A Massively Parallel Implementation of QC-LDPC Decoder on GPU

Guohui Wang, Michael Wu, Yang Sun, and Joseph R. Cavallaro
Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77005
{wgh, mwb2, ysun, cavallaro}@rice.edu

Abstract—The graphics processor unit (GPU) is able to provide a low-cost and flexible software-based multi-core architecture for high performance computing. However, it is still very challenging to efficiently map the real-world applications to GPU and fully utilize the computational power of GPU.

As a case study, we present a GPU-based implementation of a real-world digital signal processing (DSP) application: low-density parity-check (LDPC) decoder. The paper shows the efforts we made to map the algorithm onto the massively parallel architecture of GPU and fully utilize GPU’s computational resources to significantly boost the performance. Moreover, several algorithmic optimizations and efficient data structures have been proposed to reduce the memory access latency and the memory bandwidth requirement. Experimental results show that the proposed GPU-based LDPC decoding accelerator can take advantage of the multi-core computational power provided by GPU and achieve high throughput up to 100.3Mbps.

Keywords—GPU, parallel computing, CUDA, LDPC decoder, accelerator

I. INTRODUCTION

A graphics processing unit (GPU) provides a parallel architecture which combines raw computation power with programmability. GPU provides extremely high computational throughput by employing many cores working on a large set of data in parallel. In the field of wireless communication, although power and strict latency requirements of real communication systems continue to be the main challenges for a practical real-time GPU-based platform, GPU-based accelerators remain attractive due to their flexibility and scalability, especially in the realm of simulation acceleration and software-defined radio (SDR) test-beds. Recently, GPU-based implementations of several key components of communication systems have been studied. For instance, a soft information multiple-input multiple-output (MIMO) detector is implemented on GPU and achieves very high throughput [1]. In [2], a parallel turbo decoding accelerator implemented on GPU is studied for wireless channels.

Low-density parity-check (LDPC) decoder [3] is another key communication component and the GPU implementations of the LDPC decoder have drawn much attention recently, due to its high computational complexity and inherently massively parallel nature. LDPC codes are a class of powerful error correcting codes that can achieve near-capacity error correcting performance. This class of codes are widely used in many wireless standards such as WiMax (IEEE 802.16e), WiFi (IEEE 802.11n) and high speed magnetic storage devices. The flexibility and scalability make GPU a good simulation platform to study the characteristics of different LDPC codes or to develop new LDPC codes. However, it is still very challenging to efficiently map the LDPC algorithm to the GPU’s massively parallel architecture and achieve very high performance. Recently, parallel implementations of high throughput LDPC decoders are studied in [4]. In [5], the researchers optimize the memory access and develop parallel decoding software for cyclic and quasi-cyclic LDPC (QC-LDPC) codes. However, there are still great potentials to achieve higher performance by developing better algorithm mapping according to GPU’s architecture.

This work focuses on the techniques to fully utilize GPU’s computational resources to implement a computation-intensive digital signal processing (DSP) algorithm. As a case study, a highly-optimized and massively parallel LDPC decoder implementation on GPU is presented. Several efforts have been made from the aspects of algorithmic optimizations, memory access optimizations and efficient data structures, which enable our implementation to achieve much higher performance than the prior work.

This paper is organized as follows. In Section II, the CUDA platform for GPU computing is introduced. Section III gives an overview of the LDPC decoding algorithm. Different aspects of the GPU implementation of the LDPC decoder are discussed in Section IV. Section V focuses on memory access optimization techniques. Section VI provides the experimental results for performance and throughput of the proposed implementation. Finally, Section VII concludes this paper.

II. CUDA ARCHITECTURE

Computer Unified Device Architecture (CUDA) [6] is widely used to program massively parallel computing applications. The NVIDIA Fermi GPU architecture consists of multiple stream multiprocessors (SM). Each SM consists of 32 pipelined cores and two instruction dispatch units. During execution, each dispatch unit can issue a 32 wide single instruction multiple data (SIMD) instruction which is executed on a group of 16 cores. Although CUDA provides the possibility to unleash GPU’s computational power, several restrictions prevent programmers from achieving peak performance. The programmer should pay attention to the following aspects to achieve near-peak performance.

In the CUDA model, a frequently executed task can be mapped into a kernel and executed in parallel on many threads.
in the GPU. CUDA divides threads within a thread block into blocks of 32 threads, which are executed as a group using the same common instruction (a warp instruction). As instructions are issued in-order, a stall can occur when an instruction fetches data from device memory or when there is data dependency. Stalls can be minimized by coalescing memory access, using fast on-die memory resources and through hardware support for fast thread switching.

A CUDA device has a large amount (>1GB) of off-chip device memory with high latency (400~800 cycles) [6]. To improve efficiency, device memory accesses can be coalesced. For CUDA devices, if all threads access 1, 2, 4, 8, or 16 byte words and threads within half of a warp access sequential words in device memory, the memory requests of half of a warp can be coalesced into a single memory request. Coalescing memory access improves performance by reducing the number of device memory requests.

Fast on-chip resources such as registers, shared memory, and constant memory can reduce memory access time. Access to shared memory usually takes one load or store operation. However, random accesses to shared memory with bank conflicts should be avoided since they will be serialized and cause performance degradation. On the other hand, memory access to constant memory is cached although constant memory resides in device memory. Therefore, it takes one cached read if all threads access the same location. However, other constant memory access patterns will be serialized and need device memory access, therefore, should be avoided.

Furthermore, to hide the processor stalls and achieve peak performance, one should map sufficient concurrent threads onto a SM. However, both shared memory and registers are shared among concurrent threads on a SM. The limited amounts of shared memory and registers require the programmers to effectively partition the workloads such that at least a sufficient number of blocks can be mapped onto the same processor to hide stalls.

III. INTRODUCTION TO LDPC DECODING ALGORITHM

A. LDPC Codes

Low-density parity-check (LDPC) codes are a class of widely used error correcting codes. The binary LDPC codes can be defined by the equation \( H \cdot x^T = 0 \), in which \( x \) is a codeword and \( H \) is an \( M \times N \) sparse parity check matrix. The number of 1’s in a row of \( H \) is called row weight, denoted by \( \omega_r \). If \( \omega_r \) is the same for all rows, the code is regular; otherwise, the code is irregular. Regular codes are easier to implement but have poorer error correcting performance than irregular codes.

Quasi-Cyclic LDPC (QC-LDPC) codes [7] are a special class of LDPC codes with a structured \( H \) matrix, which can be generated by the expansion of a \( Z \times Z \) base matrix. As an example, Fig. 1 shows the parity check matrix for the (1944, 972) 802.11n LDPC code with sub-matrix size \( Z = 81 \). In this matrix representation, each square box with a label \( I_x \) represents an \( 81 \times 81 \) circularly right-shifted identity matrix with a shifted value of \( x \), and each empty box represents an \( 81 \times 81 \) zero matrix. For a QC-LDPC code, \( H \) could be viewed as an array of \( M_{sub} \times N_{sub} \) sub-matrices. \( M_{sub} \) and \( N_{sub} \) are the number of sub-matrices in a column and a row, respectively. A row of \( N_{sub} \) sub-matrices is also known as a layer.

Because of its well-organized structure, QC-LDPC is very efficient for hardware implementation. In addition, irregular codes have better error correcting performance than regular codes. Therefore, irregular QC-LDPC codes have been adopted by many communication systems such as 802.11n WiFi and 802.16e WiMAX. In this paper, we take the irregular QC-LDPC codes from the 802.11n standard as a case study. However, the proposed accelerator can be used for other cyclic or quasi-cyclic LDPC codes.

B. Sum-product Algorithm for LDPC Decoder

The sum-product algorithm (SPA) is based on the iterative message passing among check-nodes (CNs) and variable-nodes (VNs) and provides a powerful method to decode LDPC codes [3]. The SPA is usually performed in the log-domain. The SPA has a computational complexity of \( O(N^3) \), in which \( N \) is normally very large which results in very intensive computations.

Let \( c_n \) denote the \( n \)-th bit of a codeword, and let \( x_n \) denote the \( n \)-th bit of a decoded codeword. The a posteriori probability (APP) log-likelihood ratio (LLR) is a soft information of \( c_n \) and can be defined as \( L_n = log((Pr(c_n = 0)/Pr(c_n = 1)) \).

1) Initialization:

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

2) Begin the iterative Decoding process:

For each VN \( n \), calculate \( Q_{mn} \) by

\[
Q_{mn} = L_n + \sum_{m' \in \{M_n \setminus m\}} R_{m'n},
\]

where \( M_n \setminus m \) denotes the set of all the CNs connected with VN \( n \) except CN \( m \). Then, for each CN \( m \), compute the new CTV message \( R'_{mn} \) and \( \Delta_{mn} \) by

\[
R'_{mn} = Q_{mn_1} \oplus Q_{mn_2} \oplus \cdots \oplus Q_{mn_k},
\]

\[
\Delta_{mn} = R'_{mn} - R_{mn},
\]
where \( n_1, n_2, \ldots, n_k \in \{ N_m \setminus n \} \) and \( N_m \setminus n \) denotes the set of all the CNs connected with VN \( n \) except CN \( m \). \( R'_{mn} \) and \( \Delta_{mn} \) are saved for the APP update and the next iteration. The \( \ominus \) operation is defined as below:

\[
    x \ominus y = \text{sign}(x)\text{sign}(y) \min(|x|, |y|) + S(x, y),
\]

\[
    S(x, y) = \log(1 + e^{-|x+y|}) - \log(1 + e^{-|x-y|}).
\]

3) Update the APP values and make hard decisions

\[
    L'_n = L_n + \sum_{m} \Delta_{mn}.
\]

The decoder makes a hard decision to get the decoded bit \( x_n \) by quantizing the APP value \( L'_n \) into 1 and 0, that is, if \( L'_n < 0 \) then \( x_n = 1 \), otherwise \( x_n = 0 \). The decoding process terminates when the codeword \( x \) satisfies \( H \cdot x^T = 0 \), or the pre-set maximum number of iterations is reached. Otherwise, go back to step 2 and start a new iteration of decoding.

C. Scaled Min-Sum Algorithm

The min-sum algorithm (MSA) reduces the decoding complexity of the SPA with minor performance loss [8][9]. The MSA approximates log-SPA by removing the correction item \( S(x) \) from (4). The \( R_{mn} \) calculation in the scaled MSA can be expressed as below:

\[
    R'_{mn} = \alpha \cdot \prod_{n' \in \{ N_m \setminus n \}} \text{sign}(Q_{mn'}) \cdot \min_{n' \in \{ N_m \setminus n \}} |Q_{mn'}|,
\]

where \( \alpha \) is the scaling factor to compensate for the performance loss in the min-sum algorithm (\( \alpha = 0.75 \) is used in this paper) [9].

IV. Mapping LDPC Decoding Algorithm onto GPU

In this work, we map both the MSA and the log-SPA onto GPU architecture. In this section, we present modified decoding algorithms to reduce the decoding latency and the bandwidth requirement of device memory. The implementations of the LDPC decoding kernels are also described.

A. Loosely Coupled LDPC Decoding Algorithm

It can be noted that the log-SPA or min-sum algorithm could be simplified by reviewing the equations from (1) to (6). Instead of saving all the \( Q \) values and \( R \) values in the device memory, only \( R \) values are stored [10]. To compute the new CTV message \( R' \), we first recover the \( Q \) value based on \( R \) and APP value from the previous calculation. Based on this idea, Equation (1) can be changed to:

\[
    Q_{mn} = L_n - R_{mn}.
\]

Due to the limited size of the on-chip shared memory, \( R_{mn} \) values can only fit in the device memory. The loosely coupled LDPC decoding algorithm significantly reduces device memory bandwidth by reducing the amount of device memory used. For example, with an \( M \times N \) parity-check matrix \( H \) with the row weight of \( \omega_r \) (the number of 1’s in a row of \( H \)), we save up to \( M \omega_r \) memory slots. Also, the number of memory accesses to the device memory is reduced by at least \( 2M \omega_r \), since each \( Q_{mn} \) is read and written to at least once per iteration.

B. Two-Pass Recursion CTV Message Update

Based on (2), we need to traverse one row of the \( H \) matrix multiple times to update all the CTV messages \( R_{mn} \). Each traversal pass requires \( (\omega_r - 2) \) compare-select operations to find the minimum value from \( (\omega_r - 1) \) data. Moreover, \( (\omega_r - 1) \) XOR operations are also needed to compute the sign of \( R_{mn} \). In addition, the traversal introduces some branch instructions which cause an unbalanced workload and reduce the throughput.

The traversal process could be optimized by using a forward and backward two-pass recursion scheme [11]. In the forward recursion, some intermediate values are calculated and stored. During the backward recursion, the CTV message for each node is computed based on the intermediate values from the forward recursion. By roughly calculating, the number of operations can be reduced from \( M \omega_r (\omega_r - 2) \) to \( M (3\omega_r - 2) \) in one iteration of LDPC decoding. For the 802.11n (1944, 972) code, this two-pass recursion scheme saves us about 50% of the operations. This significantly reduces the decoding latency. The number of memory accesses and the number of required registers are also greatly reduced.

C. Implementation of the LDPC Decoder Kernel on GPU

According to Equations (2), (3), (6) and (8), the decoding process can be split into two stages: the horizontal processing stage and the APP update stage. So we can create one computational kernel for each stage. The relationship between the host (CPU) code and the device (GPU) kernel is shown in Fig 2.

1) CUDA Kernel 1: Horizontal Processing: During the horizontal processing stage, since all the CTV messages are calculated independently, we could use many parallel threads to process these CTV messages. For an \( M \times N \) \( H \) matrix, \( M \) threads are spawned, and each thread processes a row. Since all non-zero entries in a sub-matrix of \( H \) have the same shift value (one square box in Fig 1), threads processing the same layer have almost exactly the same operations when calculating the CTV messages. In addition, there is no data dependency among different layers. Therefore, each layer is processed by a thread block. \( M_{sub} \) thread blocks are used and
Algorithm 1 CUDA Kernel 1: Horizontal processing
1: $i_{Layer} = blockIdx.x$; /*the index of a layer in $H$*/
2: $i_{SubRow} = threadIdx.x$; /*the row index in the layer*/
3: Calculate new CTV message using two-pass recursion;
4: Write CTV messages ($R_{mn}, \Delta_{mn}$) back into device memory;

2) CUDA Kernel 2: APP value update: During the APP update stage, there are $N$ APP values to be updated. Similarly, the APP value update is independent among different variable nodes. So, $N_{sub}$ thread blocks are used to update the APP values, with $Z$ threads in each thread block. In the APP update stage, there are 1944 threads which are grouped into 24 thread blocks working concurrently for the 802.11n (1944, 972) LDPC code. The algorithm of CUDA kernel 2 is described in Algorithm 2.

Algorithm 2 CUDA Kernel 2: Update APP value
1: $i_{BlkCol} = blockIdx.x$; /*the column index of a sub-matrix in $H$*/
2: $i_{SubCol} = threadIdx.x$; /*the column index within the sub-matrix*/
3: Calculate the device memory address of APP value;
4: Read the old APP value;
5: for all All the sub-matrices in $i_{BlkCol}$ column of $H$ do
6: Read the corresponding $\Delta_{mn}$ value;
7: Update the APP value using (6);
8: end for
9: Write the updated APP value into the device memory;
10: Make a hard decision for the current bit.

D. Multi-codeword Parallel Decoding

Since the number of threads and thread blocks are limited by the dimensions of the $H$ matrix, it is hard to keep all the cores fully occupied by decoding a single codeword. Multi-codeword decoding is needed to further increase the parallelism of the workload. A two-level multi-codeword scheme is designed. $N_{CW}$ codewords are first packed into one macro-codeword (MCW). Each MCW is decoded by a thread block and $N_{MCW}$ MCWs are decoded by a group of thread blocks. The multi-codeword parallel decoding algorithm is described in Fig. 3.

Since multiple codewords in one MCW are decoded by the threads within the same thread block, all the threads follow the same execution path during the decoding process. So the workload is well balanced for the codewords in one MCW which is helpful to increase the throughput. Another benefit is that the latency of read-after-write dependencies and memory bank conflicts can be completely hidden by a sufficient number of active threads. To implement this multi-codeword parallel decoding scheme, $N_{MCW} \times N_{CW}$ codewords are written into the device memory in a specified order before kernel launch.

E. Implementation of Early Termination Scheme

The early termination (ET) algorithm is used to avoid unnecessary computations when the decoder already converges to the correct codeword. For software implementation and simulation, ET algorithm can significantly boost the throughput, especially for the high SNR (signal-to-noise ratio) scenario.

The parity check equations $H \cdot x^T = 0$ can be used to verify the correctness of the decoded codeword. Since the parity check equations are inherently parallel, a new CUDA kernel with many threads is launched to perform the ET check. We create $M$ threads, and each thread calculates one parity check equation independently. Since the decoded codeword $x$, compact $H$ matrix and parity check results are used by all the threads, on-chip shared memory is used to enable the high speed memory access. After the concurrent threads finish computing the parity check equations, we reuse these threads to perform a reduction operation on all the parity check results.
to generate the final ET check result, which indicates the correctness of the concurrent codeword.

This parallel ET algorithm is straightforward for a single codeword. However, for multi-codeword parallel decoding, the ET can help increase the throughput only if all the codewords meet the ET condition, which has a very low chance to happen. To overcome this problem, we propose a tag-based ET algorithm for multi-codeword parallel decoding. All the $N_{MCW} \times N_{CW}$ codewords are checked by multiple thread blocks. We assign one tag per codeword and mark the tag once the corresponding parity check equation is satisfied. Afterwards, the decoding process of this particular codeword is stopped in the following iterations. Once the tags for all the codewords are marked, the iterative decoding process is terminated. The experimental results in Section VI-B show that the proposed parallel ET algorithm significantly increases the throughput.

V. OPTIMIZING MEMORY ACCESS ON GPU

In this section, several memory access optimization techniques to further increase the throughput are explained.

A. Memory Optimization for H Matrix

The constant memory on the GPU device could be utilized to optimize the throughput. As mentioned in Section II, reading from the constant memory is as fast as reading from a register as long as all the threads within a half-warp read the same address. Since all the Z threads in one thread block access the same entry of the H matrix simultaneously, we can store the H matrix in the constant memory and take advantage of the broadcasting mode of the constant memory. Simulation shows that constant memory increases the throughput by about 8%.

The quasi-cyclic characteristic of the QC-LDPC code allows us to efficiently store the sparse H matrix. Two compact representations of H (H$_{kernel1}$ and H$_{kernel2}$) are designed. Here, we regard the cyclic H matrix in Fig. 1 as a $12 \times 24$ matrix H whose entry is the shift value for each sub-matrix of H. After horizontally compressing H to the left, we get H$_{kernel1}$. Similarly, vertically compressing H to the top gives us H$_{kernel2}$. The structure of H$_{kernel1}$ and H$_{kernel2}$ are shown in Fig. 4.

There are two advantages of the proposed compact representation of H. First, since the compressed format reduces the memory usage, the time used to read the H matrix from device memory (with long latency) is reduced. Second, the branch instructions cause throughput degradation. The compressed matrix shows the position of all the non-empty entries in H. Therefore, during the two-pass recursion there is no need to check whether one entry of H is empty which avoids the use of branch instructions. Taking the 802.11n (1944, 972) H matrix as an example, 40% of memory access and branch instructions are reduced by using the compressed H$_{kernel1}$ and H$_{kernel2}$.

B. Coalescing Device Memory Access

Since the device memory access suffers several hundred cycles latency to complete, optimizing the device memory access time can help to increase the throughput. In CUDA kernel 1, $R_{mn}$ and $\Delta_{mn}$ values are read from the device memory during the two-pass recursion. Once the computations are done, $R_{mn}$ and $\Delta_{mn}$ values are written back to the device memory. Since there is only one $R_{mn}$ value and one $\Delta_{mn}$ value per row in each sub-matrix of H, the compressed format can be used to store $R_{mn}$ and $\Delta_{mn}$. Two $M \times \omega_r$ matrices are used to store $R_{mn}$ and $\Delta_{mn}$. In total, memory saving for $R_{mn}$ and $\Delta_{mn}$ is more than halved.

More importantly, GPU supports very efficient coalesced access if all threads in a warp access the memory locations which have contiguous addresses. By writing the compressed $R_{mn}$ and $\Delta_{mn}$ matrices column-wise into the device memory, all memory operations for $R_{mn}$ and $\Delta_{mn}$ are coalesced. Simulation shows that 20% throughput improvement is achieved by coalescing device memory access for $R_{mn}$ and $\Delta_{mn}$.

VI. EXPERIMENTAL RESULTS

The experimental setup to evaluate the performance of the proposed architecture on the GPU consists of an NVIDIA GTX470 GPU with 448 stream processors, running at 1.215GHz and with 1280MB of GDDR5 device memory.

The implementation goals are flexibility and scalability. The key parameters such as the length of codeword and the code rate can be easily reconfigured. In addition, this LDPC decoder supports both the log-SPA algorithm and the min-sum algorithm. The implementation of log-SPA employs GPU’s intrinsic functions _logf() and __expf(), which are directly mapped onto GPU hardware and are very fast [6]. Furthermore, this LDPC decoding architecture supports different cyclic or quasi-cyclic LDPC codes, including the standard-compliant 802.11n WiFi and 802.16e WiMAX LDPC codes.

A. Decoder Error Probability Performance

Fig. 5 shows the block error rate (BLER) performance for the IEEE 802.11n (1944, 972) LDPC code versus the signal-to-noise ratio (SNR) which is represented by Eb/N0. Binary phase shift keying (BPSK) modulation and additive white Gaussian noise (AWGN) channel model are employed.
In this simulation, the maximum number of iterations is set to a typically used value of 15. The simulation result curve is the same as the CPU version, since both the algorithm and the data precision (single-precision floating-point) are the same. The simulation results show that the log-SPA algorithm outperforms the min-sum algorithm by about 0.4dB. As will be shown in the next subsection, the throughput for the log-SPA is comparable to the min-sum algorithm on GPU. This means that the GPU implementation can take advantage of the performance gain of the log-SPA while still achieving good throughput.

B. Throughput Results

Assume the length of the codeword is \( N_{\text{bits}} \), the total number of codewords is \( N_{\text{codewords}} \), the simulation number is \( N_{\text{Sim}} \), running time is \( T_{\text{total}} \), which can be expressed as \( T_{\text{total}} = T_{\text{Host-to-Device}} + T_{\text{iterations}} + T_{\text{Device-to-Host}} \). The throughput can be calculated by:

\[
\text{Throughput} = N_{\text{bits}} \times N_{\text{Sim}} \times N_{\text{codewords}} / T_{\text{total}}.
\]

Table I shows the throughput for CPU and GPU implementations for both 802.11n code and WiMAX code with different number of iterations \( (N_{\text{iter}}) \). The CPU implementation is a highly optimized single-core version running on an Intel Core i5 CPU running at 3.2GHz. The results show that GPU implementation outperforms the CPU version with more than 300x speed-up in most cases. Although using multi-core programming language can increase the performance of the CPU implementation, the speed-up we can gain is still limited by the computational resources in the CPU.

Moreover, we can also notice that, for the CPU implementation, the throughput for the log-SPA is much slower than the min-sum algorithm because the log-SPA contains more instructions and more complicated \( \log() \) and \( \exp() \) functions. However, for the GPU implementation, the throughput for the log-SPA algorithm is comparable to the min-sum algorithm. The reason is that GPU implementation employs very efficient intrinsic functions \( \_\log() \) and \( \_\exp() \). And the bottleneck for GPU implementation is in the long device memory access latency, therefore, the run time for the extra instructions in the log-SPA is hidden behind the memory access latency.

Furthermore, the results also show that the decoder for WiMAX code has higher throughput compared to the 802.11n code because of two reasons. First, the block size of the WiMAX code is longer, while the row weights \( (\omega_r) \) for these two codes are similar, which means the computational workload is comparable. Therefore, the WiMAX code with longer codeword tends to have higher throughput according to the throughput equation. Second, for longer codeword, there are more arithmetic instructions per memory access which can hide the memory access latency and data transfer overhead.

![Simulation results for early termination. Average numbers of iterations (dashed line) and throughput (solid line) of the 802.11n (1944, 972) LDPC decoder on GPU. The maximum number of iterations is set to 50.](image)

Table II compares our work with related prior work. There are two main differences among these implementations. First, block sizes of LDPC codes are different. LDPC codes with

<table>
<thead>
<tr>
<th>Code Type</th>
<th>( N_{\text{iter}} )</th>
<th>CPU (Mbps)</th>
<th>GPU (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( N_{\text{Sim}} )</td>
<td>( \text{log-SPA} )</td>
</tr>
<tr>
<td>802.11n</td>
<td></td>
<td>5</td>
<td>0.065</td>
</tr>
<tr>
<td>(1944, 972)</td>
<td></td>
<td>10</td>
<td>0.039</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>0.021</td>
</tr>
<tr>
<td>WiMAX</td>
<td></td>
<td>5</td>
<td>0.071</td>
</tr>
<tr>
<td>(2304, 1152)</td>
<td></td>
<td>10</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>0.029</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>0.023</td>
</tr>
</tbody>
</table>

The parallel early termination scheme is implemented as described in Section IV-E. Fig. 6 shows the throughput results and the average number of iterations when the early termination scheme is used. As the SNR (represented by EbN0) increases, the LDPC decoder converges faster so that the average number of iterations decreases and the decoding throughput increases. The simulation results show that the parallel early termination scheme significantly speeds up the simulation for the high SNR case. For low SNR, the ET version may be slower than the non-ET version due to overhead of the ET check kernel. Therefore, an adaptive scheme can be used to speed up the simulation from the low SNR to high SNR – the ET kernel only launches when the simulation SNR is higher than a specific threshold.

C. Comparison with Prior Work

Table II compares our work with related prior work. There are two main differences among these implementations. First, block sizes of LDPC codes are different. LDPC codes with
TABLE II
LDPC Decoding Throughput Comparison with Other Work. (ET means early termination is used.)

<table>
<thead>
<tr>
<th>Work</th>
<th>GPU</th>
<th>Code</th>
<th>Code Type</th>
<th># iteration</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wang, 2008 [12]</td>
<td>8800GT</td>
<td>(2048, 1024)</td>
<td>regular code</td>
<td>10</td>
<td>2.34 Mbps</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Falcao, 2011 [4]</td>
<td>8800 GTX</td>
<td>(1024, 512)</td>
<td>regular code</td>
<td>10</td>
<td>10.0 Mbps</td>
</tr>
<tr>
<td></td>
<td>8800 GTX</td>
<td>(4896, 2448)</td>
<td></td>
<td>10</td>
<td>17.9 Mbps</td>
</tr>
<tr>
<td>Ji, 2010 [5]</td>
<td>GTX 285</td>
<td>(1944, 972)</td>
<td>802.11n code, irregular</td>
<td>10</td>
<td>0.75 Mbps</td>
</tr>
<tr>
<td>This work</td>
<td>GTX 470</td>
<td>(1944, 972)</td>
<td>802.11n code, irregular</td>
<td>10</td>
<td>39.98 Mbps</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50 ~ 4.3 (ET)</td>
<td>22.5–100.3 Mbps (Eb/No=1.5~5dB)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52.15 Mbps</td>
</tr>
</tbody>
</table>

longer codeword have higher throughput compared to shorter codes due to the high arithmetic operations to memory access ratio. Second, most of the previous implementations use regular LDPC codes which have more balanced workload than the irregular LDPC codes. It is difficult to use massive threads to fully occupy the computation resources of the GPU when decoding the irregular LDPC codes. When processing an irregular LDPC code, imbalanced workloads cause the threads on GPU to complete the computations at different times. Therefore, runtime is bounded by the threads with the most amount of work.

Table II shows that although the irregular codes we used are theoretically harder to get higher throughput than the ones in the prior work, our decoder still outperforms the prior work with significant improvement, especially when the parallel ET scheme is used. Our work is directly comparable to [5] since they also implemented a decoder for 802.11n (1944, 972) QC-LDPC code. Although the GPU used in this work has approximately twice the amount of computation resource as in [5], our decoder achieves more than 50 times throughput compared to their work. This huge improvement can be attributed to our highly optimized algorithm mappings, efficient data structures and the memory access optimizations.

D. The Occupancy Ratio and Instruction Throughput Ratio

According to the profiling results of NVIDIA Visual Profiler tools, the occupancy of Kernel 1 is 0.833 and that of Kernel 2 is 1 (the occupancy is measured by the ratio of active warp). The occupancy ratio of our work is high which implies that our kernels have a sufficient number of threads to hide stalls. This is confirmed by looking at the instruction throughput ratio (ITR), that is the ratio of achieved instruction rate to peak single issue instruction ratio. The ITR is around 0.8 for this implementation. This shows that our algorithm mapping and workload partitioning are efficient and we are close to achieving the peak GPU throughput.

VII. CONCLUSION

This paper presents the techniques and design methodology to fully utilize GPU’s computational resources to accelerate a computation-intensive DSP algorithm. As a case study, a massively parallel implementation of LDPC decoder on GPU is presented. The challenges to map the algorithms to the computational resources in GPU are described. To achieve high decoding throughput, several techniques including the algorithmic optimizations, efficient data structures and memory access optimizations are employed. We take the LDPC decoder for the IEEE 802.11n WiFi LDPC code and 802.16e WiMAX LDPC code as examples to demonstrate the performance of our GPU-based implementation. The proposed GPU implementation of LDPC decoder shows great flexibility and reconfigurability. The simulation results exhibit that our LDPC decoder can achieve high throughput around up to 100.3Mbps.

REFERENCES