BER Guaranteed Optimization and Implementation of Parallel Turbo Decoding on GPU

Xiang Chen[†1,2], Ji Zhu[*], Ziyu Wen[‡], Yu Wang[‡], Huazhong Yang[‡]

[†1]Aerospace Center, School of Aerospace, Tsinghua University, Beijing 100084
[†2]Aerospace Communications and Terminal Application Technologies Engineering Laboratory, Shenzhen, 518057
[*]Department of Electronic Engineering, Xidian University, Xi'an, Shanxi, 710126
[‡]Department of Electronic Engineering, TNList, Tsinghua University, Beijing 100084, China
Email: chenxiang@tsinghua.edu.cn

***Abstract*—In this this paper, we present an optimized parallel implementation of a Bit Error Rate (BER) guaranteed turbo decoder on a General Purpose Graphic Process Unit (GPGPU). Actually, it is a critical task to implement complex communication signal processing over GPGPUs, since the parallelism over GPGPUs in general requires independent data streams for processing. So we explore both the inherent parallelisms and the extended sub-frame level parallelisms in turbo decoding and map them onto the recent GPU architecture. A guarding mechanism called Previous Iteration Value Initialization with Double Sided Training Window (PIVIDSTW) is used to minimize the loss of BER performance caused by sub-frame level parallelism, while the high throughput is still maintained. In addition, to explore the potential of parallelization in Turbo decoding on GPUs, the theoretical occupancy and scalability are analyzed with the consideration of the number of sub-frames per frame. Compared with previous work in [5] and [7], we achieve a better trade-off between BER performance and throughput concerns.** [1]

I. Introduction

Turbo codes [3] have become one of the most widely used channel codes in wireless standards benefited from their excellent Bit Error Rate (BER) performance near the Shannon limit. However, the inherent serial decoding structure and complex iterative decoding algorithm make it a great challenge to implement a Turbo decoder for any real time systems. As a result, Turbo decoders are often implemented on ASIC or FPGA [4], which performs high throughput at the expense of flexibility.

Compared with ASIC or FPGA solutions in wireless communication applications, GPU provides much more flexibility while maintaining high throughput and becomes one of the best GPP for SDR applications. Some GPU implementations of Turbo decoder have been proposed [2], [5], [7]. Wu et al. [5] implemented a Turbo decoder with a high throughput at the cost of reduced BER performance while Yoge et al. [7] presented the PIVIDSTW guarding mechanism, which can efficiently mitigate this BER performance degradation. Both Yoge et al. [7] and Wu et al. [5] explain the exploitable parallelisms in Turbo decoding, which consists of the trellis state level parallelism and sub-frame level parallelism.

Compared to the previous work, we further explore the parallelism in Turbo decoding algorithms. In addition to trellis state level parallelism, two other kinds of inherent parallelism, traversal-direction level parallelism and trellis stage level parallelism of log-likelihood ratios (LLRs), are discussed. Instead of doing backward traversal through the trellis after the completion of forward traversal, we perform the two-direction traversals simultaneously. And to achieve further parallelism and hence larger throughput in LLRs computation, we split the computation of backward state metrics and state LLRs, which allows the LLRs at different stages to be calculated in parallel.

Instead of assigning a single sub-frame per thread block [5], [7], we allocated multiple warps [2] per block to process the corresponding sub-frames, which increases the thread utilization on GPU. In addition, we evaluate the occupancy of our GPU implementation and show how the number of sub-frames affects it. To minimize the BER performance degradation caused by sub-frame level parallelism, proper guarding mechanism called PIVIDSTW [7] is applied. Benefited from the optimization on parameters in the GPU implementation, we achieve a better trade-off between BER performance and throughput concerns.

Through it is reported in [6] that the throughput of Turbo decoding by Max-Log-MAP algorithm on a GPU is approaching 30Mbps with 5 ierations, it depends on a batch loading of 2048 codewords (i.e., frames) into GPU memory and decoding all of them at a time in parallel. Such operation will bring large latency to cache the received so many codewords and to output to next step in the system, which will be unacceptable for general communication systems with critical round-trip protocol processing delay constraints. So our work focuses on optimal parallelism in decoding one frame, but not multiple frames, and could be easily combined with the frame level parallelism in [6]. In this paper, we take 3GPP LTE standard Turbo codes as an example.

[1]This work was supported by National Basic Research Program of China (2013CB329000), National Science and Technology Major Project (2011ZX03003-003-01, 2013ZX03003013-003), National Natural Science Foundation of China (No.61132002, 61028006, 61076035), and Tsinghua University Initiative Scientific Research Program. The graphics hardware was supported by NVIDIA Corporation.

[2]The definition of warp will be given in Section III.

The paper is organized as follows. In Section II, we first review the Max-Log-MAP decoding algorithm, and analyze the inherent sub-frame-level parallelisms in Turbo decoding. Then we map the Turbo decoding onto GPUs with considering Kernel and thread allocation, computation and traversal of LLRs, memory allocation and access. Some implementation results are provided in Section IV to illustrate the good performance of the proposed method, in both BER and throughput. Finally, a conclusion is drawn in Section V.

II. TURBO DECODING ALGORITHM AND PARALLELISM ANALYSIS

A Turbo decoder consists of two half-decoders which perform the same sequence of computation during each iteration and exchange appropriately interleaved extrinsic LLRs with each other as inputs for the next iteration.

A. Max-Log-MAP Algorithm

The algorithm running on both decoders is the maximum a posteriori probability (MAP) algorithm. The original form of MAP algorithm involves exponential computation, which has a high computational complexity. The Max-Log-MAP algorithm is proposed to simplify the MAP with limited degradation in BER performance.

To describe the Turbo decoding algorithm, we use x_k^i, x_k^p to denote the information bit and party bit at stage k generated by the encoder respectively. Let y_k^i, y_k^p denote the information bit and party bit at stage k received by the decoder after the transmission in AWGN channel. And let s_k denote the state at stage k.

Fig. 1 shows the state transition diagram for the 3GPP LTE Turbo codes from trellis stage k to stage $k+1$. In the diagram, each state has two incoming paths as well as two outgoing paths, one path for $x_k^i = 0$ and the other for $x_k^i = 1$. For each iteration, both half-decoders execute a forward traversal and a backward traversal through the trellis to compute the extrinsic LLRs.

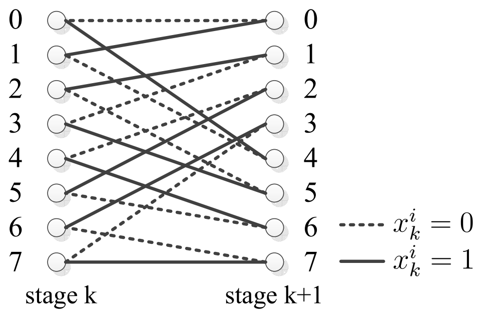

Fig. 1. 3GPP LTE Turbo code trellis with 8 states.

In Max-Log-MAP algorithm, the branch metric $\gamma_k(s_{k-1}, s_k)$ is defined as

$$\gamma_k(s_{k-1}, s_k) = (L_c(y_k^i) + L_a(k))x_k^i + L_c(y_k^p)x_k^p, \quad (1)$$

where $L_c()$ is the LLR received from channel, $L_a(k)$ is the priori information from another RSC decoder, and bit x_k^i leads to the state transition $s_k \to s_{k+1}$, which generates the corresponding parity bit x_k^p.

The forward state metric for a state s_k at stage k, $\alpha_k(s_k)$, is defined as

$$\alpha_k(s_k) = \overset{\star}{\max_{s_{k-1} \in \mathcal{S}}}(\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)), \quad (2)$$

where $\mathcal{S}$ is the set of all possible states at stage $k-1$ from which a state transition is to the state s_k at stage k.

The backward state metric for a state s_k at stage k, $\beta_k(s_k)$, is defined as

$$\beta_k(s_k) = \overset{\star}{\max_{s_{k+1} \in \mathcal{S}}}(\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k, s_{k+1})) \quad (3)$$

where $\mathcal{S}$ is the set of all possible states at stage $k+1$ from which a state transition is to the state s_k at stage k.

After computing α, β, and γ, we compute two state LLRs per trellis state, $\Lambda(s_k \mid x_k^i = 0)$ and $\Lambda(s_k \mid x_k^i = 1)$, for two incoming paths which correspond to $x_k^i = 0$ and $x_k^i = 1$, respectively. They are defined as:

$$\Lambda(s_k)_{x_k^i=0} = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta(s_k) \quad (4)$$
$$\Lambda(s_k)_{x_k^i=1} = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta(s_k) \quad (5)$$

where the two paths of state transitions from state s_{k-1} to state s_k correspond to $x_k^i = 0$ or 1, respectively.

Finally, we evaluate the extrinsic LLR for x_k^i using the following equation:

$$\begin{aligned} L_e(k) = & \max_{s_k \in \mathcal{S}}(\Lambda(s_k \mid x_k^i = 1)) \\ & - \max_{s_k \in \mathcal{S}}(\Lambda(s_k \mid x_k^i = 0)) \\ & - L_c(y_k^i) - L_a(k) \end{aligned} \quad (6)$$

where $\mathcal{S}$ is the set of all possible states in stage k.

B. Inherent Parallelisms

Although the MAP algorithm and the iterative decoding process make Turbo decoding with natural data serialism and strong data dependencies, there are a certain degree of inherent parallelisms in it. For example, in the forward recursion and the backward recursion, all state metrics α and β at each stage can be computed in parallel, which is called trellis state level parallelism [5], [7]. In addition to that, we have made further exploration on the inherent parallelisms including traversal-direction level parallelism and trellis stage level parallelism of LLRs.

It has always been ignored that although the forward and backward traversal themselves are totally serial, the computations of state metrics α and β are independent of each other and can be executed simultaneously if the initial value of α and β are known respectively. This traversal direction level parallelism is suitable to perform on GPU and helps to hide the delays caused by accessing high latency memories.

The computation of two state LLRs Λ_0, Λ_1 is given by Eq. (4) and Eq. (5), respectively. This computation shows complete data parallelism, i.e. Λ_0 and Λ_1 can be computed in parallel for all the possible state transitions at all trellis stages. Eq. (6) give the computation of the extrinsic LLR L_e. Likewise, extrinsic LLRs of different trellis stages can be evaluated concurrently. However, as is shown in the equation, the evaluation of a single extrinsic LLR needs to compare the values of Λ_0 and Λ_1 for all the 8 states at each trellis stage, which may introduce parallelism overhead.

C. Sub-frame-level Parallelism and Guarding Mechanism

To maximize the advantage of the GPU's multi-core feature, we divide one frame into several smaller sized sub-frames so that the decoding workload can be distributed across more cores in GPU. However, this parallelism will break the natural serial structure of Turbo decoding algorithm and inevitably cause degradation in BER performance.

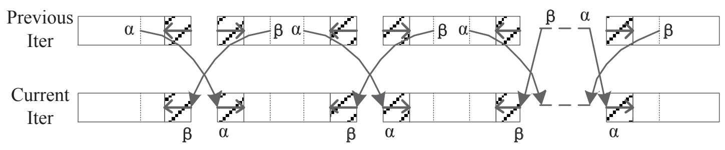

Fig. 2. PIVIDSTW guarding mechanism in [7].

To implement the sub-frame level parallelism, all sub-frames should be provided with initial values of state metrics α and β respectively, which are totally independent among all the sub-fames. A simple way is to initialize the state metrics α and β of all states to equal values at the start of forward and backward recursion, which leads to severe BER performance degradation. To minimize the BER performance loss, three guarding mechanisms, Previous Iteration Value Initialization (PIVI), Double Sided Training Window (DSTW) and PIVIDSTW were discussed in [7]. And the PIVIDSTW guarding mechanism combines the features of both DSTW and PIVI. As shown in Fig. 2, in PIVIDSTW, there are training windows running on either sides of the sub-frame. At the beginning of the window, the initial values of α and β metrics are set equal to the values from the trellis stages at the end of the training windows from the previous iteration [7]. We apply PIVIDSTW in our implementation as it shows the best performance in minimizing the BER degradation caused by the sub-frame-level parallelisms among the three guarding mechanism.

III. Mapping TURBO Decoding Algorithm onto GPU

In our implementation, we map the decoding algorithm on to GTX580, a Compute Unified Device Architecture (CUDA) processor of compute capability 2.0 from Nvidia. There are 512 stream processors (SPs) grouped to 16 stream multi-proce-ssors (SMs). In CUDA programing, a kernel function is defined to give a sequence of operations and the parallel computation is mapped onto GPU by assigning each thread block with multiple threads to an SM. In a thread block, every 32 threads are packed into a group which is called warp. At the beginning, the data are transferred from the host memory for CPU to the device memory for GPU, i.e., global memory.

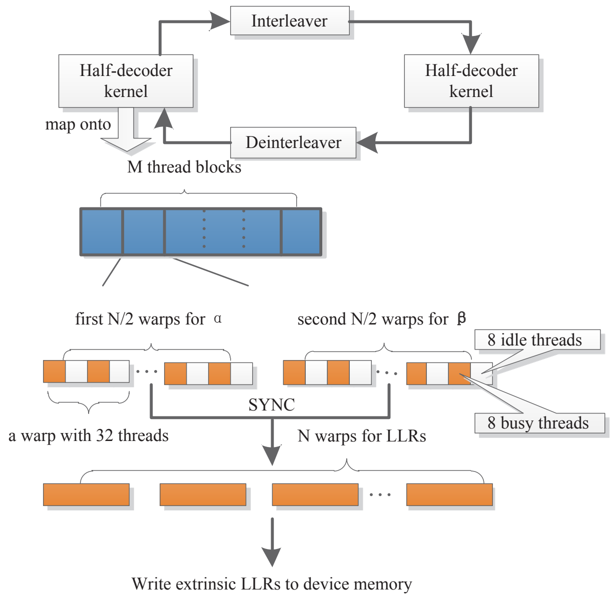

Fig. 3. Mapping of Turbo decoding algorithm on GPU.

The global memory is large (>1GB) but slow, while faster but smaller caches like registers and shared memory are also provided.

In our implementation, we allocated multiple warps within a thread block to improve the occupancy while subjecting to the register and shared memory constraint. The overall mapping of the Turbo decoding algorithm on the GPU is shown in Fig. 3. And the details of the implementation are described in the following sections.

A. Half-Decoder Kernel and Thread Allocating Approach

In the iterative decoding algorithm, the single iteration consists of two half-iterations which are executed on two half-decoders respectively. The same kernel definition is used for both the half-decoders as they perform the same sequence of computations.

We divide a frame of length L (6144) into P sub-frames, so the length W of each sub-frame is equal to L/p. There are M thread blocks allocated with N warps each block to compute the corresponding extrinsic LLRs for P ($P = M \times N$) sub-frames. To make the forward and backward recursion perform concurrently, we assign the first $N/2$ warps and the second $N/2$ warps in each block to compute α and β metrics of N stages respectively. Since each warp consists of two half-warps with 16 threads, to map the trellis state-level parallelism, the first 8 threads within each half-warp are assigned to compute the state metrics at corresponding stage and the rest 8 threads are left idle. As a shared memory request for a warp is split into two memory requests, one for each half-warp, which are issued independently, there can be no bank conflict between threads belonging to different half-warps [1]. After the computation of α and β state metrics, we assign all the N warps each block to compute extrinsic LLRs of the corresponding N stages in parallel.

B. Forward and Backward Traversal

Before evaluating the extrinsic LLRs, the forward and backward traversal through the trellis must be performed to compute and store the state metrics α and β. Eq. 2 and Eq. 3 show the computation of $\alpha_k(s_k)$ and $\beta_k(s_k)$ respectively. As shown in Fig. II-A, each state has two incoming paths as well as two outgoing paths, one for $x_k^i = 0$ and the other for $x_k^i = 1$. Each thread is assigned to compute $\alpha_k(s_k)$ or $\beta_k(s_k)$ for one state through two incoming paths or outgoing paths respectively.

C. Computation of LLRs

After evaluating and storing the values of state metrics α and β for all the stages, two state LLRs $\Lambda 0$ and $\Lambda 1$ and extrinsic LLRs are computed. As discussed above, the two state LLRs and extrinsic LLRs at all the trellis stage can be computed in parallel. Then the extrinsic LLR L_e at each stage is computed from the state LLRs of 8 states. Instead of assigning 8 threads to evaluate a single L_e immediately after β metric computation at each stage, we assign W threads per warp for the corresponding W trellis stages of each sub-frame. Each thread handles one stage of Λ_0 and Λ_1 to compute a L_e, which increases the thread utilization. Since there are 16 shared memory banks, Λ_0 and Λ_1 at even trellis stages with the same index would share the same memory bank, which causes bank conflicts. To alleviate the problem, we re-organize the shared memory access pattern based on thread ID as **Algorithm 1** in TABLE I. Here, the party bits $x_k^p | x_k^i = 0$, $x_k^p | x_k^i = 1$, are associated with the paths of state transitions.

D. Memory Allocation and Access Pattern

At beginning, we pre-fetch the input data from global memory into shared memory. Since the computation of LLRs depends on the results of forward and backward traversals, a certain amount of memory must be allocated to store the values of all α and β metrics. In one thread block for N sub-frames of length W,$N \times W \times 8 \times 2$ floats ($N \times W \times 64$ bits) are needed to store the results of state metrics. Although small N and W will reduce the amount of shared memory required per thread block, they also leads to the decrease of active warps and BER performance respectively.

Instead of allocating considerable amount of shared memory to store all α and β metrics, we only store one stage of α and β during the forward and backward traversal, and use global memory to store all the state metrics for the computation of LLRs. For example, assume all 8 state metrics $\alpha_{k-1}(s_{k-1})$ at previous stage $k-1$ are stored in the shared memory. After calculating $\alpha_k(s_k)$ at current stage k using $\alpha_{k-1}(s_{k-1})$, we store all 8 metrics $\alpha_k(s_k)$ in global memory and update shared memory with $\alpha_k(s_k)$ for the computation of next stage. By doing so, there is enough amount of shared memory left for Λ_0 and Λ_1. We need $N \times W \times 8 \times 2$ floats to store them per block. Additionally, the read-only constant data, such as the structure data of the trellis, is stored in the constant memory.

TABLE I: Computation algorithm of LLRs on GPU

```
Algorithm 1: Computation of $\Lambda_0$, $\Lambda_1$ and $L_e$
$k =$ Thread ID &31
while $k < W$ do
  for $j = 0$ to 7 do
    $index = (k + j)\&7$
    $\gamma_0 \leftarrow 0.5 \times ((L_c y_k^i + L_e(k))(-1) + L_c y_k^p(x_k^p | x_k^i = 0))$
    $\gamma_1 \leftarrow 0.5 \times (L_c y_k^i + L_e(k) + L_c y_k^p(x_k^p | x_k^i = 1))$
    $\Lambda_0(index) \leftarrow \alpha_{k-1}(index) + \gamma_0 + \beta_k(index)$
    $\Lambda_1(index) \leftarrow \alpha_{k-1}(index) + \gamma_1 + \beta_k(index)$
  end for
  $\lambda_0 = \Lambda_0(0)$
  $\lambda_1 = \Lambda_1(0)$
  for$j = 0$ to 7 do
    $index = (k + j)\&7$
    $\lambda_0 = \max(\lambda_0, \Lambda_0(index))$
    $\lambda_1 = \max(\lambda_1, \Lambda_1(index))$
  end for
  $L_e(k) = (\lambda_1 - \lambda_0) - L_c y_k^i - L_a(k)$
  $k \leftarrow k +$ warpsize(32)
end while
```

IV. RESULTS: BER PERFORMANCE AND THROUGHPUT

The BER performance and throughput of our Turbo decoder are tested on a Windows7 platform with 8GB DDR3 memory. The GPU used in our experiment is the Nvidia Geforce GTX 580 graphic card running at 1.6 GHz with 1.5GB of GDDR3 memory.

A. BER performance

As shown in Fig. 4, simply portioning the decoding workload across cores without applying any guarding mechanism (No guarding, $W = 32$) causes a severe loss of BER performance. As expected, it shows a significant improvement of BER performance after using PIVIDSTW guarding mechanism. The performance is improved by increasing the length of sub-frames. We see that, for $P = 64$, $W = 96$ and for a window size $g = 8$, the decoder with PIVIDSTW guarding mechanism provides a BER performance that is within 0.01dB of the optimal case of a single frame.

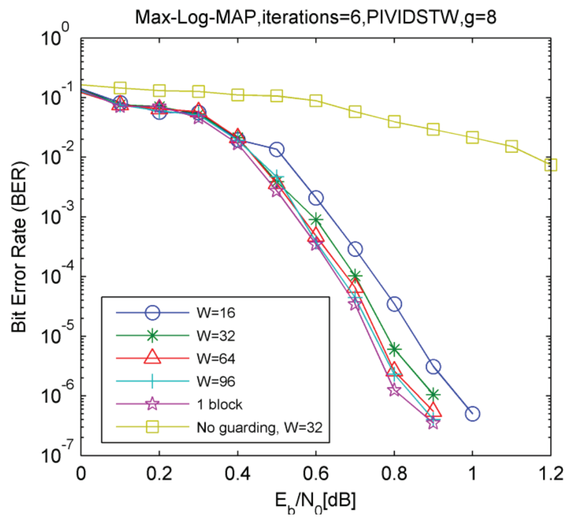

Fig. 4. BER performance of Max-Log-MAP algorithm on GPU.

TABLE II: Throughput vs. W, with PIVIDSTW ($g = 8$)

Iter	Max-Log-MAP with PIVIDSTW (Mbps)			
	W=16	W=32	W=64	W=96
1	9.8	8.1	6.3	5.2
2	7.3	6.1	4.4	3.4
3	5.9	4.9	3.3	2.5
4	4.9	4.1	2.7	2.0
5	4.2	3.5	2.3	1.7
6	3.7	3.1	2.0	1.5
7	3.3	2.7	1.7	1.3

TABLE III: Throughput of our proposal vs. [7], $P = 192$

Iter	Max-Log-MAP with PIVI (Mbps)	
	W=16	W=32
1	4.7	9.4
2	3.8	6.9
3	3.1	5.5
4	2.6	4.6
5	2.4	3.9
6	2.1	3.4
7	1.9	3.1

B. Throughput

Like the measurement method described in [7], the time measured for calculating throughput in our implementation consists of both memory transfer and kernel execution time, and the memory transfer is done separately for each frame including the input channel values transferring from the host to the device and returning decoded bits back to the host. We use the Max-Log-MAP algorithm and PIVIDSTW [7] guarding mechanism for a window size $g = 8$ in our implementation. TABLE II presents the throughput results. We see that larger number of iterations leads to lower throughput of the decoder.

In order to compare with the throughput in [7], we also apply the PIVI [7] guarding mechanism to our implementation. As shown in TABLE III, our throughput is about twice as faster as those in [7] for $P = 192$, $W = 32$. The throughput results may not be as high as expected, because our implementation performs further parallelism at the cost of increased times of accessing global memory (fetch α and β metrics).

C. Occupancy

In [5], [7], a single sub-frame is assigned to each thread block with one warp. Only 8 threads per warp is allocated to perform the state metrics computation while the other 24 threads in a warp are left idle, which leads to a considerable waste of on-chip resources. Different from their mapping approach, we allocate multiple warps in a thread block to process the corresponding multiple sub-frames, so that the computation latency can be better hided by switching the instruction to another warp when any stall occurs.

To find the optimal number of warps to be allocated per thread block to achieve higher throughput, we vary N, which denotes the warp number per block, for different values while subjecting to the shared memory constraint. TABLE IV gives the results for different sub-frame length W. As shown in the table, since the increase of length W leads to more amount of shared memory required for each sub-frame, the number of warps allocated per thread block is reduced to enable more active blocks per SM.

TABLE IV: Number of warps per block vs. W

W	16	32	64	96
N	12	6	2	2

We calculate the occupancy in our implementation and compare it with that in [5], [7], assuming their decoders are also implemented on GTX580. The GTX580 has 16 SMs, and there are at most 48 active warps and 8 active thread blocks per SM. For example, in [5], [7], when the length of a sub-frame is 32, 192 thread blocks are allocated for a whole frame. Thus, 12 blocks are assigned per SM. As the max number of active thread blocks is 8 and there is only one warp allocated per block, the number of active warps per SM is 8, which leads to an occupancy of 1/6. In our work, we allocate 6 warps to process 6 sub-frames per thread block so the number of thread blocks is 32 and 2 blocks is assigned per SM. Because there are 24 active warps per SM, the occupancy is equals to 1/2, which is much higher than that in [5], [7]. Moreover, as the forward traversal or backward traversal has been executed each warp using 16 threads, the thread utilization of each warp is higher. Fig. 5 shows a comparison of occupancy between [5], [7] and our work for different values of P , which is the number of sub-frame per frame.

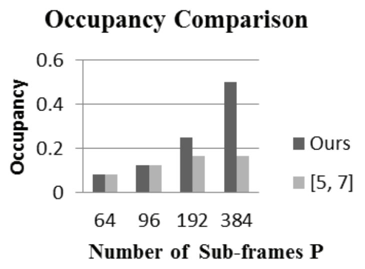

Fig. 5. The comparison of occupancy.

As mentioned above, in our implementation, a frame is divided into P sub-frames, and we allocate M thread blocks with N warps per block to compute for the corresponding sub-frames. Despite of the shared memory and register constraints, the number of active warps is $M \times N$, which equals to P . As a GTX580 has 16 SMs and there are at most 48 active warps per SM, the total number of available warps is equals to 768. Hence, the occupancy would not be more than $P/768$. As the workload to decode a single frame is limited, a larger P will distribute the workload across more cores and spawns more active warps, which will provide better occupancy. Yet, a larger P also leads to poorer BER performance. To reduce the impact of P on the occupancy and keep all the SMs busy, multiple frames should be decoded simultaneously [6]. However, the frame-level parallelism is not practical when the data are loaded in a single stream and may cause higher decoding latency.

V. Conclusions

In this paper, we implement a parallel Turbo decoder on GPU. Compared to the previous work in [5], [7], we proposed two other kinds of parallelism, traversal-direction level parallelism and trellis stage level parallelism of LLRs, and mapped them onto recent GPU architecture. We divide one frame into several sub-frames to distribute the decoding workload across cores on GPU while using PIVIDSTW guarding mechanism [7] to minimize the degradation of error correction performance. In the half-decoder kernel, the forward recursion and backward recursion to compute state metrics for each sub-frame have been executed concurrently. And as a result, the computation of backward state metrics and state log-likelihood ratios (LLRs) is split and further parallelism in LLRs computation has been performed compared to [5], [7]. Multiple sub-frames are assigned per thread block which increases the number of active warps per thread block. The results show that our decoder achieves a higher occupancy on GPU and provides about twice higher throughput than that in [7] while maintaining good BER performance.

References

[1] NVIDIA CUDA C Programming Guide Version 4.2. [Online]. Available: http://developer.nvidia.com/cuda-downloads, 2012.

[2] D. Lee, M. Wolf and H. Kim, “Design space exploration of the Turbo decoding algorithm on GPUs,” *Proc. 2010 International conference on compilers, architectures and synthesis for embedded systems (CASES’10)*, Scottsdale, AZ, USA, October 24-29, 2010, pp. 217-226.

[3] William E. Ryan, “A Turbo code tutorial," *Proc. IEEE Globecom’98*, Sydney, Australia, November 8-12, 1998.

[4] C.-C. Wong, Y.-Y. Lee and H.-C. Chang, “A 188-size 2.1 mm^2 reconfigurable Turbo decoder chip with parallel architecture for 3GPP LTE system," *Proc. 2009 Symposium on VLSI Circuits*, Kyoto, Japan, June 16-18, 2009, pp. 288-289.

[5] M. Wu, Yang Sun and J. R. Cavallaro, “Implementation of a 3GPP LTE Turbo decoder accelerator on GPU," *Proc. 2010 IEEE Workshop on Signal Processing Systems (SIPS’10)*, San Francisco, USA, Oct. 6-8, 2010, pp. 192-197.

[6] M. Wu, Yang Sun and J. R. Cavallaro, “Implementation of a High Throughput 3GPP Turbo Decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, pp. 171-183, Sept. 2011.

[7] D. R. N. Yoge and N. Chandrachoodan, “GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications," *Proc. 25th International Conference VLSI Design (VLSID’12)*, Hyderabad, India, Jan. 7-10, 2012, pp. 149-154.