International Journal of Parallel Programming (2020) 48:566–579
https://doi.org/10.1007/s10766-019-00632-3

GPU Accelerated Parallel Algorithm of Sliding-Window Belief Propagation for LDPC Codes

Bowei Shan[1] · **Yong Fang**[1]

Received: 31 May 2018 / Accepted: 12 March 2019 / Published online: 4 April 2019

Abstract

Low-Density Parity-Check (LDPC) codes are widely used from hard-disk systems to satellite communications. Sliding-Window Belief Propagation (SWBP) is an effective decoding algorithm of LDPC codes for time-varying channels and demonstrates near-optimal performance in many experiments. However, to adaptively find the best window size, SWBP may need very long computing time. Inspired by Graphics Processing Unit and Compute Unified Device Architecture, in this paper we propose a novel method to address the issue of SWBP's computing complexity. Different from sequential SWBP, we simultaneously compute the metrics of different window sizes in parallel, which enables us to quickly find the best window size. We use coalesced memory access to accelerate reading and writing processes. Registers and shared memory are also considered in our program to reduce memory latency. On the GTX 1080Ti platform, experimental results show that parallel SWBP can achieve about 14 $\times$ to 118 $\times$ speedup ratio for different regular LDPC codes, and about 8 $\times$ to 120 $\times$ speedup ratio for different irregular LDPC codes, respectively. According to the trend of our experiments, we strongly believe that, as the length of LDPC codes increases, a higher speedup ratio can be obtained.

Keywords LDPC · SWBP · GPU · CUDA

This work was supported by the Fundamental Research Fund for the Central Universities of China (Grant no. 300102249304), the Provincial Science Foundation of Shannxi, China-Key Project (Grant no. 2016JZ024), and the Provincial Foundation for Sci-Tech Youth Nova of Shaanxi, China (Grant no. 2014KJXX-41).

✉ Yong Fang
fy@chd.edu.cn

[1] School of Information Engineering, Chang'an University, Xi'an 710064, Shaanxi, China

1 Introduction

Low-Density Parity-Check (LDPC) codes are a class of advanced error-correcting codes broadly used in modern telecommunication. It was first invented by Gallager [1] in 1962 and rediscovered by MacKay and Neal [2] in 1996. Up to now, LDPC codes have been widely applied in many communication systems, e.g., 4G/5G, 10GBase-T Ethernet (IEEE 802.3an), Wi-Fi (IEEE 802.11n& 802.11ac), WiMAX (IEEE 802.16e), and DVB-S2.

The decoding of channel codes is an important issue. Belief Propagation (BP) is the optimal decoding algorithm of LDPC codes, in the sense that channel states are exactly known. Nevertheless, in the absence of exact knowledge of channel states, the optimal decoding of LDPC codes still remains open. In 2012, Fang [3] proposed the Sliding-Window BP (SWBP) algorithm to handle this issue, which can exactly trace time-varying channel state. In many experiments, this novel technique exhibits near-limit performance. Meanwhile, it is easy to be implemented and insensitive to initial settings. In addition, another outstanding advantage of SWBP is its convenience for parallelization. Although SWBP is very attractive in practice, it suffers from intensive computation. In SWBP, the optimal window size is the key factor for estimating channel state, which is found by exhaustively evaluating all possible window sizes. If SWBP is implemented serially, more running time is needed for longer LDPC codes. Therefore, parallelization of SWBP is indispensable.

In recent years, Graphics Processing Units (GPUs) demonstrate powerful computing ability and evolve into General Purpose Computing Units. GPUs can accelerate many computationally intensive problems by thousands of parallel executing threads. In addition, NVIDIA provides the Compute Unified Device Architecture (CUDA) as a programming interface for researchers to develop applications on GPUs in C-like languages.

Since BP-based LDPC decoding algorithm can be easily implemented in parallel, many LDPC decoders accelerated by GPUs have been presented by researchers [4,5]. However, to our best knowledge, parallel SWBP still remains uninvolved. In this paper, we speed up the SWBP algorithm on GPUs. There are two major bottlenecks in SWBP. The first one is belief-passing of standard BP between Variable Nodes (VNs) and Check Nodes (CNs). Since this problem has been addressed by many parallel algorithms, we will not further discuss it in this paper, and the reader can refer to [6–8] for details. The second one is estimating the best window size, which is a time-consuming process. We will tackle this problem by simultaneously computing the metrics of different window sizes. The coalesced memory access will be applied to accelerate the reading and writing operations. The intermediate variables will be stored in registers and shared memory to reduce memory latency. We will use three different algorithms, i.e., thrust, cuBLAS, and reduction kernel, to find the minimum element of an array.

The rest of this paper is arranged as below. Section 2 reviews the background knowledge of SWBP and GPU. Section 3 introduces the parallelized SWBP on GPUs using CUDA. Section 4 reports experimental results. Finally, Sect. 5 concludes this paper.

2 Review on Background

2.1 Preliminaries

Let n be the length of the used LDPC code, which is specified by an $m \times n$ parity check matrix $\mathbf{H}$. Let $\mathbf{x} = (x_1, \ldots, x_n)^\top$ be the codeword sent by the encoder and $\mathbf{y} = (y_1, \ldots, y_n)^\top$ be the codeword received at the decoder. Let us multiply $\mathbf{H}$ with $\mathbf{x}$ to get syndrome $\mathbf{c} = (c_1, \ldots, c_m)^\top = \mathbf{Hx}$. For simplicity, this paper considers the memoryless Binary Symmetric Channel (BSC) model, i.e., $\Pr(x_i \neq y_i) = p_i$. If the channel is stationary, then $p_1 = \cdots = p_n$.

BP is a popular algorithm to decode LDPC code. To introduce it, we first define some notations. Let $\mathbf{v} = (v_1, \ldots, v_n)^\top$ denote the overall Log-likelihood-Ratios (LLRs) of VNs, including both intrinsic and extrinsic LLRs. Let q_{ij} be the LLR propagated from the ith VN to the jth CN and r_{ji} be the LLR propagated from the jth CN to the ith VN. Let dv_i be the degree of the ith VN and dc_j be the degree of the jth CN. Let $\tilde{\mathbf{x}}$ be the estimation of $\mathbf{x}$. We illustrate the standard BP in Algorithm 1.

Algorithm 1 BP Algorithm

Input: $\mathbf{c}$, $\mathbf{y}$, and $\mathbf{p}$
Output: $\tilde{\mathbf{x}}$
Initialization: $v_i^{(0)} = (1 - 2y_i) \log \frac{1-p_i}{p_i}$ and $r_{ji}^{(0)} = 0$
1: **for** $l = 1 :$ MAX_ITERATION **do**
BP from VNs to CNs:
2: **for** $i = 1 : n$ **do**
3: $q_{ij}^{(l)} = v_i^{(l-1)} - r_{ji}^{(l-1)}$
4: **end for**
BP from CNs to VNs:
5: **for** $j = 1 : m$ **do**
6: $\tanh\left(\frac{r_{ji}^{(l)}}{2}\right) = (1 - 2c_j) \prod_{k=1}^{dc_j} \tanh\left(\frac{q_{kj}^{(l)}}{2}\right) \Big/ \tanh\left(\frac{q_{ij}^{(l)}}{2}\right)$
7: **end for**
Computing overall beliefs for VNs:
8: **for** $i = 1 : n$ **do**
9: $v_i^{(l)} = v_i^{(0)} + \sum_{k=1}^{dc_i} r_{ki}^{(l)}$
10: **end for**
Hard decision:
11: $\tilde{x}_i = \begin{cases} 0, & \text{if } v_i > 0 \\ 1, & \text{if } v_i < 0 \end{cases}$
12: **if** $\mathbf{H}\tilde{\mathbf{x}} = \mathbf{s}$ **then**
13: quit loop
14: **end if**
15: **end for**

The LDPC decoder with SWBP algorithm includes three phases: standard BP, window size setting, and local bias probability refinement. Detailed explanation of SWBP can be found in [3]. Since many researchers have used GPUs to accelerate BP algorithm in LDPC decoder [4,5], we will ignore standard BP phase in this paper.

2.2 Bias Probability Estimation

To estimate the bias probability of each BSC sub-channel, we re-compute local bias probability $\tilde{p}_i$ by averaging the overall beliefs of neighboring variable nodes in a size-s (an odd) window around y_i. Let $b_i = \frac{1}{1+\exp(v_i)}$. Then

$$\tilde{p}_i = \frac{-b_i + \sum_{i'=\max(1,i-h)}^{\min(i+h,n)} b_{i'}}{\min(i+h,n) - \max(1,i-h)}, \tag{1}$$

where $h = \lfloor s/2 \rfloor$. Obviously, (1) can be easily deduced as

$$\tilde{p}_i = \begin{cases} \tilde{p}_{i-1} + \dfrac{b_{i-1} - b_i + b_{i+h} - \tilde{p}_{i-1}}{i+h-1}, & 2 \le i \le (1+h) \\ \tilde{p}_{i-1} + \dfrac{b_{i-1} - b_i + b_{i+h} - b_{i-h-1}}{i+h-1}, & 2+h \le i \le (n-h) \\ \tilde{p}_{i-1} + \dfrac{b_{i-1} - b_i + b_{i-h-1} - \tilde{p}_{i-1}}{n-i+h}, & n-h+1 \le i \le n \end{cases}. \tag{2}$$

2.3 Window-Size Setting

Estimating an appropriate s is the key step of SWBP, where s is the window-size. To address this problem, [3] uses Mean Squared Error (MSE) as the metric. For each possible s, $\tilde{\boldsymbol{p}}$ is first calculated according to (2) and then the MSE between $\boldsymbol{b}$ and $\tilde{\boldsymbol{p}}$ is calculated. Finally, the best s that gives the smallest MSE is obtained. This problem can be solved by Algorithm 2.

Algorithm 2 Window-Size Setting Algorithm

Input: $\boldsymbol{b}$: overall beliefs
Output: optimal s^*
1: **Initialization:** $s = 1$ and σ^2 =FLOAT_MAX;
2: **for** $s < n$ **do**
3: compute $h = \lfloor s/2 \rfloor$;
4: compute $\tilde{p}_i$ by equation (2);
5: compute $\sigma_s^2 = \frac{1}{n} \sum_{i=1}^{n} (b_i - \tilde{p}_i)^2$;
6: **if** $\sigma^2 > \sigma_s^2$ **then**
7: $\sigma^2 = \sigma_s^2$;
8: $s^* = s$;
9: **end if**
10: $s = s + 2$;
11: **end for**

It is very reasonable that the best window-size s should minimize the MSE between overall beliefs and local bias probabilities.

Table 1 Running time of different parts of SWBP algorithm

Part	Running time (s)
Load LDPC code	0.92
Encode	2.38
Decode preparing	0.49
Standard BP	98.44
Select best window size	178.51

2.4 Complexity Analysis of SWBP

In the window size setting phase, for each possible s, about $4n$ additions/subtractions and n divisions are performed to compute $\tilde{\boldsymbol{p}}$. Then n subtractions and n multiplications are performed to compute each $(b_i - \tilde{p}_i)^2$. Finally, n-1 additions and 1 division are performed to get $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(b_i - \tilde{p}_i)^2$. Therefore, we need in total $8n$ operations of addition/subtraction/multiplication/division for each s. To find the best s, one needs to try each odd s between 1 to n, which needs $h = \lfloor s/2 \rfloor$ window size setting iterations, where $\lfloor \cdot \rfloor$ denotes the flooring function. In [3], the minimum search step of different window-size was set to 20 to reduce the computing complexity. In our experiment, we find that this interval is so big that the best window-size s may be omitted. Thus we fix the minimum interval of searched window-size to 2.

In the local bias probability refinement phase, about $4n$ additions/subtractions and n division are performed to refine $\tilde{\boldsymbol{p}}$ in each BP iteration. Since the outputs of successive BP iterations are usually very similar, it is unnecessary to refine $\tilde{\boldsymbol{p}}$ after each BP iteration. In [3], source bias probability is re-estimated after every 10 BP iterations (except explicit declaration), which is a good tradeoff between performance and complexity. Compared to window size setting phase, it is clear that the computing complexity of this phase can be ignored.

According to the above analysis, the total computing complexity is $\mathcal{O}(4n^2)$ in each SWBP iteration. The bottleneck of the SWBP lies in window size setting phase (standard BP is ignored in this paper). It is obvious that for long LDPC codes (n is large), SWBP is an algorithm with heavy computing complexity.

To verify aformentioned analysis, we employ a regular LDPC code with length 2000 as the input of SWBP algorithm, and measure the running time of different modules. The test result is listed in Table 1, which shows that selecting best window size is the bottleneck .

2.5 Features of GPUs

Originally, GPUs are designed to accelerate the creation of images for output to a display device. With its rapid progress, highly parallel structure of modern GPUs makes them more efficient to compute the large block of data in parallel. For example, with its 3584 CUDA cores, NVIDIA GTX 1080Ti has powerful computing ability. One GPU has thousands of threads. All threads can run in parallel. The memory architecture of the GPU is depicted in Fig. 1. The Global, Local and Texture memory

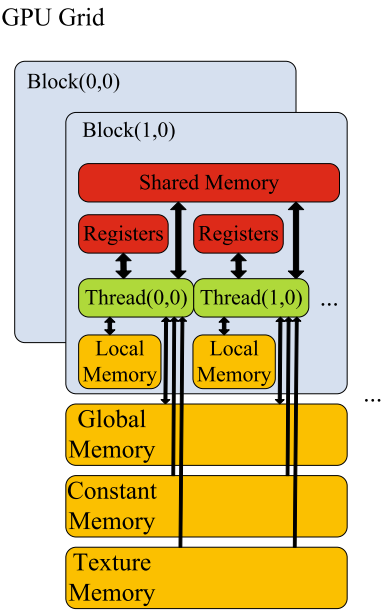

Fig. 1 GPU memory architecture

has larger capacity with lower speed while Shared memory and registers are fast but scarce. To optimize program, we should take care of memory allocation.

Meanwhile, NVIDIA presented CUDA as a parallel computing platform and application programming interface. The CUDA platform is designed to work with C-like programming languages. Therefore, software engineers can easily use CUDA to development parallel program to use GPU resources. On CUDA platform, we can access the compute kernels which is the parallel computational elements. CUDA compute capability version specifies the maximum parallel elements, such as threads and blocks.

2.6 Testing Platform

In our experiments, two different test platforms will be used to investigate the speed up effect. The sequential SWBP will run on CPU platform and the parallel SWBP will run on GPU platform, respectively. The specifications of these two platforms are listed in Table 2.

3 Parallel SWBP

3.1 Algorithm

In sequential SWBP algorithm, each window size setting iteration will generate one MSE σ^2 and each σ_i^2 is calculated by b_i, s_i and p_i. Therefore, any two MSEs σ_i^2 and σ_j^2 $(i \neq j)$ have no data correlation and can be simultaneously calculated. In our

Table 2 Specification of CPU platform and GPU platform

	CPU platform	GPU platform
Name	Intel Core i7 6850K	NVIDIA GTX 1080Ti
Core name	Broadwell	Pascal
OS	Window server 2008 enterprise	Window server 2008 enterprise
Frequency	3.60 GHz	1.58 GHz
Technology	14 nm	16 nm
Details	Cores: 6	CUDA capability version: 6.1
	Threads: 12	CUDA cores: 3584
	L1 cache: 384 KB	SMs: 28
	L2 cache: 1.5 MB	Threads per block: 1024
	L3 cache: 15 MB	Each dimension of a block: 1024 × 1024 × 64
	Memory: 64 GB DDR4	Warp size: 32
		Constant memory: 64 KB
		Shared memory per block: 48 KB
		Memory: 11 GB GDDR5

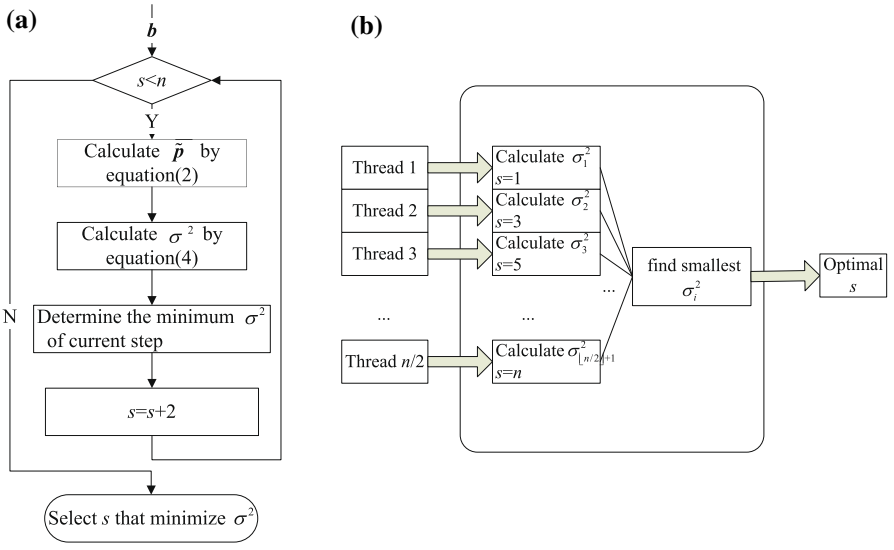

Fig. 2 **a** Sequential SWBP algorithm. **b** Parallel SWBP algorithm

parallel SWBP, all σ_i^2's will be calculated simultaneously by thousands of threads on GPU. These two algorithms are illustrated in Fig. 2.

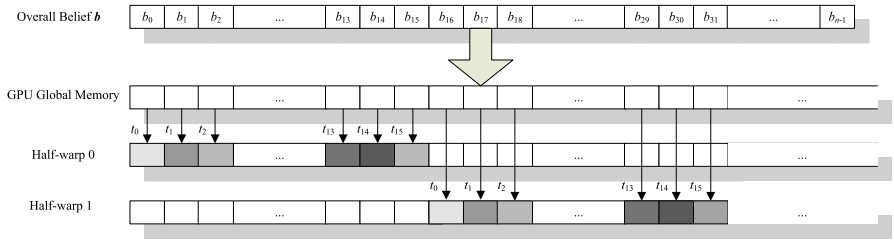

Fig. 3 Coalesced memory access

3.2 Coalesced Memory Access

In our SWBP algorithm, the input is overall belief vector $\boldsymbol{b}$, which is stored in global memory of GPU. Because global memory has large capacity, it can be easily made use of by large array. But global memory is an off-chip memory, much slower than on-chip memory. In order to reduce the access time to global memory while reading $\boldsymbol{b}$, coalesced memory access should be considered. Instead of performing 16 individual memory accesses, all the 16 threads of a half-wrap should access the global memory of GPU in a single read. The elements of $\boldsymbol{b}$ have to lie on a contiguous memory block, where the kth thread accesses the kth data element. The principle of coalesced memory access is illustrated in Fig. 3.

3.3 Using Shared Memory

Shared memory is an on-chip memory with lower latency and high bandwidth. The speed of shared memory is the same as that of registers and hundreds times faster than that of global memory. Note that the capacity of shared memory is small and has only 48KB per block. The shared memory should be allocated to variables carefully in order not to exceed the capacity. We can find in Fig. 3 that all threads in one block can access variables in the same shared memory. Therefore, we define an array in shared memory, and two variables in registers:

```
1 __shared__ float b[n];
2 float bias;
3 float sum;
```

where b[n] loads the overall belief stored in global memory into shared memory. $bias$ stores the local bias probability $\tilde{p}_i$ calculated by Eq. (2), and sum stores the current sum of $(b_i - \tilde{p}_i)$. Since these two variables are frequently used in each thread, the total access time will be reduced. σ_i^2 will be calculated by accumulating sum. Because σ_i^2 is only used for comparison purpose, the operation of division by n can be ignored.

3.4 Find Best Window Size

In global memory, we define an array dev_sigma2 with length n to store all σ_i^2's, each of which is calculated by one thread. The best window size s should give the smallest

σ_i^2. Then the problem becomes finding the smallest element of array dev_sigma2. To solve this problem, we investigated three parallel algorithms: (1) thrust; (2) cuBLAS; and (3) reduction kernel algorithm.

3.4.1 Thrust

Thrust is a powerful library of parallel algorithms and data structures [10]. Thrust provides a flexible, high-level interface for GPU programming. In thrust library, $min\_element()$ function could return the minimum element of array. Therefore, we use the following code to call thrust library, where the variable $index$ is the index of the minimum element.

```
thrust::device_ptr<float> d_ptr
  = thrust::device_pointer_cast(dev_sigma2);
thrust::device_vector<float>::iterator d_it
  = thrust::min_element(d_ptr, d_ptr + code_length);
index
  = d_it-(thrust::device_vector<float>::iterator)d_ptr;
```

3.4.2 cuBLAS

The cuBLAS [9] library is a fast GPU-accelerated implementation of the standard Basic Linear Algebra Subroutines (BLAS). In cuBLAS library, $cublasIsamin()$ function will return the minimum element of array. Therefore, we use the following code to call cuBLAS library, where the variable $index$ is the index of the smallest element.

```
cublasStatus_t status;
cublasHandle_t h;
status = cublasCreate(&h);
cublasIsamin (h, code_length, dev_sigma2,  1, &index);
cublasDestroy(h);
```

3.4.3 Reduction Kernel

To achieve the fastest speed, we design a reduction search algorithm. We implement a kernel function $min\_kernel()$ and present the code as following.

```
#define nB 256
#define MAX_KERNEL_BLK 30
#define MAX_BLK ((n/nB)+1)
#define MIN(a,b) ((a>b)?b:a)
#define FLOAT_MAX 1.0e30
__device__ volatile float block_vals[MAX_BLK];
__device__ volatile int   block_idxs[MAX_BLK];
__device__ int   block_num = 0;
__global__ void min_kernel(const float *data,
                 const int dsize, int *result){
__shared__ volatile float   vals[nB];
__shared__ volatile int idxs[nB];
__shared__ volatile int last_block;
```

```
int idx = threadIdx.x+blockDim.x*blockIdx.x;
last_block = 0;
float this_val = FLOAT_MAX;
int this_idx = -1;
while (idx < dsize){
if (data[idx] < this_val)
  {this_val = data[idx];
   this_idx = idx;}
idx += blockDim.x*gridDim.x;}
vals[threadIdx.x] = this_val;
idxs[threadIdx.x] = this_idx;
__syncthreads();
for (int i = (nB>>1); i > 0; i>>=1){
if (threadIdx.x < i)
if (vals[threadIdx.x] > vals[threadIdx.x + i])
  {vals[threadIdx.x] = vals[threadIdx.x+i];
   idxs[threadIdx.x] = idxs[threadIdx.x+i];}
__syncthreads();}
if (!threadIdx.x){
block_vals[blockIdx.x] = vals[0];
block_idxs[blockIdx.x] = idxs[0];
if (atomicAdd(&block_num, 1) == gridDim.x - 1)
last_block = 1;}
__syncthreads();
if (last_block){
idx = threadIdx.x;this_val = FLOAT_MAX;this_idx = -1;
while (idx < gridDim.x){
if (block_vals[idx] < this_val)
  {this_val = block_vals[idx]; this_idx =
  block_idxs[idx];}
idx += blockDim.x;}
vals[threadIdx.x] = this_val; idxs[threadIdx.x] =
 this_idx;
__syncthreads();
for (int i = (nB>>1); i > 0; i>>=1){
if (threadIdx.x < i)
if (vals[threadIdx.x] > vals[threadIdx.x + i])
  {vals[threadIdx.x] = vals[threadIdx.x+i];
  idxs[threadIdx.x] = idxs[threadIdx.x+i]; }
__syncthreads();}
  if (!threadIdx.x)
  *result = idxs[0];}
}
```

In our *main*() function, we call *min_kernel*() to get the index of the smallest element.

```
min_kernel<<<MIN(MAX_KERNEL_BLK, ((n+nB-1)/nB)), nB>>>
(dev_sigma2, n, index);
```

In Sect. 4, codes of different lengths are used in our experiments. It is found that our own designed reduction kernel performs the best.

Table 3 Regular LDPC code parameters (N is codeword length, K is information bit number)

Test	1	2	3	4	5	6
N	504	1008	2000	4000	8000	10,000
K	252	504	1000	2000	4000	5000
Maximum degree	7	8	7	7	7	7
Code type	Regular	Regular	Regular	Regular	Regular	Regular

Table 4 Parallel SWBP running time with different algorithms

Codeword length	504	1008	2000	4000	8000	10,000
With thrust (ms)	4.209	6.530	12.264	22.337	42.555	52.936
With cublas (ms)	2.607	5.107	9.996	20.100	40.250	50.557
With reduction kernel (ms)	2.539	5.061	9.941	20.056	40.210	50.514

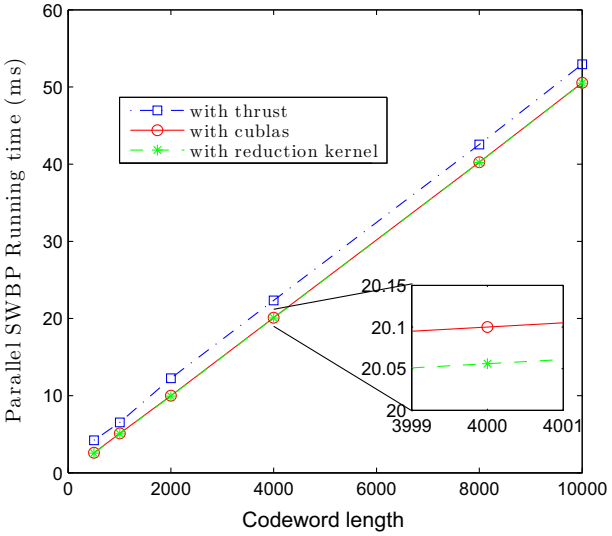

Fig. 4 Parallel SWBP running time with different algorithms

4 Experiment Results

Our experiment platforms are listed in Table 2. We will perform 3 experiments to investigate the speed up effect of our parallel SWBP algorithm. The first and second experiments will use regular LDPC codes as input. The parameters of LDPC code are listed in Table 3.

In our first experiment, we compare 3 parallel versions of SWBP with different algorithms listed in Sect. 3.4. In order to eliminate randomness, we perform 200 times tests and calculate the average running time. Our experiment results are listed in Table 4 and illustrated in Fig. 4. The experiment result shows that our own designed reduction ker-

Table 5 Running time and speedup ratio

Codeword length	504	1008	2000	4000	8000	10,000
Sequential SWBP (ms)	36.25	128.20	280.74	1020.57	3856.91	5991.36
Parallel SWBP (ms)	2.539	5.061	9.941	20.056	40.210	50.514
Speedup ratio	14.28	25.33	28.24	50.89	95.92	118.61

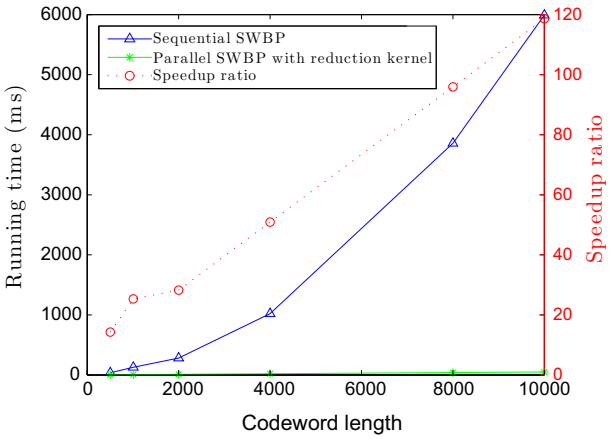

Fig. 5 Running time and speedup ratio (regular LDPC code)

nel algorithm achieves the fastest speed. Thrust algorithm is the slowest and cuBLAS algorithm is in between. Although the speed of cuBLAS is almost the same as that of reduction kernel, it should be noticed that we only count the running time of function $cublasIsamin()$, while that of functions $cublasCreate()$ and $cublasDestroy()$ is not included. In fact, these two functions cost about 100 milliseconds on this platform.

In our second experiment, we compare sequential SWBP and parallel SWBP algorithm. Since reduction kernel has achieved the fastest performance, we select parallel SWBP with reduction kernel as the representative of parallel SWBP. Our experiment results are listed in Table 5 and illustrated in Fig. 5. The experiment results show that our parallel SWBP algorithm obtained 14 $\times$ to 118 $\times$ speedup ratio for different LDPC codeword lengths, and as codeword length increases, the speedup ratio rises tremendously. According to the trend of Fig. 5, we believe that if we use longer LDPC codes, higher speedup ratio can be obtained.

In our last experiment, we use irregular LDPC codes as the input to investigate our parallel SWBP algorithm. The parameters of irregular LDPC code are listed in Table 6. Our experiment results are listed in Table 7 and illustrated in Fig. 6. The experiment results show that the parallel SWBP obtained 8 $\times$ to 120 $\times$ speedup ratio and the changing trend of speedup ratio is the same as that of regular LDPC codes.

Table 6 Irregular LDPC code parameters (N is codeword length, K is information bit number)

Test	1	2	3	4	5	6
N	504	1008	2000	4000	8000	10,000
K	252	504	1000	2000	4000	5000
Maximum degree	8	9	10	8	8	8
Code type	Irregular	Irregular	Irregular	Irregular	Irregular	Irregular

Table 7 Runnning time and speedup ratio (irregular LDPC code)

Codeword length	504	1008	2000	4000	8000	10,000
Sequential SWBP (ms)	20.691	109.795	371.414	999.975	3843.97	5991.36
Parallel SWBP (ms)	2.529	4.986	9.847	19.819	39.4884	50.514
Speedup ratio	8.18	22.02	37.72	50.46	97.34	120.34

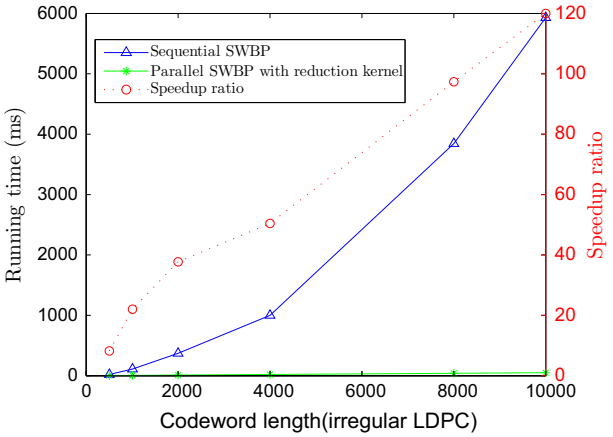

Fig. 6 Running time and speedup ratio (irregular LDPC code)

5 Conclusion

We proposed a parallel SWBP algorithm to decode LDPC codes. This algorithm was implemented on CUDA platform and accelerated by NVIDIA GTX 1080Ti GPU. Different from sequential SWBP, parallel SWBP simultaneously estimates the metrics of different window sizes by thousands of threads of GPU. By taking good care of memory architecture of GPU, the reading and writing time were also reduced. We carefully design a reduction kernel to find the smallest element of a long array in parallel, and this algorithm achieved better performance than thrust and cuBLAS algorithms.

To investigate the speedup effect, we use CPU and GPU platforms for sequential SWBP and parallel SWBP respectively. The experiment results show that parallel SWBP achieved about 14 × to 118 × speedup ratio for different regular LDPC codes,

and about 8 × to 120 × speedup ratio for different irregular LDPC codes. From the trend of above experiments, we expect higher speedup ratio for longer LDPC codes.

All source codes of this paper can be found in [11]. Readers can download it for academic purpose.

Acknowledgements We would like to thank colleagues in School of Information Engineering, Chang'an University, for their useful suggestions.

References

1. Gallager, R.G.: Low-density parity-check codes. IRE Trans. Inf. Theory **8**(1), 21–28 (1962)
2. Mackay, D.J.C., Neal, R.M.: Near shannon limit performance of low density parity check codes. Electron. Lett. **32**(6), 457–458 (1997)
3. Fang, Y.: Ldpc-based lossless compression of nonstationary binary sources using sliding-window belief propagation. IEEE Trans. Commun. **60**(11), 3161–3166 (2012)
4. Chang, C.C., Chang, Y.L., Huang, M.Y., Huang, B.: Accelerating regular LDPC code decoders on GPUS. IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. **4**(3), 653–659 (2011)
5. Dai, Y., Fang, Y., Yang, L., Jeon, G.: Graphics processing unit-accelerated joint-bitplane belief propagation algorithm in DSC. J. Supercomput. **72**(6), 2351–2375 (2016)
6. Pai, Y.S., Shen, Y.C., Wu, J.L.: High efficient distributed video coding with parallelized design for LDPCA decoding on CUDA based GPGPU. J. Vis. Commun. Image Represent. **23**(1), 63–74 (2012)
7. Park, J.Y., Chung, K.S.: Parallel LDPC decoding using CUDA and OPENMP. Eurasip J. Wirel. Commun. Netw. **2011**(1), 1–8 (2011)
8. Falcao, G., Sousa, L., Silva, V.: Massively LDPC decoding on multicore architectures. IEEE Trans. Parallel Distrib. Syst. **22**(2), 309–322 (2010)
9. NVIDIA: cublas. https://developer.nvidia.com/cublas. Accessed 10 Apr 2018
10. Thrust. http://thrust.github.io/. Accessed 10 Apr 2018
11. Source codes. http://js.chd.edu.cn/xxgcxy/dbw_en/list.htm. Accessed 10 Apr 2018