High Throughput Low Latency LDPC Decoding on GPU for SDR Systems

Guohui Wang, Michael Wu, Bei Yin, and Joseph R. Cavallaro
Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77005
Email: {wgh, mbw2, by2, cavallar}@rice.edu

***Abstract*—In this paper, we present a high throughput and low latency LDPC (low-density parity-check) decoder implementation on GPUs (graphics processing units). The existing GPU-based LDPC decoder implementations suffer from low throughput and long latency, which prevent them from being used in practical SDR (software-defined radio) systems. To overcome this problem, we present optimization techniques for a parallel LDPC decoder including algorithm optimization, fully coalesced memory access, asynchronous data transfer and multi-stream concurrent kernel execution for modern GPU architectures. Experimental results demonstrate that the proposed LDPC decoder achieves 316 Mbps (at 10 iterations) peak throughput on a single GPU. The decoding latency, which is much lower than that of the state of the art, varies from 0.207 ms to 1.266 ms for different throughput requirements from 62.5 Mbps to 304.16 Mbps. When using four GPUs concurrently, we achieve an aggregate peak throughput of 1.25 Gbps (at 10 iterations).**

***Index Terms*—LDPC codes, software-defined radio, GPU, high throughput, low latency.**

I. Introduction

Low-Density Parity-Check (LDPC) codes are a class of error-correction codes which have been widely adopted by emerging standards for wireless communication and storage applications, thanks to their near-capacity error-correcting performance. Because LDPC decoding algorithms are very computationally intensive, researchers have been exploring GPUs' parallel architecture and used GPUs as accelerators to speed up the LDPC decoding [1–9].

Falcão first introduced GPU-based LDPC decoding using NVIDIA's Compute Unified Device Architecture (CUDA) [10], and studied algorithm mapping onto GPU, data packing methods, and memory coalescing techniques [1, 2]. In [3], compact $\mathbf{H}$ matrix representations and optimized memory access are studied for Quasi-Cyclic LDPC codes. The forward-backward algorithm (FBA), optimized memory access and tag-based parallel early termination algorithm are discussed in our previous work [4]. Later, researchers studied the methodology to partition the workload based on availability of GPU's resources, so that scalable LDPC decoding can be achieved on different GPU architectures [5, 6]. Kang proposed LDPC decoding based on unbalanced memory coalescing [7]. Recently, Falcão presented a portable LDPC decoding implementation using OpenCL [8].

Depending on the LDPC code structures and decoding algorithms, the current GPU-based LDPC decoding can normally achieve 50~150 Mbps peak throughput by packing a large number of codewords. As a side effect, the decoding latency becomes very high due to the data aggregation. Attracted by the highly parallel architecture and easy-to-use parallel programming environment provided by modern GPUs, researchers are attempting to build GPU-based software-defined radio (SDR) systems. In this scenario, reducing decoding latency is as important as increasing throughput.

In this paper, we present a new GPU-based implementation of LDPC decoder targeting at future GPU-based SDR systems. Our goal is to achieve both high throughput and low latency. To improve decoding throughput, several optimization strategies are explored, including two-min decoding algorithm, fully coalesced memory access, and data/thread alignment. In addition, we use asynchronous memory data transfer and multi-stream concurrent kernel execution to reduce the decoding latency.

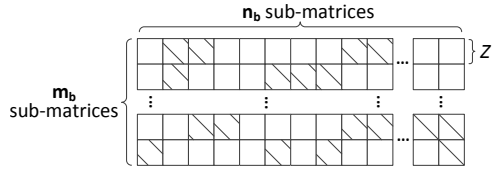

Fig. 1. Matrix $\mathbf{H}$ of a QC-LDPC code (Slashes represent 1's in sub-matrices).

II. LDPC Codes and Decoding Algorithm

A. Quasi-Cyclic LDPC (QC-LDPC) Codes

A binary LDPC code is a linear block code defined by a sparse $M \times N$ parity-check matrix $\mathbf{H}$, which can be represented by a Tanner graph containing M check nodes (CNs) and N variable nodes (VNs). Number of nonzero entries in a row (or column) of $\mathbf{H}$ is called row (or column) weight, denoted as ω_r (or ω_c).

QC-LDPC codes are a class of well-structured codes, whose matrix $\mathbf{H}$ consists of an array of shifted identity matrices with size Z. QC-LDPC codes have been adopted in many standards such as IEEE 802.16e WiMAX and 802.11n WiFi, due to their good error-correction performance and efficient hardware implementation. Fig. 1 shows a typical $\mathbf{H}$ of QC-LDPC codes, which contains $m_b \times n_b$ shifted identity matrices with different shift values. The WiMAX (2304, 1152) code and WiFi (1944, 972) code have similar structures, in which $m_b = 12$ and $n_b = 24$. $Z = 96$ and $Z = 81$ are defined in WiMAX (2304, 1152) code and WiFi (1944, 972) code, respectively.

B. Scaled Min-Sum Algorithm for LDPC Decoding

The sum-product algorithm (SPA) algorithm is usually used to decode LDPC codes, in which belief messages are passed and processed between check nodes and variable nodes. The Min-Sum algorithm (MSA) is a simplification of the SPA based on the processing of *a posteriori* probability (APP) log-likelihood ratio (LLR). Let c_n denote the n-th bit of a codeword, and let x_n denote the n-th bit of a decoded codeword. LLR is defined as $L_n = log((Pr(c_n = 0)/Pr(c_n = 1))$. Let Q_{mn} and R_{mn} denote the messages from VN n to CN m and the message from CN m to VN n, respectively. The major steps of the MSA can be summarized as follows.

1) Initialization:

L_n and VN-to-CN (VTC) message Q_{mn} are initialized to channel input LLRs. The CN-to-VN (CTV) message R_{mn} is initialized to 0.

2) Check node processing (CNP):

$$R_{mn}^{new} = \alpha \cdot \prod_{n' \in \{N_m \setminus n\}} sign(Q_{mn'}^{old}) \cdot \min_{n' \in \{N_m \setminus n\}} \mid Q_{mn'}^{old} \mid, \quad (1)$$

where "old" and "new" represent the previous and the current iterations, respectively. $N_m \setminus n$ denotes the set of all VNs connected with CN m except VN n. α is a scaling factor to compensate for performance loss in the MSA (typical value is $\alpha = 0.75$).

3) Variable node processing (VNP):

$$L_n^{new} = L_n^{old} + \sum_m (R_{mn}^{new} - R_{mn}^{old}), \quad (2)$$

$$Q_{mn}^{new} = L_n^{new} - R_{mn}^{new}. \quad (3)$$

4) Tentative decoding:

To appear at IEEE Global Conference on Signal and Information Processing (GlobalSIP), December 2013, Austin, Texas, USA

Algorithm 1 TMA for check node processing.

```
1: sign_prod = 1;   /* sign product; 1:postive, -1:negtive */
2: sign_bm = 0;   /* bitmap of Q sign; 0:postive, 1:negtive */
3: for i = 0 to ω_r − 1 do
4:    Load L_n and R from device memory;
5:    Q = L_n − R;
6:    sq = Q < 0;   /* sign of Q; 0:postive, 1:negtive */
7:    sign_prod ∗= (1 − sq ∗ 2);
8:    sign_bm | = sq << i;
9:    if |Q| < min_1 then
10:      update min_1, idx and min_2;
11:   else if |Q| < min_2 then
12:      update min_2;
13:   end if
14: end for
15: for i = 0 to ω_r − 1 do
16:   sq = 1 − 2 ∗ ((sign_bm >> i)&0x01);
17:   R^new = 0.75 · sign_prod · sq · (i ! = idx ? min_1 : min_2);
18:   dR = R^new − R;
19:   Store dR and R^new into device memory;
20: end for
```

The decoder makes a hard decision to get the decoded bit x_n by checking the APP value L_n, that is, if $L_n < 0$ then $x_n = 1$, otherwise $x_n = 0$. The decoding process terminates when a pre-set number of iterations is reached, or the decoded bits satisfy the check equations if early termination is allowed. Otherwise, go back to step 2 and start a new iteration.

III. Improving Throughput Performance

In this section, we describe parallel LDPC decoding algorithms and optimization techniques to improve throughput.

A. Parallel LDPC Decoding Algorithm

The message values are represented by 32bit floating-point data type. Similar to [4], CNP and VNP are mapped onto two separate parallel kernel functions. Matrix $\mathbf{H}$ is represented using compact formats, which are stored in GPU's constant memory to allow fast data broadcasting. To fully utilize the stream multi-processors of GPU, we use multi-codeword decoding algorithm. N_{MCW} macro-codewords (MCWs) are defined, each of which contains N_{CW} codewords, so the total number of codewords decoded in parallel is $N_{codeword} = N_{CW} \times N_{MCW}$ (typically $N_{CW} \in [1, 4]$, and $N_{MCW} \in [1, 100]$). To launch the CNP kernel, the grid dimension is set to $(m_b, N_{MCW}, 1)$ and the thread block dimension is set to $(Z, N_{CW}, 1)$. For the VNP kernel, the grid dimension and the thread block dimension are $(n_b, N_{MCW}, 1)$ and $(Z, N_{CW}, 1)$, respectively. By adjusting N_{MCW} and N_{CW}, we can easily change the scalable workload for each kernel. For data storage, since we can use R_{mn} and L_n to recover Q_{mn} according to (3), we only store R_{mn} and L_n in the device memory and compute Q_{mn} on the fly in the beginning of CNP. Please refer to [4] for the above implementation details.

To support both the SPA and the MSA algorithms, a forward-backward algorithm (FBA) is used to implement the CNP kernel in [4]. In this paper, we employ the two-min algorithm (TMA) to further reduce the CNP complexity [8, 11]. It is worth mentioning that FBA and TMA provide the same error-correcting performance when implementing the MSA. According to (1), we can use four terms to recover all R_{mn} values for a check node: the minimum of $|Q_{mn}|$ (denoted as min_1), the second minimum of $|Q_{mn}|$ (denoted as min_2), the index of min_1 (denoted as idx), and product of all signs of Q_{mn} (denoted as $sign\_prod$). R_{mn} can be determined by $R_{mn} = sign\_prod \cdot sign(Q_{mn}) \cdot ((n \neq idx)?min1 : min2)$. The TMA is described in Algorithm 1. Since we do not store Q_{mn} values, the sign array of Q_{mn} needs to be kept for the second recursion. To save storage space, we use a char type $sign\_bm$ to store the bitmap of the sign array. Bitwise shift and logic operations are needed to update this bitmap or extract a sign out of the bitmap. The $sign\_prod$ can be updated by using either bitwise logic operations or floating-point (FP) multiplication. However, since the instruction throughput for FP multiplication is higher than bitwise logic operations (192 versus 160 operations per clock cycle per multiprocessor) [10], FP multiplication is chosen to update $sign\_prod$ value efficiently.

TABLE I
Complexity comparison for CNP using a "native" implementation, the FBA and the TMA.

	"Naive"	FBA	TMA
CS operations	$M\omega_r(\omega_r - 1)$	$M(3\omega_r - 2)$	$M(\omega_r - 1)$
Memory accesses	$M\omega_r^2$	$M(3\omega_r - 2)$	$2M\omega_r$

Table I compares the complexity of a naive implementation of (1), the FBA and the TMA. Since compare-select (CS) is the core operation in the Min-Sum algorithm, we use the number of CS operations to indicate algorithmic complexity. Table I indicates that the TMA has lower complexity compared to the other two algorithms. It is worth mentioning that Algorithm 1 is targeted at decoding more challenging irregular LDPC codes (ω_c is not constant). If we decode regular LDPC codes, the loops in Algorithm 1 can be fully unrolled to avoid branching operations to further increase the throughput.

B. Memory Access Optimization

Accesses to global memory incur long latency of several hundred clock cycles, therefore, memory access optimization is critical for throughput performance. In our implementation, to minimize the data transfer on the PCIe bus, we only transfer the initial LLR values from host to device memory and the final hard decision values from device to host memory. All the other variables such as R_{mn} and dR_{mn} (storing $(R_{mn}^{new} - R_{mn}^{old})$ values needed by (2) in VNP) are only accessed by the kernel functions without being transferred between host and device. To speed up data transfers between host and device, the host memories are allocated as page-locked (or pinned) memories. The page-locked memory enables a direct memory access (DMA) on the GPU to request transfers to and from the host memory without the involvement of the CPU, providing higher memory bandwidth compared to the pageable host memory [10]. Profiling results indicate that throughput improves about 15% by using page-locked memory.

GPUs are able to coalesce global memory requests from threads within a warp into one single memory transaction, if all threads access 128-byte aligned memory segment [10]. Falcão proposed to coalesce memory reading via translation arrays, but writing to memory is still uncoalesced [2]. In [7], reading/writing memory coalescing is used in VTC messages, but CTV message accesses are still not coalesced. In this section, we describe a fully coalesced memory access scheme which coalesces memory accesses for both reading and writing in both CNP and VNP kernels.

In our implementation, accesses to R_{mn} (and dR_{mn}) in CNP kernels and memory accesses to APP values L_n are naturally coalesced, as is shown in Fig. 2-(a). However, due to the random shift values, memory accesses to L_n in CNP and memory accesses to R_{mn} (and dR_{mn}) in VNP are misaligned. For instance, in Fig. 2-(b), three warps access misaligned R_{mn} data, and warp 2 even accesses nonconsecutive data, so multiple memory transactions are generated per data request. As is shown in Fig. 2-(c), we use fast shared memory as cache to help coalesce memory accesses (size of shared memory: $\omega_r \cdot N_{CW} \cdot Z \cdot \text{sizeof}(float)$). We first load data into shared memory in

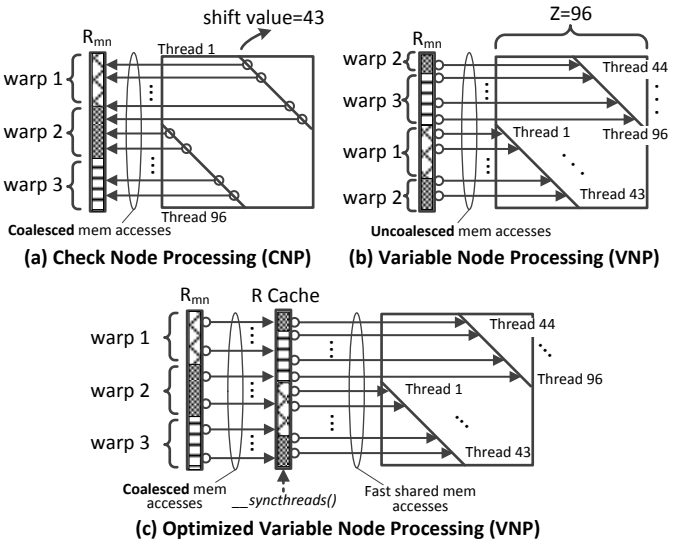

Fig. 2. Optimized coalesced memory access. A shifted identity matrix from WiMAX code ($Z = 96$) with shift value 43 is shown. Combining CNP from (a) and VNP from (c), we achieve fully coalesced memory accesses.

a coalesced way using parallel threads. After a barrier synchronization is performed, the kernels can access data from the shared memory with very low latency. Finally, the kernels write cached data back to device memory in a coalesced way. Profiling results from NVIDIA development tools indicate the proposed method effectively eliminates uncoalesced memory accesses. Since all the device memory accesses become coalesced which leads to a reduction in the number of global memory transactions, the decoding throughput is increased.

C. Data and Thread Alignment for Irregular Block Size

Data alignment is required for coalesced memory access, so it has a big impact on the memory access performance. For the WiMAX (2304, 1152) code, the shifted identity matrix has a size of $Z = 96$, which is a multiple of warp size (32). Therefore, the data alignment can be easily achieved. However, since $Z = 81$ is defined in the WiFi (1944, 972) code, with straightforward data storing order and thread block assignment, few data are aligned to 128-byte addresses. Therefore, we optimize LDPC decoding for irregular block sizes (such as WiFi codes) by packing dummy threads, which means that the thread block dimension becomes $((Z + 31)/32 \times 32, N_{CW}, 1)$. Similarly, for data storage, dummy spaces are reserved to make sure all memory accesses are 128-byte aligned. Although we waste some thread resources and a few memory slots, the aligned thread and data enable efficient memory accesses, and therefore, improves the throughput by approximately 20%.

IV. Reducing Decoding Latency

All the aforementioned optimization strategies applied to the decoding kernels will not only improve the throughput, but also help reduce the decoding latency. In this section, we present optimization techniques to reduce the LDPC decoding latency.

A. Asynchronous Memory Transfer

The current generation NVIDIA GPU contains two memory copy engines and one compute engine. Therefore, we are able to hide most of the time required to transfer data between the host and device by overlapping kernel execution with asynchronous memory copy. Fig. 3 shows how the memory transfers overlap with CNP/VNP kernels. According to our experiments, this technique improves performance by 17% for a typical kernel configuration ($N_{CW} = 2$, $N_{MCW} = 40$).

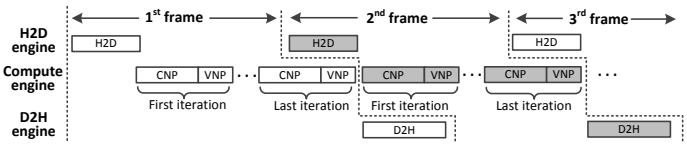

Fig. 3. Asynchronous data transfer. H2D: host to device data transfer. D2H: device to host data transfer.

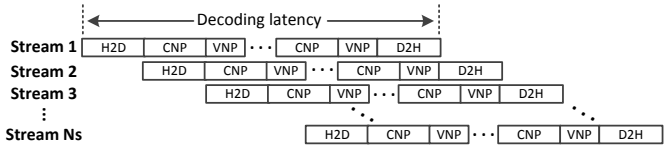

Fig. 4. Multi-stream LDPC decoding.

B. Multi-stream Scheduling for Concurrent Kernels

Computation kernels and memory operations in multiple streams can execute concurrently if there is no dependency between streams. Since the Kepler GK110 architecture, NVIDIA GPUs support up to 32 concurrent streams. In addition, a new feature called Hyper-Q is provided to remove false dependencies between multiple streams to fully allow concurrent kernel overlapping [10]. We take advantage of these new features and further reduce the LDPC decoding latency.

Algorithm 2 Depth-first multi-stream scheduling.

```
for i = 0 to N_Stream − 1 do
    memcpyAsync(streams[i], host→device);
    for j = 0 to N_iter − 1 do
        CNP_kernel(streams[i]);
        VNP_kernel(streams[i]);
    end for
    memcpyAsync(streams[i]), device→host);
end for
for i = 0 to N_Stream − 1 do
    streamSynchronize(streams[i]);
end for
```

In the literature, high throughput is usually achieved via multi-codeword decoding in order to increase the occupancy ratio of parallel cores [4, 5, 7–9]. One drawback of multi-codeword decoding is long latency. To overcome this drawback, we partition codewords into independent workloads and distribute them across multiple streams, so that each stream only decodes a small number of codewords. Multi-stream decoding not only keeps high occupancy thanks to concurrent kernel execution, but also reduces decoding latency. Breadth-first and depth-first GPU command issuing orders are two typical ways to schedule multiple streams. Our experimental results indicate that both issuing orders result in similar decoding throughput, but the depth-first scheduling listed in Algorithm 2 leads to much lower latency. Therefore, we choose the depth-first scheduling algorithm.

Fig. 4 demonstrates a timeline for the multi-stream LDPC decoding. The degree of kernel overlapping depends on the kernel configurations (such as parameters N_{CW} and N_{MCW}). In a practical SDR system, we can use multiple CPU threads with each managing one GPU stream, so that all the GPU streams can run independently. The decoding latency is determined by the latency of each stream.

V. Experimental Results

The experimental platform consists of an Intel i7-3930K six-core 3.2GHz CPU and four NVIDIA GTX TITAN graphics cards. The GTX TITAN has a Kepler GPU containing 2688 CUDA cores running at 837MHz, and 6GB GDDR5 memory. Graphics cards are connected to the system via PCIe x16 interfaces. CUDA toolkit v5.5 Linux 64bit

TABLE II
ACHIEVABLE THROUGHPUT. $N_S = 16$, $N_{CW} = 2$, $N_{MCW} = 40$.

Code	# of iterations	Throughput (Mbps)
WiMAX (2304, 1152)	5	621.38
	10	316.07
	15	204.88
WiFi (1944, 972)	5	490.01
	10	236.70
	15	154.30

TABLE III
LOWEST ACHIEVABLE LATENCY FOR DIFFERENT THROUGHPUT GOALS ($N_{iter} = 10$). WIMAX (2304, 1152) CODE. (T: THROUGHPUT)

T_{goal}(Mbps)	N_S	N_{CW}	N_{MCW}	Latency (ms)	T(Mbps)
50	1	2	3	0.207	62.50
100	1	2	6	0.236	110.25
150	8	1	10	0.273	155.43
200	16	2	7	0.335	201.39
250	16	2	10	0.426	253.36
300	32	2	25	1.266	304.16

version is used. NSight v3.5 is used for profiling. In the experiments, two typical codes from the 802.16e WiMAX and 802.11n WiFi standards are employed. The processing time is measured using the CPU timer, so the kernel processing time plus the overhead including CUDA runtime management and memory copy time are counted.

Table II shows the achievable throughput when using one GPU. N_S denotes the number of concurrent streams. 16 concurrent streams are used, and experiments show that using 32 streams provides similar throughput performance. We achieve the peak throughput of 316.07 Mbps (@10 iters) when decoding the WiMAX code. We also notice that there is still a gap in throughput results between WiMAX codes and WiFi codes, although specific optimizations have been performed for WiFi LDPC codes as discussed in Section III-C. The reason is two fold. Firstly, by aligning the size of a thread block to a multiple of the warp size, 15.6% threads (15 out of 96) are idle; while for the WiMAX codes, all threads perform useful computations. Secondly, the $\mathbf{H}$ matrix of the WiFi LDPC code has 13.16% more edges than the WiMAX codes, which requires more computations.

Table III shows the minimum workload per stream (so as to get the lowest latency) needed to achieve different throughput goals. The workload can be configured by changing parameters (N_S,N_{CW},N_{MCW}) to meet different latency/throughput requirements. We sweep through all combinations of (N_S,N_{CW},N_{MCW}) for $N_S \in [1, 32]$, $N_{CW} \in [1, 5]$ and $N_{MCW} \in [1, 150]$. We searched the whole design space and found the configurations that meet the T_{goal} Mbps performance with the lowest latency, which are reported in Table III. For example, to achieve throughput higher than 50 Mbps, one stream ($N_S = 1$) with $N_{CW} = 2$ and $N_{MCW} = 3$ is configured. With this configuration, we can actually achieve 62.5 Mbps throughput while the latency is only 0.207 ms. As is shown in Table IV, this work achieves much lower decoding latency than other GPU-based LDPC decoders.

In this paper, we focus on improving the raw performance of the computation kernels. Please note that we can still apply the tag-based parallel early termination algorithm and achieve the corresponding speedup as we reported in [4].

The above experiments are performed on a single GPU. We have successfully further pushed the throughput limit by using all four GPUs in our test platform. In order to distribute the decoding workload evenly across four GPUs, we create four independent CPU threads using OpenMP APIs, with each CPU thread managing a GPU, as shown in Fig. 5. As a result, an aggregate peak throughput of 1.25 Gbps (at 10 iterations) is achieved for decoding the WiMAX (2304, 1152) LDPC code. The workload configuration for each CPU thread is $N_S = 16$, $N_{CW} = 2$, and $N_{MCW} = 40$.

TABLE IV
DECODING LATENCY COMPARISON WITH OTHER WORKS. (N_C: NUMBER OF CODEWORDS; T: THROUGHPUT; L: LATENCY)

	LDPC code	GPU	N_{iter}	N_C	T (Mbps)	L (ms)
[2]	(1024, 512)	8800GTX	10	16	14.6	1.12
[3]	(2304, 1152)	GTX280	10	1	1.28	1.8
[4, 6]	(2304, 1152)	GTX470	10	300	52.15	13.25
[5]	(2304, 1152)	9800GTX	5	256	160	3.69
[7]	(2048, 1723)	GTX480	10	N/A	24	N/A
[8]	(8000,4000)	HD5870	10	500	209	19.13
[9]	(64800, 32400)	M2050	17.42	16	55	18.85
This work	(2304, 1152)	GTX TITAN	10	6	62.50	0.207
				12	110.25	0.236
				14	201.39	0.335
				50	304.16	1.266

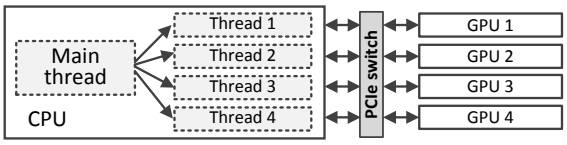

Fig. 5. Multi-GPU LDPC decoding managed by multiple CPU threads.

VI. CONCLUSION

In this paper, we present our effort to improve LDPC decoding on GPU to achieve both high throughput and low latency for potential SDR systems. Several optimization strategies are described to improve throughput performance. Moreover, asynchronous data transfer and multi-stream concurrent kernel execution are employed to reduce decoding latency. Experimental results show that the proposed LDPC decoder achieves 316 Mbps peak throughput for 10 iterations. We also achieve low latency varying from 0.207 ms to 1.266 ms for different throughput requirements from 62.5 Mbps to 304.16 Mbps. An aggregate peak throughput of 1.25 Gbps (at 10 iterations) is achieved by distributing workload to four concurrent GPUs.

ACKNOWLEDGMENT

This work was supported in part by Renesas Mobile, Texas Instruments, Xilinx, and by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, and EECS-0925942.

REFERENCES

[1] G. Falcão, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding," in *Proc. ACM Int. conf. Supercomputing*, 2009, pp. 390–399.
[2] G. Falcão, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, pp. 309–322, 2011.
[3] H. Ji, J. Cho, and W. Sung, "Memory access optimized implementation of cyclic and Quasi-Cyclic LDPC codes on a GPGPU," *Springer J. Signal Process. Syst.*, vol. 64, no. 1, pp. 149–159, 2011.
[4] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Proc. IEEE Symp. Application Specific Processors (SASP)*, 2011, pp. 82–85.
[5] K. K. Abburi, "A scalable LDPC decoder on GPU," in *Proc. IEEE Int. Conf. VLSI Design (VLSID)*, 2011, pp. 183–188.
[6] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "GPU accelerated scalable parallel decoding of LDPC codes," in *Proc. IEEE Asilomar Conf. Signals, Systems and Computers*, 2011, pp. 2053–2057.
[7] S. Kang and J. Moon, "Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2012, pp. 3692–3697.
[8] G. Falcão, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC Decoding on Multicores Using OpenCL," *IEEE Signal Process. Mag.*, vol. 29, no. 4, pp. 81–109, 2012.
[9] G. Falcão, J. Andrade, V. Silva, S. Yamagiwa, and L. Sousa, "Stressing the BER simulation of LDPC codes in the error floor region using GPU clusters," in *Proc. Int. Symp. Wireless Commun. Syst. (ISWCS)*, August 2013.
[10] NVIDIA CUDA C programming guide v5.5. [Online]. Available: http://docs.nvidia.com/cuda/
[11] K. Zhang, X. Huang, and Z. Wang, "High-throughput layered decoder implementation for quasi-cyclic LDPC codes," *IEEE J. Sel. Areas in Commun.*, vol. 27, no. 6, pp. 985–994, 2009.