.11a and 802.16 PHY for preforming one frame transmission on the GPU. The throughput is calculated by dividing the number of processed bits or samples per module by the processing time. Note that the IFFT and FFT modules are implemented by the CUBLAS function library [26] presented by CUDA.

In **Table 6**, we can find that in a protocol, the processing time of each module with four modulation types is equivalent, except for the synchronizer. This result can be attributed to the fact that the total amount of processed signal is equivalent and that data dependence among the thread-blocks is eliminated. All input OFDM symbols can be distributed to idle blocks for parallel computing, in which the bits or samples are mapped to multiple threads. When the CUDA cores are sufficient, the processing time of all bits or samples is equal to that of a single bit or sample. As for the blocked modules, the number of symbols is equal to the required number of thread-blocks, when the CUDA cores are not enough to process all the symbols for

parallel computing, the processing time increases. Therefore, the decrease in the number of symbols results in the decrease in the processing time of each blocked module.

In other cases, some modules still demonstrate data dependence among the thread-blocks. For example, the synchronizer fetches the maximum value from all the samples within a given frame. The thread-blocks in the synchronizer should wait for all values to be processed completely. The maximum value is then fetched from the multiple thread-blocks. Thus, a decrease in the number of symbols results in a decrease in the number of thread blocks that require synchronizing by the GPU, thereby shortening the processing time.

The memory copy time shown in **Table 6** is the memory copy time between the GPU and the CPU. the memory copy time ranges from 64 to 82 us per frame, depending on the data size. The memory copy time occupies about 4% to 9% of the total time. The processing time for one frame in the PHY decreases with the increase in the transmitter rate, thus shortening the processing time of the synchronizer.

Table 6. The execution time and throughput of modules in 802.11a and 802.16 PHY on GTX580 GPU, measure format:time(us)[throughput(Mbps or Msps)]

Module	**802.11a**				**802.16**			
	BPSK	QPSK	16QAM	64QAM	BPSK	QPSK	16QAM	64QAM
Scrambler	8.9 [1424]	8.8 [1440]	8.7 [1457]	8.9 [1432]	9.3 [1370]	8.5 [1499]	8.6 [1488]	9.0 [1500]
RS Encoder	N/A	N/A	N/A	N/A	130.2 [131]	145.8 [117]	152.2 [95]	164.5 [88]
Convolutional Encoder	48.1 [527]	47.2 [537]	47.0 [539]	46.8 [545]	52.4 [649]	52.5 [647]	51.2 [563]	50.5 [570]
Interleaver	10.2 [1656]	9.5 [1779]	10.2 [1656]	10.1 [1682]	10.1 [2243]	10.6 [2137]	10.6 [1811]	10.6 [1631]
Mapper	8.4 [2011]	7.8 [1083]	7.9 [535]	8.2 [345]	8.4 [2697]	7.9 [1434]	7.7 [623]	9.7 [297]
P/G Insertion	28.0 [984]	26.4 [514]	24.8 [234]	23.8 [170]	28.3 [1213]	25.4 [637]	24.5 [243]	24.2 [225]
IFFT	22.9 [525]	21.9 [269]	24.1 [128]	22.2 [89]	24.9 [673]	23.7 [346]	26.3 [138]	17.1 [225]
CP Insertion	12.7 [2217]	12.7 [1164]	10.4 [677]	10.4 [454]	12.9 [2927]	11.6 [1628]	10.7 [748]	12.8 [375]
Synchronizer	163.4 [172]	98.6 [143]	57.5 [122]	54.0 [87]	712.8 [53]	385.0 [49]	170.7 [47]	145.1 [33]
CP Removal	10.9 [2583]	10.1 [1394]	8.7 [809]	8.7 [543]	11.0 [3433]	10.1 [1869]	9.3 [860]	9.2 [522]
FFT	24.1 [935]	22.6 [498]	23.1 [244]	23.1 [163]	24.5 [1233]	24.5 [616]	24.4 [262]	18.9 [203]
Channel Estimator	48.3 [466]	40.4 [257]	40.4 [139]	40.0 [94]	48.5 [623]	42.3 [357]	39.5 [162]	39.0 [98]
Demapper	10.3 [1640]	12.6 [880]	12.6 [335]	12.4 [228]	9.9 [2288]	9.5 [1192]	10.0 [480]	12.9 [223]
Deinterleaver	10.7 [1579]	10.5 [1724]	10.5 [1609]	10.6 [1603]	10.1 [2243]	10.4 [2178]	11.2 [1714]	13.3 [1300]
Viterbi Decoder	225.1 [113]	219.5 [115]	201.7 [126]	198.4 [128]	343.4 [99]	332.3 [102]	325.9 [88]	310.3 [93]
RS Decoder	N/A	N/A	N/A	N/A	131.4 [129]	143.6 [118]	153.2 [94]	161.5 [89]

Descrambler	9.1 [1393]	8.3 [1527]	8.8 [1440]	9.2 [1385]	9.2 [1385]	8.4 [1517]	8.7 [1471]	9.1 [1484]
N	352	176	88	59	118	59	25	15
Memcpy time(us)	64.4	64.0	64.1	64.6	81.8	80.7	71.7	68.7
Frame time(us)	705.5	620.1	559.5	551.4	1659.1	1332.8	1116.4	1086.4
FTWS(us)	542.1	521.5	502.0	497.4	946.3	947.8	945.7	941.3

By contrast, the frame time without synchronizer (FTWS) in **Table 6** is somehow equal for all modulation modes. As we known, the synchronizer only works in the first frame of a transmitting packet. After the first frame, the frame processing time will thus be less than 1ms, falling within the frame duration of the 802.11a and 802.16 protocols. **Fig. 7** and **Fig. 8** show the module throughput relative to the required throughput for real-time performance in the 802.11a and 802.16 protocols, respectively. Some modules can archive more than 477 times throughput to the requirement throughput in the 802.16 protocol. The slowest block is the Viterbi decoder, with 178% of the target data rate in the 64QAM modulation mode, and with the code rate of 3/4 in the 802.11a protocol. Thus, all processing modules can meet the real-time requirement. Furthermore, **Fig. 7** and **Fig. 8** show that the speedup for the required throughput in each module decreases along with the increase in the transmitting rate configuration. This result can be attributed to the fact that the processing throughput of each module is steady as the required throughput increases.

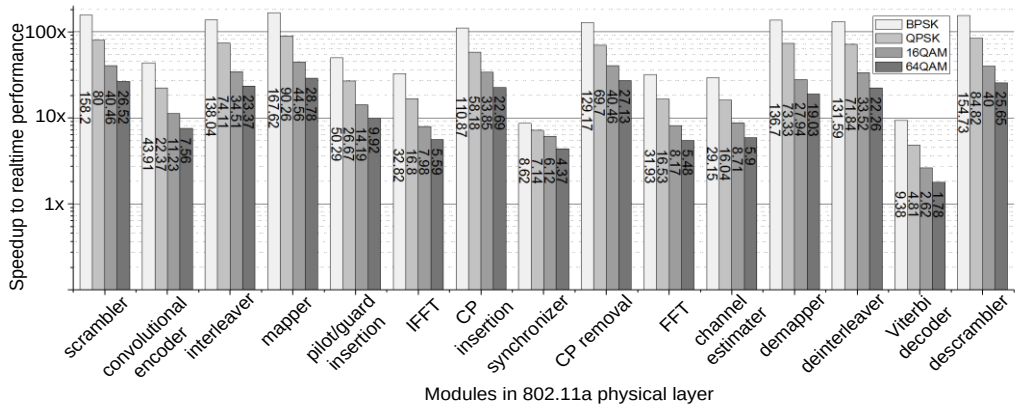


Fig. 7. 802.11a Module throughput relative to required throughput for real-time performance

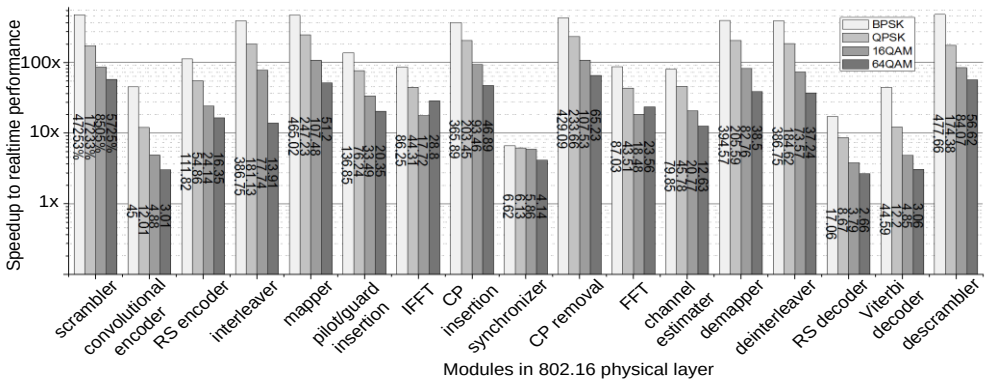


Fig. 8. 802.16 Module throughput relative to required throughput for real-time performance

7.3 BER performance

To evaluate the BER performance of the GPU-based PHY modules, we measure the BER of our 802.11a and 802.16 PHY under the AWGN channel in the simulation, which is shown in **Fig. 9**. The BER of proposed GPU-based PHY is close to the theoretical BER performance in the Matlab simulation. This result proves the practicality of the GPU-based 802.11a and 802.16 PHY.

7.4 Performance Comparison

In this section, we will compare the performance of our GPU-based algorithms with the current GPU, DSP and FPGA-based implementations. We also discuss the different characteristics among the various platforms and the reason of the performance gap of them.

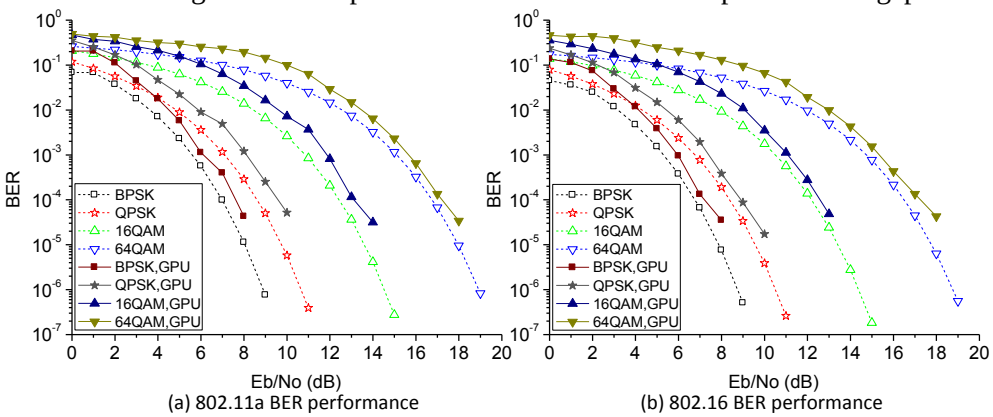


Fig. 9. BER performance of GPU-based 802.11a and 802.16 PHY under AWGN channel

7.4.1 Comparison with Other GPU-based Implementations

In order to compare the performance of the algorithms in this study with that of other GPU-based implementations, we list the performance of the modules in the 802.16 PHY described in [7]. In [7], the modules with 16QAM modulation and 1/2 code rate were implemented. **Table 7** provides a comparison of the performance of the GPU-based modules in [7] and those in the current study in 16QAM modulation of the 802.16 protocol.

Table 7. Performance comparison between different GPU-based implementation of SDR algorithms

	Throughput(Mbps/Msps)			**Speedup**	
	[7]	Ours	Ours	Ours vs.[7]	Ours vs.[7]
platform	GTX275	GTX275	GTX580	GTX275	GTX580
Conv. Encoder	24.5	270.1	562.5	11.0	23.0
Interleaver	97.7	862.5	1811.3	8.8	18.5
Mapper	98.8	301.2	623.4	3.0	6.3
P/G Insertion	139.5	172.4	261.2	1.2	1.9
IFFT	72.4	73.3	243.3	1.0	3.4
Synchronizer	7.7	23.8	46.9	3.1	6.1
FFT	74.1	74.2	262.3	1.0	3.5
Cha.Estimator	43.2	76.9	162.0	1.8	3.8
Deinterleaver	92.4	810.3	1714.3	8.8	18.6
Viterbi Decoder	16.2	45.5	88.4	2.8	5.5

For a fair comparison of the result, we also implement our modules on the GTX275 GPU, which has 240 CUDA cores. On the same GPU platform, our modules archive a speedup of about 1.8x-11.0x compared with the modules in [7], except for the following three modules. The FFT/IFFT modules have approximately the same performance because similar to [7], both are implemented by the CUBLAS function library. The module of P/G insertion attains only a speedup of 1.2x because it involves a serial operation, which has low parallel potentiality. Our modules obtain better throughput than that in [7] because we fully exploit the parallelism of the algorithms. T For instance, the Viterbi decoding algorithm proposed in the current study is a fully parallel truncated Viterbi algorithm. Hence, the entire trellis is partitioned to form multiple sub-trellises, and the forward and track-back phases of each sub-trellis are executed independently. By contrast, the Viterbi algorithm in [7] only adopts one thread-block for the forward computing phase, wherein each thread calculates one ACS operation. In addition, the trace-back procedure is performed in a thread. Compared with that in [7], the degree of parallelism in our modules is larger, and can produce a speedup of 2.8x. On the GTX580 GPU, we can archive an improved speedup of about 1.9x to 23.0x because of the large number of CUDA cores. Based on the information listed on **Table 7**, we can conclude that our implementation is superior to that in [7].

7.4.2 Comparison with DSP Implementations

In this section, we discuss the differences between the GPU and DSP implementations of SDR algorithms. As an example, in [10] the receiver of 802.11a protocol was implemented at AsAP2 platform, which is a DSP chip multiprocessor with a clock frequency of 590MHz. The performance comparison of each module under DSP and GPU implementations is shown in **Table 8**. The throughput of our GPU-based modules can archive an improved speedup of 2.5x to 31.3x compared with that in [10], except for the FFT module, which is realized by the DSP library [27]. Although the DSP has a high clock frequency, the number of concurrent instructions is limited and depends on the architecture of the DSPs. Some modules with great parallelism potential, such as the channel estimator, demapper, or deinterleaver, cannot be totally parallel executed in the DSP platform. Meanwhile, many CUDA cores in the GPU facilitate the parallel computation of the OFDM symbols; such computation may result in an enhanced throughput.

Table 8. Performance comparison between DSP and GPU implementation of modules for the 54Mbps mode in 802.11a receiver

Module	**Throughput(Mbps/Msps)**		**Speedup**
	DSP[10]	GPU[Ours]	Ours vs.[10]
Synchronizer	6	87	14.5
CP Removal	214	543	2.5
FFT	184	163	0.9
Cha.Estimator	3	94	31.3
Demapper	12	228	19.0
Deinterleaver	85	1603	18.9
Viterbi Decoder	43	128	3.0
Descrambler	59	1385	23.5

7.4.3 Comparison with FPGA Implementations

In this section, we discuss the differences between the GPU and FPGA implementations of SDR algorithms. As a representative example [12] realized the 802.11a transmitter on the Xilinx Virtex-II Pro XC2VP50 FPGA, using the multi-clock pipelined scheme. In the 54Mbps mode, modules in the PHY can be divided into four stages of clock area, each of which has independent clock frequency. The throughput of the modules of the transmitter both in [12] and in our implementation is shown in the **Table 9**. Compared with the FPGA implementation, GPU-based modules can achieve superior throughput with a speedup ranging from 2.0x to 14.4x. However, the FPGA can adopt the pipelined technology due to its architecture. The latency of transmitting one bit or sample through all the modules is only 57us. Meanwhile, the entire signal in a GPU-based module should be processed completely before being passed to the next module. This requirement results in latency that is equal to the processing time for a whole frame. **Table 9** shows that the latency of the GPU-based transmitter is 2.3 times that of the FPGA implementation.

Table 9. Performance comparison between DSP and GPU implementation of modules for the 54Mbps mode in 802.11a receiver

Module	**Throughput(Mbps/Msps)**		**Speedup**
	FPGA[12]	GPU[Ours]	Ours vs.[12]
Scrambler	126	1432	11.4
Conv. encoder	126	545	4.3
Interleaver	117	1682	14.4
Mapper	117	345	2.9
P/G insertion	57	159	2.8
IFFT	57	170	3.0
CP Insertion	226	454	2.0
latency(us)	57	130	2.3

8. Conclusion

In this study, generic baseband modules are implemented using GPUs for two different OFDM protocols, namely, 802.11a and 802.16. An efficient strategy is presented to map the PHY modules of two OFDM protocols to the GPU platform. A parameterized method is employed to allow for the reuse and customization of the baseband module across different protocols. The modules are distributed on the numerous CUDA cores of the GPU. To ensure the flexibility of the modules, the grid configuration of each module is set according to the parameters and the hardware resource of the specified GPU platform. According to the dependence of multiple OFDM symbols, the modules in the OFDM baseband transceiver are initially divided into two categories: the blocked module and the consecutive module. To eliminate data dependence among the OFDM symbols, a fully truncated parallel strategy is presented. Through the efficient strategy, the OFDM-based protocols can generate the corresponding efficient GPU-based PHY modules. Finally, the 802.11a and 802.16 PHY are implemented in all modulation configurations and the performance of the GPU-based modules is evaluated. The experiments show that the modules implemented on the GPUs can attain throughput that exceeds the real-time requirement and satisfactory BER performance. The performance comparison of various algorithms shows that our GPU-based parallel algorithm for each module is superior to other GPU-based implementations. The GPU can thus serve as

an alternative to the traditional DSP and FPGA solutions for wireless applications, particularly in simulations and software-defined wireless test beds.

References

[1] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide version 4.0," 2011.

[2] C. Yang, Q. Wu, T. Tang, F. Wang, and J. Xue, "Programming for scientific computing on peta-scale heterogeneous parallel systems," *Journal of Central South University*, vol. 20, no. 5, pp. 1189-1203, May, 2013. Article (CrossRef Link).

[3] X. Yang, T. Tang, G. Wang, J. Jia, and X. Xu, "MPtostream: an OpenMP compiler for CPU-GPU heterogeneous parallel systems," *Science China-information Sciences*, vol. 55, no. 9, pp. 1961-1971, September, 2012. Article (CrossRef Link).

[4] C. Yang, Q. Wu, H. Hu, Z. Shi, J. Chen, and T. Tang, "Fast weighting method for plasma PIC simulation on GPU-accelerated heterogeneous systems," *Journal of Central South University*, vol. 20, no. 6, pp. 1527-1535 , June, 2013. Article (CrossRef Link).

[5] S. Gronroos, K. Nybom and J. Bjorkqvist, "Complexity analysis of software defined DVB-T2 physical layer," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 131-142, December, 2011. Article (CrossRef Link).

[6] J. Kim, H. Seungheon and C. Seungwon, "Implementation of an SDR system using graphics processing unit," *IEEE Communication Magazine*, vol. 48, no. 3, pp. 156-162, March, 2010. Article (CrossRef Link).

[7] C. Ahn, J. Kim, J. Ju, J. Choi, B. Choi and S. Choi, "Implementation of an SDR platform using GPU and its application to a 2x2 MIMO WiMAX system," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2, pp. 107-117, December, 2011. Article (CrossRef Link).

[8] C. Ahn, S. Bang, H. Kim, S. Lee, J. Kim, S. Choi, and J. Glossner, "Implementation of an SDR system using an MPI-based GPU cluster for WiMAX and LTE," *Analog Integrated Circuits and Signal Processing*, vol. 73, no. 2, pp. 569-582, November, 2012. Article (CrossRef Link).

[9] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. A. Lai, J. W. Webb, E. W. Work, T. Mohsenin, and B. M. Baas, "Architecture and evaluation of an asynchronous array of simple processors," *Journal of Signal Processing Systems*, vol. 53, no. 3, pp. 243-259, December, 2008. Article (CrossRef Link).

[10] A. T. Tran, D. N. Truong, and B. M. Baas, "A complete real-time 802.11a baseband receiver implemented on an array of programmable processors," in *Proc. of 42nd Asilomar Conference Signals, Systems and Computer,* pp. 165-170, October 26-29, 2008. Article (CrossRef Link).

[11] H. Lee, C. Chakrabarti, and T. Mudge, "A low-power DSP for wireless communications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 9, pp. 1310-1322, September, 2010. Article (CrossRef Link).

[12] M. Mizani, and D. Rakhmatov, "Multi-clock pipelined design of an IEEE 802.11a physical layer transmitter," in *Proc. of 20th International Parallel and Distributed Processing Symposium*, pp. 21-27, April 25-29, 2006. Article (CrossRef Link).

[13] J. S. Park and T. Ogunfunmi, "Efficient FPGA-Based Implementations of MIMO-OFDM Physical Layer," *Circuits Systems and Signal Processing*, vol. 31, no. 4, pp. 1487-1511, August, 2012. Article (CrossRef Link).

[14] M. J. Canet, J. Valls, V. Almenar and J. Marin-Roig, "FPGA implementation of an OFDM-based WLAN receiver," *Microprocessors and Microsystems*, vol. 36, no. 3, pp. 232-244, May, 2012. Article (CrossRef Link).

[15] T. Nylanden, J. Janhunen, O. Silven and M. Juntti, "A GPU implementation for two MIMO-OFDM detectors," in *Proc. of International Conf. Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 293-300, July 19-22, 2010. Article (CrossRef Link).

[16] M. Wu, Y. Sun, S. Gupta and J. R. Cavallaro, "Implementation of a high throughput soft MIMO detector on GPU," *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 123-136, July, 2011. Article (CrossRef Link).
[17] G. Falcao, L. Sousa and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309-322, February, 2011. Article (CrossRef Link).
[18] H. Ji, J. Cho and W. Sung, "Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPGPU," *Journal of Signal Processing System*, vol. 64, no. 1, pp. 149-159, July 2011. Article (CrossRef Link).
[19] F. J. Martinez-Zaldivar, A. M. Vidal-Macia, A. Gonzalez and V. Almenar, "Tridimensional block multiword LDPC decoding on GPUs," *Journal of Supercomputing*, vol. 58, no. 3, pp. 314-322, December, 2011. Article (CrossRef Link).
[20] M. Wu, Y. Sun, and J. R. Cavallaro, "Implementation of a 3GPP LTE turbo decoder accelerator on GPU," in *Proc. of IEEE Workshop Signal Processing Systems*, pp. 192-197, October, 2010. Article (CrossRef Link).
[21] C. Lin, W. Liu, W. Yeh, L. Chang, W. Hwu, S. Chen, and P. Hsiung, "A Tiling-Scheme Viterbi Decoder in Software Defined Radio for GPUs," in *Proc. of 2011 7th International Conf. Wireless Communications, Networking and Mobile Computing*, pp. 1-4, September 23-25, 2011. Article (CrossRef Link).
[22] R. W. Chang, "Symthesis of band-limited orthogonal signals for mulltichannel data transmission," *Bell System Technical Kournal*, vol. 45, pp. 1775-1796, 1966. Article (CrossRef Link)
[23] IEEE, "Std 802.11a-1999, Part 11: wireless LAN, medium access control (MAC) and physical layer (PHY) specifications: high-speed physical layer in the 5 GHz band, supplement to IEEE 802.11 Standard," 1999.
[24] IEEE, "IEEE standard 802.16. Air interface for fixed broadband wireless access systems," 2004.
[25] S. Choi, K. Kang and S. Choi, "A two-stage radix-4 Viterbi decoder for multiband OFDM UWB system," *ETRI Journal*, vol. 30, no. 6, pp. 850-852, December, 2008. Article (CrossRef Link).
[26] NVIDIA Corporation, "CUBLAS Library version 4.0," 2011.
[27] Texas Instruments, "TMS320C64x DSP Library Programmer's Reference," 2002.

Rongchun Li received the B.S. in Computer Science and Technology from Wuhan University, Wuhan, China, in 2007, and M.S. in Computer Science and Technology from National University of Defense Technology, Changsha, China, in 2009. Currently, he is a Ph.D. candidate in the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology. His research interests include wireless algorithms on GPU and reconfigurable architectures, and high performance wireless transceiver designs.

Yong Dou received his B.S., M.S., and Ph.D. degrees in Computer Science and Technology at National University of Defense Technology in 1995. Now he is a professor and Ph.D. supervisor in the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology. He is senior membership of China Computer Federation and a member of the IEEE and ACM. His research interests include high performance computer architecture, high performance embedded microprocessor, reconfigurable computing, and software defined radio.

Jie Zhou received his D.S. degree in Computer Science and Technology at National University of Defense Technology in 2011, and now he is an assistant professor at National University of Defense Technology. His research interests include Software-defined Radio, MIMO, and high performance computer architecture.

Baofeng Li received his D.S. degree in Computer Science and Technology at National University of Defense Technology in 2009, and now he is an assistant professor at National University of Defense Technology. His research interests include design of supercomputer, reconfigurable computing and high performance computer architecture.

Jinbo Xu received his D.S. degree in Computer Science and Technology at National University of Defense Technology in 2009, and now he is an assistant professor at National University of Defense Technology. His research interests include DSP processing, reconfigurable computing and high performance computer architecture.