A High Performance GPU-based Software-defined Basestation

Kaipeng Li, Michael Wu, Guohui Wang, Joseph R. Cavallaro
Department of Electrical and Computer Engineering
Rice University, Houston, Texas 77005
Email: {kl33, mbw2, wgh, cavallar}@rice.edu

***Abstract*—We present a high performance GPU-based software-defined basestation. The key idea is to explore the feasibility of using GPU as a baseband processor for supporting software-defined basestation to achieve both real-time high performance and high reconfigurability, considering the numerous computing resources and flexible programming interface of GPU. Based on an existing WARPLab SDR framework, we put effort on the exploration of the data level parallelism and algorithm level parallelism of baseband kernels for GPU acceleration, as well as the task level parallelism in the system for the task pipelining of packet data transfer and data processing. As a case study, an OFDM system is implemented to better demonstrate the concept of our system architecture and optimization strategies. In this case, our GPU-based basestation can not only achieve less than 3ms latency and more than 50Mbps throughput for processing streaming frames in real-time, but also offer software-defined flexibility and scalability for supporting future wireless standards.**

***Index Terms*—GPU, Software-defined Radio, High performance, Reconfigurability, Basestation**

I. Introduction

Software-defined radio (SDR) system is a communication system that implements baseband processing modules in a fully programmable way instead of designing hardware circuits of fixed functionality. The programmability and reconfigurability of SDR show high potential for supporting current and next generation wireless communication standards.

Field programmable gate arrays (FPGAs) and general purpose processors (GPPs) like CPU are often used for accelerating the baseband processing algorithms in a SDR. FPGA-based SDR systems, such as 802.11 reference design on wireless open-access research platform (WARP), usually have high performance satisfying real-time demand, but suffer from low programming accessibility and flexibility and high development cost. GPP-based SDR systems, such as GNU radio [1], Microsoft Sora platform [2] and WARPLab [3], offer higher programming capability based on high-level programming language such as C or Matlab for rapid prototyping and developing of PHY layer algorithms, but may meet some performance bottleneck when dealing with computationally-intensive baseband algorithms such as channel decoding.

Recently, general purpose computing on graphics processing unit (GPGPU) leads a new trend in the high performance computing area [4]. The parallel architecture and numerous computational resources on graphics processing unit (GPU) are beneficial for the efficient processing of high workload digital signal, at the same time, we can use an easy-to-use programming interface, such as Open Computing Language (OpenCL) and Compute Unified Device Architecture (CUDA), for convenient development of specific applications. As an alternative to a of baseband processor, GPU has the potential to combine the high performance of FPGA and high flexibility of GPP for supporting a real-time SDR system. The feasibility of accelerating baseband algorithms using GPU was explored in some previous work,

This work was supported in part by the US National Science Foundation under grants ECCS-1408370, CNS-1265332, and ECCS-1232274.

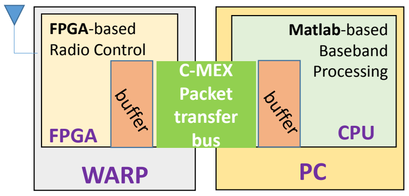

Figure 1. WARPLab transceiver model

such as GPU-based Turbo decoder [5] and LDPC decoder [6], which can achieve quite high performance as expected. Also, there are some recent work on the implementation of GPU-based SDR for mobile WiMAX standard [7] or LTE standard [8], but the performance of those systems is not quite satisfactory for real-time applications. This motivates us to further optimize the utilization of GPU for supporting a software-defined basestation with both high performance and flexibility.

In this paper, we propose to build a software-defined basestation using Nvidia GPU and a modified WARPLab [3] framework. For simplicity but without losing generality, we will show a case study on the implementation of a single input single output (SISO) OFDM WiFi uplink system. Considering the reconfigurability and scalability of our framework and system, it would not be difficult to modify and extend our design for supporting future wireless standards.

This paper is organized as follows. In Section II, we briefly overview the WARPLab framework and the design flow of our proposed basestation. In Section III, we show the design and implementation details of our basestation, focusing on the explanation of how we use GPU for accelerating each baseband kernel. We evaluate the performance of our basestation in Section IV and conclude our work in Section V.

II. System Overview

A. Overview of WARPLab

WARPLab is a flexible SDR platform for rapid prototyping of PHY layer baseband algorithms. We can use a PC and WARP version3 nodes to set up the experimental platform. The PC and WARP are connected and communicate via Ethernet cable to complete a transceiver, and the transmitter (TX) and receiver (RX) can communicate through the wireless channel. The basic transceiver model of the latest WARPLab 7.4 version is shown in Figure 1.

As shown in Figure 1, the WARPLab framework includes three major design components: a Matlab-based design running on the CPU within the PC for baseband processing and transceiver parameter configuration; a FPGA-based design for radio control and interface running on the Xilinx Virtex-6 FPGA integrated in WARP version3 node; a C-MEX based design for the user datagram protocol (UDP) packet transfer and buffer between PC and WARP in the private Ethernet network. At the TX side, the PC will transform the original data into I/Q samples in baseband, wrap the samples and some

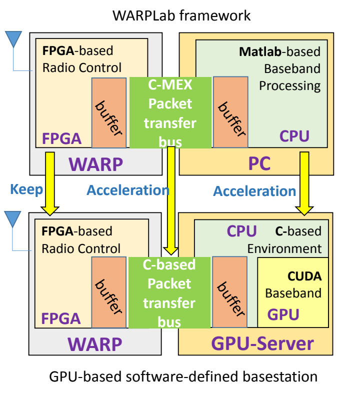

Figure 2. Design flow and proposed system architecture

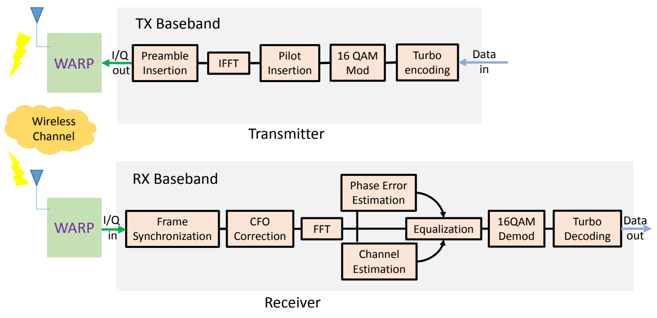

Figure 3. OFDM baseband prototype in the case study

transceiver configuration information (such as RF gain, etc) into UDP packets and send to the TX WARP node. The WARP will extract the samples and configuration information from the packets, execute the configuration, pass the I/Q samples through the DAC, upconvert them to radio frequency (2.4GHz/5GHz) and transmit them over the air. The RX WARP node will capture the signal from the air, complete downconversion, ADC, and send the I/Q samples to the PC for baseband processing and recovery of the original data. For more implementation details of the original WARPLab, please refer to [3].

B. High level architecture of GPU accelerated basestation

We use the WARPLab framework as a starting point of our proposed GPU-based SDR system and overcome its several limits to satisfy the real-time demand of a practical SDR system. First, the Matlab-based baseband processing running on CPU is too slow for real-time performance. To overcome this limit, we implement a GPU-based software-defined solution using CUDA [9], a C-based interface for programming the GPU, to accelerate the baseband kernels and meet the high throughput and low latency demand of a real-time SDR. Second, the execution of the C-MEX packet transfer function from Matlab will introduce high overhead, reducing the Ethernet throughput between PC and WARP, and finally the throughput of the whole system. To overcome this limit, we modify the packet transfer module totally in native C language to remove the unnecessary overhead in the packet transfer and baseband processing by enabling a more efficient data sharing interface. The design flow and high level architecture of our GPU accelerated basestation are shown in Figure 2.

In the following sections, we will illustrate the design and implementation details of a SISO OFDM system for a WiFi uplink as a case study. In this case, we use our GPU-based software-defined basestation as a receiver basestation considering the baseband processing algorithms are more complex at RX. We implement the OFDM baseband signal processing for TX in C language running on CPU since the C-based implementation can meet the performance requirements for the TX.

III. System Design and Implementation

A. Experimental platform setup and baseband prototype

Our GPU-based RX baseband design is deployed on a GPU server, which includes an Intel i7-3930K six-core 3.2GHz CPU and four NVIDIA GTX TITAN graphic cards. Each TITAN contains a 837MHz 2688 core Kepler GPU and 6GB GDDR5 memory. Nsight Eclipse edition 6.0 and CUDA 6.0 tookit are used to design, debug, profile and compile the codes on a Linux 64-bit operating system. We use the fftw and cuFFT library for the FFT/IFFT computation on CPU and GPU respectively. The FPGA design configuration is borrowed from the latest WARPLab version 7.4 for WARP version3 board. The PC and WARP v3 board are connected via 1Gbps Ethernet. Figure 3 shows the baseband prototype of our OFDM system for the case study.

B. GPU accelerated baseband processing

In the case study, we implement a GPU-based software-defined solution of a whole RX baseband chain for WiFi standard on our basestation. For convenience, here we list the definitions of some parameters which will be used to describe the parallelism of kernels.

N_{samp}: The number of samples in a TX or RX frame. It can be configured in WARPLab by "txlength" and "rxlength", indicating the transmitted or received frame length in each transmission cycle. The transmission and processing of each frame will be kept in a loop for data streaming. The maximum frame length is 32768, which is limited by the buffer capacity on WARP.

N_{sym}: The number of OFDM symbols in each frame.

N_{sub}: The number of subcarriers, excluding cyclic prefix, in each OFDM symbol.

N_{data}: The number of subcarriers with payload data, excluding cyclic prefix, pilot and zero subcarriers, in each OFDM symbol.

N_{code}: The number of codewords in each frame for Turbo encoding/decoding.

1) Frame synchronization: Frame synchronization is used to detect the start of the frame and then extract the whole frame for the following processing. In this case study, we add two 64-sample long training sequence (LTS) as the preamble of the frame at TX. At RX, we use another training sequence to calculate the correlation with received frames. The detected correlation peaks with a 64-sample interval indicate a start of the data payload in a frame. In frame synchronization, the most computationally-intensive part is the convolution of training sequence and received frame when calculating the correlation. Usually, we can use FFT and IFFT to implement fast convolution by calculating circular convolution. In our case, the length of LTS *len_lts* is significantly shorter than the length of a frame N_{samp}. To perform direct FFT and IFFT based circular convolution, we need to extend the length of both the LTS and frame to {*len_lts*+N_{samp}-1} by padding zeros, especially more zeros for shorter LTS, which will introduce extra computation overhead. Here, we use the overlap-add method [10] to calculate the convolution of the training sequence and received frame more efficiently. In this method, the long digital signal will be divided into several short segments, and each short segment will do convolution with the short signal respectively to get a batch of short convolution results, and the final convolution of the long signal and short signal is the

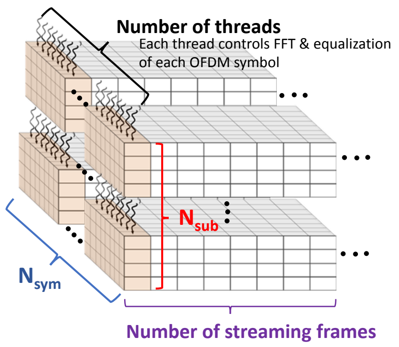
Number of threads
Each thread controls FFT & equalization of each OFDM symbol
N_{sub}
N_{sym}
Number of streaming frames

Figure 4. GPU processing model for FFT and equalization workload

overlap-add sum of the short convolution results. When computing the short convolutions, we can use the FFT and IFFT to reduce the computational complexity.

To implement the synchronization kernel on GPU, we divide each received frame into 64-sample short segments, so the total amount of segments is N_{samp}/64. For each segment, we calculate its convolution with the training signal by using FFT and IFFT, which can be easily implemented on GPU using the cuFFT library. According to the overlap-add method, we can get the final convolution result by calculating a batch of small length FFT and IFFT on GPU in parallel, and the batch number is the amount of small segments in a frame N_{samp}/64, which indicates the parallelism in this synchronization kernel. The convolution results of training sequence and received frames are then used to detect the correlation peaks to find the start of each frame and extract the I/Q samples for following processing.

2) Carrier frequency offset (CFO) correction: Carrier frequency offset (CFO) will result in a spin of the received constellation. We can estimate the CFO by calculating the phase difference of identical samples in the two LTS and averaging the 64 phase differences. The CFO result is stored in the register on GPU for fast fetching when we apply it on the received I/Q samples for offset recovery. Considering that there is no data dependency between each I/Q sample when applying CFO recovery, we generate N_{samp} threads in the GPU for recovering the CFO of each I/Q sample in parallel in this kernel.

3) FFT and Equalization: After correcting CFO, we remove the cyclic prefix and push the time domain samples into the FFT to generate frequency domain samples. For equalization, we need to use the results of channel estimation and phase error estimation. Channel information can be estimated by averaging the estimates from the two LTS, and phase error can be calculated by the phase difference of inserted pilot tones in each OFDM symbol. We store the channel estimation and phase error estimation results in the GPU register in the equalization kernel for efficiently correcting the amplitude and phase error resulting from propagation in the wireless channel. Then, a zero-forcing equalizer is used for simplicity.

In each frame, we have N_{sym} OFDM symbols, each symbol contains N_{sub} subcarriers and has no data dependency when performing FFT and the following equalization. For the FFT part, we have N_{sym} batches of N_{sub} point FFT calculated by cuFFT in parallel, and then the frequency domain samples are pushed into the N_{sym} threaded kernel for equalization, in which each thread controls the processing of N_{sub} samples in each OFDM symbol synchronously. Assuming there are several streaming frames to be processed, the GPU processing model for FFT and equalization workload can be described in Figure 4.

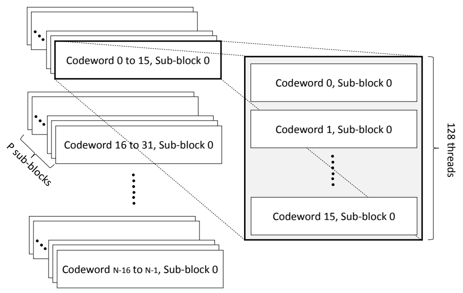

Figure 5. GPU processing model for Turbo decoder

Table I
PARALLELISM DEGREE OF RX KERNELS

Kernel	Parallelism degree
Frame synchronization	N_{samp}/64
CFO correction	N_{samp}
FFT+equalization	N_{sym}
Demodulation	$N_{data} \times N_{sym}$
Decoding	128×P×N_{code}/16

4) Demodulation: After equalization, we extract our payload samples, excluding pilots and zero carriers, for demodulation. We use a 16QAM hard-decision demodulator in our case study for the trade-off between data rate and bit error rate performance. Considering the reconfigurability of our design, we can also replace it with a QPSK or 64QAM demodulator when needed. In the demodulation kernel, we generate $N_{data} \times N_{sym}$ threads for mapping each I/Q payload sample in a frame to corresponding code bits in parallel since each I/Q sample is independent from others when demodulating.

5) Decoder: To improve the error correction performance, we integrated a GPU-based Turbo decoder developed in our previous work [5] into our current software-defined basestation. At the transmitter side, we have a rate 1/3 encoder developed in C.

Consider that we have N_{code} codewords to decode in a frame, we first divide each codeword into P sub-blocks, then group sub-blocks of 16 codewords into a processing unit. For each unit, we generate 128 threads for parallel decoding, so we have totally 128×P×N_{code}/16 threads to deal with all the decoding workload in a frame, as shown in Figure 5. Besides, shared memory and registers are wisely used in kernels to store frequently used parameters for efficient log-likelihood ratio computation. More implementation details can be found in reference [5].

6) Optimization strategies for GPU implementation: As we discussed above, we develop a GPU-based software-defined solution for the entire WiFi RX baseband. In this design process, we carefully explore the parallelism of each kernel and optimize the hierarchical memory usage on GPU for better utilization of GPU computing resources and system performance.

For parallelism exploration, we analyze the data level parallelism and algorithm level parallelism of each baseband kernel function and deploy corresponding GPU threads for parallel processing. The parallelism degree of each kernel can be measured by the number of generated threads on GPU when executing a kernel, which are listed in Table I.

For memory access optimization, we judicially utilize the hierarchical memory on GPU to reduce the memory access latency. Considering the large overhead of memory copy between host memory and GPU device memory, we only copy the received raw frame data to GPU at the beginning of the baseband chain and copy the decoded results back to CPU at the end of the baseband, to avoid the data

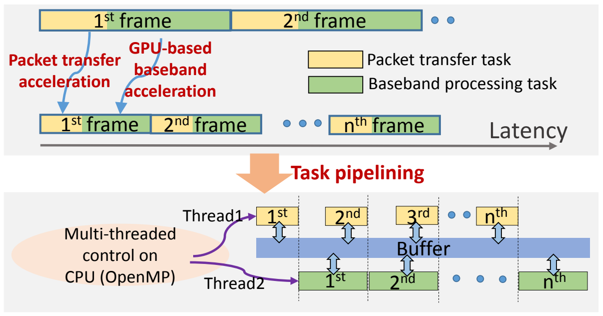

Figure 6. RX optimization flow

copy overhead. Device global memory is used for sharing the results and information between different kernel functions. When the kernels need to fetch data from device memory for processing, most of the kernels are designed to access the device memory in a coalesced way, that is, the parallel threads will be grouped into 32-thread wraps to access a batch of memory with contiguous addresses for reducing the device memory access overhead. For frequently used parameters and data in some kernels, such as equalization and Turbo decoding kernel, the shared memory and registers are used to achieve fast reading and writing of temporary data inside a kernel. For example, the channel estimation and phase error estimation results will be passed into the equalization kernel from device memory and stored in local registers for efficient access in that kernel. We should also note that the shared memory and register resources are quite limited in a GPU, such as several tens of KB in a thread block, so they should be carefully used and organized to avoid the performance bottleneck introduced by the resource access competition between different threads.

C. *Packet transfer bus acceleration*

The efficiency of packet transfer bus between PC and WARP node is also a determining factor of the Ethernet throughput and the performance of the whole system. The C-MEX based UDP transport design and Matlab environment in the original WARPLab will introduce high initialization and interface overhead. To avoid such overhead, we use native C language to redesign the transfer bus, from parameter initialization to the core write buffer and read buffer functions. In this C-based environment, the packet transfer bus and baseband processing task can share data and communicate more efficiently with a smoother interface.

D. *Further optimization*

The C-based environment also offers us the convenience of using a compatible API, such as OpenMP, for concurrent execution of packet transfer and baseband processing to realize the task level parallelism. Considering the data dependency between packet transfer and baseband processing, we use a "producer-consumer" model to pipeline these two tasks. For example, at the receiver side, the streaming frames are transferred from WARP node to PC, "producing" the data to be processed, and the baseband processing chain will "consume" the data once the data is ready. A buffer is allocated on the host as the communication interface between the Ethernet packet transfer task and baseband processing task, and two threads generated by OpenMP interface will control each task respectively for the task pipelining. In this way, we can overlap the packet transfer latency and baseband processing latency for lower cycle latency on processing each streaming frame and higher system throughput. The RX optimization flow is shown in Figure 6 as an example.

Table II
PARAMETER CONFIGURATION.

Kernel	Configuration in each frame
Sync, CFO	$N_{samp} = 32768$ (16 cyclic prefix per OFDM symbol, 2 LTS preambles, payload data, padded zeros)
FFT, IFFT, equalization	$N_{sym} = 384$ OFDM symbols, 64-point FFT/IFFT $N_{sub} = 64$ (48 data subcarriers, 4 pilots, 12 zero subcarriers)
Mod/Demod	16 QAM, $N_{data} = 48$, $N_{mod\_samp} = N_{data} \times N_{sym}$
Turbo enc/dec	1/3 rate, 4608 bits codeword, $N_{code} = 16$, $P = 64$, 6 max iters

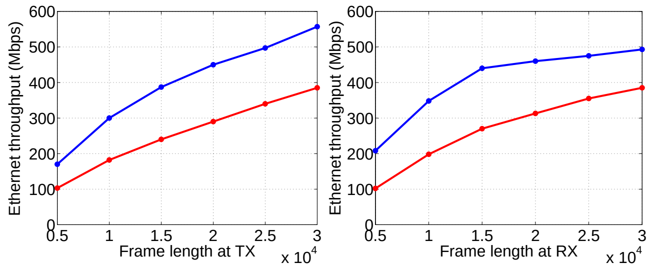

Figure 7. Packet transfer improvement results

IV. SYSTEM EVALUATION

A. *Parameter configuration for the case study*

We benchmark the timing, throughput and BER performance using our experimental setup stated earlier. Most of the computation is done in floating point on both CPU and GPU. The parameter configuration of our OFDM system for case study is listed in Table II.

B. *Packet transfer improvement results*

We benchmark the raw Ethernet throughput of packet transfer without adding baseband processing at both TX and RX, and compare the results of the original Matlab-based packet transfer design and our modified C-based packet transfer design at different frame length per cycle. Figure 7 shows the benchmark results.

From the results, we can find that our C-based packet transfer design achieves evident Ethernet throughput improvement. Also, with the configuration of longer frame length per cycle, the Ethernet throughput tends to be higher. That is because the frame will be divided into several UDP packets when transferring between PC and WARP node, and the frame length is usually not exactly divisible by the packet length, so a fragment packet will be generated at the tail of each frame. In general, compared to longer frame, a smaller frame will generate more fragment packets given a certain transmission workload, which will result in lower Ethernet throughput. The maximum value of frame length in a transmission cycle is 32768 samples in WARPLab, and around 500Mbps Ethernet throughput of packet transfer can be achieved at that frame length with our modified design while excluding the baseband processing of the packets.

C. *Timing performance of system*

We implement the TX baseband in C without GPU acceleration for simplicity considering its low processing complexity, while at the RX side, we have a GPU accelerated RX baseband in CUDA and also a serial C implementation for timing performance comparison. Table III shows the kernel runtime comparison results at RX baseband on processing one frame (32768 samples). The runtime is measured by wall-clock time difference between the start and end of a kernel execution based on the release version compiled by Nsight Eclipse.

From the comparison result, we can find our GPU implementation achieves obvious speedup on each baseband kernel, and usually, the

Table III
TIMING PERFORMANCE COMPARISON

Kernel	C	CUDA	Parallelism	Speedup
Synchronization	5.915ms	0.473ms	512	12.51×
CFO	1.327ms	0.036ms	32768	36.86×
FFT+equalization	2.547ms	0.693ms	384	3.68×
Demodulation	0.081ms	0.006ms	18432	13.50×
Turbo decoding	8.160ms	0.756ms	8192	10.79×

Table IV
LATENCY OF TX AND RX

TX		RX	
Packet transfer (C)	Baseband processing (C)	Packet transfer (C)	Baseband processing (CUDA)
1.326 ms	0.461 ms	2.185 ms	2.598 ms

kernel with higher parallelism achieves even better acceleration since there is greater GPU streaming processor utilization.

We also benchmark the entire baseband latency and packet transfer latency of both TX and RX, which is shown in Table IV. Note that the baseband processing latency includes both kernel runtime and memory copy and access overhead. The latency is measured by the difference of the wall-clock time at the start and end of a task.

By the task pipelining of the packet transfer and baseband processing using multithreaded control on CPU, we can totally overlap TX baseband latency by packet transfer latency, and overlap most of baseband processing latency at RX side, to further accelerate our system as stated before. We should note that our GPU-accelerated RX baseband evidently reduces the baseband processing latency to make it close to packet transfer latency, which contributes to the better overlap and task pipelining of packet transfer and baseband. By above optimization, our GPU-based software-defined basestation can achieve less than 3ms latency on processing a streaming frame in real time.

D. Throughput performance of system

The benchmark results of Ethernet data rate over the PC-WARP link (Ethernet throughput) and the original data resources transmission rate (over-the-air throughput) is shown in Table V. "Not pipelined" means the packet transfer task and baseband processing task proceed in serial, and "Pipelined" means packet transfer task and baseband processing task execute concurrently under "producer-consumer" model. In this benchmark, we use GPU-accelerated RX baseband.

By the task pipelining, the TX throughput does not improve too much, since the TX baseband is already quite fast, which will not provide significant latency overlap. At the RX side, the pipelined design achieves an 82.3% throughput improvement, which proves the efficient latency overlap between packet transfer and RX baseband processing. After all of the above acceleration, our GPU-based software-defined basestation can achieve more than 50Mbps throughput for a WiFi uplink.

E. Bit error rate performance

We benchmark the bit error rate (BER) performance of our system in an indoor channel based on over-the-air experiments, and plot BER against the RF gain of TX, which is proportional to SNR. The BER performance of our system with 16QAM modulation is shown in Figure 8.

V. CONCLUSION

In this paper, we present a high performance GPU-based software-defined basestation. As a case study of GPU accelerated baseband processing, we explore the data level and algorithm level parallelism of baseband kernels and efficiently utilize the hierarchical memory resources on GPU to implement a high performance OFDM RX baseband. The task level parallelism is further studied for the task pipelining of packet transfer and baseband processing to achieve higher system performance. Benchmark results show that our GPU accelerated basestation can achieve less than 3ms processing latency on streaming frames and more than 50Mbps over-the-air throughput for a WiFi uplink configuration. The software-defined high flexibility and scalability of our basestaion also have the potential for supporting future wireless standards.

Table V
THROUGHPUT PERFORMANCE

	TX		RX	
	Not pipelined	Pipelined	Not Pipelined	Pipelined
Ethernet throughput	377.3Mbps	409.4Mbps	220.5Mbps	402.7Mbps
Over-the-air throughput	42.7Mbps	51.2Mbps	27.6Mbps	50.3Mbps

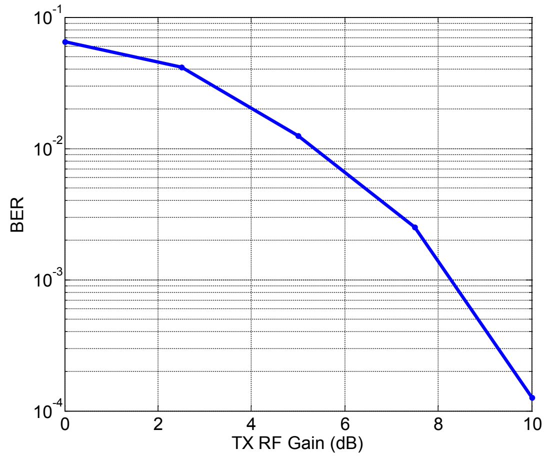

Figure 8. BER performance

REFERENCES

[1] *GNU Radio*. [Online]. Available: http://gnuradio.org
[2] *Sora*. [Online]. Available: http://research.microsoft.com/en-us/projects/sora
[3] *WARPLab*. [Online]. Available: http://warpproject.org/trac/wiki/WARPLab
[4] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[5] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro, "Implementation of a high throughput 3GPP turbo decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, no. 2, pp. 171–183, 2011.
[6] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *IEEE Global Conference on Signal and Information Processing*, 2013, pp. 1–4.
[7] J. Kim, S. Hyeon, and S. Choi, "Implementation of an SDR system using graphics processing unit," *Communications Magazine, IEEE*, vol. 48, no. 3, pp. 156–162, 2010.
[8] S. Bang, C. Ahn, Y. Jin, S. Choi, J. Glossner, and S. Ahn, "Implementation of LTE system on an SDR platform using CUDA and UHD," *Analog Integr. Circuits Signal Process.*, vol. 78, no. 3, pp. 599–610, Mar. 2014.
[9] *CUDA*. [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[10] L. R. Rabiner and B. Gold, *Theory and application of digital signal processing*, 1975.