Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 7048482, 6 pages
http://dx.doi.org/10.1155/2016/7048482

Research Article

LDPC Decoding on GPU for Mobile Device

Yiqin Lu, Weiyue Su, and Jiancheng Qin

School of Electronic and Information Engineering, South China University of Technology, Tianhe District, Guangzhou, China

Correspondence should be addressed to Weiyue Su; weiyue.su@gmail.com

Received 17 May 2016; Revised 17 August 2016; Accepted 25 August 2016

Academic Editor: Mariusz Głąbowski

A flexible software LDPC decoder that exploits data parallelism for simultaneous multicode words decoding on the mobile device is proposed in this paper, supported by multithreading on OpenCL based graphics processing units. By dividing the check matrix into several parts to make full use of both the local memory and private memory on GPU and properly modify the code capacity each time, our implementation on a mobile phone shows throughputs above 100 Mbps and delay is less than 1.6 millisecond in decoding, which make high-speed communication like video calling possible. To realize efficient software LDPC decoding on the mobile device, the LDPC decoding feature on communication baseband chip should be replaced to save the cost and make it easier to upgrade decoder to be compatible with a variety of channel access schemes.

1. Introduction

Low Density Parity Check (LDPC) error correcting code is a kind of linear block codes, proposed by Gallager in 1962 [1] and rediscovered by Mackay and Neal in 1996 [2]. It takes its name from its sparse check matrix. LDPC codes are capacity-approaching codes, which means that it allows the noise threshold to be set very close to the Shannon limit for a symmetric memoryless channel; thus, the practical constructions of LDPC code exists.

Good performance of the LDPC code is at the cost of a very large amount of calculation. DCP decoding computation has very high parallel computation. The current commercial LDPC decoder is based on the hardware implementation, which only allows several kinds of specific LDPC codes at the same time and is difficult to upgrade. There are a large number of studies using FPGA to realize the efficient LDCP decoder [3, 4]. With the rapid development of the graphics processing units (GPU) on the desktop, there are a lot of researches using CUDA framework for LDPC decoding [5, 6]. The LDPC code is widely used in the fourth generation of mobile telecommunications technology, which makes it significant to develop efficient software LDPC decoding on the mobile device. At the same time, software LDPC code can dynamically change the parameters, including code length, code rate, and the number of iterations to quickly deal with all kinds of network environment.

Open Computing Language (OpenCL) [7] is a framework for writing programs that execute across heterogeneous platforms consisting of CPU, GPU, DSP, FPGA, and other processors or hardware accelerators. This technical specification was reviewed by the Khronos members and approved for public release on 2008. Compute Unified Device Architecture (CUDA) [8] also enables developers to develop parallel computing program on GPU at the desktop. OpenCL appears later, but it supports more scenarios. With the rapid development of mobile devices, many mobile devices especially mobile phone began to have their own high-performance GPU chips. Some vendors such as Qualcomm, Imagination PowerVR, ARM, and Vivante are beginning to support the OpenCL on their mobile GPU [9], which make developing parallel computing program on mobile devices based on GPU easier. In this article, we tried to develop a LDPC decoder on the mobile GPU based on the OpenCL. Nevertheless, the global memory is limited on a mobile GPU; therefore, the performance is not as good as on the desktop GPU. We improve the decoding through making full use of the local memory of each computing unit and the private memory of each processing unit. At the same time, we properly reduce the number of threads per code word and add code-words in decoding process, and better performance is obtained. In our experiments, as the best result in the decoder, the throughput reached 160 Mbps, which can satisfy the current mobile wireless communication in many cases, and delay time is less than 2 milliseconds (ms), which can satisfy many real-time applications like video calling.

2. MSA for LDPC Decoding

Belief propagation (BP) algorithm is a kind of important message passing algorithm, often used in the field of artificial intelligence [9]. Algorithm between each node transfers the belief information. For example, the belief information from bit node BN_n to check node CN_m depends on the observation of BN_n and all the check nodes BN_n connected with, except CN_m. Similarly, the belief information from check node CN_m to bit node BN_n depends on the observation of CN_m and all the bit nodes CN_m connected with, except BN_n. As a BP algorithm, the Min Sum Algorithm (MSA) is a very efficient LDPC decoding algorithm [10]. It is based on the belief propagation between nodes connected as indicated by the Tanner graph [11] edges. Figure 1 shows the Tanner graph of a particular 4×8 $\mathbf{H}$ matrix. MSA, proposed by Gallager, operates in the logarithmic probabilistic domain.

LDPC code is a special form of linear (N, K) block code, defined by sparse binary parity check $\mathbf{H}$ matrices of dimension $M \times N$, while $M = N - K$. We assume that the channel is an additive white Gaussian noise (AWGN) channel with the mean 0 and the variance σ^2. BPSK modulation maps a code-word $\mathbf{c} = (c_1, c_2, \ldots, c_N)$ onto the sequence $\mathbf{x} = (x_1, x_2, \ldots, x_N)$, according to $x_i = (-1)^{c_i}$. The received sequence is $\mathbf{y} = (y_1, y_2, \ldots, y_N)$, with $y_i = x_i + n_i$. In the case of receiving y_n, the logarithmic a priori probability of x_n is Lp_n^0. MSA is as shown in Figure 2.

Before entering the loop iteration, we use the received sequence $\mathbf{y}$ to initialize the prior probabilities of BN_n as follows:

$$\begin{aligned} Lp_n^0 &= \ln \frac{p_n^0(0)}{p_n^0(1)} = 2\frac{y_n}{\sigma^2}, \\ Lq_{nm}^0 &= Lp_n^0. \end{aligned} \tag{1}$$

In this algorithm, we do not compute the posterior probabilities of BN_n and CN_m directly; instead, we compute the message transferring between the bit nodes and check nodes as well as the posterior probabilities before hard decoding.

In the step of updating message CN_m to BN_n, for ith iteration, accessing $\mathbf{H}$ in row-major order, Lr_{mn}^i as the message sent from CN_m to BN_n is updated according to any bit nodes connected to CN_m in Tanner graph, except the BN_n. The update process, called minimum step, is as follows:

$$Lr_{mn}^i = \prod_{n' \in N(m) \setminus n} \operatorname{sign}\left(Lq_{n'm}^{i-1}\right) \min_{n' \in N(m) \setminus n} \left| Lq_{n'm}^{i-1} \right|. \tag{2}$$

Using the $\mathbf{H}$ matrix and Tanner graph in Figure 1, for instance, $Lr_{0,0}^i$ is updated by BN_1 and BN_2, as in Figure 3, $r_{0,0}^i = f(Lq_{1,0}^{i-1}, Lq_{2,0}^{i-1})$.

The posterior probabilities of BN_n is updated by the prior probabilities of BN_n and all the check nodes connected to BN_n:

$$Lp_n^i = Lp_n^0 + \sum_{m \in M(n)} Lr_{mn}^i. \tag{3}$$

Similarly, in the step of updating message BN_n to CN_m, for ith iteration, Lq_{nm}^i, as the message sent from BN_n to CN_m is updated according to any check nodes connected to BN_n in Tanner graph, except the CN_m. The update process is called sum step.

$$Lq_{nm}^i = Lp_n^i - Lr_{mn}^i. \tag{4}$$

Using the Tanner graph in Figure 1, for instance, $Lq_{0,0}^i$ is updated by CN_2, as in Figure 4, $Lq_{0,0}^i = f(Lr_{2,0}^i)$.

Actually, the steps of updating Lp_n^i and Lq_{nm}^i can be exchanged. If we update Lp_n^i first, the result of Lp_n^i can be used to update Lq_{nm}^i, which reduces the repeated computation.

The final hard decoding is performed at the end of an iteration.

$$c_n^i = \begin{cases} 1 & \text{if } Lp_n^i < 0 \\ 0 & \text{if } Lp_n^i > 0. \end{cases} \tag{5}$$

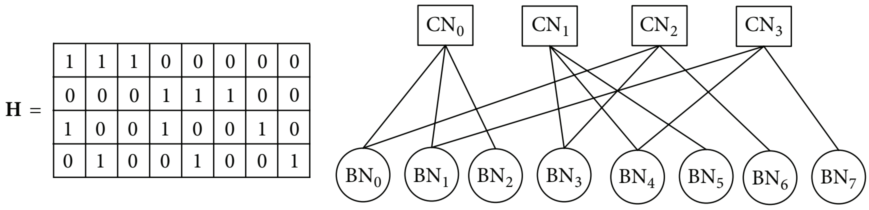
The iteration procedure is stopped if the decoded word $\mathbf{c}$ verifies all parity check equations $\mathbf{c}\mathbf{H}^T = 0$, or the maximum iteration is reached.

The implementation of decoder is achieved by a flood scheduling algorithm [12]. It guarantees that the bit nodes would not interfere with each other in the update step and when updating check nodes, check nodes will not interfere with each other too. Using this principle allows the true parallel execution of MSA for LDPC decoding based on the stream-based computing method.

3. OpenCL for Mobile GPU

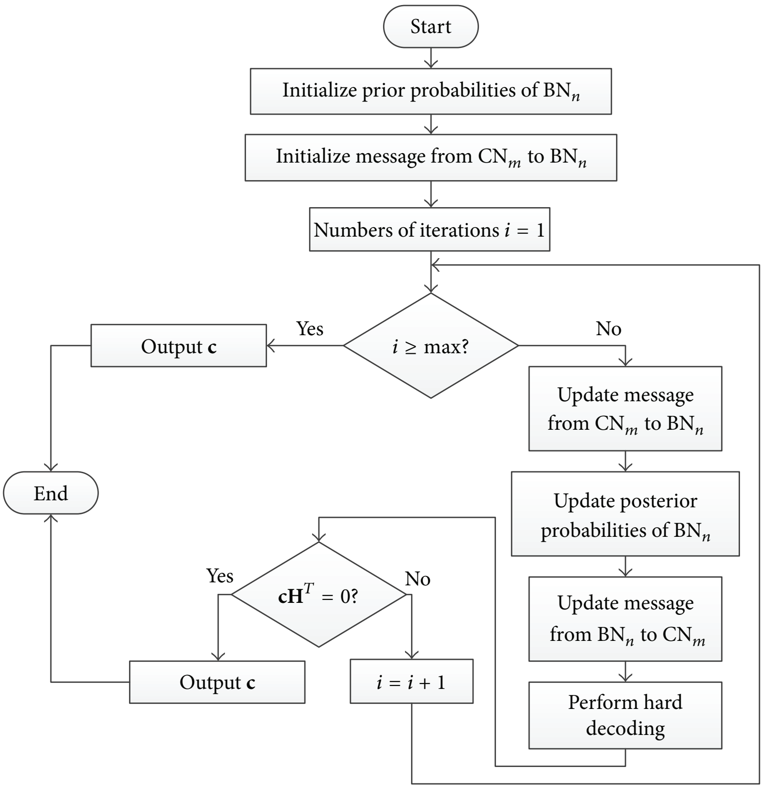
Modern GPU is based on ultra high parallel computing ability and programmable pipeline. Stream processor of GPU is able to do general-purpose computation [13]. GPU is more efficient than CPU floating point performance especially when we deal with the single instruction multiple data (SIMD) and the completion of compute-intensive tasks, in which data processing operation needs far more time than the data scheduling and data transmission [14].

Unlike the dedicated GPU for desktop computers, a mobile GPU is typically integrated into an application processor, which also includes a multicore CPU, an image processing engine, DSPs, and other accelerators [15]. Recently, modern mobile GPUs such as the Qualcomm Adreno GPU [16], the Imagination PowerVR GPU, ARM Mali, and GPGPU on Vivante tend to integrate more compute units in a chip. Mobile GPUs have gained general-purpose parallel computing capability thanks to the multicore architecture and emerging frameworks such as OpenCL, and they are likely to offer flexibility similar to vendor specific solutions designed for desktop computers, such as CUDA of Nvidia.



$$\mathbf{H} = \begin{array}{|c|c|c|c|c|c|c|c|}\hline 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

Figure 1: A 4 × 8 **H** matrix and its Tanner graph representation.



Start
Initialize prior probabilities of BN$_n$
Initialize message from CN$_m$ to BN$_n$
Numbers of iterations i = 1
$i \geq$ max?
Yes
No
Output **c**
End
Update message from CN$_m$ to BN$_n$
Update posterior probabilities of BN$_n$
Update message from BN$_n$ to CN$_m$
Perform hard decoding
Yes
$\mathbf{cH}^T = 0$?
No
Output **c**
$i = i + 1$

Figure 2: Process of Min Sum Algorithm.

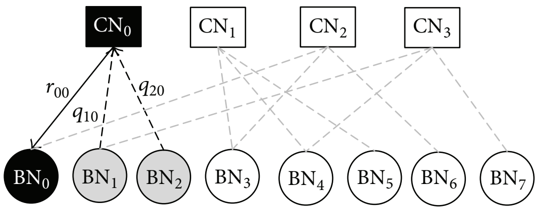


Figure 3: Example for updating $Lr^i_{0,0}$, the message from CN$_0$ to BN$_0$.

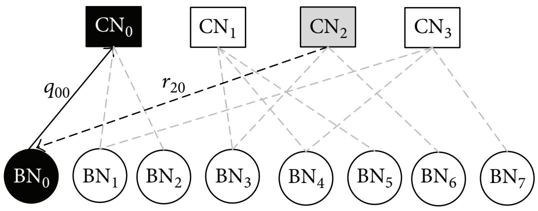


Figure 4: Example for updating $Lq^i_{0,0}$, the message from BN$_0$ to CN$_0$.

OpenCL is a programming framework designed for heterogeneous computing across various platforms [17]. In OpenCL, a host processor (typically a CPU) manages the OpenCL context and is able to offload parallel tasks to several compute devices (for instance, GPU).

The parallel jobs can be divided into work-groups, and each of them consists of many work-items which are the basic processing units to execute a kernel in parallel.

OpenCL defines a hierarchical memory model containing a large global memory but with long latency and a small but fast local memory which can be shared by work-items in the same work-group; what is more, each work-item has its own memory, which is not shared with other items and is fastest accessing.

To efficiently and fully utilize the limited computation resources on a mobile processor for better performance, we partition the tasks between CPU and GPU and explore the algorithmic parallelism, and memory access optimization needs to be carefully considered.

On embedded platform, to handle various tasks is becoming a trend. OpenCL specification describes a subset of the OpenCL specification for handheld and embedded platforms.

The OpenCL embedded profile has some restrictions; for instance, there are optional support for 3D images and no support for 64-bit integers and no support for 64-bit integers. The details of the OpenCL embedded profile can be found in Khronos's website [17].

Despite these specification restrictions, it is possible to use OpenCL to accelerate the program on the mobile devices. The compute-intensive computation on the mobile device is transferred to the GPU or other devices supporting OpenCL; not only these tasks can perform even more efficiently, but also CPU can handle more tasks that it is good at. Actually, LDPC decoding is a kind of traditional compute-intensive computation.

4. Parallel MSA LDPC Decoding on Mobile GPU

MSA is an intensive processing, which should be processed in a high-performance specific computing engine, or in a highly parallel programmable device. On the mobile device, the GPU is a good choice. This general model, supported by GPU using OpenCL, executes kernels in parallel on several multiprocessors. Each processor is composed by several cores that dispatch multiple threads. In this section, a parallel processing to save the information of matrix **H** into work-items is showed. In order to save the private memory, each work-item only keeps the compressed information that related to its own computation. After that, the specific parallel algorithm in OpenCL kernel is introduced. Given an (N, K) LDPC code, it is important to manage the computation to reduce the expenditure in parallel programming. Instead of using $M \times w_c$ work-items (w_c is the maximum column weight of matrix **H**), the model uses $M \times 1$ work-items in each work-group, and each work-item updates the message about one check node, which means M work-items work for M check nodes, respectively.

4.1. Compact Representation of the Tanner Graph. The Tanner graph of a LPDC code is defined as **H**. We propose it in two separate data structures, namely, H_{BN} and H_{CN}. This is because one iteration of the LDPC decoder can be decomposed into horizontal and vertical processing, which means we update message from CN_n to BN_m and message from BN_m to CN_n, respectively.

The data structure used in the horizontal step is defined as H_{BN}. It is generated by scanning the matrix **H** in a row-major order and mapping only the bit nodes' edges associated with nonnull elements in **H** used by a single check node equation in the same row. Algorithm 1 describes this procedure in detail for a matrix having M rows and N columns. H_{BN} is saved in the private memory. Because each work-item updates the message in the whole row, H_{BN} is not necessary to be accessed by any other work-items.

```
as the work-item $k$ in a work-group: do
    $m = k$
    for all $\mathrm{BN}_n$ (columns in $H_{mn}$): do
        if $H_{mn} == 1$ then
            $H_{\mathrm{BN}}[idx{+}{+}] = n$
```

Algorithm 1: Generating compact H_{BN} from matrix **H**.

```
as the work-item $k$ in a work-group: do
    for offset = from 0 to $N/M - 1$: do
        $n = k \times N/M + \text{offset}$
        for all $\mathrm{BN}_n$ (columns in $H_{mn}$): do
            if $H_{mn} == 1$ then
                $H_{\mathrm{CN}}[idx{+}{+}] = n$
```

Algorithm 2: Generating compact H_{CN} from matrix **H**.

The H_{CN} data structure is used in the vertical processing step. It can be defined as a sequential representation of the edges associated with nonnull value in **H**. It is generated by scanning the **H** matrix in a column-major order. H_{CN} is also saved in the private memory. Because each work-item updates the message in the neighbor N/M rows, H_{CN} is not necessary to be accessed by other work-items too.

4.2. Programming the MSA on the OpenCL Grid. Each work-group contains M work-items that represent threads. Instead of the whole matrix **H** or H_{BN}, each work-item can save the necessary part of information of H_{BN} in the private memory, which make access to perform the update faster. Again, the same principle applies to the update of Lq^i_{nm} messages.

According to LDPC code length, the CPU on mobile do allocate memory in GPU, including the global memory for storing the check matrix **H**, input data, output data, and the local memory for saving the message data sent from bit nodes to check nodes, marked as Lr^i_{mn} and from check nodes to the bit nodes, marked as Lq^i_{nm} (Algorithm 3).

In step (2), the compact H_{BN} and H_{CN} are generated in private memory by Algorithms 1 and 2.

The same as the normal MSA algorithm, the loop execution from step (3) will end until the output code word is current or it reaches the maximum loop times.

It executes a horizontal processing, a vertical processing, and a synchronization for all threads in steps (5)–(9). Generally, all threads should be synchronized after the horizontal and vertical processing, but in this algorithm, every work-item takes charge of its own check node, and Lr^i_{mn} data is not shared with other work-items, so it is able to cancel the synchronization after horizontal processing to improve performance. Lq^i_{nm} data is still shared with all work-items, so the synchronization after vertical processing is retained.

```
(1)  Initialize the work-group size (or number of work-item per work-group).
(2)  Generating compact $H_{\mathrm{BN}}$, $H_{\mathrm{CN}}$ from matrix $\mathbf{H}$
(3)  **while** $(\mathbf{cH}^{T} \neq 0 \cap i < I)$
(4)      **as** the work-item $k$ on an $M$ work-group: **do**
(5)          **for** all $H_{\mathrm{BN}}$: **do**
(6)              $m = k$
(7)              update the message sent from $\mathrm{BN}_n$ to $\mathrm{CN}_m$
(8)              update the message sent from $\mathrm{CN}_m$ to $\mathrm{BN}_n$
(9)          Synchronize all threads
(10)         **for** offset = 0 to $N/M - 1$: **do**
(11)             $n = k \times N/M + \text{offset}$
(12)             **for** all $H_{\mathrm{CN}}$[offset]: **do**
(13)             update the posterior probabilities of $\mathrm{BN}_n$
(14)         Synchronize all threads
(15)         perform hard decoding
```

Algorithm 3: MSA kernel executing on the GPU grid.

After the synchronization, it calculates the posterior probabilities of BN_n and every work-item deals with N/M bit nodes as in steps (10)–(13). After the second synchronization, it performs the hard decoding by posterior probabilities, according to the method described in Section 2.

True parallel execution is conducted and the overall processing time required to decode a code word can be significantly reduced as a result, as it will be seen in the next section. More data parallelism can be exploited by decoding several code words simultaneously, but it was not considered in this work.

5. Implementation and Experimental Results

The experimental setup to evaluate the performance of the proposed parallel LDPC decoder on the GPU consists of a PowerVR G6200 with 256 MB global memory and 4 KB local memory and was programmed using the C language and the OpenCL programming interface (version 1.1). In this algorithm, each code word is decoded in a work-group. Because of the limited local memory, only small LDPC code can be used in this test mobile phone. However, the work-group number can be large due to the relatively large global memory on the GPU.

To decode a batch of code words, whose original size is 1 Mbit, the variation in performance is minimal and in Figure 5 we show only the best results achieved. As a 144 × 576 matrix, the work-items per work-group are equal to their row number, which means we use 144 work-items per work-group and 1000 work-groups in this experiment.

The decoding times reported in Figure 5 define global processing times, including data transmission time and decoding time. The decoding time increases along with the increase of iterations. They have a linear relation. The computation capacity of GPU is fully used. The throughput decreases as iterations increase when iterations increase when the size of data for decoding remains the same.

On the mobile device we attach as much importance to the delay as the throughput. Figure 6 shows the decoding delay when the speed is from 10 Kbps to 100 Mbps. With the speed exponential increase, the delay increases but slowly. Actually, the size of data for decoding on the GPU in a decoding cycle is too small that some capacity of GPU is waste and the parallel effect is not obvious with low speed.

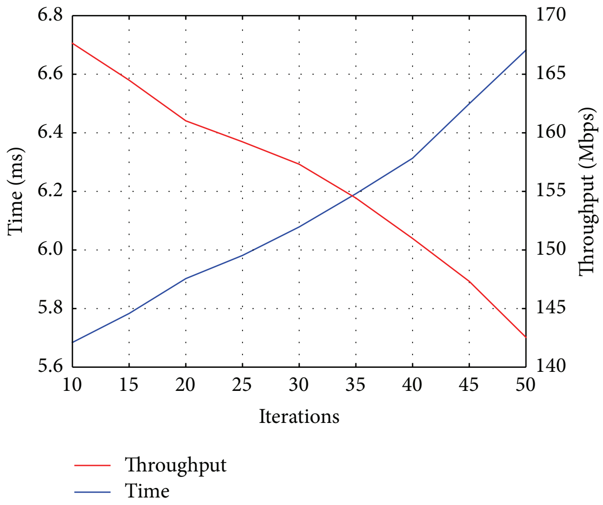


Figure 5: LDPC decoding times on the GPU and corresponding throughput using MSA.

It is obvious that the delay increases when code words for GPU decoding increase. However, the time of a decoding cycle, which is the most important part of the delay, increases but slowly thanks to the more fully use of the computation capacity. The mean time for a code word decreases in higher speed. Thus, it can be applied for some high-speed mobile services, like large file transmission, and delay-sensitive services like video calling.

6. Conclusion

This paper proposes a multicode word parallel LDPC decoder using a GPU on the mobile device running OpenCL. LDPC is widely used in the fourth generation of mobile telecommunications technology, so it is significant to realize high-speed LDPC decoding on the mobile devices.

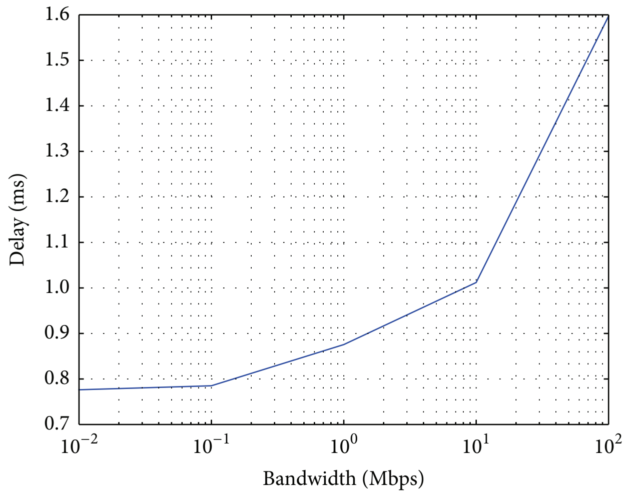


Figure 6: Decoding delay time in when the speed is from 10 Kbps to 100 Mbps.

For an instance, popular video calling software, Skype, has its bandwidth requirements noticed on its official website [18]. The bandwidth required by Skype depends on the type of calls. The minimum speeds required for normal screen sharing video calling, high-quality video calling, and HD video calling are 0.3 Mbps, 0.5 Mbps, and 1.5 Mbps. In the experiment above the decoding delay is 0.84 ms, 0.86 ms, and 0.98 ms in Figure 6. The HD video calling has less than 1 ms delay. It can meet its requirements apparently.

With the software realization of LDPC decoding on mobile devices, LDPC can dynamically change the parameters, including code length, code rate, and the number of iterations. All of them can be fast dynamic switched on OpenCL device, which can quickly deal with all kinds of network environment. With the bad network the code rate can be reduced to improve the ability of error correction, while the code rate can be improved when the network is fine. Compared with the traditional way of hardware decoding, our proposed decoding algorithm based on the software implementation of decoding on the mobile GPU is more efficient, for it can switch at any time according to actual environment.

Competing Interests

The authors declare that they have no competing interests.

References

[1] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[2] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, no. 18, pp. 1645–1646, 1996.

[3] T. Zhang and K. K. Parhi, "A 54 Mbps (3,6)-regular FPGA LDPC decoder," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SIPS '02)*, October 2002.

[4] D. Chang, F. Yu, Z. Xiao et al., "FPGA verification of a single QC-LDPC code for 100 Gb/s optical systems without error floor down to BER of 10-15," in *Proceedings of the 2011 Optical Fiber Communication Conference and Exposition and the National Fiber Optic Engineers Conference (OFC/NFOEC '11)*, Los Angeles, Calif, USA, March 2011.

[5] G. Falcão, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding," in *Proceedings of the 23rd International Conference on Supercomputing*, Yorktown Heights, NY, USA, June 2009.

[6] G. Falcão, L. Sousa, and V. Silva, "Massive parallel LDPC decoding on GPU," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pp. 83–90, Salt Lake City, UT, USA, February 2008.

[7] https://www.khronos.org/opencl/.

[8] http://www.nvidia.com/object/cuda_home_new.html.

[9] R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 2, pp. 140–152, 1998.

[10] J. Zhao, F. Zarkeshvari, and A. H. Banihashemi, "On implementation of min-sum algorithm and its modifications for decoding low-density parity-check (LDPC) codes," *IEEE Transactions on Communications*, vol. 53, no. 4, pp. 549–554, 2005.

[11] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, 1981.

[12] S. Lin and D. J. Costello, *Error Control Coding*, Pearson Education India, 2004.

[13] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for GPGPU stream architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, Vienna, Austria, September 2010.

[14] V. W. Lee, C. Kim, J. Chhugani et al., "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010.

[15] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU—a case study," in *Proceedings of the 38th IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '13)*, IEEE, Vancouver, Canada, May 2013.

[16] https://developer.qualcomm.com/category/tags/opencl.

[17] A. Munshi, *The OpenCL Specification*, Khronos OpenCL Working Group 1, 2009.

[18] http://skype.gmw.cn/help/content_69_579.html.

Advances in Multimedia
The Scientific World Journal
International Journal of Distributed Sensor Networks
Journal of Industrial Engineering
Applied Computational Intelligence and Soft Computing
Advances in Fuzzy Systems
Modelling & Simulation in Engineering
Journal of Computer Networks and Communications
Advances in Artificial Intelligence
Advances in Computer Engineering
International Journal of Computer Games Technology
International Journal of Biomedical Imaging
Advances in Artificial Neural Systems
Advances in Software Engineering
Journal of Robotics
Advances in Human-Computer Interaction
Computational Intelligence and Neuroscience
International Journal of Reconfigurable Computing
Journal of Electrical and Computer Engineering