2012 25th International Conference on VLSI Design

GPU Implementation of a Programmable Turbo Decoder for Software Defined Radio Applications

Dhiraj Reddy Nallapa Yoge and Nitin Chandrachoodan
Dept. of Electrical Engineering
IIT Madras
Chennai, India
ee06b066, nitin@ee.iitm.ac.in

***Abstract*—This paper presents the implementation of a 3GPP standards compliant configurable turbo decoder on a GPU. The challenge in implementing a turbo decoder on a GPU is in suitably parallelizing the Log-MAP decoding algorithm and doing an architecture aware mapping of it on to the GPU. The approximations in parallelizing the Log-MAP algorithm come at the cost of reduced BER performance. To mitigate this reduction, different guarding mechanisms of varying computational complexity have been presented. The limited shared memory and registers available on GPUs are carefully allocated to obtain a high real-time decoding rate without requiring several independent data streams in parallel.**

***Keywords*-Turbo Decoder; GPU implementation; CUDA; Parallel Log-MAP; Guarding Mechanisms;**

I. Introduction

Turbo codes are an important class of forward error correcting (FEC) codes because of their low bit-error rate (BER) and frame error rate (FER) performance, and are widely used in many 3G and 4G standards such as UMTS[1], 3GPP LTE[2] etc. The turbo decoder is one of the most computationally challenging and time consuming parts of the encoding-decoding process because of the complex iterative decoding algorithm. Hence, typically ASIC implementations of the decoder are preferred for achieving a high data throughput. Graphic Processing Units (GPUs) provide an alternative for achieving the same high data throughput as achieved on dedicated ASICs, but with the added benefit of programmability in software. Implementation of a high throughput configurable turbo decoder completely in software is an important step towards a pure software defined radio (SDR) realization for testing out various evolving 4G standards. Today, even many hand held devices contain GPUs and the decoder implementation may be extended to them as well.

GPUs, with their large number of processing cores, are very good at handling tasks that exhibit gratuitous amounts of data parallelism. In this paper, we utilize the different kinds of data parallelisms inherent to the computations in the Log-MAP algorithm and also additional data parallelisms brought about by splitting the code block into several small code blocks to achieve a high throughput. We distribute the workload of the parallel Log-MAP algorithm efficiently across all the processor cores and also effectively utilize the fast on chip shared memory. The splitting of the code block into several small code blocks comes at the cost of reduced BER and FER performance and hence different guarding mechanisms to mitigate this reduction are also presented. The relative merit of each of these mechanisms in mitigating the degradation in performance and the associated impact on throughput are also discussed.

[1]Universal Mobile Telecommunications System
[2]Third Generation Partnership Project - Long Term Evolution

The turbo decoding algorithm is usually based on the BCJR algorithm and variants thereof [1], [2]. However, these algorithms are highly serial in nature, requiring a complete traversal of a block from end-to-end and then in reverse for each iteration. Windowing techniques [3] have been proposed to work around this problem to some extent, but the interleaver used in turbo codes forms yet another challenge for implementation in parallel. As a result, although there have been several VLSI implementations of decoders of different capabilities, very few parallel software approaches have been proposed, such as [4] and [5].

One important aspect that is typically used to obtain a high degree of parallelism in the decoding process is to consider the decoding of several code blocks in parallel. For example, [4] considers the decoding of 100 blocks in parallel to estimate the throughput of the decoder. While this is useful in terms of estimating resource usage and parallelism, it does not help in the case where we want a single stream of data to be decoded in software, as would be the case in a true SDR receiver.

In this paper, we have concentrated on the case of a single stream of data and tried to maximize the throughput, while keeping in consideration the time required for transfer of blocks of data into and out of device memory.

The rest of the paper is organized as follows: Section II overviews the Log-MAP algorithm and section III explores the possible parallelisms in it. Section IV describes an architecture aware mapping of the parallel Log-MAP algorithm onto the GPU. The BER performance results and the achieved throughput are presented in section V followed by the conclusion in section VI.

II. Log-MAP Algorithm

For conciseness, we do not go into the full details of how a turbo encoder and decoder are implemented, and rather concentrate only on those aspects relevant to the discussion of the parallel implementation. For further details, including the structure of the encoder and decoder, use of the interleaver, etc., excellent tutorials are available in [1] and [2].

The turbo decoder consists of two half decoders, each of which exchange appropriately interleaved extrinsic log-likelihood ratios (LLRs) after each iteration. The extrinsic likelihood ratios in each half-decoder are estimated using the Maximum a posteriori probability (MAP) algorithm [1]. The MAP algorithm in its direct form involves the computation of exponentials, and hence a simplified version of it that uses log-likelihood ratios, the Log-Map algorithm, is often used for both hardware and software implementations [2].

Let $\boldsymbol{u} = u_1, u_2, \ldots, u_N$ and $\boldsymbol{x^p} = x_1^p, x_2^p, \ldots, x_N^p$ denote the input bit-stream and the parity bit-stream generated by the constituent encoder respectively. For each half decoder, let $\boldsymbol{y^s} = y_1^s, y_2^s, \ldots, y_N^s$ and $\boldsymbol{y^p} = y_1^p, y_2^p, \ldots, y_N^p$ denote the noisy AWGN versions of $u_1, u_2, \ldots, u_N$ and $x_1^p, x_2^p, \ldots, x_N^p$ respectively. Let $L_a(k)$ denote the a priori LLR of bit u_k passed on from the other half-decoder and $L_e(k)$ denote the computed extrinsic LLR.

For a state transition from a state s_{k-1} at a trellis stage $k-1$ to a state s_k at the trellis stage k, the branch metric $\gamma_k(s_{k-1}, s_k)$ is defined as :

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

where L_c is the channel reliability value, and bit u_k causes the state transition $s_{k-1} \rightarrow s_k$ and generates the parity bit x_k^p. The possible state transitions for the 3GPP LTE standards compliant encoder are shown in Fig.1

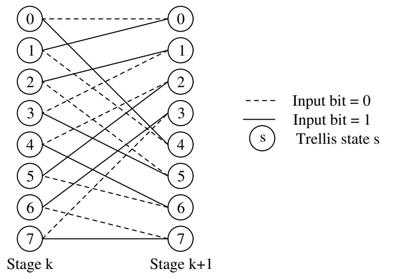

Figure 1: Trellis state transition diagram of 3GPP LTE encoder

The forward state metric for a state s_k at a stage k of the trellis $\alpha_k(s_k)$ is defined as:

$$\alpha_k(s_k) = max^*_{s_{k-1}\in S}(\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)) \qquad (2)$$

where S is the set of all states from which a state transition is possible to the state s_k.

The backward state metric for a state s_k at a stage k of the trellis $\beta_k(s_k)$ is defined as:

$$\beta_k(s_k) = max^*_{s_{k+1}\in S}(\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k, s_{k+1})) \qquad (3)$$

where S is the set of all states to which a state transition is possible from the state s_k.

After computation of α, β and γ, two LLRs per trellis state are computed. The state LLR $\Lambda(s_k|\, u_k = 0)$ and $\Lambda(s_k|\, u_k = 1)$, for a state s_k at a trellis stage k are defined as:

$$\Lambda(s_k|\, u_k = 0) = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \qquad (4)$$

$$\Lambda(s_k|\, u_k = 1) = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \qquad (5)$$

where $u_k = 0$ or 1 respectively causes the state transition $s_{k-1} \rightarrow s_k$ respectively.

The extrinsic LLR for u_k is computed as:

$$L_e(k) = max^*_{s_k\in S}(\Lambda(s_k|\, u_b = 1)) - max^*_{s_k\in S}(\Lambda(s_k|\, u_b = 0)) \\ -L_c(y_k^s) - L_a(k) \qquad (6)$$

where S is the set of all possible states and max^* is defined as

$$max^*(S) = \ln(\sum_{s\in S} e^s). \qquad (7)$$

A. max^* Function

The way in which the max* function is actually computed divides the MAP algorithm into two algorithms - the Full log-map algorithm and the Max log-map algorithm. In the full log-map algorithm, max^* is computed as:

$$max^*(a, b) = max(a, b) + \ln(1 + e^{-|b-a|}) \qquad (8)$$

In the max log-map algorithm, max^* is computed as:

$$max^*(a, b) = max(a, b) \qquad (9)$$

B. QPP Interleaver

The 3GPP LTE standard uses the Quadratic Permutation Polynomial (QPP) interleaver. The QPP interleaver is defined as:

$$\Pi(x) = f_1 x + f_2 x^2 \;\; (mod\, N) \qquad (10)$$

where f_1 and f_2 satisfy several properties detailed in [6].
A computationally less expensive way of computing the QPP interleaver function defined in Eq. (10) is

$$\Pi(x) = (f_1 + f_2 x \;\; (mod\, N))x \;\; (mod\, N) \qquad (11)$$

III. Parallelism in Log-MAP Algorithm

The Log-MAP turbo decoding algorithm, in its direct form, exhibits only a little amount of data parallelism. In this algorithm, there exists a very strong data dependency between adjacent trellis stages. The computation of both the forward and backward state metrics(SM) proceeds in a serial fashion along the trellis stages. The serial evaluation of the SMs forms the basis of the MAP algorithm and any attempts to parallelize this part would result in severe BER performance loss.

A. Trellis State Level Parallelism

The SMs α and β at each trellis stage can be computed concurrently for all the trellis states. For example, the $\alpha_k(s_k)$ or $\beta_k(s_k)$, for all the states at a stage k of the trellis can all be computed at once if, $\alpha_{k-1}(s_{k-1})$ or $\beta_{k+1}(s_{k+1})$ respectively, are known for all the states. The branch metric (BM) γ can be computed in parallel for all the possible state transitions at each stage of the trellis. In fact, the BM computation shows complete data parallelism i.e the BMs of all the trellis stages themselves can all be computed in parallel. In the 3GPP LTE standard encoder, since there are 8 trellis states and 16 possible state transitions, there is a 8-way parallelism in α and β and a 16-way parallelism in γ computations to exploit. The trellis state level parallelism is inherent to the Log-Map algorithm and hence does not cause any BER performance degradation.

B. Sub-block Level Parallelism

Though the trellis state level parallelism offers some amount of data parallelism, the degree of parallelism shown is small. Further data parallelism can be obtained by dividing the code block into several smaller sized code blocks called sub-blocks and performing the decoding on each of the sub-blocks independently in each iteration [7]. During each iteration, in each sub-block, the forward and backward recursions run only over the length of the sub-block. Hence, during each iteration, the forward and backward recursions of all the sub-blocks are completely data independent and can be done concurrently. As before, after each iteration, the extrinsic LLRs computed by each of the two half-decoders are exchanged with appropriate interleaving. Fig.2 contrasts the forward and backward recursions in the MAP algorithm with and without sub-block level parallelism.

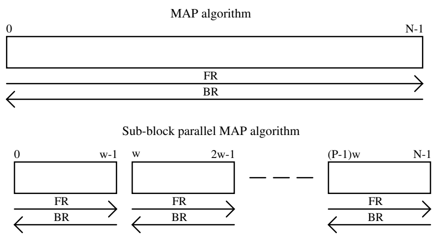

Figure 2: Recursions in parallel MAP algorithm

Let a code block of length N be split into P sub-blocks each of length $w = \frac{N}{P}$. The degree of data parallelism thus brought about is equal to the number of sub-blocks P. An increase in P increases the amount of data parallelism possible but it comes at the cost of reduced BER performance. The BER performance degradation occurs because the α and β metrics at the start and end of a sub-block respectively are no longer known. However, the decrease in BER performance can be reduced by adopting various guarding mechanisms to estimate the α and β metrics at the start and the end of a sub-block respectively.

C. Guarding Mechanisms

Simply initializing the α and β metrics of all states to equal values at the start and the end of the sub-blocks leads to severe performance degradation. To minimize the degradation in BER performance the following three guarding mechanisms as shown in Fig. (3) are used.

1) Previous Iteration Value Initialization (PIVI): Here, the α metrics at the beginning of a sub-block are initialized with the α metrics at end of the previous sub-block from the previous iteration. Similarly, the β metrics at the end of a sub-block are initialized with the β metrics at the beginning of the next sub-block from the previous iteration. This mechanism requires extra memory but involves no extra arithmetic computations.

2) Double Sided Training Window (DSTW): Here, training windows are run on either sides of the sub-block to allow the α and β metrics to develop into better estimates [3]. At the start of the training window, both α and β metrics for all the states are set to equal values. A larger training window results in better BER performance but would come at the cost of extra computations.

3) Previous Iteration Value Initialization with Double Sided Training Window (PIVIDSTW): Here, the features of both of the above mechanisms are combined. As in DSTW, this also has training windows running on either sides of the sub-block. But, at the beginning of the training window, instead of initializing the α and β metrics of all states to equal values, they are set equal to the α and β metrics from the trellis stages at the end of the training windows from the previous iteration. Naturally, since this

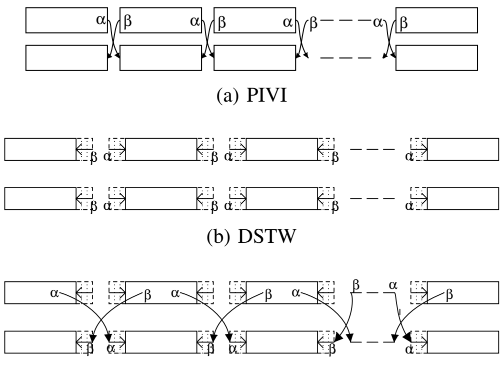

Figure 3: Types of Guarding Mechanisms

guarding mechanism combines both the above two mechanisms, it requires extra memory and involves extra computations.

IV. Mapping the Log-MAP Algorithm on to GPU

To obtain a high throughput on the GPU, an architecture aware [8] mapping of the algorithm is paramount. The GPU architecture differs significantly from that of a CPU [9]. In this paper, we have made use of the Compute Unified Device Architecture (CUDA) processors from Nvidia, and some of the implementation decisions are influenced by the restrictions on registers, shared memory etc. imposed by the specific processor available to us. However, we have tried to explain the principles behind the parallelization and memory usage in general terms that make it easy to adapt the design to a different architecture with different constraints.

A CUDA compatible GPU has a large number of processing cores and is capable of running several hardware threads concurrently but has no hardware managed cache. Four different kinds of device memories are presented to the programmer [10]. The sizes and relative latencies of each of these memories directly impact the mapping of any algorithm on to the GPU. A fast on chip memory is available in the form of shared memory. The computations wherein all accesses are to the shared memory occur much faster than those wherein one or more accesses involve the global memory [11].

Occupancy, which is a measure of the number of threads being actually run concurrently on a streaming multi-processor (SMP) [12], is determined by the number of threads per thread block, the shared memory allocated per thread block and the number of registers per each thread. Higher the occupancy, better would GPU be in hiding the delays associated with accesses to high latency memories The principles followed here for mapping of the Log-MAP turbo decoding algorithm are general and apply to any generator function in the encoder, any code block length and any interleaver function. However, the mapping has been performed for the 3GPP standard specifications.

A. Half-Decoder Kernel

The decoding is done iteratively by the two half-decoders. In each half-decoder, the decoding proceeds with a forward traversal followed by a backward traversal through the length of the sub-block for all the sub-blocks. The same kernel definition is used for both the half-decoders since, both perform the same set of

Table I: Operands for α_k computation [4]

	u=0		u=1	
Thread id(i)	s_α^0	x_α^{p0}	s_α^1	x_α^{p1}
0	0	0	1	1
1	3	1	2	0
2	4	1	5	0
3	7	0	6	1
4	1	0	0	1
5	2	1	3	0
6	5	1	4	0
7	6	0	7	1

computations on the input data and are identical in all aspects other than in reading the input a priori LLRs $L_a(k)$ and the systematic channel values $\boldsymbol{y^s}$ in direct or interleaved order, and returning the computed extrinsic LLRs $L_e(k)$ in direct or deinterleaved order. The extrinsic LLRs returned by both the half-decoders are in direct order. A kernel is launched with 8 threads [5] and P thread blocks.

The computation of α, β and extrinsic LLRs all require the γ metrics. The gamma metrics can either be computed once, stored and fetched when required or be computed each time they are required. The latter approach has been chosen in the current implementation because of the shared memory size and global memory latency constraints [13].

B. Forward Traversal

In the forward traversal, $\alpha_k(s)$ for each trellis stage k is computed for the length of the sub-block w. The computation of $\alpha_k(s)$ is given by Eq. (2). For each state s, at a trellis state k, there are exactly 2 states at trellis state $k-1$ from which a state transition to state s is possible, one each for $u_k = 0$ & 1. Each thread evaluates the $\alpha_k(s)$ for one state. To compute the $\alpha_k(i)$ for a state i, the thread needs to know the states s_α^0 and s_α^1 from which there is a state transition possible to the state i and the parity bits x_α^{p0} and x_α^{p1} associated with the state transitions $s_\alpha^0 \to i$ and $s_\alpha^1 \to i$. The Table. I summarizes the operands needed for α computation.

During the forward traversal the α metrics are computed and stored, since they are required for the computation of extrinsic LLRs computed during the backward traversal. The pseudo code for the α computation during the forward traversal is given by the Algorithm 1.

Algorithm 1 Forward Traversal - Thread i computes $\alpha_k(i)$

```
for $k = 1$ to $w$ do
    $\gamma^0 \leftarrow 0.5 \times ((L_c y_k^s + L_k^e)(-1) + L_c y_k^p(x_\alpha^{p0} | u_k = 0))$
    $\gamma^1 \leftarrow 0.5 \times (L_c y_k^s + L_k^e + L_c y_k^p(x_\alpha^{p1} | u_k = 1))$
    $\alpha^0 \leftarrow \alpha_{k-1}(s_\alpha^0 | u_k = 0) + \gamma^0$
    $\alpha^1 \leftarrow \alpha_{k-1}(s_\alpha^1 | u_k = 1) + \gamma^1$
    $\alpha_k(i) \leftarrow max^*(\alpha^0, \alpha^1)$
    SYNC
end for
```

C. Backward Traversal

In the backward traversal, the $\beta_k(s)$ and the extrinsic LLR $L^e(k)$ are computed for each trellis stage k along the length of the sub-block. The extrinsic LLR is computed immediately after β computation, thereby, removing the need to store the β metrics for the entire length of the sub-block. The computation of $\beta_k(s)$ is given by Eq. (3). For each state s, at a trellis state k, there are exactly 2 states at trellis stage $k+1$ to which a state transition from state s is possible, one each for $u_k = 0$ & 1. Each thread evaluates the $\beta_k(s)$ for one state. To compute the $\beta_k(i)$ for a state i, the thread needs to know the states s_β^0 and s_β^1 to which there is a state transition possible from the state i and the parity bits x_β^{p0} and x_β^{p1} associated with the state transitions $i \to s_\beta^0$ and $i \to s_\beta^1$. A table similar to the α computation applies for the β as well.

The state LLRs Λ_0 and Λ_1 for all the 8 states, as given by Eq. (4), are computed concurrently with each thread computing the state LLRs for one state. At each trellis stage, the computation of the extrinsic LLR from the state LLRs, as given by Eq. (6), can be done immediately after β metric computation. However, this would require the serial computation of the max^* function over all the the 8 states, thereby leaving all but one of the 8 threads in each block idle. To utilize all the threads, the state LLRs $\Lambda(s_k | u_k = 0)$ and $\Lambda(s_k | u_k = 1)$ for 8 consecutive trellis stages are stored and then the extrinsic LLRs for these 8 stages are computed concurrently using the 8 available threads. The pseudo code for β and extrinsic LLR computations is given in the Algorithm 2

Algorithm 2 Backward Traversal - Thread i computes $\beta_k(i)$ and $L^e(k)$

```
for $l = 1$ to $\frac{w}{8}$ do
    for $j = 1$ to $8$ do
        $k \leftarrow 8l + j$
        $\beta^0 \leftarrow \beta(s_\beta^0 | u_k = 0) + \gamma^0$
        $\beta^1 \leftarrow \beta(s_\beta^1 | u_k = 1) + \gamma^1$
        $\beta_{hold}(i) \leftarrow max^*(\beta^0, \beta^1)$
        SYNC
        $\beta(i) \leftarrow \beta_{hold}(i)$
        SYNC
        $\gamma^0 \leftarrow 0.5 \times ((L_c y_k^s + L_k^e)(-1) + L_c y_k^p(x_\beta^{p0} | u_k = 0))$
        $\gamma^1 \leftarrow 0.5 \times (L_c y_k^s + L_k^e + L_c y_k^p(x_\beta^{p1} | u_k = 1))$
        $\Lambda_0(8i + j) \leftarrow \gamma^0 + \beta(s_\beta^0 | u_k = 0)) + \alpha_k(i)$
        $\Lambda_1(8i + j) \leftarrow \gamma^1 + \beta(s_\beta^1 | u_k = 1)) + \alpha_k(i)$
    end for
    SYNC
    $L_{e0}, L_{e1} \leftarrow 0$
    for $j = 1$ to $8$ do
        $L_{e0} \leftarrow max^*(L_{e0}, \Lambda_0(8i + j))$
        $L_{e1} \leftarrow max^*(L_{e1}, \Lambda_1(8i + j))$
    end for
    $L_e(k) = (L_{e1} - L_{e0}) - L_c y_k^s - L_a(k)$
end for
```

D. Memory Allocation

1) Global Memory: Storing data in the global memory is the only way of exchanging data between different kernel launches. The extrinsic LLRs need to be exchanged between the different kernel launches corresponding to each of the half-decoders after every iteration and hence a copy of them is stored in the global memory. In addition, the PIVI and PIVIDSTW guarding mechanisms present the need to store α and β metrics at the ends of the sub-block in global memory as these need to be passed on to the next iteration.

2) Shared Memory: The shared memory is almost as fast as the registers and is the fastest available memory that the programmer can directly control. Shared memory is allocated for a thread block. The γ_k computation requires channel values and extrinsic LLRs. Hence, these are fetched from the global memory on to the shared memory. For a sub-block size of w, this would require storing $3w$ floats. The extrinsic LLR computation during the backward traversal requires the α_k values computed during the forward traversal. Hence, the α_k values for the entire size of the sub-block need to be stored in the shared memory. This requires storing another $8w$ floats. These two constitute the bulk of the shared memory allocated for each block. In addition, β_k computation requires β_{k-1} and hence 16 floats are required for β. In the extrinsic LLR computation given by Eq. (6), the $\Lambda(s_k| u_b = 0)$ and $\Lambda(s_k| u_b = 1)$ values for all states are stored for 8 trellis stages. This requires the storage of 64 floats each for $\Lambda(s_k| u_b = 0)$ and $\Lambda(s_k| u_b = 1)$. Extra shared memory is required for implementing the various guarding mechanisms. The PIVI requires 16 floats each for α and β initializations. The DSTW requires $6g$ floats and the PIVIDSTW requires $32 + 6g$ floats for a guard window of size g.

3) Constant Memory: The constant memory is ideal for storing all those values which remain constant during a kernel execution. It has been used for storing the operands mentioned in Table. I and the similar table required for the computation of α and β metrics respectively, as well as the indices required for deinterleaving.

V. BER Performance and Throughput

The BER performance and the throughput of the designed turbo decoder depends on the type of the decoding algorithm used - Full Log-MAP or Max Log-MAP, the number of parallel sub-blocks P and the type of guarding mechanism used - PIVI, DST or PIVIDSTW. The effect of each of these factors on the BER performance and the throughput has been presented below.

A. BER Performance

As is to be expected, the Full Log-MAP algorithm fares better than Max Log-MAP algorithm and this fact remains unchanged with the number of parallel sub-blocks. As shown in Fig.4, the BER performance degradation increases with a increase in the number of parallel sub-blocks. Among the three guarding mechanisms, the DSTW mechanism fares the worst and hence has been discarded. The PIVI mechanism can be used as a possible guarding mechanism. As shown in Fig.5, for the number of parallel sub-blocks P = 96, where the size of each sub-block w is only 64, the decoder with PIVI guarding mechanism provides a BER performance that is within 0.1dB and a FER performance that is within 0.2dB of the optimal case of a single code block. The PIVIDSTW guarding mechanism, which is a combination of both PIVI and DST, can be used to further improve the BER and FER performance. For $P = 96$, $w = 64$ and for a window size g = 8, the decoder with PIVIDSTW guarding mechanism provides a BER performance that is within 0.01dB and a FER performance that is within 0.02dB of the optimal case of a single code block.

B. Throughput

The BER performance and throughput of the designed turbo decoder have been evaluated by testing it on a 64-bit Linux

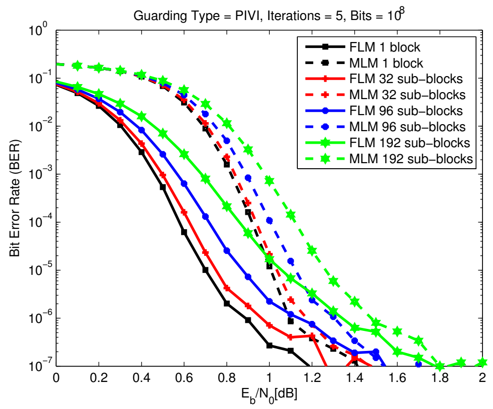

Figure 4: Effect of P on BER performance for Max Log-MAP and Full Log-MAP

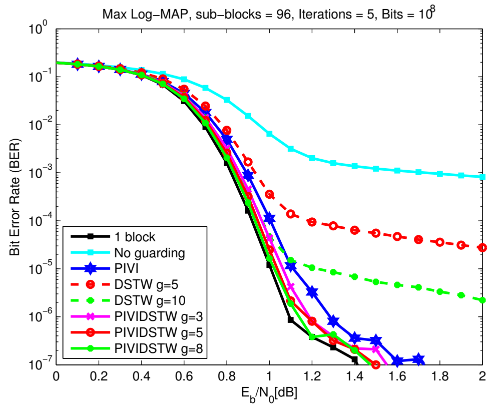

Figure 5: Effect of the type of guarding mechanism on BER performance

platform with 4GB DDR2 memory running at 800MHz and an AMD Phenom 9750 Quad-Core Processor running at 2.4GHz. The GPU used in the implementation is Nvidia GeForce 9800 GX2 graphics card of compute capability 1.1 running at 1.5GHz with 512MB of GDDR3 memory running at 1GHz. It has 128 streaming processors batched into 8SMPs; 8192 registers and 16KB of shared memory per SMP. The time measured for calculating throughput in our implementation is the cumulative sum of the times to transfer the input channel values from the host to the device, perform the decoding on the device and return the decoded bits back to the host. In addition, as would be the requirement for any real time usage of the turbo decoder, the data transfer is done separately for each code block. For a code block size of 6144 and for an Max Log-MAP implementation with sub-blocks P=96, the time for data transfer is 0.99ms and is quite significant compared to the time for one decoding iteration on the GPU, which is equal to 0.41ms.

The Table. II showcases the speed up achieved using the GPU over an implementation done purely on the CPU for both Max-Log-MAP and Full Log-MAP implementations. For a Max Log-MAP turbo decoder with 5 iterations, the GPU implementation with 96

Table II: Throughput on CPU vs GPU for $P = 96$

Iters	MLP/FLP Decoder Throughput (Kbps)		
	CPU	GPU	
		PIVI	PIVIDSTW g=5
1	991/116	4380/4050	4300/3990
2	524/56	3390/3010	3200/2840
3	351/37	2760/2400	2580/2240
4	264/28	2330/1990	2170/1850
5	211/21	2020/1700	1860/1580
6	176/19	1780/1490	1630/1370
7	151/17	1590/1320	1450/1210

Table III: Throughput vs P, with PIVI

Iters	MLP/FLP Decoder Throughput (Mbps)				
	P=32	P=64	P=96	P=128	P=192
1	3.0/2.6	3.9/3.5	4.4/4.1	4.6/4.3	4.7/4.4
2	1.9/1.6	2.9/2.5	3.4/3.0	3.7/3.3	3.8/3.4
3	1.4/1.2	2.3/1.9	2.8/2.4	3.1/2.7	3.1/2.7
4	1.1/0.9	1.9/1.6	2.3/2.0	2.7/2.3	2.6/2.3
5	1.0/0.8	1.6/1.3	2.0/1.7	2.3/2.0	2.4/2.0
6	0.8/0.7	1.4/1.1	1.8/1.5	2.1/1.8	2.1/1.8
7	0.7/0.6	1.2/1.0	1.6/1.3	1.9/1.6	1.9/1.6

parallel sub-blocks is more than an order of magnitude faster than the CPU implementation. The C code run on the CPU is compiled using gcc with -O3 optimization flag and is single threaded i.e it does not utilize any parallelism on multiple CPU cores.

As is to be expected, the throughput of the Full Log-MAP algorithm is lesser than that of the Max Log-MAP algorithm and the throughput of PIVIDSTW guarding mechanism is lesser than that of PIVI guarding mechanism. For 5 iterations, compared to the Max Log-MAP algorithm, the Full Log-MAP algorithm is slower by approximately 300Kbps. The throughput achieved by PIVIDSTW guarding mechanism with window size $g = 5$ is approximately 150Kpbs lesser than that achieved by PIVI guarding mechanism.

The throughput achieved by the designed turbo decoder for different number of sub-blocks P is shown in Table. III. The throughput increases initially with increase in P as the occupancy increases because of the decrease in the shared memory usage per each thread block. The throughput tends to saturate with a further increase in P as now the register usage and not the shared memory becomes the constraining factor for occupancy.

C. Comparison

As explained earlier, there are currently hardly any published results on implementation of turbo decoders on GPUs. [4] presents results of implementation on an Nvidia Tesla C1060 GPU, and they are able to obtain a speed of up to 6.77 Mbps for a Max Log-MAP implementation with $w = 64$ and 5 iterations on this machine. A direct comparison between the two architectures is complicated by the fact that they operate at different frequencies and the Tesla architecture has a greater number of registers, thus alleviating one of the main bottlenecks in the implementation.

Another important factor that makes comparisons difficult is the fact that the implementation in [4] uses 100 blocks loaded into GPU memory at a time. Even though they account for the memory transfer time, the presence of independent blocks makes it easier to find parallelism. In our implementation, we have considered only one data stream at a time, in order to focus on getting a practical SDR implementation suitable for single user terminals.

VI. Conclusion

A 3GPP standards compliant configurable turbo decoder has been implemented completely in software. More than an order of magnitude speed up over an implementation done purely on the CPU has been achieved. Different guarding mechanisms to mitigate the degradation in BER performance of the parallelized Log-MAP algorithm have been presented. The PIVIDSTW guarding mechanism as been shown to be capable of producing a BER performance that is within 0.01dB of the optimal case of a single code block. The principles used in the mapping of the Log-MAP algorithm can be readily extended to the newer architectures of Nvidia GPUs with larger number of cores and larger sized low latency memories to achieve a further increase in throughput.

References

[1] W. E. Ryan, “A Turbo Code Tutorial,” New Mexico State University, Tech. Rep., 1997.

[2] S. A. Abrantes, “From BCJR to turbo decoding: MAP algorithms made easier,” University of Porto, Tech. Rep., 2004.

[3] M. Marandian, J. Fridman, Z. Zvonar, and M. Salehi, “Performance analysis of turbo decoder for 3GPP standard using the sliding window algorithm,” in *Personal, Indoor and Mobile Radio Communications, 2001 12th IEEE International Symposium on*, vol. 2, sep/oct 2001, pp. E–127–E–131 vol.2.

[4] M. Wu, Y. Sun, and J. Cavallaro, “Implementation of a 3GPP LTE turbo decoder accelerator on GPU,” in *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, oct. 2010, pp. 192–197.

[5] D. Lee, M. Wolf, and H. Kim, “Design space exploration of the turbo decoding algorithm on GPUs,” in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, ser. CASES ’10. New York, NY, USA: ACM, 2010, pp. 217–226.

[6] J. Sun and O. Takeshita, “Interleavers for turbo codes using permutation polynomials over integer rings,” *Information Theory, IEEE Transactions on*, vol. 51, no. 1, pp. 101–119, jan. 2005.

[7] O. Muller, A. Baghdadi, and M. Jezequel, “Exploring Parallel Processing Levels for Convolutional Turbo Decoding,” in *Information and Communication Technologies, 2006. ICTTA ’06. 2nd*, vol. 2, 0-0 2006, pp. 2353–2358.

[8] S. Hong and H. Kim, “An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness,” Georgia Institute of Technology, Tech. Rep., 2009.

[9] D. Kirk, W. Hwu, and W. Hwu, *Programming massively parallel processors: a hands-on approach*, ser. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010.

[10] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[11] NVIDIA Corporation, *NVIDIA CUDA C Best Practices Guide Version 3.2*, 2010. [Online]. Available: http://developer.nvidia.com/cuda-downloads

[12] ——, *NVIDIA CUDA C Programming Guide Version 3.2*, 2010. [Online]. Available: http://developer.nvidia.com/cuda-downloads

[13] M. Harris, “Optimizing Parallel Reduction in CUDA,” *NVIDIA Developer Technology*, 2007.