# GPU IMPLEMENTATION OF A PROGRAMMABLE TURBO DECODER FOR SDR

*A Project Report*

*submitted by*

**DHIRAJ REDDY N.Y**

*in partial fulfilment of the requirements*
*for the award of the dual degrees of*

**BACHELOR OF TECHNOLOGY**
**and**
**MASTER OF TECHNOLOGY**


**DEPARTMENT OF ELECTRICAL ENGINEERING**
**INDIAN INSTITUTE OF TECHNOLOGY MADRAS.**

**JUNE 2011**

Dedicated to the most loving and wonderful parents

***Parvathi*** and ***Dhanunjaya Reddy***

# THESIS CERTIFICATE

This is to certify that the thesis titled **GPU Implementation of a Programmable Turbo Decoder for SDR**, submitted by **Dhiraj Reddy N.Y**, to the Indian Institute of Technology, Madras, for the award of the degrees of **Bachelor of Technology** and **Master of Technology**, is a bona fide record of the research work done by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Dr.Nitin Chandrachoodan**
Research Guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 18th June 2011

# ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Nitin Chandrachoodan for his constant guidance and support throughout the project. I would like to particularly thank him for allowing me do to the work at my own pace and for being supportive at times when the learning curve in the project was a bit steep. I would also like to thank Dr. Andrew Thangaraj for his video lectures on Error Control Codes and also for providing valuable suggestions on the project work.

A special thanks to Dr. David Koilpillai, Dr. Krishna Vasudevan, Dr. Nandita Dasgupta and Dr. Shankar Balachandran for all the courses taught by them in such an inspiring manner.

I am grateful to have spent 5 years of my life in the beautiful campus of IIT Madras. I would like to thank my friends Shravan, Aditya, Karthik, Ashok, Krishnagiri, Shankar Ganesh, Srinivas, Aanveeksha, Chaitanya and Venkatesh for making the stay here so memorable and enjoyable. A special thanks to Vamsi Krishna, Ranadheer and Goutham for the wonderful time we had playing Cricket and Tennis together.

Above all, I would like to thank my parents and sister Vineesha for their love, support and encouragement for without which this would not have been possible.

# ABSTRACT

KEYWORDS: Turbo Decoder, GPU implementation, CUDA, Parallel Log-MAP, Guarding Mechanisms

The project involves the implementation of a 3GPP standards compliant programmable turbo decoder on GPU. The primary motivation of the project is to implement a high throughput turbo decoder completely in software. Since the implementation is done in software, the decoder can be readily configured for the various evolving standards of mobile telecommunication. The challenge in implementing a turbo decoder on a GPU is suitably parallelizing the Log-MAP decoding algorithm and doing an architecture aware mapping of the parallel algorithm on to the GPU. Parallelization of the Log-MAP algorithm comes at the cost of a reduced BER performance and hence ways of mitigating the degradation in BER performance needs to be studied. Finding ways to efficiently distribute the workload across cores and effectively utilizing the fast on chip memory available on the device form the critical parts of the mapping process.

In the current implementation on GPU, more than an order of magnitude high throughput over an implementation done purely on the CPU has been achieved. In addition, effect of the approximations done in parallelizing the Log-MAP algorithm on the BER and FER performance has been analyzed for both Full Log-MAP and Max Log-MAP algorithms. The effectiveness of the 3 different kinds of guarding mechanisms in mitigating this degradation in performance has also been presented.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# List of Algorithms

# ABBREVIATIONS

| | |
|---|---|
| **GPU** | Graphic Processing Unit |
| **SDR** | Software Defined Radio |
| **CUDA** | Compute Unified Device Architecture |
| **FEC** | Forward Error Correction |
| **BER** | Bit Error Rate |
| **FER** | Frame Error Rate |
| **UMTS** | Universal Mobile Telecommunications System |
| **3GPP** | Third Generation Partnership Project |
| **CPU** | Central Processing Unit |
| **MAP** | Maximum a posteriori Probability |
| **BPSK** | Binary Phase Shift Keyring |
| **AWGN** | All White Gaussian Noise |
| **RSC** | Recursive Systematic Convolutional |
| **LLR** | Log Likelihood ratio |
| **BM** | Branch Metric |
| **SM** | State Metric |
| **API** | Application Programming Interface |
| **SMP** | Streaming Multi-Processor |
| **SP** | Streaming Processor |
| **SIMT** | Single Instruction Multiple Thread |
| **FR** | Forward Recursion |
| **BR** | Backward Recursion |
| **PIVI** | Previous Iteration Value Initialization |
| **DSTW** | Double Sided Training window |
| **PIVIDSTW** | Previous Iteration Value Initialization with Double Sided Training Window |
| **QPP** | Quadratic Permutation Polynomial |

# NOTATION

| | |
|---|---|
| $E_b/N_0$ | Energy per bit to noise power spectral density ratio |
| N | Code Block Length |
| E1 | Encoder 1 |
| E2 | Encoder 2 |
| $u$ | Input bits |
| $x_1^p$ | Bits encoded by E1 |
| $x_2^p$ | Bits encoded by E2 |
| s | Trellis state |
| k | Trellis stage |
| D1 | Half-Decoder 1 |
| D2 | Half-Decoder 2 |
| $y^s$ | Channel output values of systematic bits |
| $y_1^p$ | Channel output values of bits encoded by E1 |
| $y_1^p$ | Channel output values of bits encoded by E2 |
| $L_e^{12}$ | Extrinsic LLR passed from D1 to D2 |
| $L_e^{21}$ | Extrinsic LLR passed from D2 to D1 |
| $\alpha$ | Forward Metric alpha |
| $\beta$ | Backward Metric beta |
| $\gamma$ | Branch Metric gamma |
| $\Lambda$ | State LLR |
| B | Blocks launched by a kernel |
| T | Threads launched by a kernel |
| P | Number of parallel sub-blocks |
| w | Size of a sub-block |
| g | Size of the training window on either sides |
| i | Thread Id |

# CHAPTER 1

# INTRODUCTION

## 1.1 Motivation

Turbo codes are an important class of forward error correction(FEC) codes because of their low Bit error rate(BER) performance and are widely used in many 3G and 4G standards such as UMTs, 3GPP LTE etc. The turbo decoder is the most computationally challenging and time consuming part of the encoding-decoding process because of the complex iterative decoding algorithm. Hence, typically ASIC implementations of the decoder are preferred for achieving high data throughput. GPUs provide an alternative for achieving the same high data throughput as achieved on dedicated ASIC implementations in software. Also, implementation of a high throughput turbo decoder in software forms an important part of Software Defined Radio (SDR) for testing out various evolving 4G standards in software. Today, even many hand held devices contain GPUs and the decoder implementation may be extended to them as well.

## 1.2 Objective

The turbo decoding algorithm is inherently a highly serial algorithm and a direct implementation of it on a GPU would not benefit much as not much of it can be parallelized. Hence, there is a need to modify the algorithm to facilitate the possibility of parallelizing it. The modifications tend to cause a degradation in BER performance and hence ways of minimizing the degradation needs to be studied.

Finally, a complete implementation of a turbo decoder is to be done on the GPU while optimizing it for performance. The speed up, if any, that can achieved through a implementation on the GPU over an implementation on the CPU needs to be analyzed.

## 1.3 Organization

The organization of the rest of the thesis is as follows:

**Chapter 2** briefly describes the Log-MAP turbo decoding algorithm.

**Chapter 3** presents the architecture of a CUDA enabled Nvidia GPU, briefly describing the software model, the hardware execution model and the different kinds of device memories present it.

**Chapter 4** discusses ways of parallelizing the Log-MAP turbo decoding algorithm. Different guarding mechanisms are presented to mitigate the associated BER performance degradation.

**Chapter 5** discusses an architecture aware mapping of the parallel Log-MAP algorithm on to the GPU. The mapping is done for the 3GPP LTE standard turbo decoder.

**Chapter 6** present BER vs $E_b/N_0$ plots and data throughput results for the designed turbo decoder on the GPU.

# CHAPTER 2

# TURBO DECODING

The Fig. (2.1) illustrates the complete encoding and turbo decoding process. First, a input bit-stream of length $N$ to be transmitted over a noisy channel is encoded by passing it through a turbo encoder. The encoder adds redundancy to the input bit-stream and outputs $2N$ or $3N$ length bit-stream depending on it being a rate $\frac{1}{2}$ or rate $\frac{1}{3}$ encoder respectively. Next, BPSK modulation is performed on the encoded bit-stream and the modulated bit-stream is then transmitted over an AWGN channel. Finally, the channel output values are fed into the turbo decoder which then performs the iterative decoding and returns the decoded bits.

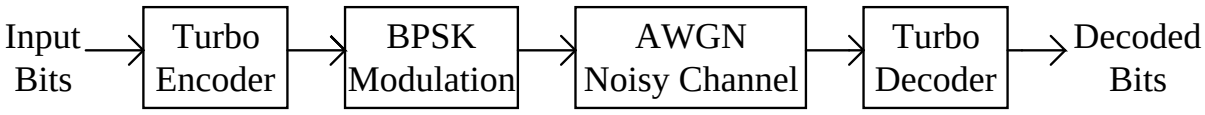



Figure 2.1: Encoding and Decoding Process

## 2.1 Turbo Encoder

Turbo encoder consists of two recursive systematic convolutional encoders separated by a $N$ bit interleaver, as shown in the Fig. (2.2). The two encoders are connected in the so called *parallel* concatenation fashion Ryan (1997). It is a rate $1/3$ encoder when no puncturing is performed i.e. it maps $N$ input-bits to $3N$ encoded-bits. The encoder $E1$ performs encoding on the input bit stream whereas the encoder $E2$ performs encoding on the interleaved bit stream.

Both the recursive systematic encoders are characterized by the same generator matrix

$$G(D) = \begin{bmatrix} 1 & \frac{g_2(D)}{g_1(D)} \end{bmatrix} \tag{2.1}$$

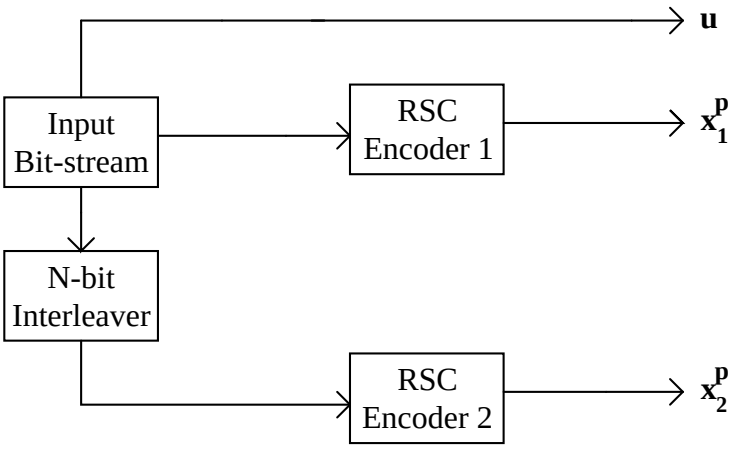



Figure 2.2: Turbo Encoder

For the 3GPP LTE standard, $g_1(D)$ and $g_2(D)$ are as follows:

$$g_1(D) = 1 + D^2 + D^3 \tag{2.2}$$

$$g_2(D) = 1 + D + D^3 \tag{2.3}$$

Hence,

$$G(D) = \begin{bmatrix} 1 & \frac{1+D+D^3}{1+D^2+D^3} \end{bmatrix} \tag{2.4}$$

The trellis state transition diagram of the above 3PP LTE standard encoder is as shown in the Fig. (2.3). The trellis has 8 states and the encoder starts from the trellis state 0. Both the next state and output parity bit of the encoder are functions of the current state and the input parity bit. Depending upon the input bit sequence, the encoder traverses through the different trellis states while producing the corresponding encoded bits.

## 2.2 Turbo Decoder

The structure of the Turbo Decoder is as shown in the Fig. (2.4). It Consists of two identical half-decoders, both of which perform the same set of computations but on different inputs. The inputs to the first half-decoder $D1$ are the deinterleaved extrin-

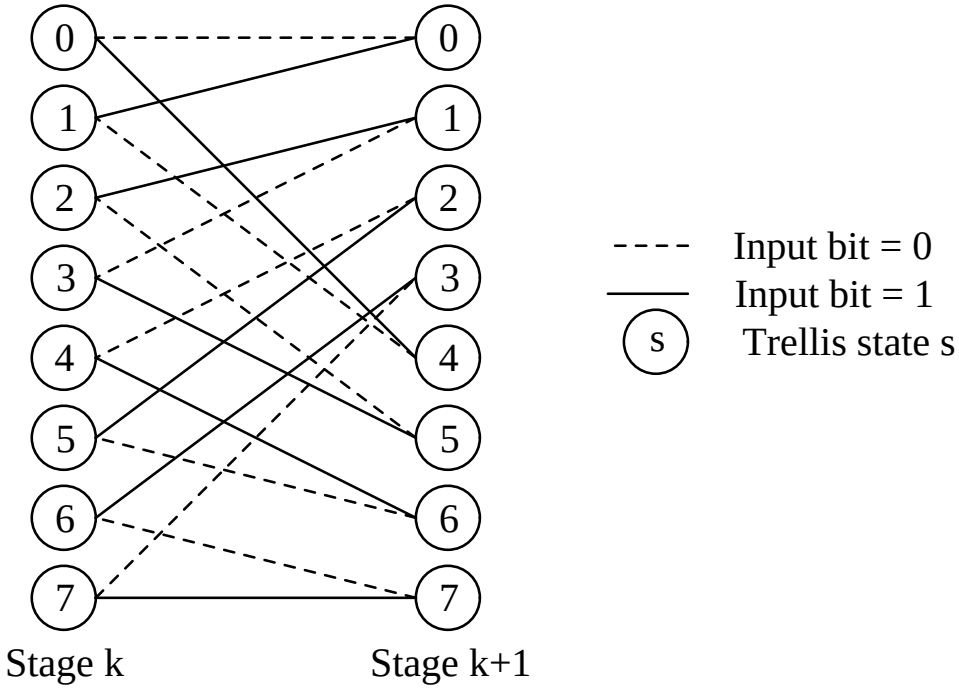



Figure 2.3: Trellis state transition diagram of 3GPP LTE encoder

sic Log-likelihood ratios(LLRs) from the second half-decoder $D2$, the noise corrupted channel values of the systematic bit stream and the noise corrupted channel values of the encoded bit stream from $E1$. The inputs to the second half-decoder $D2$ are the interleaved extrinsic Log-likelihood ratios(LLRs) from the first half-decoder $D1$, the interleaved noise corrupted channel values of the systematic bit stream and the noise corrupted channel values of the encoded bit stream from $E2$. The exchange of appropriately interleaved or deinterleaved extrinsic LLRs between the half-decoders is at the heart of the iterative turbo decoding process.

## 2.3 Log-MAP Algorithm

The extrinsic LLRs in each half-decoder can be estimated using the Maximum a posteriori probability(MAP) algorithm. The MAP algorithm provides optimal BER performance at the cost of high computational complexity Ryan (1997). The MAP algorithm in its direct form involves the computation of exponentials and a hence simplified version of it, the Log-Map algorithm is often used both for hardware and software implementations Abrantes (2004).

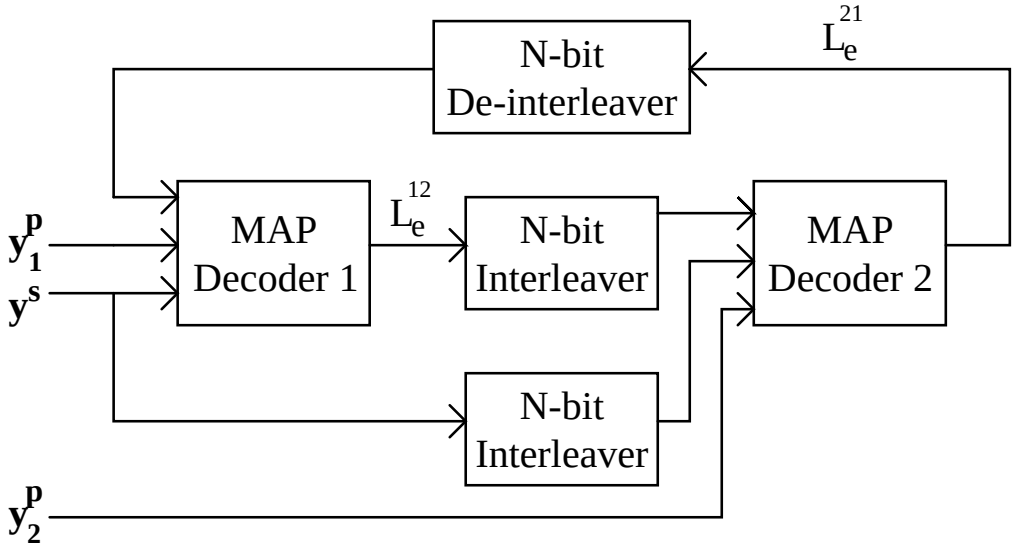



Figure 2.4: Iterative Turbo Decoder

The MAP algorithm involves computation of exponentials in the calculation of branch metric(BM) $\gamma$ in addition to expensive multiplications in the computation of extrinsic LLR from $\alpha, \beta$ and $\gamma$. The Log-Map Algorithm removes the need to compute exponentials during BM computation and substitutes the costly multiplications with additions. Hence, the Log-Map algorithm is often the preferred choice for both hardware and software implementations of the turbo decoder. Two types of Log-Map algorithms - the Full-Log-Map algorithm and the Max-Log-Map algorithm are briefly presented below:

The MAP algorithm calculates the Log-Likelihood Ratio(LLRs), the logarithm of the ratio of *a posteriori* probability (APP). The Log-MAP algorithm reduces the computational complexity of the MAP algorithm by calculating the logarithms of $\alpha$, $\beta$ and $\gamma$ used in the calculation of extrinsic LLR.

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

The LAPP or extrinsic LLR of a bit $u_k$ is the log of the ratio of the probability of $u_k$ being '1' to the probability of $u_k$ being '0' given the channel output $\boldsymbol{y}$.

$$L_e(k) = \ln \frac{P(u_k = 1|\,\mathbf{y})}{P(u_k = -1|\,\mathbf{y})} \tag{2.5}$$

After sufficient number of decoding iterations, the decision on the final decoded bits can be taken using the $L_e$ values evaluated by any of the two half-decoders. If $L_e(k)$ is positive, the bit $u_k$ is decoded to be 1 and if $L_e(k)$ is negative, the bit $u_k$ is decoded to be -1 i.e being 0.

To decode a codeword with N bits, the turbo decoder performs a forward traversal followed by a backward traversal. It computes the branch metrics $\gamma$ and then the state metrics $\alpha$ and $\beta$ during the forward and backward traversal respectively. These are then used in the computation of the extrinsic LLRs. The state transitions are defined by the particular encoder used. For the 3GPP standard encoder, characterized by the trellis structure shown in the Fig. (2.3), there are two incoming paths for each of the 8 states, one for input bit $u_b = 0$ and another for $u_b = 1$.

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 :

$$\gamma_k(s_{k-1}, s_k) = (L_c(y_k^s) + L_a(k))u_k + L_c(y_k^p)x_k^p \tag{2.6}$$

where $L_c$ is the channel reliability value, $u_k$ is the input bit which causes the trellis transition from state $s_{k-1}$ to state $s_k$ and $x_k^p$ is the parity bit generated for the state transition from $s_{k-1}$ to $s_k$.

As is evident from the Fig. (2.3), not all state transitions are possible from the trellis structure. In fact, as the number of lines in the Fig. (2.3) indicate, only 16 state transitions are permitted for the trellis structure of the 3GPP standard encoder. The $\gamma(s_{k-1}, s_k)$ values for all the impossible state transitions are set to 0.

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

$$\alpha_k(s_k) = max^*_{s_{k-1} \in K}(\alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k)) \tag{2.7}$$

where $K$ is the set of all states from which there is a possibility of a state-transition to the state $s_k$.

For the trellis of the 3GPP standard encoder, as shown in Fig. (2.3), each state at a stage $k$ of the trellis can be reached from exactly two states of the trellis at stage $k-1$. One such state transition occurs when the input bit $u_k = 0$ and the other when $u_k = 1$.

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

$$\beta_k(s_k) = max^*_{s_{k+1} \in K}(\beta_{k+1}(s_{k+1}) + \gamma_{k+1}(s_k, s_{k+1})) \tag{2.8}$$

where $K$ is the set of all states to which there is a possibility of a state-transition from the state $s_k$.

For the trellis of the 3GPP standard encoder, as shown in Fig. (2.3), from each state at a stage $k$ of the trellis, exactly two states of the trellis at stage $k+1$ can be reached, one each when the input bit $u_k = 0$ and when $u_k = 1$.

After computation of $\alpha$, $\beta$ and $\gamma$, two LLRs per trellis state are computed. One state LLR per state $s_k$, $\Lambda(s_k|\, u_k = 0)$, for the incoming path to $s_k$ that is caused by the input bit $u_k = 0$ and another LLR per state $s_k$, $\Lambda(s_k|\, u_k = 1)$, for the incoming path to $s_k$ that is caused by the input bit $u_k = 1$.

The state LLR $\Lambda(s_k|\, u_k = 0)$, for a state $s_k$ at a trellis stage $k$ is defined as:

$$\Lambda(s_k|\, u_k = 0) = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \tag{2.9}$$

where $s_{k-1}$ is the state at trellis stage $k-1$ from which the input bit $u_k = 0$ causes the transition to state $s_k$ at trellis stage $k$.

Similarly, the state LLR $\Lambda(s_k|\, u_k = 1)$, for a state $s_k$ at a trellis stage $k$ is defined as:

$$\Lambda(s_k|\, u_k = 1) = \alpha_{k-1}(s_{k-1}) + \gamma_k(s_{k-1}, s_k) + \beta_k(s_k) \tag{2.10}$$

where $s_{k-1}$ is the state at trellis stage $k-1$ from which the input bit $u_k = 1$ causes the transition to state $s_k$ at trellis stage $k$.

The extrinsic LLR for $u_k$ is computed as:

$$L_e(k) = max^*_{s_k \in K}(\Lambda(s_k|\, u_b = 1)) - max^*_{s_k \in K}(\Lambda(s_k|\, u_b = 0)) - L_c(y^s_k) - L_a(k) \tag{2.11}$$

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

$$max^*(S) = \ln(\sum_{s \in S} e^s). \tag{2.12}$$

i.e. the function $max^*(S)$ is defined over a set S such that it is equal to the logarithm of the summation of the exponentials of all $s \in S$.

The manner 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.

### 2.3.1 $max^*$ Function

As defined in Eq. (2.12), the $max^*$ function involves the computation of several exponentials and a logarithm. To completely remove or reduce the need to compute these computationally expensive functions, the following two approximations are used to compute the $max^*$ function.

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

$$max^*(a, b) = max(a, b) + \ln(1 + e^{-|b-a|}) \tag{2.13}$$

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

$$max^*(a, b) = max(a, b) \tag{2.14}$$

The $max^*(a, b, c)$ is trivially computed by recursive calls to $max^*$ as:

$$max^*(a, b, c) = max^*(a, max^*(b, c)) \tag{2.15}$$

The above procedure can be readily extended to calculate the $max^*(S)$ function over any set $S$.

## 2.4 QPP Interleaver

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

$$\Pi(x) = f_1 x + f_2 x^2 \pmod N \tag{2.16}$$

where $f_1$ and $f_2$ satisfy several properties detailed in Sun and Takeshita (2005).

A computationally less expensive way of computing the QPP interleaver function defined in Eq. (2.16) is

$$\Pi(x) = (f_1 + f_2 x \pmod N) x \pmod N \tag{2.17}$$

# CHAPTER 3

# GPU ARCHITECTURE AND SOFTWARE MODEL

The concept of parallel programming is by no means new and has been existing in the high performance computing community since decades. However, such parallel programs needed to be run on large scale and expensive computers to harness the parallel computing potential. Hence, parallel programming and the use of such expensive computers was restricted to a few elite applications. However, the advent of modern graphic cards, driven primarily by the video gaming industry, has made available *a parallel computer* to the common programmer. The graphic cards are present in almost all of the computers produced today.

The terms ‘graphic card’ and ‘GPU’ are often used interchangeably, with ‘graphic card’ generally referring to the physical hardware and ‘GPU’ referring to the graphic card being treated as a co-processor. In addition, often the terms ‘host’ and ‘device’ are used to refer to the CPU and the GPU respectively.

Most modern GPUs contain a large number of cores and each core can run a large number of hardware threads simultaneously. This is in sharp contrast to the architecture of a general purpose CPU. The GPUs are excellent for applications that are computationally intensive and which exhibit gratuitous amount of data parallelism. GPUs hands-down beat the traditional CPUs in the number of floating point calculations possible per second on them.

However, till very recently harnessing the parallel computing power of the GPUs for general purpose computing was a esoteric subject. It required the need to pose the general purpose computing problem as a graphics problem and as if that wasn’t enough, the knowledge of graphics API was necessary to program the GPUs. However, all this changed with the advent the CUDA enabled Nvidia GPUs. The Nvdia GPUs with the CUDA architecture can be easily programmed using the language CUDA C. The CUDA C language consists of only a few extensions above the C language and is fairly easy to

learn compared to knowing the graphics API. The enormous floating point computation potential of GPUs and the ease of programming them has precipitated the use GPUs for general purpose computing.

## 3.1 Architecture of a Modern GPU

The CUDA enabled Nvdia GPU consists of an array of highly threaded streaming multiprocessors(SMPs). The number of SMPs on a GPU differs from one generation of graphic cards to another. For example, the Nvidia GeForce 8600M GT card has 4 SMPs, the Nvdia GeForce 9800 GX2 card has 16 SMPs and the Nvidia Tesla C1060 has 30 SMPs. Each SMP is highly threaded in that it can run as many as 512 or 768 hardware threads concurrently. Each SMP has a number of Streaming Processors(SPs) present in it that share the control logic and instruction cache. The Geforce and Tesla generations of Nvidia GPUs have 8 SPs per each SMP. The number of SMPs and SPs present in a GPU is a measure of the true hardware level parallelism present in it. For typical implementations in CUDA C with sufficient number of thread blocks, the speed of the implementation scales with the number of SMPs present in the GPU. The number of SMPs in a GPU, number of SPs per SMP and the total number of SPs per GPU for different CUDA enabled graphic cards is as shown in the table.

Table 3.1: Number of SMPs and SPs present on different GPUs

| Type of GPU | No. of SMPs | No. of SPs per SMP | Total No. of SPs |
|---|---|---|---|
| Nvidia GeForce 8600M GT | 4 | 8 | 32 |
| Nvidia GeForce 9800 GX2 | 16 | 8 | 128 |
| Nvidia Tesla C1060 | 30 | 8 | 240 |

The SPs present on a GPU are batched into groups of 8. Each group of 8 SPs share a memory space called as shared memory. As shown in in Fig. (3.1), the 8 SPs, the shared memory and the registers allocated to them constitute one SMPs.

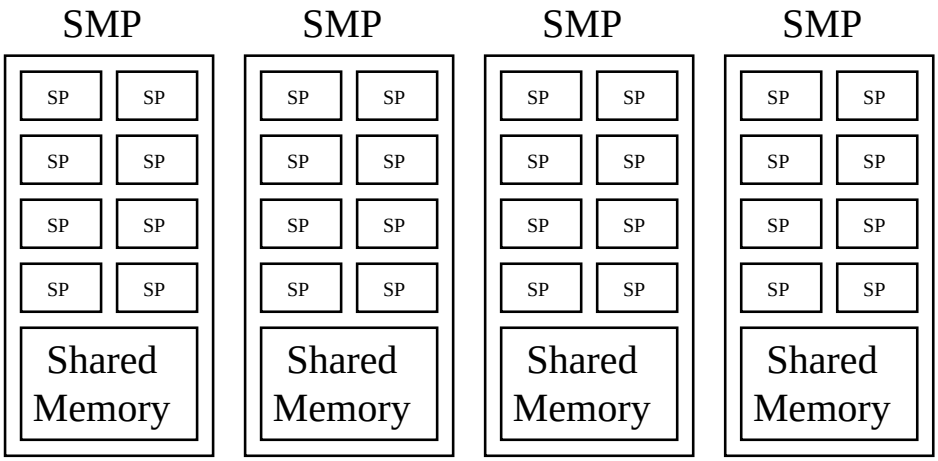



Figure 3.1: A GPU with 4 SMPs

## 3.2 CUDA Software Model

The software model presented by the CUDA architecture removes the need to know the actual number of SMPs or SPs present on the GPU. This abstraction is useful as it always keeps the code written to be generic allowing easy portability from one generation of GPUs to other. However, for fully optimizing the code for performance, the GPU in hand needs to be known.

The CUDA kernel, which is the name for the parallel function in CUDA C, can be launched by specifying the number of threads and thread blocks NVIDIA Corporation (2010*b*). The kernel is very similar to a normal function except that it is launched with a certain number of threads and thread blocks. For a kernel launched with $T$ threads and $B$ thread blocks, each thread block contains $T$ threads and the total number of threads launched by the kernel is $BT$. When there is no ambiguity, a thread block is simply referred to as a block. The thread Id and block Id differentiates each thread and block receptively from others. At the software level of abstraction, threads and blocks are a way of specifying data parallelism Sanders and Kandrot (2010). Each thread in the kernel executes the same sequence of instructions but on different data. The thread identifies this different data by using both its own thread Id and the block Id of the block containing it. All the threads from all the blocks launched by the kernel can be viewed to be executing in parallel. All the threads of one block can easily exchange data among

themselves using a fast on chip memory called shared memory.

Four different kinds of memories are presented to the programmer by the CUDA software model. Each kind of memory as a different scope, latency and size.The type memory in which the variables defined in the kernel should reside needs to be explicitly specified by the programmer at the time of writing the kernel. The different kinds of device memories and their characteristics are described in detail in the following section.

## 3.3 Device Memories

There are 5 different kinds of memories present on the device - Registers, Shared Memory, Global Memory, Constant Memory and Texture Memory. Other than the registers, all other kinds of memories need to be explicitly allocated by the programmer at the time of writing the kernel function. Each kind of memory has a different scope, latency and size. Registers and Shared memory are present on the die and hence are the fastest.

### 3.3.1 Registers

Registers are allocated to each individual thread and each thread can access only its own registers. A kernel function uses registers to hold the frequently accessed variables that are private to each thread. Unlike other memories, the programmer cannot specify which variables to store in the registers at the time of writing the kernel function. This is automatically done by the compiler when it converts the CUDA C code to object code. Since, registers reside on the die, they are the fastest available memory on the device.

### 3.3.2 Shared Memory

The Nvidia GeForce generation of GPUs have 16KB of shared memory per SMP. Shared memory also resides on the die and is almost as fast as the registers. However, unlike registers, the programmer has to specify which variables are to be stored in the shared memory at the time of writing the kernel. Hence, optimal use of shared

memory forms a crucial part of writing the kernel function.

Shared memory is allocated to a thread block. All the threads in a block can access variables in the shared memory locations allocated to that block. Shared memory is an efficient means for threads in a block to cooperate by sharing their input data and the intermediate results of their work. The scope of shared memory allocated to a block is restricted to that block i.e. the variables in the shared memory of a block are active only when the threads of that block are executing and the threads from other blocks do not have access to these variables.

Since, in the software model, all the threads are considered to be executing in parallel, at any point of time, each thread can execute up to a different number of instructions in the kernel. Often, when there is a need to exchange data computed by various threads, there is a need to ensure that all the threads have executed exactly up to a certain number of instructions. The current CUDA architecture provides synchronization among threads of blocks and not among all the threads launched by the kernel. Barrier synchronization among threads of a block is a way of ensuring that all the threads of a block have executed instructions exactly up to a pre-defined synchronization point in the kernel. The current CUDA architecture supports barrier synchronization only on writes to shared memory but not on writes to global memory. So, whenever there is a need to exchange some intermediate data among the threads of a block, the intermediate data to be exchanged needs to be necessarily stored in the shared memory allocated to that block.

### 3.3.3 Global Memory

Global memory is the slowest and the biggest sized memory available on the device. The size of the global memory varies with the graphic card in hand and is typically in the range of 256MB to 1GB. Since the global memory is present off-chip, it is much slower than the shared memory. It is in fact around 100 times slower than the shared memory. Data on which computations are to be performed on the GPU, is first transferred from the host's memory to the global memory of the device.

The scope of the global memory, unlike the shared memory, is not restricted to a

single kernel launch. Storing data in the global memory is the only way of exchanging data between different kernel launches. Hence, whenever there is a needs to exchange data among different kernel launches, a copy of the data needs to be stored in the global memory before the termination of the kernel.

### 3.3.4 Constant Memory

The constant memory can be used for storing values that remain *constant* during the entire duration of a kernel execution on the GPU. In the worst case, the latency of constant memory is the same as that of the global memory but constant memory possess a cache unlike the global memory and when the values accessed are present in the cache, the latency is much lowered. The Nvidia Geforce generation of GPUs provide 64KB of constant memory. The data in the constant memory needs to be set from the host and cannot be altered during the GPU kernel launches.

### 3.3.5 Texture Memory

The texture memory is useful when the data access pattern exhibits either a 1-dimensional or a 2-dimensional spatial locality. Since, no part of the turbo decoding algorithm exhibits such data access pattern, the texture memory has not been used in this implementation.

## 3.4 Hardware Execution Model

In the software model, all the threads of all the blocks can be treated to be executing in parallel. Trivially, the true hardware parallelism, measured in terms of the number of SPs and SMPs present, limits such a possibility.

Each SMP has certain number of registers and a certain amount of shared memory present in it. The kernel definition specifies the amount of shared memory that each thread block uses. The NVCC CUDA C compiler, optimizes for performance and determines the number of registers that are required per each thread. The number of threads

per block, the amount of shared memory used per block and the number of registers required per thread together determine the number of blocks that can be truly executed in parallel on a SMP. The GeForce generation of GPUs can support a maximum of 512 or 768 concurrent threads per SMP. Both have a maximum of 8192 registers and 16KB of shared memory. In the current architecture, a maximum of 8 blocks can be executed at once on a SMP. For the maximum of 8 blocks to be executed concurrently on a SMP, the total amount of resource usage, measured in terms of total number of threads, registers and shared memory usage by all the 8 blocks shouldn't exceed the available resources on a SMP i.e. the total number of threads from all the blocks should be less than the maximum supported concurrent threads, the total shared memory usage by all the blocks should be less than 16KB and the total number of registers used by all the threads from all the blocks should be less than 8192. Whenever, any of the resource usage is exceeded, the number of blocks assigned for concurrent execution on a SMP is reduced.

A batch of 32 threads is called as a *warp*. The hardware executes the same instructions for all the threads in a warp. This style of execution, called single-instruction, multiple-thread (SIMT), is motivated by the hardware cost constraints, as it allows the cost of fetching and processing of an instruction to be amortized over a large number of threads Kirk *et al.* (2010). When different threads in the same warp follow different execution paths because of conditional statements, the execution gets serialized and execution of the warp would require multiple passes through all the divergent paths. Since, each SMP has 8 SPs, a warp is executed on it in 4 cycles with 8 threads being executed on the 8 SPs every cycle.

The architecture of a GPU differs markedly from the CPU in the absence of a hardware managed cache. The shared memory can be treated as a substitute for a cache but it is to be managed through software. The absence of multi-levels of caches readily exposes the GPUs to delays associated with fetches and stores to high latency memories. The GPU aims at hiding this delay by performing a low cost context switch among the light weight threads being executed concurrently. The higher the number of threads and thread blocks being run concurrently on the GPU, the better it would be in hiding the delays associated with accessing slow memories.

# CHAPTER 4

# PARALLEL LOG-MAP ALGORITHM

GPUs are very good at handling tasks that show large amounts of data-level parallelism. When the same set of computations is performed on several different data sets, the GPUs can effectively utilize the large number of processing cores present in them to produce a significant speed up over an implementation done purely on the CPU. Thus, to produce a speed-up on the GPU, the presence of data-level parallelism in an algorithm is fundamental.

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 state metric $\alpha$ and the backward state metric $\beta$ proceeds in a serial fashion along the trellis stages. In the N-stage trellis, the $\alpha$ values at each trellis stage are computed using the $\alpha$ values of the preceding trellis stage. Similarly, the $\beta$ values at each trellis stage are computed using the $\beta$ values of the succeeding trellis stage. As shown in Eq. (2.7) and Eq. (2.8), $\alpha_k(s)$ depends on $\alpha_{k-1}(s)$ and $\beta_k(s)$ depends on $\beta_{k+1}(s)$ respectively.The serial evaluation of the SMs forms the basis of the turbo decoding algorithm and any attempts to parallelize this part would result in severe BER performance loss.

However, there exist other parallelisms that can be exploited. The state transition metrics $\alpha$ and $\beta$ at each trellis stage can be computed concurrently for all trellis states. For example, in an 8-stage trellis, $\alpha_k(s)$ for states $s = 0, 1..7$ can all be computed in parallel, since they do not depend on each other. This kind of data parallelism where a SM or BM can be computed concurrently for all the trellis states or state transitions respectively is termed as Trellis state level parallelism. The branch metric $\gamma(s_{k-1}, s_k)$ computation exhibits trellis stage level data parallelism in addition to trellis state level parallelism. The BMs of all the stages of the N-stage trellis can all be computed concurrently since they are completely data independent. The above two data parallelisms are inherently present in the MAP decoding algorithm. Another kind of parallelism

called as Sub-block level parallelism is also possible. Here, the codeword of length $N$ is split into several smaller codewords of length $N/P$ each, depending on the degree of parallelization $P$. Each smaller code word is treated as a sub-block and decoding is performed on each sub-block independently in the half-decoder. The trellis state level parallelism is inherently present in the algorithm and hence doesn't cause any BER performance degradation, however, the sub-block level parallelism, brings along with it some degradation in BER performance. The degradation in BER performance increases with increase in the parallelism level $P$.

## 4.1 Trellis State Level Parallelism

The $\alpha_k(s)$, for all the states at a stage $k$ of the trellis can all be calculated at once if, $\alpha_{k-1}(s)$, for all the states at the previous stage of the trellis are known. The $\alpha_k(s)$ computation given by the Eq. (2.7) shows that $\alpha_k(s_k)$ for each state $s$ at a trellis stage $k$ depends only on $\alpha_{k-1}(s_{k-1})$ and $\gamma_k(s_{k-1}, s_k)$ and hence all of these can be computed in parallel during forward recursion once $\alpha_{k-1}(s)$ for all states in the previous trellis stage and $\gamma_k(s_{k-1}, s_k)$ values for all possible state transitions from the stage $k-1$ to the stage $k$ are known.

Similarly, $\beta_k(s)$, for all the states at a stage $k$ of the trellis can all be calculated at once if, $\beta_{k+1}(s)$, for all the states at the next stage of the trellis are known. The $\beta_k(s)$ computation given by the Eq. (2.8) shows that $\beta_k(s_k)$ for each state $s$ at a trellis stage $k$ depends only on $\beta_{k+1}(s_{k+1})$ and $\gamma_{k+1}(s_k, s_{k+1})$ and hence all of these can be computed in parallel during the backward recursion once $\beta_{k+1}(s)$ for all states in the next trellis stage and $\gamma_{k+1}(s_k, s_{k+1})$ values for all possible state transitions from stage $k$ to $k+1$ are known.

The BM $\gamma$ computation, given by the Eq. (2.6) also shows trellis state level parallelism. The BMs for all the possible state transitions can all be computed in parallel at each stage of the trellis. In fact, the BM computation shows complete data parallelism and the BMs of all the trellis stages themselves can all be computed in parallel

In the 3GPP LTE standard encoder, there are 8 trellis states and hence there is a

8-way parallelism to exploit in the computations of $\alpha$ and $\beta$. There are 16 possible state transitions between two consecutive stages of the trellis and the BMs $\gamma$ for all these state transitions can be computed in parallel. Hence, there is a 16-way parallelism to exploit in BM computations. Since, the Trellis state level parallelism is inherent to the Log-Map turbo decoding algorithm, it does not cause any BER performance degradation.

## 4.2 Sub-block Level Parallelism

Though the trellis state level parallelism exhibits some amount of data parallelism, the degree of parallelism shown is quite small. For the 3GPP LTE standard, the degree of parallelism is limited to 8 in SMs $\alpha$ and $\beta$ computations and to 16 in the BM $\gamma$ computation. 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 Muller *et al.* (2006). During each iteration, in each sub-block, the forward and backward recursions run only over the length of that sub-block. Hence, during each iteration, the forward and backward recursions of all the sub-blocks are completely data independent. Hence the forward and backward recursions of all the sub-blocks can be performed concurrently. As before, after each iteration, the extrinsic LLRs computed by each of the 2 half-decoders are exchanged with appropriate interleaving. Fig. (4.1) contrasts the the forward and backward recursions in the MAP algorithm with and without sub-block level parallelism.

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 obtained is equal to the number of sub-blocks $P$.

Increase in the number of sub-blocks $P$ increases the amount of data parallelism possible but it comes at the cost of reduced BER performance. The BER performance degradation increases with the increase in the number of parallel sub-blocks $P$.

The BER performance degradation occurs because the $\alpha$ values at the start of a sub-block and the $\beta$ values at the end of a sub-block are no longer known. Without any sub-block level parallelism, the $\alpha$ values at the beginning of a sub-block would have been set by the trellis-stage preceding the sub-block during the forward recursion through

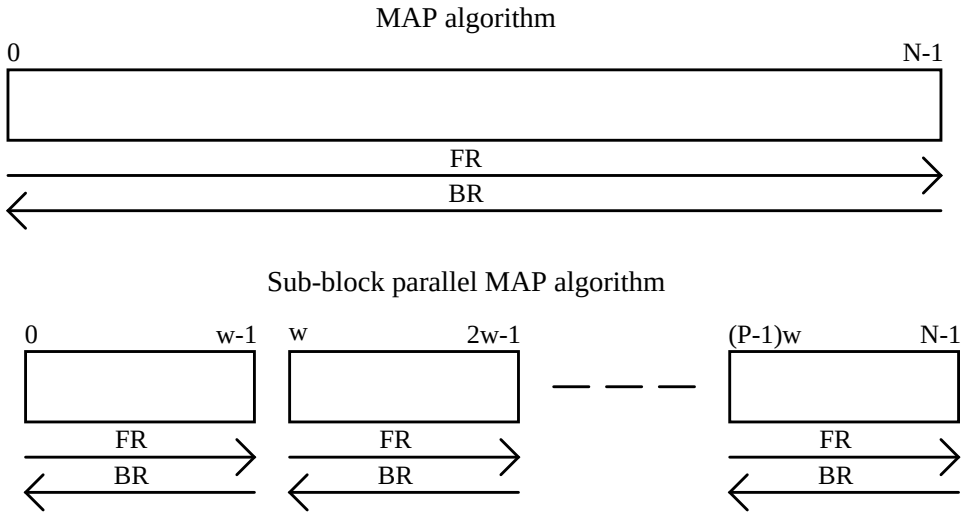



Figure 4.1: Recursions in parallel MAP algorithm

the complete trellis. Similarly, the $\beta$ values at the end of a sub-block would have been set by the trellis-state succeeding the sub-block during the backward recursion through the complete trellis. The BER performance loss is inevitable when trying to utilize this kind of parallelism because the SMs $\alpha$ and $\beta$ recursions are run over only a small region of the received sequence. However, the decrease in BER performance can be reduced by adopting various guarding mechanisms to estimate the values of $\alpha$ and $\beta$ metrics at the start and end of a sub-block respectively.

## 4.3 Guarding Mechanisms

The price paid in adopting the sub-block level parallelism is that the $\alpha$ and $\beta$ metrics at the start and end of the sub-block respectively are no longer known. Simply initializing the $\alpha$ and $\beta$ metrics of all states to equal values at the start and end of the sub-blocks leads severe performance degradation. To minimize the degradation in BER performance, three guarding mechanisms namely - Previous iteration value initialization(PIVI), Double sided training window(DSTW) and Previous iteration value initialization with double sided training window(PIVIDSTW) have been implemented and the relative merit of each of these in mitigating the degradation in BER performance has been studied. All the three guarding mechanisms have been pictorially depicted in Fig. (4.2).

### 4.3.1 Previous Iteration Value Initialization(PIVI)

This guarding mechanism is the simplest among the possible guarding mechanisms. It comes at the cost of added memory requirement and extra fetches and stores but involves no extra arithmetic computations. In this guarding mechanism, the $\alpha$ values at the beginning of a sub-block are initialized with the $\alpha$ values at end of the previous sub-block from the previous iteration. Similarly, beta values at the end of a sub-block are initialized with beta values at the beginning of the next sub-block from the previous iteration. For the first iteration, the alpha and beta values at the start and end of the sub-block are initialized to equal values and the above method is followed from the second decoding iteration onwards. The $\alpha$ initializations for the first sub-block and the $\beta$ initialization for the last sub-block do not follow the above mechanism and the $\alpha$ and $\beta$ values here are initialized in the same way as they would have been in the absence of any sub-block level parallelism.

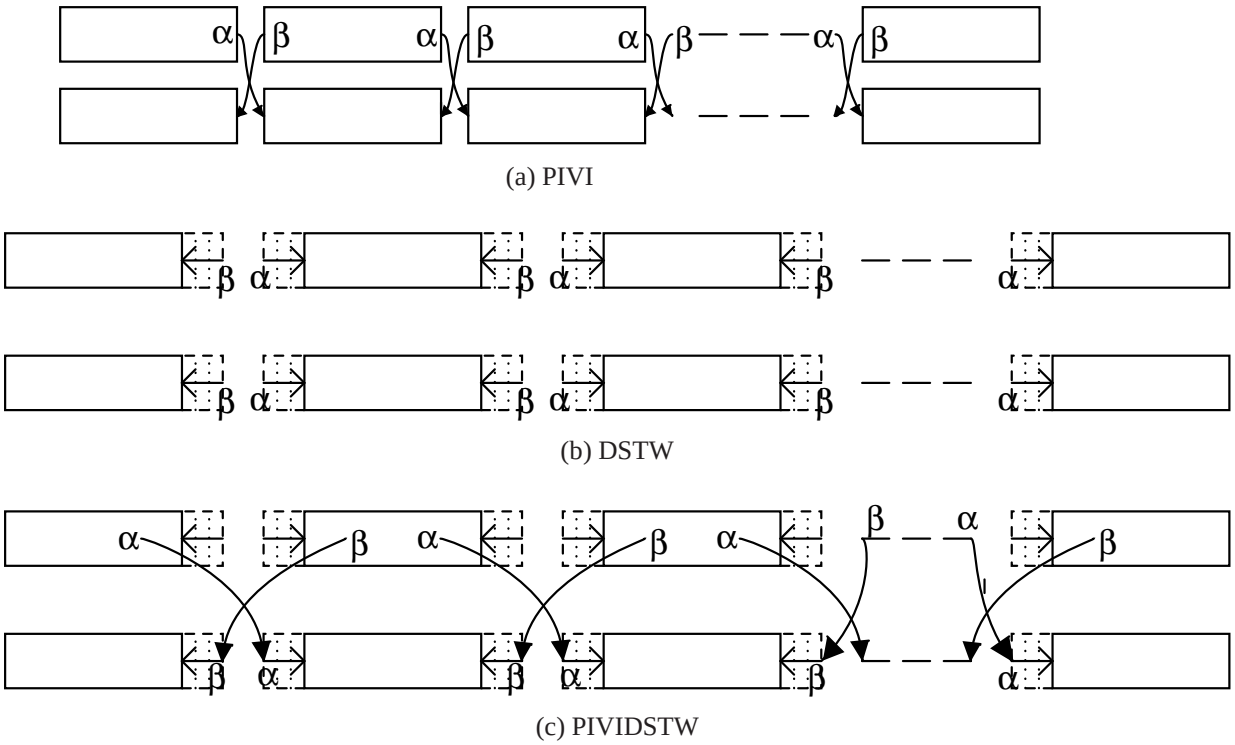



Figure 4.2: Types of Guarding Mechanisms

### 4.3.2 Double Sided Training Window(DSTW)

In Marandian *et al.* (2001) a training window based guarding mechanisms for $\beta$ metric computation in the sliding window turbo decoding algorithm is presented. Similarly, training windows can used on either sides of the sub-block to allow the $\alpha$ and $\beta$ metrics to develop into better estimates. At the start of the training window, both alpha and beta values for all the states are set to equal values and then a training window is run along using the LLRs from the previous iteration and channel output values. This guarding mechanism involves extra computations and the cost of these computations is proportional to the size of the training window. A larger training window results in better BER performance but would come at a cost of extra computations.

### 4.3.3 Previous Iteration Value Initialization with Double Sided Training Window(PIVIDSTW)

In this guarding mechanism, the features of both of the above mechanisms are combined. As in the Double sided training window guarding mechanism, 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 $\alpha$ and $\beta$ metrics of all states to equal value, they are set equal to the corresponding $\alpha$ and $\beta$ values of the trellis stage at the end of the guard windows from the previous iteration. Naturally, since this guarding mechanism combines both the above two mechanisms, it requires extra memory and involves extra computations.

# CHAPTER 5

# MAPPING THE LOG-MAP ALGORITHM ON TO THE GPU

The turbo decoding algorithm is a computationally intensive algorithm and hence traditional software implementations of it on either general purpose CPUs or dedicated DSPs have not been able to achieve significant throughput. GPUs with their large number of available parallel cores and huge computational potential present a good alternative for the implementation of a turbo decoder in software. However, as was described in chapter 3, the architecture of a graphics processor differs markedly from that of a general purpose processor. Therefore, for an implementation of an algorithm to achieve significant speed up on the GPU, an architecture aware mapping of the algorithm on to the GPU is paramount Hong and Kim (2009).

The principles of the following 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. The generator function defined by Eq. (2.4), a code block length of 6144 and the QPP interleaver have been taken as standard for the rest of the thesis.

## 5.1 Half-decoder Kernel

The turbo decoder and the parallelized turbo decoding algorithm have been described in chapters 2 and 4 respectively. At an abstract level, a rate $1/3$ turbo decoder takes as inputs, $3N$ floats, corresponding to the channel output values and returns $N$ integers corresponding to the decoded bits $0$ or $1$.The decoding is performed iteratively by two half-decoders each of which exchange appropriately interleaved extrinsic LLRs among themselves after each iteration. Let $\boldsymbol{y}^s$, $\boldsymbol{y}_1^p$ and $\boldsymbol{y}_2^p$ denote the channel output values of the systematic bits, the encoded bits from encoder $E1$ and the interleaved encoded bits

from the encoder $E2$ respectively. Let $L_e^{12}$ and $L_e^{21}$ denote the extrinsic LLRs returned by the half-decoders $D1$ and $D2$ respectively.

Both the half-decoders perform the same set of computations on the input data and are identical in all aspects other than in reading the input extrinsic LLRs and the systematic channel values $\boldsymbol{y^s}$ in direct or interleaved order and returning the computed extrinsic LLRs in direct or deinterleaved order. Hence, the same kernel function definition can be used for both the half-decoders. Appropriately defined conditional statements branching on the type of the half decoder can be used at the beginning and end of the kernel to read in the direct or interleaved systematic channel values and extrinsic LLRs and return the direct or deinterleaved extrinsic LLRs respectively.

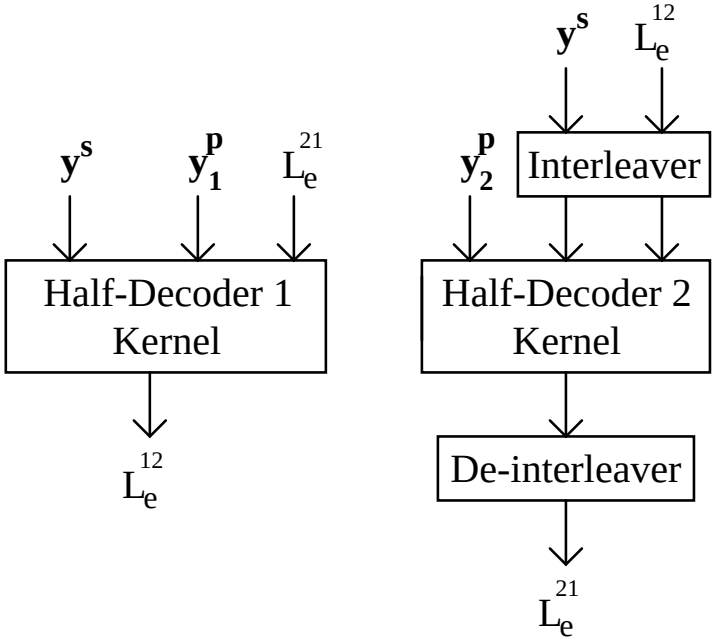



Figure 5.1: The Half-Decoder Kernels

The inputs and outputs of both the half-decoders has been illustrated in Fig. (5.1). The first half-decoder $D1$ takes as input the channel values $\boldsymbol{y^s}$, $\boldsymbol{y_1^p}$ and the extrinsic LLRs $L_e^{21}$ in direct order and returns the computed extrinsic LLRs $L_e^{12}$ in direct order. The second half-decoder $D2$ takes as input the channel values $\boldsymbol{y_2^p}$ in direct order and the systematic channel values $\boldsymbol{y^s}$ and extrinsic LLRs $L_e^{12}$ in interleaved order and returns the computed extrinsic LLRs, $L_e^{21}$ in deinterleaved order. In essence, the extrinsic LLRs returned by both the half-decoders $D1$ and $D2$ are in direct order. Hence, the decoder $D1$ directly reads the input extrinsic LLRs and returns the computed extrinsic LLRs

without any interleaving or deinterleaving whereas the decoder $D2$ reads extrinsic LLRs in interleaved order and returns computed extrinsic LLRs in deinterleaved order.

A kernel is launched with a certain number of thread and blocks depending on the amount and type of data parallelism present. The trellis state level parallelism (4.1) and sub-block level parallelism (4.2) nicely map to the threads and thread blocks software model presented by the CUDA architecture. The state level parallelism allows the computation of $\alpha_k(s)$ and $\beta_k(s)$ to be done concurrently for all the 8 states. Similarly, the $\gamma_k(s_{k-1}, s_k)$ for the 16 possible state transitions can be computed concurrently. Hence, 8 or 16 thread can be assigned per thread block for the parallel computation of $\alpha$, $\beta$ and $\gamma$. If 16 threads are assigned per thread block, 8 of these threads would be idle in the computation of $\alpha$ and $\beta$ as these computations exhibit only 8-way data parallelism. If 8 threads are assigned per thread block, two cycles would be required for the 8 threads to perform the $\gamma$ computation. The trade-offs associated in assigning 8 or 16 threads are discussed in Lee *et al.* (2010). In the current implementation, 8 threads have been chosen for each thread block. The sub-block level parallelism allows the decoding of all the sub-blocks concurrently and hence one thread block is assigned to each sub-block. So, the number of thread blocks with which the kernel is launched is equal to the the number of sub-blocks $P$ into which the original code block has been divided into. Therefore, the half-decoder kernel is launched with 8 threads and $P$ thread blocks with each thread block corresponding to one sub-block.

## 5.2 Forward and Backward Traversals

In each half-decoder, the decoding proceeds by a forward traversal followed by a backward traversal through the length of the sub-block for all the sub-blocks. During the forward traversal the $\alpha$ metrics are computed and stored, since they are required for the computation of extrinsic LLRs done during the backward traversal. During the backward traversal, the $\beta$ metrics and extrinsic LLRs are computed. The computation of $\alpha$, $\beta$ and extrinsic LLRs all require the $\gamma$ 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. Storing the $\gamma$ metrics for all the 16 possible state transition for the entire length of the sub-block would make $16w$ number of floats to be stored in the shared memory. Such high usage of shared memory per each thread block would drastically reduce the number of thread blocks that can be truly run in parallel on a SMP. Storing the $\gamma$ metrics in the global memory instead of the shared memory is not an option, because, the latency associated with a fetch from global memory is significantly more than the time of computation of the the $\gamma$ metric on the fly from the operands present in the shared memory. Hence, in the current implementation, the $\gamma$ metrics are not stored and fetched but are computed each time they are required.

### 5.2.1 Forward Traversal

In the forward traversal, $\alpha_k(s)$ for each trellis stage $k$ is computed for the length of the sub-block. The computation of $\alpha_k(s)$ is given by Eq. (2.7). 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 such transition corresponds to the input bit $u_k$ being '0' and the other corresponds to the input bit $u_k$ being '1'. Hence, $\alpha_k(s)$ can be evaluated as:

$$a_0 = \alpha_{k-1}(s^0_{k-1}) + \gamma_k(s^0_{k-1}, s_k) \tag{5.1}$$

$$a_1 = \alpha_{k-1}(s^1_{k-1}) + \gamma_k(s^1_{k-1}, s_k) \tag{5.2}$$

$$\alpha_k(s_k) = max^*(a_0, a_1) \tag{5.3}$$

where $u_k = 0$ causes the state transition $s^0_{k-1} \rightarrow s_k$ and $u_k = 1$ causes the state transition $s^1_{k-1} \rightarrow s_k$. The computation of $\gamma_k(s^0_{k-1}, s_k)$ and $\gamma_k(s^1_{k-1}, s_k)$ as given by Eq. (2.6) requires the knowledge of the parity bits $x^{p0}_k$ and $x^{p1}_k$ associated with the state transitions $s^0_{k-1} \rightarrow s_k$ and $s^1_{k-1} \rightarrow s_k$ respectively.

A kernel is launched with 8 threads and at each trellis stage, one thread evaluates the $\alpha_k(s)$ for one state. Let the states be numbered as $0, 1, 2 \ldots 7$. For the 8 threads, let a thread with thread id $i$ evaluate the $\alpha$ metric for the state $i$. To compute the $\alpha_k(i)$ for a state $i$, the thread needs to know the states $s^0_\alpha$ and $s^1_\alpha$ from which there is a state

transition possible to the state $i$. In addition, the parity bits $x_\alpha^{p0}$ and $x_\alpha^{p1}$ associated with the state transitions $s_\alpha^0 \rightarrow i$ and $s_\alpha^1 \rightarrow i$ needs to be known. Input bit $u_k = 0$ causes the state transition from $s_\alpha^0 \rightarrow i$ and generates the parity bit $x_\alpha^{p0}$, whereas $u_k = 1$ causes the state transition from $s_\alpha^1 \rightarrow i$ and generates the parity bit $x_\alpha^{p1}$. The Table. 5.1 summarizes the operands needed for $\alpha$ computation.

Table 5.1: Operands for $\alpha_k$ computation

| | u=0 | | u=1 | |
|---|---|---|---|---|
| Thread id(i) | $s_\alpha^0$ | $x_\alpha^{p0}$ | $s_\alpha^1$ | $x_\alpha^{p1}$ |
| 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 |

Each thread uses $s_\alpha^0$ and $s_\alpha^1$ indices to fetch the appropriate $\alpha_{k-1}$ values. It uses the $x_\alpha^{p0}$ and $x_\alpha^{p1}$ values in the computation of the $\gamma$ metric for these transitions. The pseudo code for the $\alpha$ 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**

### 5.2.2 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 $\beta$ computation, thereby, removing the need to store the $\beta$ values for

the entire length of the sub-block in memory. The computation of $\beta_k(s)$ is given by Eq. (2.8). 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 such transition corresponds to the input bit $u_k$ being '0' and the other corresponds to the input bit $u_k$ being '1'. Hence, $\beta_k(s)$ can be evaluated as:

$$b_0 = \beta_{k+1}(s^0_{k+1}) + \gamma_{k+1}(s_k, s^0_{k+1}) \tag{5.4}$$

$$b_1 = \beta_{k+1}(s^1_{k+1}) + \gamma_{k+1}(s_k, s^1_{k+1}) \tag{5.5}$$

$$\beta_k(s_k) = max^*(b_0, b_1) \tag{5.6}$$

where $u_k = 0$ causes the state transition $s_k \rightarrow s^0_{k+1}$ and $u_k = 1$ causes the state transition $s_k \rightarrow s^1_{k+1}$. The computation of $\gamma_{k+1}(s_k, s^0_{k+1})$ and $\gamma_{k+1}(s_k, s^1_{k+1})$ as given by Eq. (2.6) requires the knowledge of the parity bits $x^{p0}_k$ and $x^{p1}_k$ associated with the state transitions $s_k \rightarrow s^0_{k+1}$ and $_k \rightarrow s^1_{k+1}$ respectively.

A kernel is launched with 8 threads and at each trellis stage, one thread evaluates the $\beta_k(s)$ for one state. Let the states be numbered as $0, 1, 2 \ldots 7$. For the 8 threads, let a thread with thread id $i$ evaluate the $\beta$ metric for the state $i$. To compute the $\beta_k(i)$ for a state $i$, the thread needs to know the states $s^0_\beta$ and $s^1_\beta$ to which there is a state transition possible from the state $i$. In addition, the parity bits $x^{p0}_\beta$ and $x^{p1}_\beta$ associated with the state transitions $i \rightarrow s^0_\beta$ and $i \rightarrow s^1_\beta$ needs to be known. Input bit $u_k = 0$ causes the state transition from $i \rightarrow s^0_\beta$ and generates the parity bit $x^{p0}_\beta$, whereas $u_k = 1$ causes the state transition from $i \rightarrow s^1_\beta$ and generates the parity bit $x^{p1}_\beta$. The Table. 5.2 summarizes the operands needed for $\beta$ computation.

Each thread uses $s^0_\beta$ and $s^1_\beta$ indices to fetch the appropriate $\beta_{k+1}$ values. It uses the $x^{p0}_\beta$ and $x^{p1}_\beta$ values in the computation of the $\gamma$ metric for these transitions.

At each trellis stage, after the computation of the $\beta$ metric, the extrinsic LLR is computed. The immediate computation of extrinsic LLR removes the need to store the $\beta$ metrics for the entire length of the sub-block. The state LLRs $\Lambda_0$ and $\Lambda_1$ for all the 8 states, as given by Eq. (2.9) and Eq. (2.10), can be computed concurrently with each thread computing the state LLRs for one state. However, the computation of the extrin-

Table 5.2: Operands for $\beta_k$ computation

| | u=0 | | u=1 | |
|---|---|---|---|---|
| Thread id(i) | $s_\beta^0$ | $x_\beta^{p1}$ | $s_\beta^1$ | $x_\beta^{p2}$ |
| 0 | 0 | 0 | 4 | 1 |
| 1 | 4 | 0 | 0 | 1 |
| 2 | 5 | 1 | 1 | 0 |
| 3 | 1 | 1 | 5 | 0 |
| 4 | 2 | 1 | 6 | 0 |
| 5 | 6 | 1 | 2 | 0 |
| 6 | 7 | 0 | 3 | 1 |
| 7 | 3 | 0 | 7 | 1 |

sic LLR from the state LLRs, as given by the equation 2.11 requires the computation of the $max^*$ function over all the the 8 states. Since, the $max^*$ of $\Lambda(s_k|\, u_k = 0)$ and $\Lambda(s_k|\, u_k = 1)$ over all states needs to be computed serially, there exists no state level parallelism in this computation. If the $max^*$ computation is done serially using just one thread, the remaining 7 threads would be left idle. To induce parallelism into this computation and so as to utilize all the 8 threads the following method is adopted. The absence of state level parallelism in the computation of extrinsic LLR computation can be compensated by utilizing the existing stage level parallelism i.e. the extrinsic LLRs for 8 consecutive stages of the trellis can be computed concurrently by the 8 available threads. However, this would require storing the state LLR values $\Lambda(s_k|\, u_k = 0)$ and $\Lambda(s_k|\, u_k = 1)$ for all the 8 states over the length of 8 trellis stages. The pseudo code for the $\beta$ computation and the extrinsic LLR computation is given in the Algorithm 2

## 5.3 Memory Allocation

Typical execution of a program on the GPU follows the following pattern. First, the data on which the computations are to be performed on the GPU is transferred from the host memory to the device memory. The computations are then performed by the GPU on the data in the device memory. The results to be communicated back are then transferred from the device memory to the host memory.

The memory architecture of the GPU differs from that of the CPU in several as-

**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**
    $k \leftarrow 8l + j$
    $\beta^0 \leftarrow \beta(s^0_\beta |\, u_k = 0) + \gamma^0$
    $\beta^1 \leftarrow \beta(s^1_\beta |\, u_k = 1) + \gamma^1$
    $\beta_{hold}(i) \leftarrow max^*(\beta^0, \beta^1)$
    SYNC
    $\beta(i) \leftarrow \beta_{hold}(i)$
    SYNC
    $\gamma^0 \leftarrow 0.5 \times ((L_c y^s_k + L^e_k)(-1) + L_c y^p_k (x^{p0}_\beta |\, u_k = 0))$
    $\gamma^1 \leftarrow 0.5 \times (L_c y^s_k + L^e_k + L_c y^p_k (x^{p1}_\beta |\, u_k = 1))$
    $\Lambda_0(8i + j) \leftarrow \gamma^0 + \beta(s^0_\beta |\, u_k = 0)) + \alpha_k(i)$
    $\Lambda_1(8i + j) \leftarrow \gamma^1 + \beta(s^1_\beta |\, u_k = 1)) + \alpha_k(i)$
  **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**
  $L_e(k) = (L_{e1} - L_{e0}) - L_c y^s_k - L_a(k)$
**end for**

pects. First, there are no multiple levels of hardware managed cache in the GPU as is present on the CPU. On traditional CPUs, caches help in circumventing the delay associated with data fetches and stores from the slower DRAM. The absence of such hardware managed cache readily exposes the GPU to unavoidable delays when data is to be fetched from or stored to the global memory. The shared memory present on the device can be treated as being the equivalent of a cache both because of it having more than a couple of orders of magnitude lesser latency and being extremely small in size compared to the global memory. However, the shared memory, unlike the typical cache in the CPU, is software managed and needs to be completely controlled by the programmer.

Secondly, the CPU architecture presents only one kind of memory to the programmer whereas the GPU architecture presents different kinds of device memories to the programmer, each with a different size and a different latency. As is to be expected, the fastest memory is the smallest in size and the largest memory is the slowest. Primarily, there are four types of memories are presented to the programmer - Shared memory,

Constant memory, Global memory and Texture memory.

The sizes of each of these memories and their relative latencies directly impact the mapping of any algorithm on to the GPU. Hence, deciding on which variables to store in which kind of memory becomes an important part of the mapping process. Since, the shared memory is more than a couple of orders of magnitude faster than the global memory and extremely small in size, a judicious and efficient use of shared memory is necessary. Deciding on how much of shared memory to use per each thread block and which data to store in it becomes a crucial part of the mapping of a algorithm. The constant memory is the next fastest memory and is a better choice than global memory for storing data that remains constant during a kernel execution. All those values which remain constant during a kernel execution and which cannot be fitted into the shared memory can be as a next best measure stored in the constant memory. Even though the size of the constant memory is 4 times the size of the shared memory, its still much smaller than the global memory and hence not all values that remain constant during a kernel execution can be stored here. All the data that needs to be exchanged among different kernel launches needs to be compulsorily stored in the global memory as the scope of the shared memory is restricted to a single kernel launch. This data can be stored in the shared memory during computations and then transferred back to the global memory before the termination of the kernel, as the need may be.

Transfer of data between the CPU and the GPU is very slow and should be minimized as much as possible, even if it involves performing computations on the GPU that do not exhibit any data parallelism. Since, the decoding is to be performed on the GPU, first, the channel values are transferred from the host memory to the device memory. After the completion of the decoding on the GPU, the decoded bits are the transferred back from device to the host. There is no other data transfer between the host and device other than these two compulsory transfers.

### 5.3.1 Global Memory Allocation

The global memory is present in abundance on the device as far as the turbo decoding algorithm is concerned. Hence, practically, all the inputs, the intermediate data and the

outputs can all be directly stored in the global memory and accessed from there. However, since global memory is the slowest available memory on the device, throughput of such a mapping would be extremely less and could even be lesser than a implementation done purely on the CPU.

Storing data in the global memory is the only way of exchanging data between different kernel launches. The extrinsic LLRs values 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 needs to be necessarily stored in the global memory part of the device memory. In addition, the guarding mechanisms described in the sub-sections (4.3.1) and (4.3.3) present the need to initialize the $\alpha$ and $\beta$ metrics at the ends of a sub-block or the training window respectively with the corresponding $\alpha$ and $\beta$ metrics from the previous iteration for each half-decoder. Since, the decoding by each half-decoder happens through a separate kernel launch in every iteration, a copy of these $\alpha$ and $\beta$ metrics also needs to be stored in the global memory.

Global memory can also be used for storing the $\alpha$, $\beta$ and $\gamma$ metrics of each sub-block, but as is argued later, the speed of execution of such a implementation would be very less. Whenever possible, its best to avoid the global memory for storing any intermediate computed values as the latency of stores and fetches from global memory is very high. Often, because of the limited size of the shared memory, it is not possible to fit all the intermediate data in it and hence sometimes the need to store some intermediate data in the global memory becomes unavoidable. In such cases, sometimes, recomputing the intermediate values instead of storing and fetching the computed values from the global memory could be better. In fact, as is explained in the ensuing sub-section (5.3.2), its better to recompute $\gamma$ values rather than to compute them once, store in the global memory and fetch from there.

### 5.3.2 Shared Memory Allocation

The shared memory is almost as fast as the registers and is the fastest available memory that the programmer can directly control. Effective use of the shared memory goes a long way in determining the speed-up that can be achieved using a GPU. The size

of the available shared memory in the Nvidia GeForce generation of GPUs is 16KB per SMP. At the time of kernel definition, shared memory is allocated for each thread block. However, often not all of the 16KB of shared memory is allocated to a thread block. The number of thread blocks that can be executed in parallel on a SMP is determined by the amount of shared memory allocated to each thread block. Since, the shared memory present on a SMP is physically assigned to each thread block running on it, the cumulative sum of the shared memory used by all the thread blocks running concurrently on a SMP should be less than 16KB, the total available shared memory on a SMP. As would be explained in detail in sub-section (5.4.1), the number of thread blocks running in parallel on a SMP directly impacts the speed of execution. Hence, the amount of shared memory allocated to a thread block should be kept as low as possible. The trade-off between the increased speed of execution of one thread block because of the use of extra shared memory and the impact on overall speed of execution because of the change in the number of thread blocks running in parallel on a SMP should always be analyzed before zeroing on the amount of shared memory to be allocated per thread block.

The decoding process in each half-decoder comprises of a forward traversal and a backward traversal. The forward traversal involves the computation of $\alpha_k$ and backward traversal involves the computation of $\beta_k$ and extrinsic LLRs for each stage of trellis of the sub-block. The $\alpha_k$, $\beta_k$ and LLR computation all require the computation of the branch metric $\gamma_k$. In addition, the LLR computation during the backward traversal requires all $\alpha_k$, $\beta_k$ and $\gamma_k$. Since, the $\beta_k$ values are anyway computed during the backward traversal itself, there is no need to store them. However, the $\alpha_k$ values computed during the forward traversal need to be stored for the entire length of the sub-block. The $\gamma_k$ values, which are used both in the forward traversal and the backward traversal can either be computed once, stored and fetched the next time or recomputed each time. The $\gamma_k$ computation requires the channel values and the extrinsic LLRs input to the half-decoder.

The computations wherein all the fetches are from shared memory and all the writes are to the shared memory occur much faster than those wherein one or more of the fetches or writes involve the global memory. This is because of the latency of the global

memory being more than 2 orders of magnitude greater than that of the shared memory. Hence, a typical strategy in optimizing for speed on a GPU, whenever possible, is to fetch all the required data from the global memory on to the shared memory at once and perform the computations on the data present in the shared memory. The results of the computation can then all be at once transferred back from the shared memory to the global memory. This is in sharp contrast to involving fetches from and writes to the global memory during each and every computation.

The $\gamma_k$ computation requires channel values and extrinsic LLRs. Hence, these are first fetched from the global memory on to the shared memory. For a sub-block size of $w$, this would require storing $3w$ floats in the shared memory. The extrinsic LLR computation requires the $\alpha_k$ values computed during the forward traversal. Hence, the $\alpha_k$ values for the entire size of the sub-block need to be stored in the shared memory. Since, each $\alpha_k$ requires 8 floats for the 8 states in the trellis, the total shared memory requirement for storing all the $\alpha_k$'s of the entire sub-block is $8w$. These two constitute the bulk of the shared memory allocated for each block. In addition, $\beta_k$ computation requires $\beta_{k-1}$ and hence 8 floats are required for $\beta$. The need to compute all the 8 $\beta_{k-1}$ in parallel makes another 8 floats necessary for $\beta$ storage.

Different amounts of shared memory is required for implementing each of the guarding mechanisms presented in 4.3. For a training window of length $g$ on either sides, instead of fetching $3w$ floats corresponding to the channel values and extrinsic LLRs, $3(w+2g)$ floats need to fetched to the stored memory. The previous value initialization guarding mechanism requires another 16 floats each for $\alpha$ and $\beta$ initializations. Hence, the previous value initialization mechanism requires an additional 32 floats, the double sided training window of length $g$ on each side requires an additional $6g$ floats and the double sided training window with previous value initialization requires additional $6g + 32$ floats to be stored in the shared memory for the purpose of guarding.

In the extrinsic LLR computation given by Eq. (2.11), 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 $\Lambda(s_k|\, u_b = 1)$. This is done to do away with the series evaluation of $max^*_{s_k \in K}(\Lambda(s_k|\, u_b = 1))$ and $max^*_{s_k \in K}(\Lambda(s_k|\, u_b = 0))$. This requires an additional storage of 128 floats in the shared memory.

Shared memory has not been used for storing the computed $\gamma$ metrics. Storing the gamma metrics for the entire length of the sub-block would necessitate $16w$ floats to be stored in the shared memory. Such prohibitively large usage of shared memory drastically reduces the number of thread blocks being run concurrently nullifying the benefit of not requiring to compute the $\gamma$ metrics again.

### 5.3.3 Constant Memory Allocation

The constant memory is ideal for all storing those values which remain constant during the execution of a kernel. The channel values and the extrinsic LLRs all remain constant during the execution of a kernel and hence, these can be potentially stored in the constant memory. But the code block size of 6144 and the size of the available constant memory restricts such a possibility. Since, anyway all the channel values and extrinsic LLRs corresponding to each sub-block are fetched into shared memory before performing any computations the benefit of storing them in constant memory would be restricted only to reducing the time to fetch them into the shared memory. Hence, the constant memory has not been used to store any of the channel values or extrinsic LLRs. These values are stored in the global memory and are fetched from there into the shared memory of each thread block.

The constant memory has been used only for storing the operands mentioned in Table. 5.1 and Table. 5.2 required for the computation of $\alpha$ and $\beta$ metrics respectively. These operands are few in number and are used frequently by each thread and hence need to be stored in the memory with the least latency. Hence, they could alternatively be stored in the shared memory as well. The impact on speed for the storage of these operands in constant memory or shared memory has been found to be negligible. Since the shared memory is already a very scarce resource, the constant memory has been chosen for storing these constant operands.

The constant memory has also been used to store the indices required for deinterleaving. The kernel launch of the second half-decoder involves interleaving and deinterleaving. The indices for the interleaving can be directly computed using the the Eq. (2.17). The indices for deinterleaving are stored in memory and a simple fetch from

an array of indices is used to perform deinterleaving. Since, these indices are constants, they can be stored in the constant memory. Storing these indices in constant memory would require storing 6144 integers in the constant memory. The available size of the constant memory is sufficient to store them and hence the indices for deinterleaving have been stored in the constant memory.

## 5.4 Occupancy and Performance Considerations

Parallel computing is all about performance as otherwise a serial program could be written instead. Both NVIDIA Corporation (2010*a*) and Harris (2007) describe several optimizations strategies for getting the maximum throughput out of a GPU. To a large extent, the mapping of the algorithm determines the speed of its execution. However, further speed up can be achieved by fine tuning for optimum occupancy for the GPU in hand.

### 5.4.1 Occupancy

Thread instructions are executed sequentially in CUDA, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy Kirk *et al.* (2010). Occupancy NVIDIA Corporation (2010*a*) is a metric related to the number of active warps on a multiprocessor is therefore is important in determining how effectively the hardware is kept busy. The following factors together determine the occupancy on a GPU.

- Number of threads per each thread block
- Register usage per each thread
- Shared memory usage per each thread block

The occupancy can be calculated using the CUDA Occupancy calculator tool provided by Nvidia.

The number of threads per thread block is fixed by the mapping of the algorithm. The number of parallel sub-blocks $P$ into which the code has been split into and the

mapping of the algorithm together determine the shared memory usage per each thread block. The number of registers used per each thread is determined by the compiler and depends primarily on the way of implementation of the algorithm. Increase in the number of registers per thread contributes positively to speed of execution of one thread but impacts negatively on the number of thread blocks that can be run in parallel on a SMP. The compiler analyzes this trade-off and determines the optimum number of registers to be used per one thread. This registers usage per thread impacts the occupancy. Though, it is possible to force the compiler to use lesser number of registers to allow for increased occupancy, the overall speed of execution isn't positively affected by this because of the increased number of instructions. The increase in number of instructions is caused due to the usage of lesser number or registers. The number of threads per thread block is fixed by the mapping. For a device with compute capability of 1.1, eight threads per thread block allows a maximum occupancy of 0.33. Since, the number of threads per thread block is fixed by the inherent data parallelism in the algorithm, the occupancy limit set by it is the maximum that can be aimed for. In addition, the shared memory and register usage constraints further reduce the occupancy.

### 5.4.2 Shared Memory Bank Conflicts

Shared memory is divided into equally-sized memory modules, called banks, all which can be accessed simultaneously. There are 16 banks present in the shared memory. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized NVIDIA Corporation (2010*a*). The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflict. An array is stored in the shared memory with 16 banks in such a way that the 16 consecutive elements of the array are stored in different banks of the shared memory NVIDIA Corporation (2010*b*).

In the computations of all $\alpha$, $\beta$ and $\gamma$, the 8 threads present make a read or write request to 8 consecutive elements of on array. Since, all these elements would be stored in different memory banks of the shared memory, all the 8 memory requests would be serviced simultaneously. There are no shared memory bank conflicts present in the the computation of $\alpha$, $\beta$ and $\gamma$ metrics.

However, shared memory bank conflicts exist during the computation of extrinsic LLRs from $\Lambda_0$ and $\Lambda_1$. 8 elements of $\Lambda_0$ for each stage are stored in 8 consecutive addresses. Since there are 16 memory banks, elements $\Lambda_0$ and $\Lambda_1$ of stages $k$ and $k+2$ with the same index would be stored in the same memory bank. The set of threads 0, 2, 4, 8 and 1, 3, 5, 7 would access the same bank. This would lead to a 4-way memory bank conflict in the computation of the extrinsic LLRs since 4 threads access the same shared memory bank. A method to decrease the shared memory bank conflicts in the computation of extrinsic LLRs is presented in Wu *et al.* (2010). The method works well for the Tesla GPU used in Wu *et al.* (2010) but adversely affects the speed of execution on the target GPU in the current implementation, the Nvidia GeForce. This is because, though employing the above method reduces shared memory bank conflicts it increases the number of register used per each thread. The Tesla GPU of compute capability 1.3 have 16384 registers wheres the GeForce GPU has only 8192 registers. Because of having lesser number of registers, the benefit of reduced shared memory bank conflicts is nullified by the decrease in occupancy.

# CHAPTER 6

# BER PERFORMANCE AND THROUGHPUT

The BER performance and throughput of the designed turbo decoder depends on the following factors:

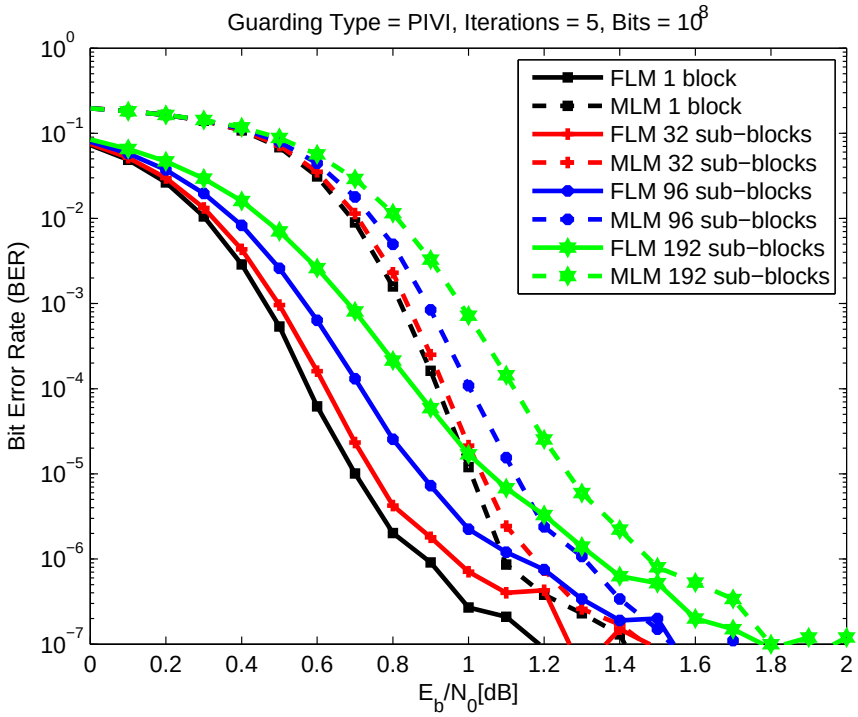
- The type of decoding algorithm used - Full Log-MAP or Max Log-MAP.
- The type of guarding mechanism used - PIVI, DST or PIVIDST, and the size of the training window in case of DST and PIVIDST guarding mechanisms.
- Number of parallel sub-blocks $P$ into which the code block has been divided into.

The effect of each of these factors on the BER performance and throughput has been presented in the following sections.

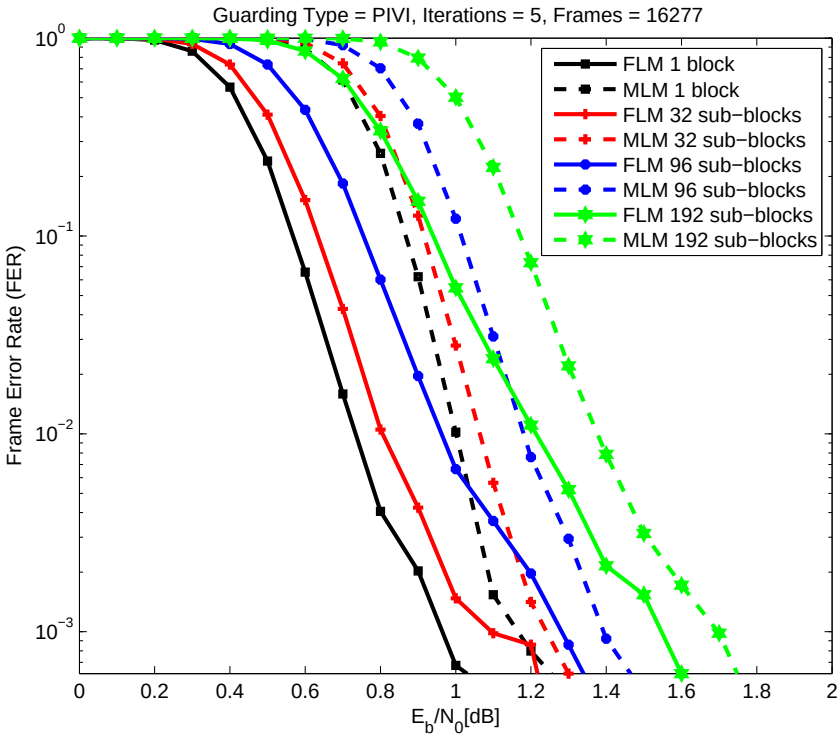
## 6.1 BER Performance

The dependence of BER performance on all the above factors has been presented in this section. Two of the above factors have been fixed and the dependence of the BER performance on the third factor has been studied for all the three factors. All the BER vs $E_b/N_0$ and FER vs $E_b/N_0$ have been obtained by performing simulations using $10^8$ bits or equivalently 16277 frames each of length 6144.

Sub-section 6.1.1 contrasts the BER and FER performance of the Max Log-MAP and Full Log-MAP algorithms. The effect of the type of guarding mechanism and the number of parallel sub-blocks on the BER and FER performance is similar for both these algorithms. Hence, the BER vs $E_b/N_0$ and FER vs $E_b/N_0$ plots to demonstrates the effects of these factors have been shown only for the Max Log-MAP algorithm. The corresponding plots for the Full Log-MAP algorithm are similar.




(a) BER vs $E_b/N_0$




(b) FER vs $E_b/N_0$

Figure 6.1: Contrasting BER and FER performance of Full Log-MAP and Max Log-MAP algorithms

### 6.1.1 Effect of using Max Log-MAP or Full Log-MAP Algorithm

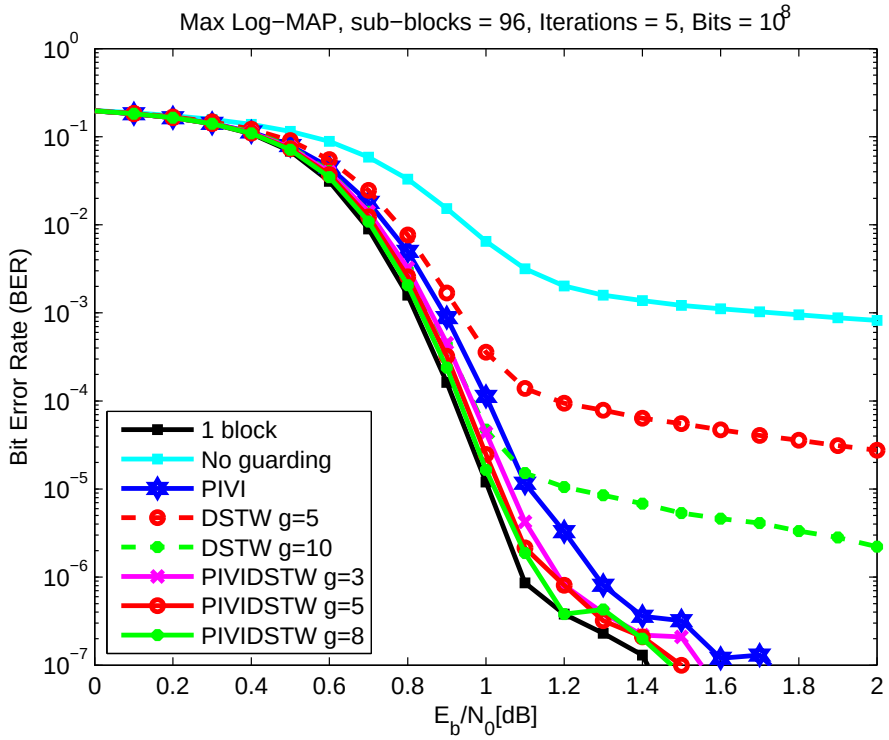
In the Full Log-MAP algorithm, the $max^*$ function is computed using 2.13 and in the Max Log-MAP algorithm, the $max^*$ function is computed using 2.14. The avoidance of logarithms and exponentials in the computation of $max^*$ function in the Max Log-MAP algorithm comes at the cost of reduced BER and FER performance. Fig. (6.1) illustrates the difference in BER and FER performance of Max-Log MAP and Full-Log MAP algorithms. It plots BER vs $E_b/N_0$ and FER vs $E_b/N_0$ for different number of parallel sub-blocks $P$. The guarding mechanism chosen is PIVI and the number of decoding iterations = 5.

As is evident from the BER vs $E_b/N_0$ and FER vs $E_b/N_0$ plots in Fig. (6.1), the Full Log-MAP algorithm gives a better BER and FER performance than the Max Log-MAP algorithm. This fact remains unchanged with the number of parallel sub-blocks.

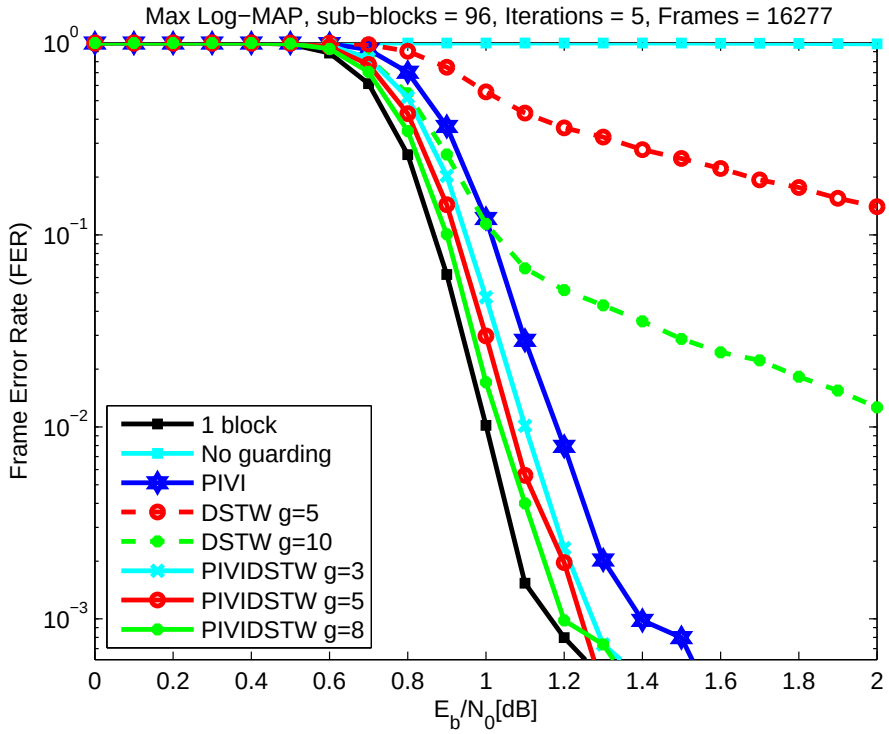
### 6.1.2 Effect of the type of Guarding Mechanism

Three types of guarding mechanisms, namely - Previous Iteration Value Initialization(PIVI), Double Sided Training Window(DSTW) and Previous Iteration Value Initialization with Double Sided Training Window(PIVIDSTW) have been discussed in section 4.3. The effect of each of these guarding mechanisms on the BER and FER performance is shown in the Fig. (6.2). The BER vs $E_b/N_0$ and FER vs $E_b/N_0$ plots in Fig. (6.2) have been obtained for the Max Log-MAP turbo decoding algorithm with number of parallel sub-blocks $P$ = 96 and number of decoding iterations = 5. The fact that some guarding mechanism is absolutely essential is demonstrated in the FER vs $E_b/N_0$ plots, wherein, the FER without any guarding is 1 even for values of $E_b/N_0$ as high as 2.

As the plots in Fig. (6.2) indicate, the DSTW fares worst among the three guarding mechanisms. Even for a window size $g$ = 10, this guarding mechanism fares much worse than the PIVI guarding mechanism. In addition, the BER and FER values do not drop to acceptable levels even for values of $E_b/N_0$ as high as 2. A larger size of the guard window might help, but for typical sizes of the sub-block that would add a huge




(a) BER vs $E_b/N_0$




(b) FER vs $E_b/N_0$

Figure 6.2: Effect of the type of guarding mechanism on BER and FER performance

computational overhead. Therefore, it is not worthwhile to use the DSTW as a possible guarding mechanism. Hence, the use of DSTW has a possible guarding mechanism has been dropped in the current implementation.

The PIVI guarding mechanism fares much better than the DSTW and can be used as a possible guarding mechanism. For the number of parallel sub-block $P$ = 96, where the size of each sub-block 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.

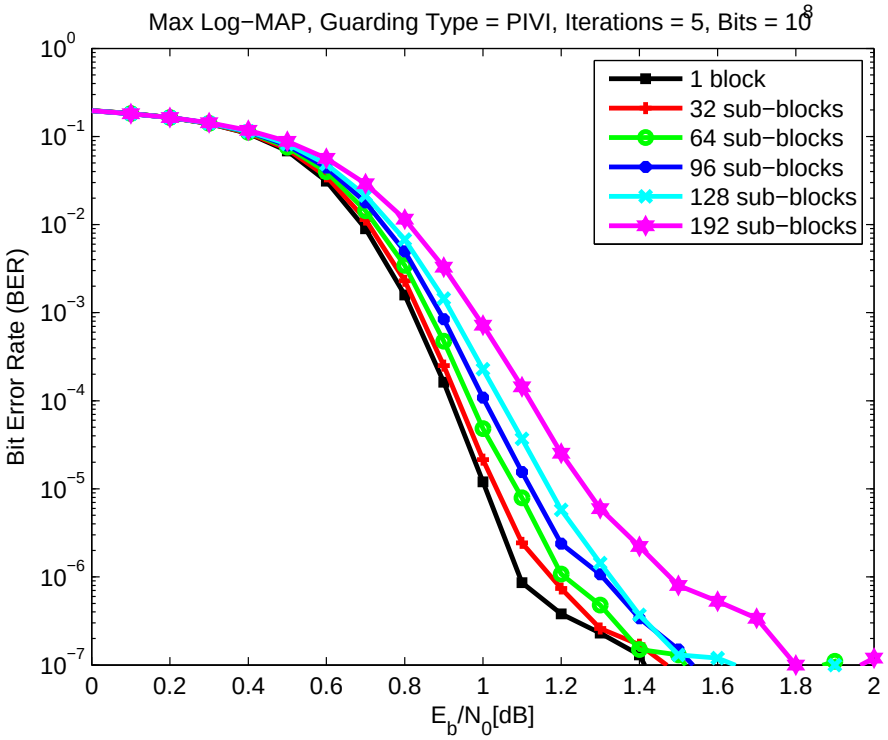
The PIVIDST guarding mechanism, which is a combination of both PIVI and DST, can be used to further improve the BER and FER performance. For the number of parallel sub-block $P$ = 96, where the size of each sub-block is only 64, the decoder with PIVIDST guarding mechanism, for a window size $g$ = 8, 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.

Hence, among the three guarding mechanisms, the DSTW guarding mechanism has been discarded. The PIVI guarding mechanism produces satisfactory BER and FER performance. The PIVIDSTW guarding mechanism has to potential to produce a BER and FER performance that is almost as good as that of the optimal case of a single code block.

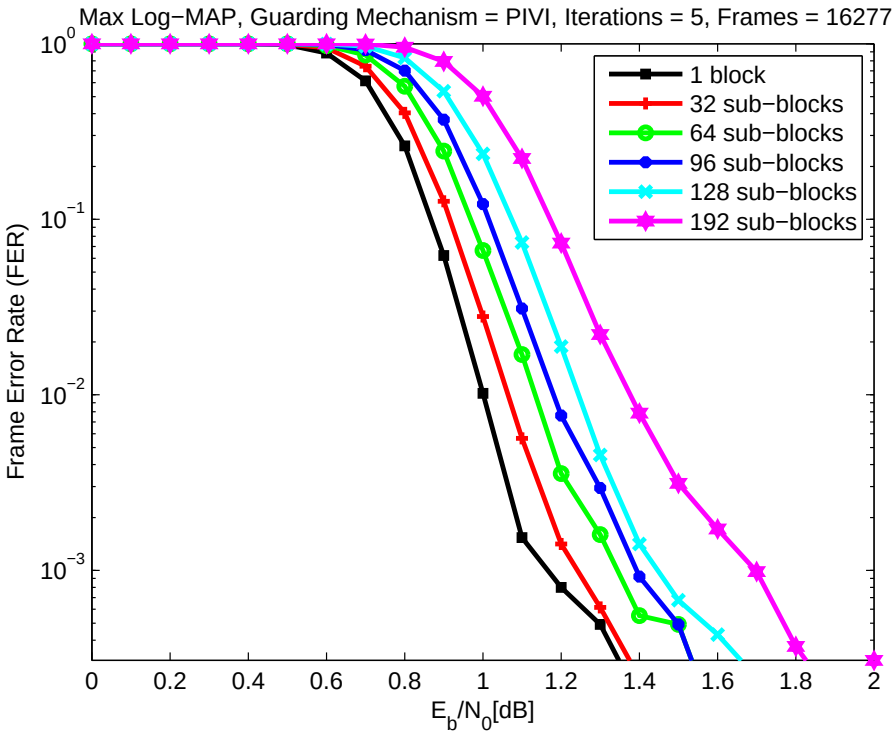
### 6.1.3 Effect of the Number of Parallel Sub-blocks

The effect of the number of parallel sub-blocks on BER and FER performance for Max Log-MAP algorithm and Full Log-MAP algorithm is as shown in Fig. (6.3) and Fig. (6.4) respectively. The guarding mechanism chosen is PIVI and the number of decoding iterations = 5.

As is to expected from theoretical discussion in chapter 4, the degradation in BER performance increases with the increase in the number of parallel sub-blocks. This degradation in BER performance is similar for both the Max Log-MAP and the Full Log-MAP decoding algorithms.

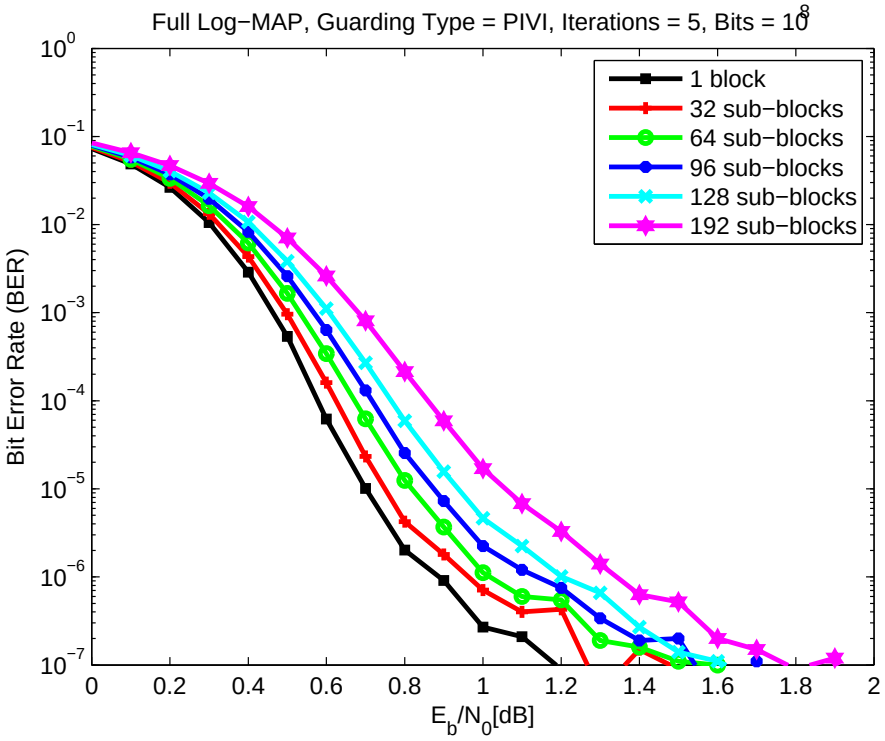



(a) BER vs $E_b/N_0$

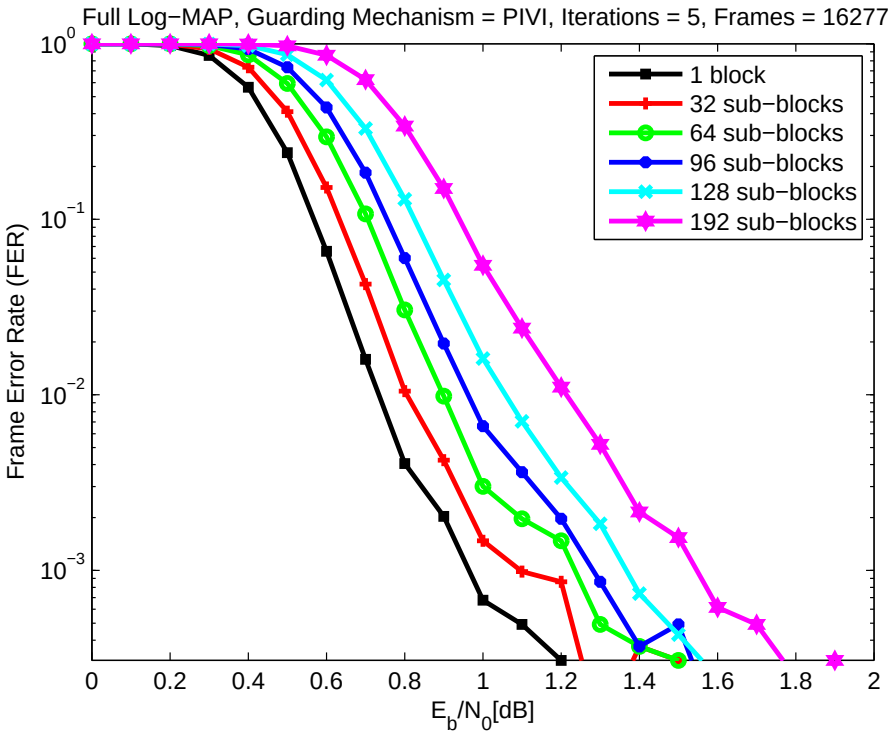



(b) FER vs $E_b/N_0$

Figure 6.3: Effect of the number of parallel sub-blocks $P$ on BER and FER performance for Max Log-MAP algorithm




(a) BER vs $E_b/N_0$




(b) FER vs $E_b/N_0$

Figure 6.4: Effect of the number of parallel sub-blocks $P$ on BER and FER performance for Full Log-MAP algorithm

As shown in Fig. (6.3) and Fig. (6.4), the BER and FER performance degradation increases with increase in the number of parallel sub-blocks $P$. However, the increase in performance degradation with the increase in the number of parallel sub-blocks $P$ is different for the two types of guarding mechanisms - PIVI and PIVIDSTW.

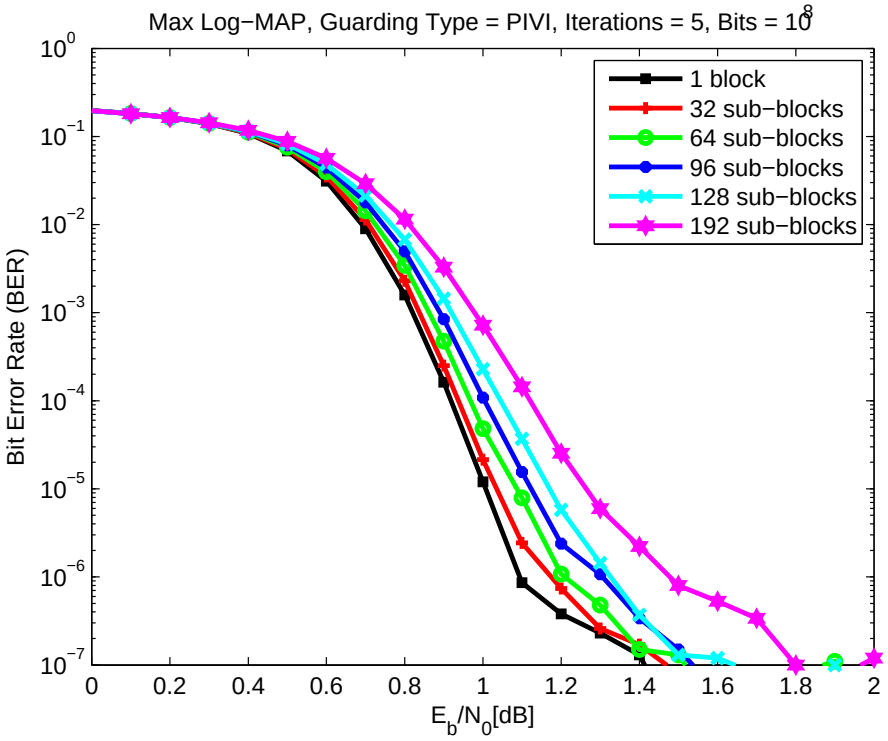
This fact is demonstrated in Fig. (6.5) and Fig. (6.6). Here, BER vs $E_b/N_0$ and FER vs $E_b/N_0$ plots have been obtained for different number of sub-blocks. In Fig. (6.5a) and Fig. (6.6a), the type of guarding mechanism used is PIVI. In Fig. (6.5b) and Fig. (6.6b), the type of guarding mechanism used is PIVIDST. The BER vs $E_b/N_0$ plots in Fig. (6.5) and FER vs $E_b/N_0$ plots in Fig. (6.6) illustrate that the performance degradation with the increase in the number of sub-blocks is less prominent for the PIVIDSTW guarding mechanism as compared to the PIVI guarding mechanism, even for a training window size as small as 5.

Hence, when the number of sub-blocks is large, the PIVIDSTW guarding mechanism functions significantly better than the PIVI guarding mechanism. Hence, to mitigate the degradation in BER and FER performance when a *large* number of sub-blocks are used in the decoder, the PIVIDSTW type of guarding mechanism is a much better choice than the PIVI.

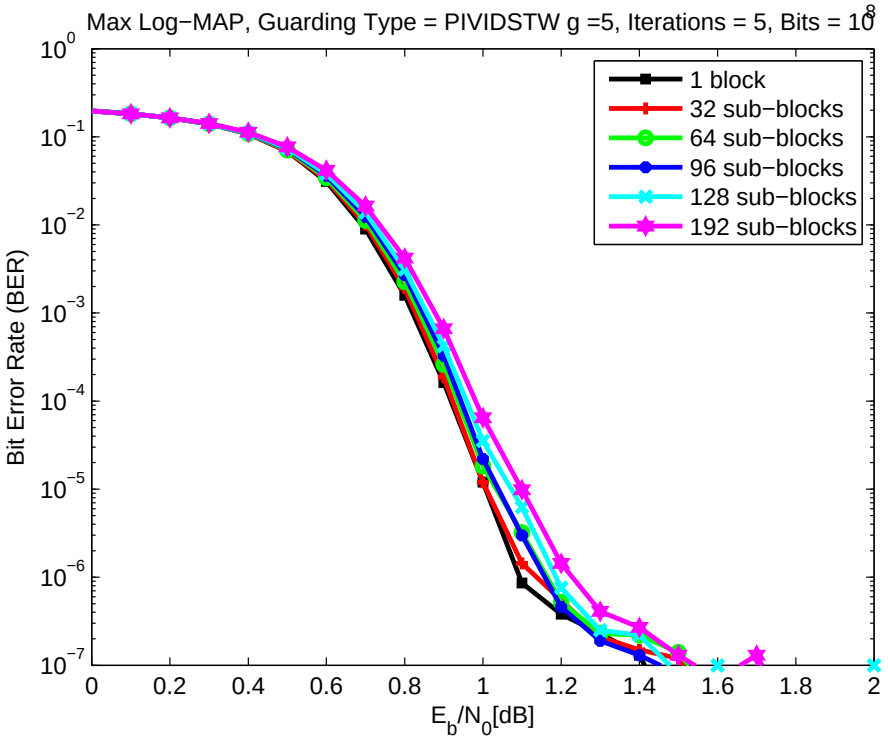
## 6.2 Throughput

The BER performance and throughput of the designed turbo decoder has been evaluated by testing it on a 64-bit Linux 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. This graphics card has 2 GPUs. Each GPU has 128 SPs running at 1.5GHz with 512MB of GDDR3 memory running at 1GHz. The compute capability of the GPU is 1.1 and it has 8192 registers and 16KB of shared memory per SMP. Each GPU on the graphics consists of 16 SMPs each of which has 8 SPs.

The speed of execution of the algorithm or the throughput is measured in terms of the number of bits decoded per second. The time measured in the calculation of

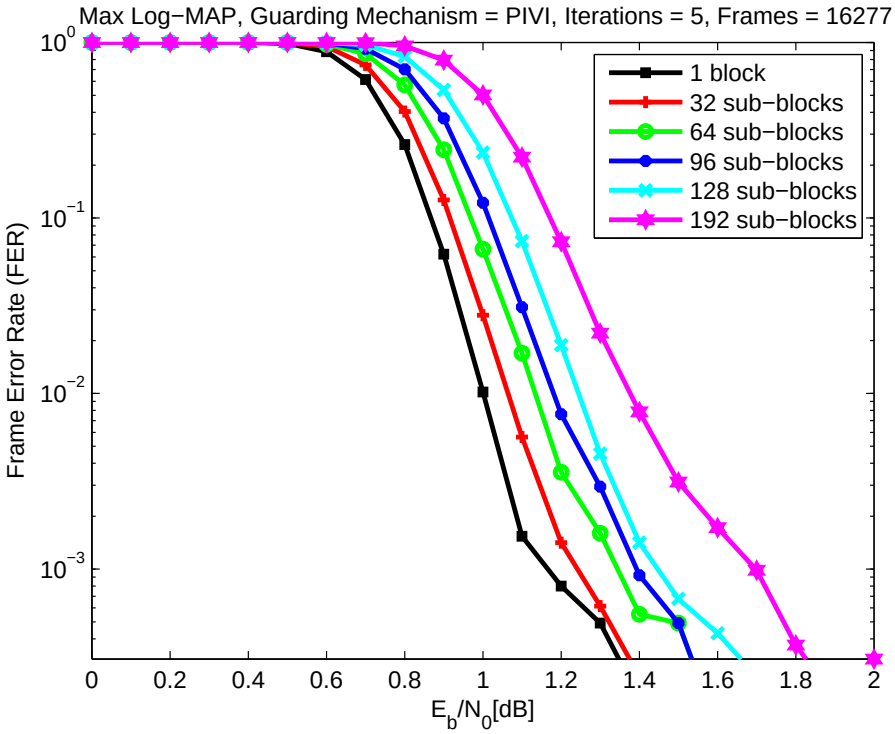



(a) BER vs $E_b/N_0$ for PIVI

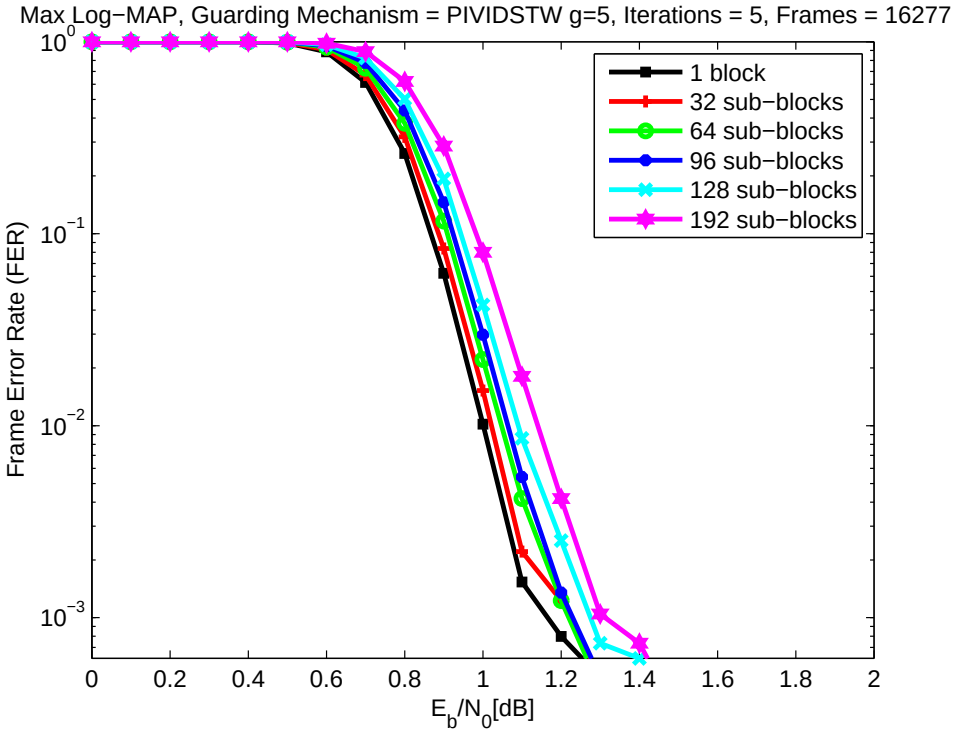



(b) BER vs $E_b/N_0$ for PIVIDSTW

Figure 6.5: Effect of the number of parallel sub-blocks $P$ on BER performance for the guarding mechanism PIVI




(a) FER vs $E_b/N_0$ for PIVI




(b) FER vs $E_b/N_0$ for PIVIDSTW

Figure 6.6: Effect of the number of parallel sub-blocks $P$ on FER performance for the guarding mechanism PIVIDSTW with g=5

throughput is the time interval between reading the channel output values from the DRAM of the host and writing the decoded bits to the DRAM of the host. This time includes the time to transfer the data from the host to device, perform decoding on the device and transfer the decoded bits back from device to host.

The throughput of the decoder would decrease with a increase in the computational complexity of the algorithm used. Since, the Full Log-MAP is computationally more complex than the Max Log-MAP algorithm, it is expected to have lesser throughput. Similarly, since the PIVIDSTW guarding mechanism involves more computations than the PIVI guarding mechanism, it is expected to have slightly lesser throughput. The increase in the number of parallel sub-blocks should ideally increase the throughput by the corresponding factor. However, such a thing would happen, only if, the increase in the number of sub-blocks increases the number of thread blocks being truly run in parallel in hardware. The increase in speed with the number of parallel sub-blocks saturates beyond a certain number of sub-blocks for the current implementation on the Nvidia GeForce 9800 GX2 GPU. The causes of the above behavior are explained in the sub-section 6.2.3

### 6.2.1 C Implementation on the CPU vs Parallel Implementation on the GPU

The parallel implementation of the turbo decoder on the GPU is significantly faster than a C implementation done purely on the CPU. The Table. 6.1 showcases the speed up achieved using the GPU over an implementation done purely on the CPU for both Max-Log-MAP algorithm and Full Log-MAP algorithm implementations. For demonstrating the speed up on the GPU, a parallel implementation with number of parallel sub-blocks $P = 96$ and PIVI guarding mechanism has been chosen. The C implementation has not been completely optimized for speed and is only an indicator of the order of the throughput achievable on the CPU.

The Table 6.1 shows that for Max Log-MAP turbo decoder with 5 iterations, the GPU implementation with 96 parallel sub-blocks is more than 25 times faster.

Table 6.1: Contrasting the decoder throughput of a implementation on the CPU with a parallel implementation on the GPU for $P = 96$

| | Decoder throughput (Kbps) | | | |
|---|---|---|---|---|
| | C Implementation on CPU | | Implementation on GPU | |
| Iterations | Max Log-MAP | Full Log-MAP | Max Log-MAP | Full Log-MAP |
| 1 | 362 | 96 | 4380 | 4050 |
| 2 | 185 | 49 | 3390 | 3010 |
| 3 | 124 | 33 | 2760 | 2400 |
| 4 | 94 | 24 | 2330 | 1990 |
| 5 | 76 | 19 | 2020 | 1700 |
| 6 | 63 | 16 | 1780 | 1490 |
| 7 | 54 | 14 | 1590 | 1320 |

### 6.2.2 Effect of using Max Log-MAP or Full Log-MAP Algorithm and the type of Guarding Mechanism

The throughput obtained for the Max Log-MAP algorithm and the Full Log-MAP algorithm for the different types of guarding mechanisms is as shown in the Table. 6.2

Table 6.2: Decoder throughput for Max Log-MAP and Full Log-MAP algorithms for the two guarding mechanisms PIVI and PIVIDSTW for $P = 96$

| | Decoder throughput (Mbps) | | | |
|---|---|---|---|---|
| | PIVI | | PIVIDSTW with g=5 | |
| Iterations | Max Log-MAP | Full Log-MAP | Max Log-MAP | Full Log-MAP |
| 1 | 4.38 | 4.05 | 4.30 | 3.99 |
| 2 | 3.39 | 3.01 | 3.20 | 2.84 |
| 3 | 2.76 | 2.40 | 2.58 | 2.24 |
| 4 | 2.33 | 1.99 | 2.17 | 1.85 |
| 5 | 2.02 | 1.70 | 1.86 | 1.58 |
| 6 | 1.78 | 1.49 | 1.63 | 1.37 |
| 7 | 1.59 | 1.32 | 1.45 | 1.21 |

As is to be expected, the throughout 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. In Table. 6.2, 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.

### 6.2.3 Effect of the Number of Parallel Sub-blocks

The throughout achieved by the designed turbo decoder for different number of sub-blocks $P$ is shown in Table. 6.3.

Table 6.3: Decoder throughput for different number of sub-blocks $P$

| | Decoder throughput (Mbps) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | P=32 | | P=64 | | P=96 | | P=128 | | P=192 | |
| Iterations | MLM | FLP | MLM | FLM | MLM | FLM | MLM | FLM | MLM | FLM |
| 1 | 2.95 | 2.56 | 3.92 | 3.54 | 4.38 | 4.05 | 4.57 | 4.27 | 4.70 | 4.39 |
| 2 | 1.93 | 1.60 | 2.88 | 2.48 | 3.39 | 3.01 | 3.68 | 3.32 | 3.77 | 3.38 |
| 3 | 1.44 | 1.17 | 2.26 | 1.91 | 2.76 | 2.40 | 3.08 | 2.72 | 3.07 | 2.68 |
| 4 | 1.14 | 0.92 | 1.87 | 1.55 | 2.33 | 1.99 | 2.65 | 2.30 | 2.64 | 2.28 |
| 5 | 0.95 | 0.76 | 1.59 | 1.30 | 2.02 | 1.70 | 2.33 | 2.00 | 2.35 | 2.01 |
| 6 | 0.81 | 0.65 | 1.38 | 1.13 | 1.78 | 1.49 | 2.07 | 1.76 | 2.08 | 1.77 |
| 7 | 0.71 | 0.56 | 1.22 | 0.99 | 1.59 | 1.32 | 1.87 | 1.58 | 1.89 | 1.58 |

The throughput appears to increase with a increase the number of sub-blocks initially and then saturate for higher number of sub-blocks. In other words, the increase in throughput is not proportional to the increase in the number of sub-blocks for higher number of sub-blocks. With increase in the number of parallel sub-blocks $P$, the size of each sub-block $w$ given by $w = 6144/P$ reduces. As a result, the shared memory requirement per each thread block reduces. As explained in Section 5.4, a decrease in the shared memory requirement increases the occupancy on the GPU, i.e. increases the number of thread blocks being truly run in parallel on a SMP,thereby increasing the throughput. Such a increase in occupancy with a decrease in the shared memory usage would occur if the shared memory is the constraining factor for occupancy. For large values of $w$, i.e small number of sub-blocks, this is in fact the case. This explains the increase in throughput with a increase in the number of sub-blocks initially. However, for higher number of sub-blocks, the shared memory usage is no longer the constraining factor for occupancy. The occupancy, at this low usage of shared memory

is constrained by the number of registers used per thread. Since, the number of registers used per thread is characteristic of the way of implementation of the algorithm and does not depend on the number of sub-blocks, beyond a stage, the occupancy and hence the throughput does not increase with a increase in the number of parallel sub-blocks.

An important factor to note here is that, this particular behavior is to a good extent characteristic of the GPU used. In fact such behavior would be absent if the total number of register available per SMP were to be larger on a different GPU. This is in fact the case for the next generation of the Nvidia GPUs. The Tesla generation of GPUs, with a compute capability of 1.3 have twice the number of registers as the GeForce generation of GPUs of compute capability 1.1. Hence, on a Tesla GPU, the saturation of increase in speed with number of sub-blocks would not occur at 192 sub-blocks.

# CHAPTER 7

# CONCLUSION AND FUTURE WORK

## 7.1 Conclusion

A 3GPP standard compliant turbo decoder has been implemented completely in software. Since, the implementation is done in software, it is completely configurable. The block length, trellis state structure of the encoder and the QPP interleaver can all be readily changed to support any other codes.

A throughput of 2Mbps has been achieved for a Max Log-MAP turbo decoder with 5 iterations. 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. With only a little computational overhead above the PIVI mechanism, the PIVIDSTW mechanism has been shown to be much better in mitigating the degradation in BER and FER performance than the PIVI mechanism, especially for large number of sub-blocks.

## 7.2 Future Work

The principles of the mapping of the algorithm can be easily extended to the newer architectures of the NVDIA GPUs. Hence, with minor modification, the decoder implementation could be readily optimized for the new architectures. Implementations on the newer GPUs, with larger number of parallel cores have the potential to achieve even higher throughput.

# REFERENCES


1. **Abrantes, S. A.** (2004). From BCJR to turbo decoding: MAP algorithms made easier. Technical report, University of Porto.

2. **Harris, M.** (2007). Optimizing Parallel Reduction in CUDA. *NVIDIA Developer Technology*.

3. **Hong, S.** and **H. Kim** (2009). An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. Technical report, Georgia Institute of Technology.

4. **Kirk, D.**, **W. Hwu**, and **W. Hwu**, *Programming massively parallel processors: a hands-on approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010. ISBN 9780123814722.

5. **Lee, D.**, **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*, CASES '10. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-903-9. URL http://doi.acm.org/10.1145/1878921.1878953.

6. **Marandian, M.**, **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*, volume 2. 2001.

7. **Muller, O.**, **A. Baghdadi**, and **M. Jezequel**, Exploring Parallel Processing Levels for Convolutional Turbo Decoding. *In Information and Communication Technologies, 2006. ICTTA '06. 2nd*, volume 2. 2006.

8. **NVIDIA Corporation** (2010*a*). *NVIDIA CUDA C Best Practices Guide Version 3.2*. URL http://developer.nvidia.com/cuda-downloads.

9. **NVIDIA Corporation** (2010*b*). *NVIDIA CUDA C Programming Guide Version 3.2*. URL http://developer.nvidia.com/cuda-downloads.

10. **Ryan, W. E.** (1997). A Turbo Code Tutorial. Technical report, New Mexico State University.

11. **Sanders, J.** and **E. Kandrot**, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010. ISBN 9780131387683.

12. **Sun, J.** and **O. Takeshita** (2005). Interleavers for turbo codes using permutation polynomials over integer rings. *Information Theory, IEEE Transactions on*, **51**(1), 101–119. ISSN 0018-9448.

13. **Wu, M.**, **Y. Sun**, and **J. Cavallaro**, Implementation of a 3GPP LTE turbo decoder accelerator on GPU. *In Signal Processing Systems (SIPS), 2010 IEEE Workshop on*. 2010. ISSN 1520-6130.