THE REPUBLIC OF TURKEY

BAHÇEŞEHİR UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING

HIGH-SPEED BIDIRECTIONAL FANO ALGORITHM IMPLEMENTATION

Master Thesis

ÖZGÜR ATEŞ

İSTANBUL, January 2014

THE REPUBLIC OF TURKEY

BAHÇEŞEHİR UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

COMPUTER ENGINEERING

HIGH-SPEED BIDIRECTIONAL FANO ALGORITHM IMPLEMENTATION

Master Thesis

ÖZGÜR ATEŞ

Supervisor: PROF. DR. TAŞKIN KOÇAK

İSTANBUL, January 2014

THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING MASTER PROGRAM

Name of the thesis: High-Speed Bidirectional Fano Algorithm Implementation
Name/Last Name of the Student: Özgür ATEŞ
Date of Thesis Defence: January, 2014

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Assoc. Prof. Tunç BOZBURA
Director
Signature

I certify that this thesis meets all the requirements as a thesis for the degree of Master of Arts.

Asst. Prof. Dr. Tarkan AYDIN
Program Coordinator
Signature

This is to certify that we have read this thesis and we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Arts.

Examining Committee Members: Signature

Thesis Supervisor
Prof. Dr. Taşkın KOÇAK: ----------

Committee Member
Asst. Prof. Dr. Tevfik AYTEKİN: ----------

Committee Member
Asst. Prof. Dr. Devrim ÜNAY: ----------

ACKNOWLEDGEMENTS

My acknowledgments start with my thesis supervisor Prof. Dr. Taskin KOCAK for his support and advices in writing this report. I would like to also thank Yelda ERDOGAN for her precious advices in formatting a report.

I cannot thank enough Assoc. Dr. Sezer GOREN – UGURDAG and Assoc. Dr. Fatih H. UGURDAG for making me appreciate the electrical engineering part of computer engineering throughout their numerous lectures at Bahcesehir University.

Speaking of Bahcesehir University, I should also thank Enver YUCEL and Pervin KAPLAN for believing in me and making me resume studies even after my staying away from university for over two years. Also, I would like to thank Aysegul CELIKER for her efforts in my thesis' well continuation.

This thesis would not have been completed if I had not received constant warmth and support from my once students and who are now my friends. Especially, my thanks go to Gokce GUNDUZ, Gozde Ayse TATAROGLU and Meltem CINAR.

I would also like to address my endless gratitude to my close friends Can KANSIN, Erdem ERZURUM, Jbid ARSENYAN and Kathleen SHAPPELL who endured my countless rebellions, yet persevered helping me go through and finish this thesis.

Special thanks should be credited to the person that helped me the most. This help ranged from moral support to technical advising while writing this thesis. That person is no other than Dr. Leyla KACAR – ATES, my mother.

Istanbul, January 2014 Ozgur ATES

ABSTRACT

HIGH-SPEED BIDIRECTIONAL FANO ALGORITHM IMPLEMENTATION

Ates, Ozgur

Computer Engineering
Thesis Advisor: Prof. Dr. Taskin KOCAK

January 2014, 95 Pages

This thesis is about the implementation of Bidirectional Fano algorithm (BFA) and Unidirectional Fano algorithm (UFA or conventional Fano algorithm) in C and CUDA.

BFA is derived from the Fano algorithm used in high speed convolutional code decoding. It is a simultaneous forward and backward codeword search decoder. The high throughput demanded by the latest wireless digital interfaces like WirelessHD may benefit from a parallel computing specific baseband processing unit that runs BFA at a high speed on CUDA.

In this thesis, BFA reached a throughput of 4.4Gbps in CUDA with a single thread per codeword. Its iteration is characterised by one thread decoding first forward then backward. Another BFA decoding at 3.1Gbps was performed using dual threads per codeword. One thread is a forward decoder while the other is a backward decoder. Finally, on the same board, GTX650, UFA was found to have 5.0Gbps of throughput. It was concluded that additional memory transactions and check conditions were the source of throughput loss of BFA in comparison to UFA. However, BER, NoI and TpI analysis gave improved results for BFA, meaning that it decoded more efficiently than UFA.

In order to obtain such throughputs, several optimization measures were taken such as the use of look-up tables instead of metric, output bit, state calculations. Another technique was to use an indexed circular queue to hold the previous eight steps in memory instead of using a conventional array.

This thesis proposes BFA implementation in CUDA application as a complement to the research done by Xu et al. in 2009, in which simulation was conducted in MATLAB. Later, it has been implemented in FPGA at 100Mbps, and here we are aiming a throughput in several Gbps.

The research done by Xu and Koçak (2010) proposed the use of LUT for checking merge conditions. LUT were used in this thesis, too, but this time to pre-calculate next metric, state, output bit calculations, and this contribution was applied to both UFA and BFA.

Keywords: Bidirectional Fano Algorithm, CUDA, High Throughput Decoding

ÖZET

YÜKSEK SÜRATLİ VE ÇİFT YÖNLÜ FANO ALGORİTMA UYGULAMASI

Ateş, Özgür

Bilgisayar Mühendisliği
Tez Danışmanı: Prof. Dr. Taşkın KOÇAK

Ocak 2014, 95 Sayfa

Bu tezin konusu, yüksek süratli ve çift yönlü Fano algoritmasının C ve CUDA dillerinde uygulanmasıdır.

Kısa adı BFA olan çift yönlü Fano algoritması, yüksek süratte evrişimli kod çözmede kullanılır. Bu, simültane bir biçimde ileri ve geri kod çözme algoritmasıdır.

WirelessHD gibi modern kablosuz dijital iletişim araçları yüksek hız gerektirir. İşte bu yüksek BFA hızı için, CUDA ile paralel işlem yapabilecek bir ana bant işlem birimi kullanılabilir.

Bu tezde, CUDA ile BFA'nın 4.4Gbps'lik bir hıza erişmesi sağlanıyor. Her kod ayrı bir thread kullanılarak çözülüyor. Her adımda bir thread başlangıçtan sona doğru kod çözüyor, ardından da sondan başa doğru. 3.1Gbps hızındaki diğer bir BFA kod çözümünde ise her kod için çift thread kullanılıyor. Burada da bir thread başlangıçtan sona doğru, öbür thread sondan başlangıca doğru kod çözüyor. UFA, aynı GTX650 cihazında 5.0Gbps hızına ulaşabiliyor. BFA'nın yavaşlığının, UFA'ya daha fazla hafıza kullanmasından ve ek kontrol mekanizmalarından kaynaklandığını düşünüyoruz. Ancak, BER, NoI ve TpI analizlerinden BFA için daha iyi sonuçlar elde edilebiliyor. Bundan da anlaşılan şu ki BFA daha yavaş olmasına karşı UFA'dan daha verimli kod çözüyor.

Bu hızlara ulaşabilmek için metrik, çıkan değer v.b. hesap yerine taramalı tablo kullanımı gibi bazı optimizasyonlar yapıldı. Bir başka yöntem, bir önceki sekiz adımı hafızada tutmak için klasik bir dizi yerine indeksli çember sıra kullanmak oldu.

Xu et al. (2009) tarafından yapılan BFA araştırmasının simülasyonu MATLAB'da yapılmıştı. Başka bir simülasyon da FPGA'da 100Mbps hızında olmuştu. Bu tezde, CUDA ile birkaç Gbps hızında BFA araştırması simülasyon yapmayı hedefliyoruz.

Xu ve Koçak (2010) tarafından yapılan araştırmada ileri ve geri kod çözücülerinin birleşmesinin kontrol edilmesinde taramalı tablo kullanılmasını önermişti. Bu tezde de metrik, çıkan değer v.b. hesap yerine taramalı tablo okuma yapılmış; üstelik bu hem UFA hem BFA için uygulanmıştır.

Anahtar kelimeler: Çift Yönlü Fano Algoritması, CUDA, Süratli Kod Çözme

CONTENT

APPENDICES

TABLES

FIGURES

ABBREVIATIONS

AWGN	:	Added White Gaussian Noise
API	:	Application Programming Interface
BER	:	Bit Error Rate
BFA	:	Bidirectional Fano Algorithm
CPU	:	Central Processing Unit
CUDA	:	Compute Unified Device Architecture
FFT	:	Fast Fourier Transform
GPU	:	Graphics Processing Unit
Gbps	:	Gigabit per second
GPGPU	:	General Purpose Graphics Processing Unit
HMPP	:	Hybrid Multi-core Parallel Programming
HPC	:	High Performance Computing
LUT	:	Look-Up Table
NoI	:	Number of Iteration
TpI	:	Throughput per Iteration
SNR	:	Signal Noise Ratio
UFA	:	Unidirectional Fano Algorithm
VA	:	Viterbi Algorithm
VLSI	:	Very Large Scale Integration

1. INTRODUCTION

This chapter is intends to provide a top-bottom view of the thesis and give the motivation behind this research.

1.1 SCOPE OF THE THESIS

The goal of this thesis is to propose a high-speed decoding solution using bi-directional Fano algorithm (Xu, 2009). The original research is the basis for this thesis and the aim will be to reach WirelessHD complying specifications (WirelessHD, 2010) which require 4 Gbps at version 1.0.

This scope of this thesis is the implementation of the BFA. Local optimizations where needed are also part of the scope but bringing structural changes to the algorithm is not part of the scope.

The main motivation of the thesis is to propose a real-life implement of BFA for high-speed wireless networks.

This research aims to propose a parallel solution to the initial problem. Therefore, a GPGPU implementation using CUDA, massive parallel programming platform from NVIDIA, will be proposed. Hardware implementation through ASIC, FPGA designing is not part of the research.

1.2 GOAL OF THE THESIS

This thesis aims to propose a high speed implementation for one of the baseband processes, namely signal decoding. For this purpose, BFA will be taken as the decoding mechanism and CUDA will be used as a parallel processing environment to optimize the computation efficiency. Also, an intermediary code in C++ will be used to accurately measure the efficiency gain between a theoretical implementation, a regular CPU based implementation and a parallel processing GPGPU based implementation.

1.3 OUTLINE OF THESIS REPORT

In Chapter 2, a literature survey covering the basics of coding and decoding mechanisms at the baseband processing level will be reviewed. Also a quick introduction to WirelessHD will be done. In Chapter 3, previous algorithms using Fano Algorithms will be described. In Chapter 4, a C++ implementation of the BFA will be shown. Chapter 5 will introduce CUDA. A CUDA implementation of the BFA will be provided in Chapter 6. Optimizations will be the subject of Chapter 7, mainly focused on parallel programming in CUDA for the BFA. In Chapter 8, the results of those algorithms will be analyzed. In Chapter 9, the conclusion and future research perspective will be discussed.

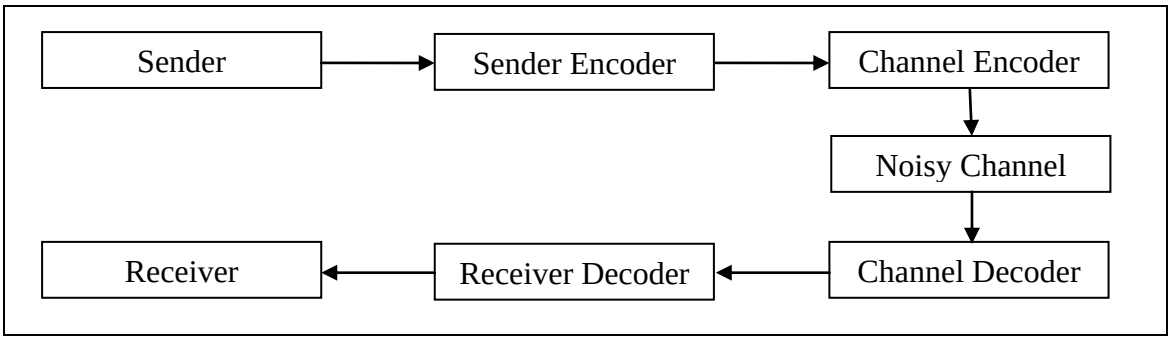
2. LITERATURE RESEARCH

2.1 CONVOLUTIONAL CODING

Convolutional coding is a subject that can be said to have started with P. Elias (1955). Inspired by Shannon's mathematical material in communication (1948) and Hamming's paper on error-correcting code (1950), Elias introduced the concept of convolutional coding. This mechanism is used in signal transmission over noisy channel, well known in telecommunication. Decoding a signal is one of the most time consuming tasks done at the baseband along with the FFT.

The concept of transmitting over a noisy channel is as follows:

Figure 2.1: Transmission over a noisy channel



The convolutional coding is the right most, bottom part of the diagram. Channel encoder applies convolutional coding to the data. In the other end, the channel decoder decodes the noisy signal which may be subjected to noise (AWGN), phase shifting, and interfaces.

In convolutional coding m-bit input decoded into n-bit symbols.

$$Code\,rate\,R = \frac{m}{n}\ (For\ n \geq m) \qquad \textbf{(2.1)}$$

Decoding is dependent on the last symbol *L* where *L* is the code length.

Encoders can be seen as FIR filters. An example of the mechanism of an encoding would be as follows.

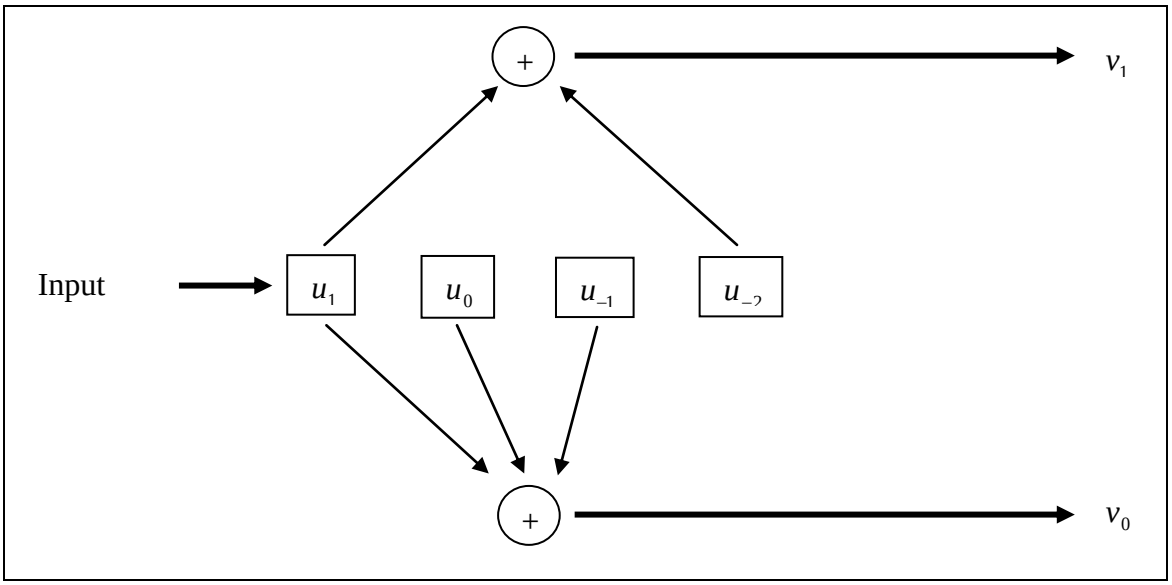
An imaginary encoder has a binary code (n,k,m) such that $n=2, k=1, m=3$ where n is the number of output bits, k is the number of output bits, m is the number of memory registers.

Considering a generator sequence:

$$\begin{cases} g_0\,(1001) \\ g_1\,(1110) \end{cases} \qquad \textbf{(2.2)}$$

The convolutional structure would be then:

Figure 2.2: Convolution structure of a (2,1,3) code



Initially all the registers are set to 0. At each time, a new input bit is sent to the encoder and n encoded v_1 and v_0 bits are generated from it. When no input is left, 0 is used by default. The encoding process ends when all the encoder's memory registers are set back to initial state, 0.

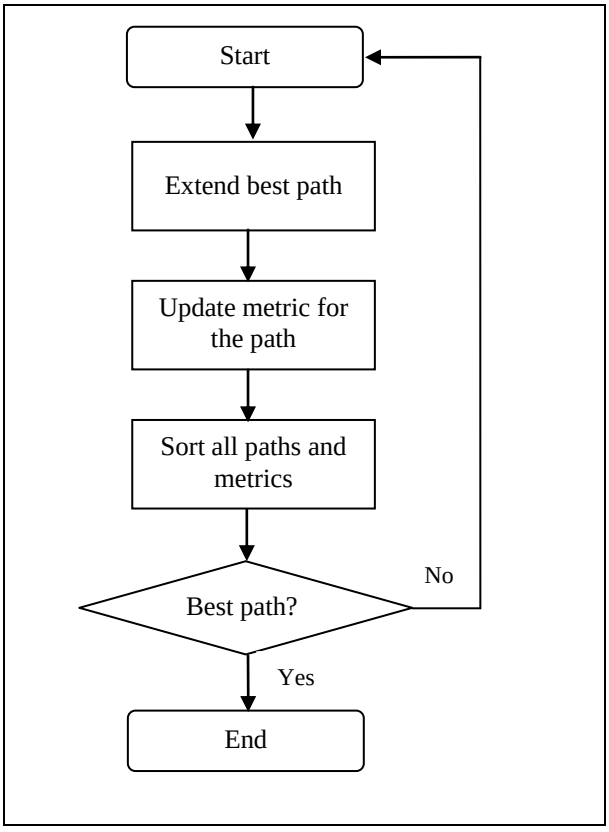
2.2 STACK ALGORITHM

The stack algorithm or ZJ algorithm is a tree searching algorithm. It is characterized by its relative simplicity compared to Fano algorithm, another sequential decoding algorithm, or compared to Viterbi which can be visualized as a search for the shortest path through a trellis diagram. The aim of both stack and Fano algorithms is to propose an efficient search that may not be the maximum likelihood result. However; the resulting path is negligibly different from the maximum likelihood result but with the benefit of not traversing all the possibilities before giving that answer.

Zigangirov (1966) and independently by Jelinek (1969), they proposed their implementations of the stack algorithm. The idea behind their result was to set a stack with the Fano metric 0 to the starting node, then calculate its successors, delete the top path from the stack and insert the new path, sort decreasingly the metrics. If the top path ends with the final node we have a result for the search. If not, looping back to the calculation above is the next step.

The algorithm's working flow can be summarized as:

Figure 2.3: Flow chart of stack algorithm



The Fano metric is calculated as below:

$$\begin{cases} \log_2 2p - R & y_j \neq x_{ij} \\ \log_2 (1-p) - R & y_j = x_{ij} \end{cases} \quad \textbf{(2.3)}$$

(Anderson, 1983)

Where,

p is the crossover probability of a bit,

R is the code rate,

x is the partial code word,

y is the received code word,

i is the starting bit number,

j is the ending bit number.

A variant (D. Haccoun, 1975) of the ZJ algorithm was brought, characterized by the extension of the most likely paths instead of the systematic top node. The algorithm was proven to add some decoding effort and also additional memory requirement. However; the result was a reduced average decoding effort.

2.3 FANO ALGORITHM

The Fano algorithm (1963) is a tree searching algorithm characterized by a good performance with low average complexity at reasonably high SNR. The algorithm is a sequential decoding algorithm. Fano does not claim to propose the maximum likelihood decoding schema as in the Viterbi algorithm (1967). However, for only slight decoding accuracy deterioration, the proposed algorithm obtains a nearly optimal decoding performance with significantly less decoding effort.

The tree is composed of branches and nodes. Each branch of the tree has a weight that is also called branch metric. Paths are sequences of branches. The weight of a path is simply the sum of all of the metrics of its branches.

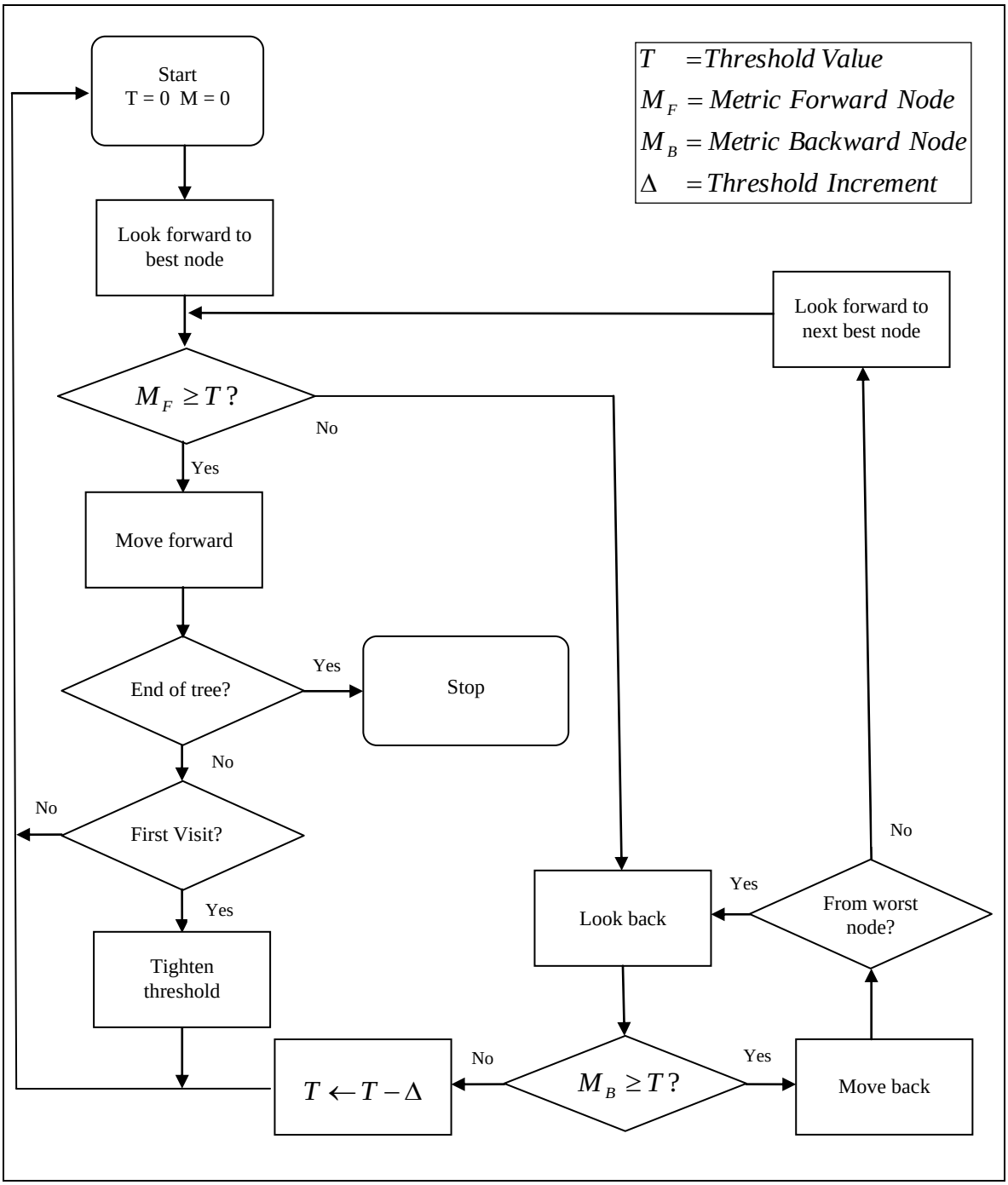
This is a search algorithm in the sense that we are searching for the minimum weight for a path from the root to a leaf. The search is sequential and is done from one node to its neighbouring nodes and so on. The algorithm is a depth-first tree-searching algorithm. Searching goes on as long as the current node is not a leaf node.

Fano algorithm moves forward if a branch metric is above a certain threshold T. On the other hand, it moves backward if there is no possible forward move and searches for other branch candidates. In case neither is possible, T is tightened.

The threshold T has a calculation that updated dependently to the branch metrics statistics. T is initially selected as a multiple of delta.

The workflow of the algorithm is as follows:

Figure 2.4: Flow chart of Fano algorithm



2.4 VITERBI ALGORITHM

Viterbi introduced the convolutional decoding known as the Viterbi algorithm (1967). This is an algorithm which finds the shortest path for a weighted graph (Omura, 1969).

It is the most used decoding schema in convolutional code because it fits well hardware implementation. The method used by the algorithm is proven to obtain the MLD (Forney, 1973) of a transmission with inter-symbol interferences. However, the algorithm is notorious for having an exponential complexity growth in correlation with a long constraint length.

The algorithm finds the shortest path along the trellis diagram. The term trellis corresponds to the structure obtained by expanding the encoder's state diagram over time. L + m + 1 time units or levels are produces from this procedure (John B. Anderson, 1983).

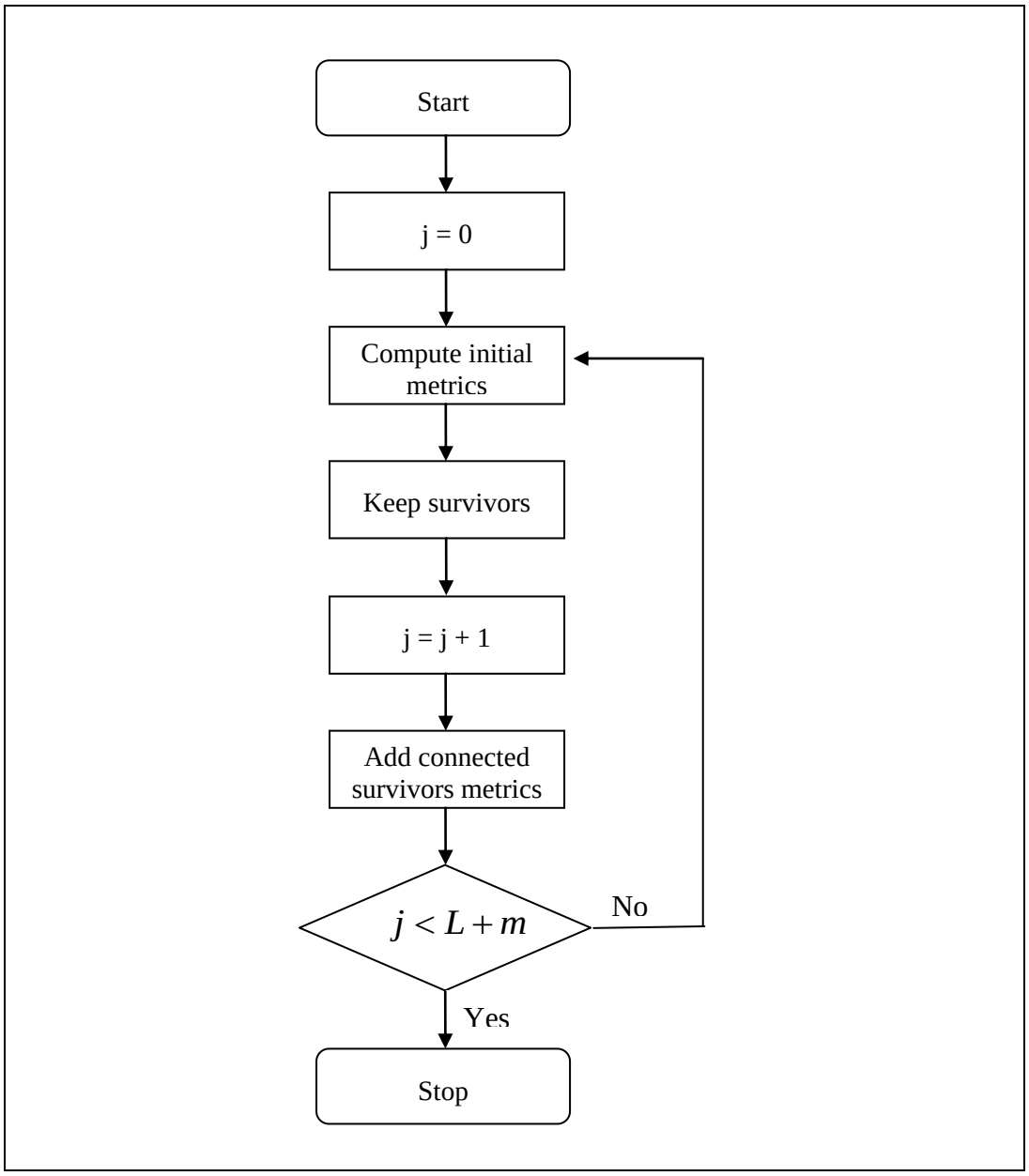
Viterbi algorithm starts by computing a metric for every path. Each path is continued to be processed. If multiple paths converge to the same node, only the one with the higher metric survives and other converging paths are dropped.

A bit stream of N bits can have 2^{kL} distinct words where k is the number of input bits and L is the constraint length. The algorithm limits the selection of the possible output by using maximum-likelihood comparisons.

The comparison type can lead to having a hard decision decoding or soft decision decoding algorithm. In a hard decision decoding, the comparison is determined using Hamming distance as the metric. On the other hand, a soft decision decoding decoder uses Euclidean distance as the measure.

The workflow of the algorithm is as follows:

Figure 2.5: Flow chart of Viterbi algorithm



2.5 WIRELESS HD

A review of some of the latest high-speed solutions is as follows.

Table 2-1: High-speed networking solutions

Technology	Speed	Area of use
HDMI (v1.3)	10Gbps (HDMI, 2012)	HD TV & WLAN
WHDI (v1.0)	4.5Gbps (WHDI, 2012)	HD TV
WiGig (v1.1)	7Gbps (Alliance, 2010)	HD TV & WLAN
WirelessHD (v1.0)	4Gbps (WirelessHD, 2008)	HD TV & WLAN
WirelessHD (v2.0)	10-28Gbps (WirelessHD, 2010)	HD TV & WLAN
WirelessUSB(v1.1)	480Mbps (LSI Corporation, Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NECCorporation, ST-Ericsson, Samsung Electronics Co., Ltd., 2010)	Video streaming, Scanners, MP3 Players, etc.

WirelessHD is a protocol primarily designed for wireless video networking. It operates at the unlicensed 60 GHz band of the frequency spectrum. At v1.0, WirelessHD supported 4 Gbps, at the latest v2.0, it provides up to 28 Gbps data streaming for a typical 10 m range. At such speed, the company can naturally claim to allow streaming high definition audio\video (HD A\V) as uncompressed 1080p A/V transmission or uncompressed 7.1 surround sound audio support. Notice that in this thesis we will focus on the first version of WirelessHD.

A device using WirelessHD supports several different features. We can cite examples as:

i. File transfer data using OBject EXchange (OBEX)

ii. Smart antenna technology for non-line of site (NLOS)

iii. 3D video support

iv. User customized data privacy

v. 1 Gbps connectivity for low power portable devices

3. PREVIOUS WORKS

3.1 UNIDIRECTIONAL FANO ALGORITHM

3.1.1 Typical Implementation

A typical UFA implementation is characterized by the avoidance of stack as present in the ZJ algorithm or heuristic path search as in the Viterbi case. However, this results in a higher decoding effort for low SNR.

Two concepts directly influence such implementation.

One of the crucial areas is the variable delta. As said earlier, it is used while tightening the threshold to a lower T. When delta is too small, the result is the many unnecessary decoding steps from it. On the other hand, a delta value too high would also be ineffective since the tightening would result in an increase of new wrong paths.

Another concept to pay attention to is the rapid initial column distance growth (Li, 1991).

The overall design would require the following memory allocation (Ran Xu, 2009).

i. A visit record is needed to keep track of each bit and in which state it was visited; thus, its size would be the number of states multiplied by the size of the frames. This information is used in the selection of the next node.

ii. State history is used to keep track of the state of each frame bit for back tracing. Therefore, this requires a memory space as much as the size of the frame.

iii. The metric history is also kept as the state history and again it needs the allocation of frame size memory space.

iv. Flag history indicates which frame bit needs to look forward to the next best node operation.

The final result of the memory requirement for each frame calculation would be calculated as the following:

Memory requirement = (number of states + 3) x size of the frame. In this memory assumption we ignore the negligible temporary values as delta, T, M, etc.

3.1.2 Hardware Implementation

Viterbi algorithm is said to be more practical for a hardware implementation but here we will present how Fano algorithm had been designed as VLSI circuits.

We can mention an earlier high speed implementation (Forney & Bower, 1971) where the design was claimed to give 5Mb/s.

For hardware implementation, normalization of data takes an important role. It is found that such practice would give better performance, make smaller circuits and prevent hardware overflow (Beerel, 2010). The normalization may concern the usage of smaller variables that directly affect bandwidth or the usage of variables such that current node's metric is always equal to zero, which makes it unnecessary to add/subtract while making metric checks.

Unlike typical or software implementations, memory limitations are heavy. For example, we may have to limit the number of nodes we can back track.

Also, in the hardware implementation specialized blocks will be used. In their publications Beerel et. al (2010) proposed the usage threshold adjust units, branch metric unit, skip-ahead units (an interesting unit that tries to guess branch bits and calculates ahead the possible decision).

Several other improvements are also proposed. For example for high SNR channel, error-free regions will be encountered most of the time. Therefore, optimizations in those regions for such events are recommended.

3.2 BIDIRECTIONAL FANO ALGORITHM

3.2.1 Original Implementation

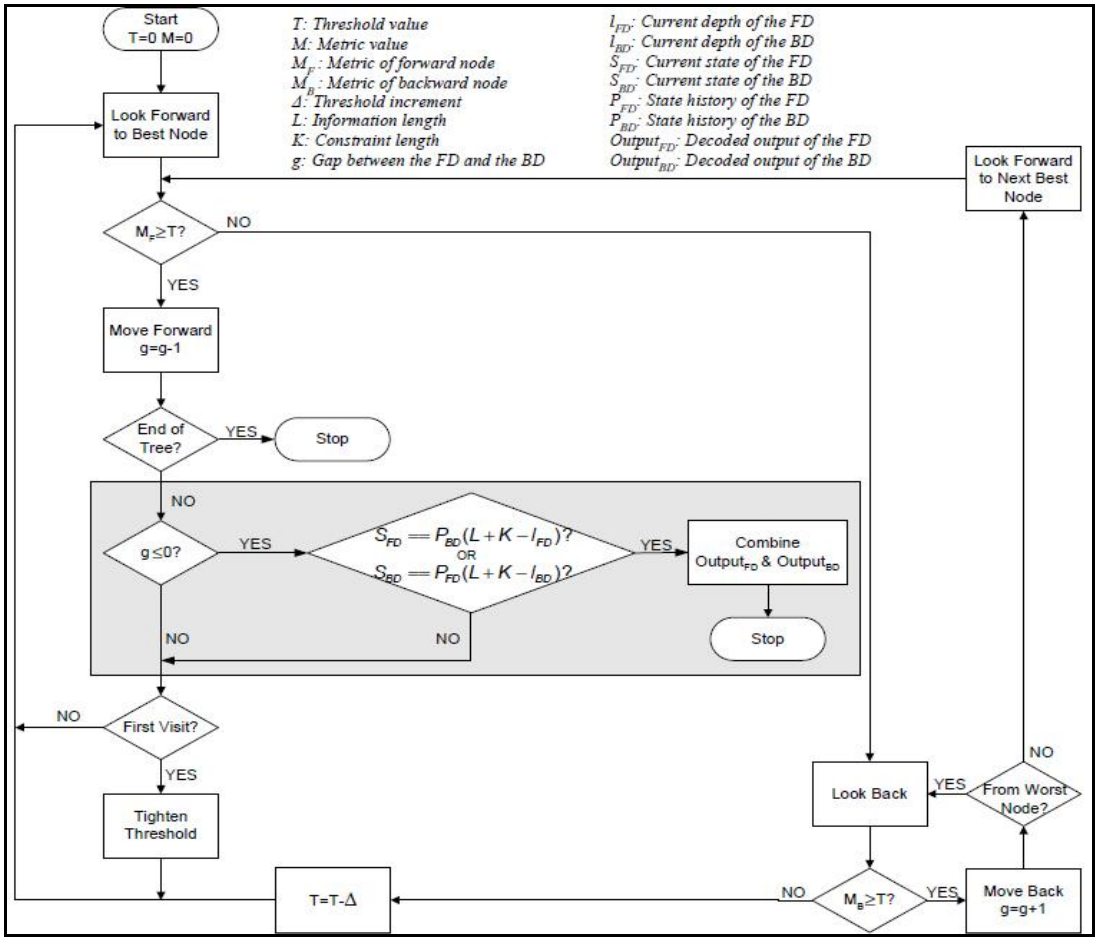
The algorithm was proposed in Bidirectional Fano Algorithm Throughput Sequential Decoding (Ran Xu, 2009) paper. The algorithm was found to have at least 50 percent speed up compared to unidirectional Fano algorithm with a better decoding throughput at low SNR.

The research was originally inspired by the idea of bidirectional search from Forney (1967) and the possibility of using a forward decoder and backward decoder starting at the opposite ends of the sequence and trying to reach a merge condition with the other decoder. The decoding ends when either a merge condition, corresponding to a successful decoding, or a decoder reaching the other end is detected, which corresponds to a failed decoding.

The simulation result of the paper (Ran Xu, 2009) showed throughput up to 300 percent improvement for 200 bits of information with $E_b / N_0 = 3\,dB$ and 1 merged state. Another note from the result is that as the number of merged states increase, BFA performances are close to UFA and VA. However, the research also points out the fact that a rigorous merging check still obtains throughput improvement.

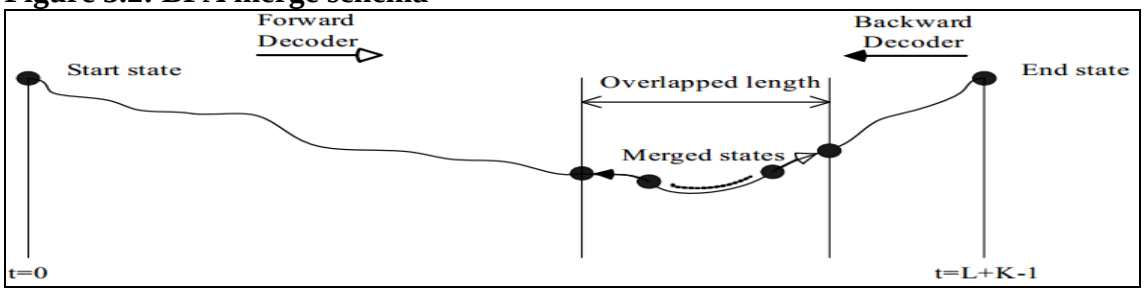
The conclusion of the paper was that the average decoding delay was reduced compared to both UFA and VA. Moreover, the throughput gain was even more distinct for low SNR values.

Figure 3.1: BFA flowchart



Source Xu, 2009

Figure 3.2: BFA merge schema



Source Xu, 2009

When consecutive merged states are detected, their number is called overlapped length. The algorithm typically has smaller length for quicker output but higher BER and, of course, expecting more states to merge is slower but gives a more accurate decoding.

3.2.2 Parallel Implementation

A high throughput parallel Fano algorithm was proposed (Xu et al. 2011). The paper proposes eight decoders to be run simultaneously.

Authors designed three possible solutions:

The first solution is a synchronous design with each decoder fed with one codeword and resulting in a synchronous result reading. However, even though the decoding time is upper bounded, the decoding is not constant since it is dependent on the input and SNR level. Hence, variability in the decoding termination time is a source of idle time for the decoder finishing earlier which is a waste of parallelism.

The second solution offers the usage of a "dynamic scheduler". The scheduler works as a dispatcher for the data to be processed and controls the collaboration between decoders. The scheduler can adjust the group effort such that for a high SNR in BFA situation may cause the circuit to take the decision to switch from BFA to UFA. This is because at high SNR values, both UFA and BFA produce similar results. Therefore instead of using two decoders, one forward and one backward, only a single decoder per codeword is used.

In this second solution, attempts to use additional one or two decoders to start next decoding codeword(s) were done. The selection of the next codeword(s) is made to be among the ones presenting the higher probability to finish earlier.

The third solution is a decoding with "static scheduling". The term refers to a design of four pairs of decoders. They assist only their pairing decoder rather than using a more complex scheduler schema as in the second solution. The motivation for this solution was to make a compromise between speed and hardware complexity.

4. BFA C++ IMPLEMENTATION

4.1 MOTIVATION

The essential motivation behind a C++ implementation is to propose a solution that could go beyond the simulation which was the greatest problem with the former MATLAB implementation. This is the case because high speed constraint, which is crucial for real-time applications, is not met.

Also, we may use this implementation to measure the speed gain in comparison to the simulations. Depending on the throughput, we can also use this implementation in real environment.

The choice of C++ was taken to avoid abstraction and try to speed the processing by focusing on lower level programming. C++ is still a language that is widely used especially on the networking environment.

Finally, C++ is a language that is used in several parallel programming environments that lead to new perspectives in that direction.

4.2 STRENGTHS & WEAKNESSES

Compared to the original code, we can truly start speaking about speed when we use C++. It is an old but still improving (ISO/IEC 14882:2011 (TR1), 2011) language that is more suitable for time critical tasks. This is the case because the language presents the advantages of low level language that can literally be optimized for specific CPU usages. In the same time, the language allows some abstraction which allows smaller coding effort.

A possible advantage is the possibility to call a C++ code from just a command line and take the result as an output. This is practical when thinking about the usability of the code. With the correct programming style, the BFA code may be used in a batch mode which is very useful for both real life usage and scripting environment.

So, compared to the MATLAB code, the result is expected to be faster. However some considerations should not be neglected. For example, in C++ there are no built-in libraries or functions to respond to the mathematical demand. Therefore, such problems should be considered. An example is the error function that must be written to run the algorithm.

There is also a technicality limitation with the change between earlier implementation and C++. For instance, the memory management is left to the programmer. It may be an advantage to allow local optimization. However, it is also a disadvantage since it will make the programmer work with stream of data that needs constant checking

4.3 APPLICATION

4.3.1 Overall Structure Of The Program

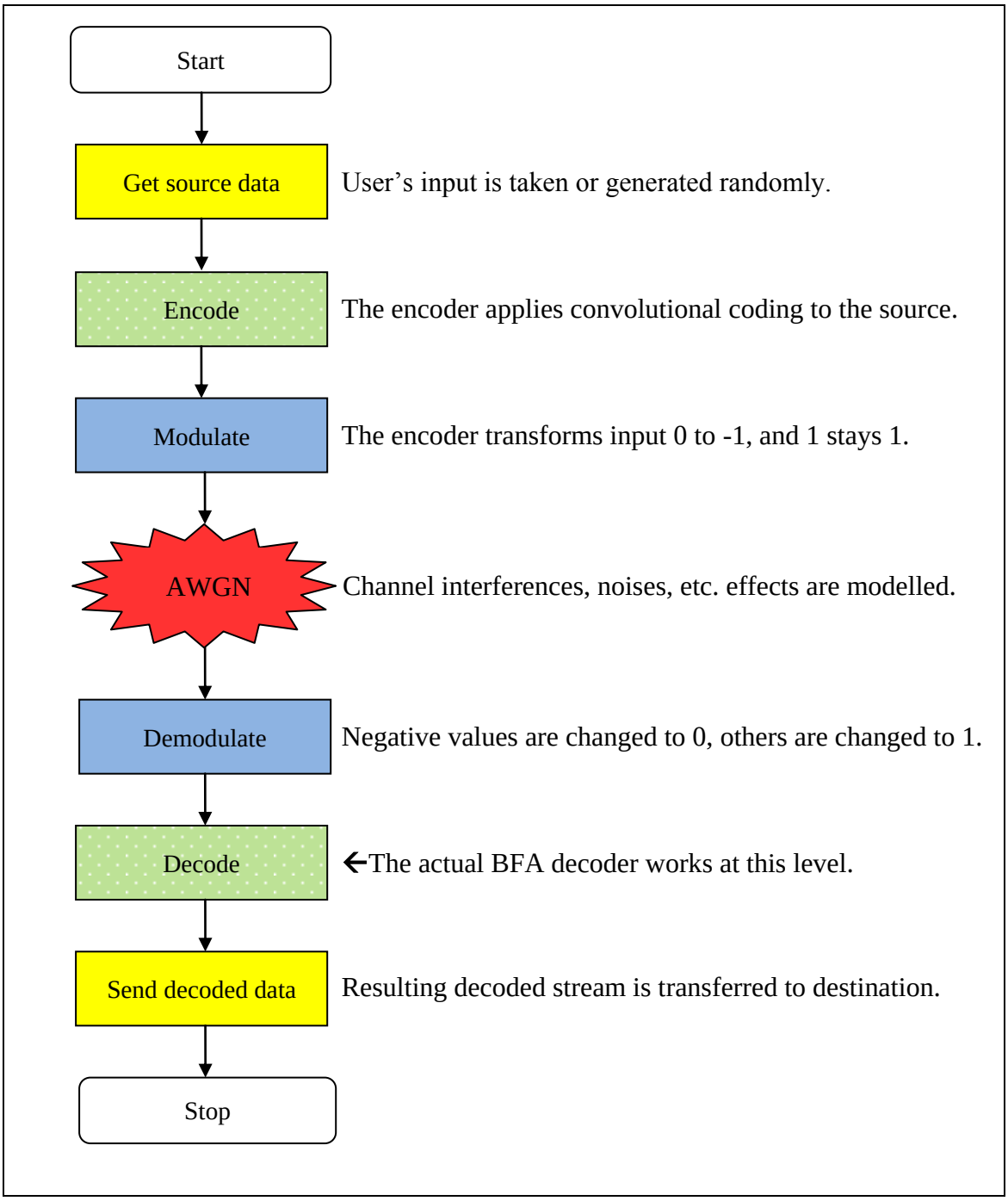
This section is very important; we will refer back to this point in a next implementation.

It is important that the core design is inspired by the MATLAB code from the first BFA paper (Ran Xu, 2009). My contribution here is not to change the overall code, which stays somehow similar, but to implement it as a C++ code.

A later implementation will need drastic modifications at several points. The motivation behind it will be to make the algorithm work as a massively parallel implementation in CUDA, which in fact is the main goal of this research.

The overall work flow of the encoder - decoder system can be summarized as:

Figure 4.1: Decoder's work flow



4.3.2 Encoder

For the encoder, the generator sequence was taken from the WirelessHD specifications document (2010).

$$\begin{cases} g0(1011011) \\ g1(1111001) \\ g2(1110101) \end{cases} \quad \textbf{(4.1)}$$

Let OUTA, OUTB, OUTC be the generated output from g0, g1, g2.

Let also d0, d1…d6 be the memory registers of the encoder, d6 be the current output, d0 being the input entered 6 steps before, we have:

$$\begin{cases} OUTA = d6 \oplus d4 \oplus d3 \oplus d1 \oplus d0 \\ OUTB = d6 \oplus d5 \oplus d4 \oplus d3 \oplus d0 \\ OUTC = d6 \oplus d5 \oplus d4 \oplus d2 \oplus d0 \end{cases} \quad \textbf{(4.2)}$$

The function loops through the input bits with six additional iterations to allow the tailing bits to be processed entirely and the encoder to return to the initial state.

When the result is calculated, right shifting is done and d0 becomes d1's value, d1 becomes d2's value, etc… d6 is set to the new input bit, or in the case no new bit is available, it is set to 0.

Naturally, if S is the size of the input, then the output will be 3xS wide.

Performance wise, an encoder is limited by its memory. This makes it very difficult to implement in a real parallel executed code.

4.3.3 Modulator \ Demodulator

Modulator \ Demodulator

The working mechanisms of both the modulator and demodulator are trivial.

Let us consider the example of an encoded stream.

Table 4-1: Modulator\Demodulator example

	Input							Output						
Modulation	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**-1**	**-1**	**-1**	**-1**	**1**	**1**
Demodulation	**0.1**	**-1.2**	**-0.7**	**-1.0**	**-3.0**	**0.2**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**

Unlike the encoder, these blocks can easily be parallelized.

4.3.4 Channel Noise With AWGN

This calculation of the noise is not a difficult problem. It consists of the usage of the normally generated random numbers not supported in the C, C++ libraries. An example code (1999) from Thomas Sailer was finally found over the internet. Later, Boost library was found to also support normal distribution generation.

Applying AWGN, like modulator code, can easily be done in parallel coding.

4.3.5 UFA

The Fano algorithm mechanism was briefly explained in 2.1. The coding, besides some technicalities, basically stays the same.

Few technical decisions were made during the coding. For example, pointers were often used. This is because while computing next states and metrics we would need to send multiple answers to output the result. For the next state, we have for current input bit=0 and another state for bit=1 and so on. Considering that we also have history pointers, we end up with many memory allocations at a time. Fortunately, unlike the history pointers, the answers for state or metric calculation stay at the order of O(1).

The mechanism of the Fano algorithm was slightly decomposed and portions of code that could be reused were actually run together.

We can summarize the overall coding as the following:

STEP 1 Initialization

Delta, M0, M1 are calculated and memory allocation are made

STEP 2 Loop start

Metrics and MF are computed.

STEP 3 If in move forward condition

Current states, metrics are stored in State history and Metric history.

Current state and its corresponding output bit are updated.

Flag history is updated.

Depth in the tree is incremented.

If search reaches other end, stop the program; otherwise go to STEP 2.

STEP 4 If MB < T

Current states, MB, M are computed.

Depth in the tree is decremented.

If from worst node check, Flag LFNB is set and go to STEP 2.

If MB < T is still true, tighten T and go to STEP 2.

Check overflow and stop if max operation is reached

For the UFA decoder to work, the parameters to provide are:

Table 4-2: UFA parameters list

Input	Output
Noised input	Decoded output
Length input	Count operations
SNR	Flag overflow
Metric precision	
Delta	
Max operations	

4.3.6 BFA

BFA implementation is very similar to two UFA blocks that run successively. This was to be expected from the specification of the paper (Ran Xu, 2009) and the Chapter 3.2.1.

We also took into account additional merging conditions. Also, new controlling parameters as g, showing the gap between both encoders were added. If the gap is 0, this means that both encoders are at the same point in the decoding. In such case, the algorithm takes the encoded part from the forward decoder from start to merged region and takes the encoded part from the backward decoder from the end to the merged region to constitute the decoded output.

Some remarks about the code: The coding requires twice the memory since we are effectively running two decoders at the same time. This may be a problem in memory limited environment.

All other functions remain the same, except that the decoder and the next state finder code change for the backward version.

The generator sequence for the FD was:

$$\begin{cases} g0(1101101) \\ g1(1001111) \\ g2(1010111) \end{cases} \quad \textbf{(4.3)}$$

The generator sequence used for the BD was then chosen to be:

$$\begin{cases} g0(1011011) \\ g1(1111001) \\ g2(1110101) \end{cases} \quad \textbf{(4.4)}$$

5. CUDA

5.1 BACKGROUND

To satisfy gamers, 2D processing image processing demand, as early as 1985 GPU were introduced with Commodore Amiga. Modern GPU can process 2D, 3D and make several intense arithmetic operations required by the latest games. This is accomplished by the use of multi-core design at the hardware level. In order to draw some effects like shader effects, OpenGL and DirectX required from GPU to provide GPGPU capability, such as a general purpose programming capability.

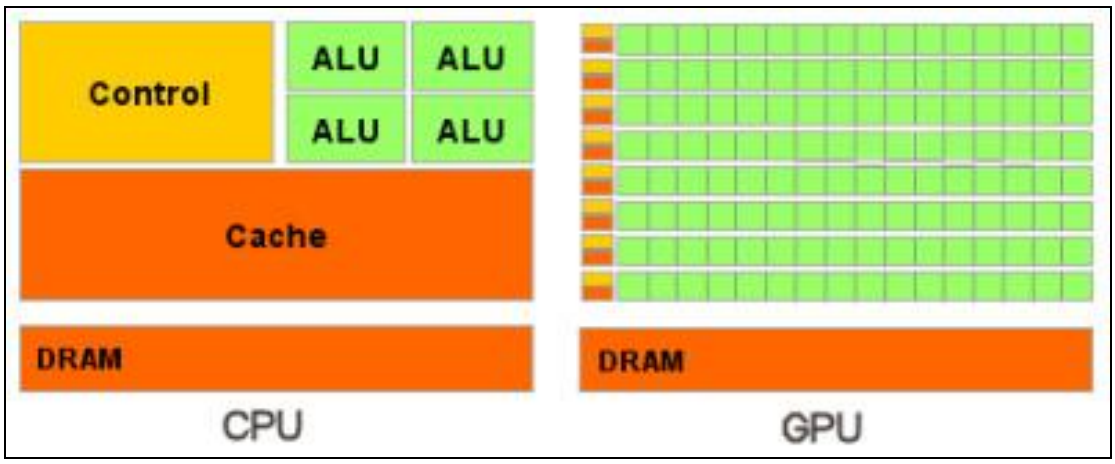
Compute Unified Device Architecture (CUDA) was specifically developed for the GPGPU purpose. Hence, NVIDIA boards now incorporate this feature allowing programmers to use the board as a parallel processing platform. Modern motherboards present on-board GPU sufficient enough for a desktop usage. In such cases, the OS puts the NVIDIA board into sleep mode for power consumption. We can say that the desktop graphics are handled by the on-board GPU, whereas the NVIDIA board handles games graphics and CUDA specific instructions. This way, programmers can employ the unused processing power of the GPGPU just like a co-processor dedicated only to execute programmer's instructions.

While generally a modern CPU has around four cores, a CUDA-capable NVIDIA board has at least two Streaming Processors (SP) having 8 cores \ SP that is 16 cores in total (NVIDIA Corporation, 2009). Current NVIDIA boards starts with at least six SP that is around 50 cores. The cores count can even go over 3000 for the latest models like Tesla K10 (NVIDIA Corporation, 2012) which is a combination of two boards working in pair. This is the result of the fact that CUDA programming is scalable.

When a CUDA programmer gives instructions to perform, he also has to indicate the number of threads that should handle those instructions. The board then gives orders to its idle cores to dispatch those instructions among its threads. This part is transparent to the coder and requires no change in the initial coding.

The following is a typical CPU compared to a CUDA-capable NVIDIA GPGPU board.

Figure 5.1: Structural comparison CPU & GPUGPU



Source: NVIDIA Corporation, 2012

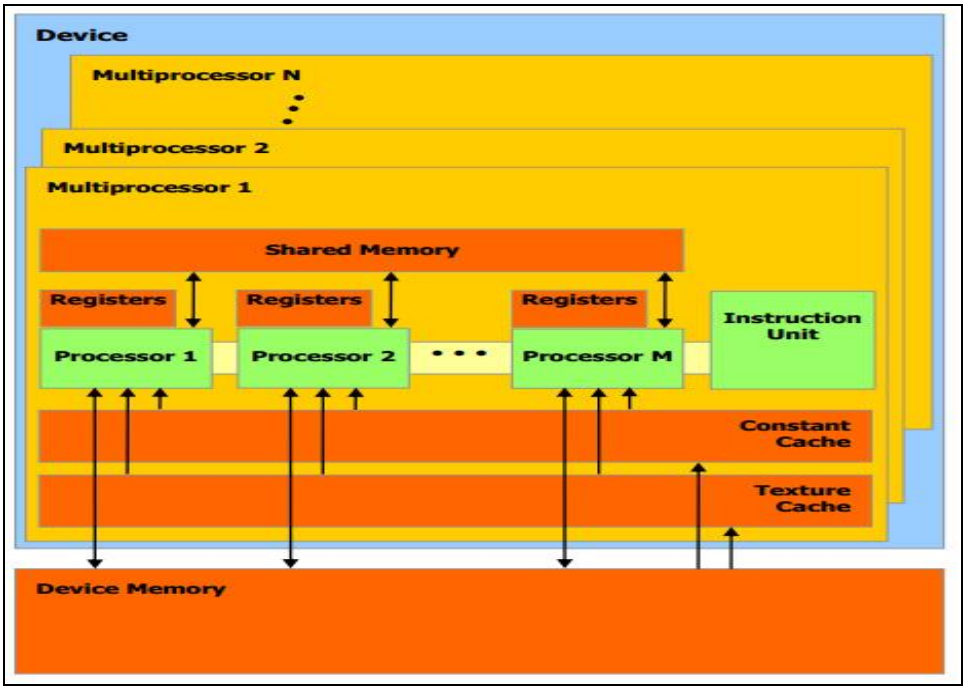
In the previous figure, we can observe that a special disposition for the memory is taken. The actual memory structure of a CUDA compliant board is as follows.

Table 5-1: Cuda memory size with example

Memory Type	Access Type	Access Scope	Example: 330M
Register	Read & Write	Thread	1 Kbyte
Local Memory	Read & Write	Thread	16 Kbytes
Shared Memory	Read & Write	Block	64 Kbytes
Constant Memory	Read only	Grid	64 Kbytes
Texture Memory	Read only	Grid	8 Kbytes
Global\Device Memory	Read & Write	Grid	1024 Mbytes

All six memories are accessible in the device. Global, constant and texture memories are also accessible from the host which allows a two way communication. This is generally used as a memory initialization as it is the only way a device can be given data or read data from. It is a large but slow memory access as it is a DRAM memory. Other memories much smaller but quicker may also transfer data with those three memory types if needed.

Figure 5.2: CUDA memory hierarchy



Source: NVIDIA Corporation, 2012

The software layer is governed by C\C++ with the addition of several CUDA specific functions like data transfer functions, thread control commands, usage of CUDA libraries etc. Several third party wrappers exist for Java (Java bindings for CUDA, 2012), Fortran (Portland Group, 2012), Python (ArrayFire CUDA Python, 2012), etc.

The programmer will be able to write kernels which are nothing more than codes destined to be run on the device. Running, instantiating kernels using threads are done by the coder. For example, we may want to have 100 percent occupancy for a 330M. For this we would ask 512 threads (the maximum threads per blocks) to be instantiated for a particular kernel run.

5.2 APPLICATIONS

CUDA works on a NVIDIA GPU board but its applications go beyond the graphical and video related applications. As mentioned earlier, the platform presents an immense processing power to programmers who have already started utilizing that opportunity (CUDA-Accelerated Applications).

As examples of applications we can refer to the following table.

Table 5-2: CUDA's areas of application

Program Name	Area of application
Accelerating bio-molecular simulations	Computational chemistry
Arbitragis Trading	Financial
CUDA Acceleration for MATLAB	Matlab integration
CUDA Voxel Rendering Engine	Video applications
Digisens: SnapCT tomographic reconstruction software	Medical imaging
DNA Sequence alignment: MUMmerGPU	Bio-informatics
GIS: Manifold	Government & defence
GPMAD : Particle beam dynamics simulator	Electrodynamics
NAMD molecular dynamics	Molecular dynamics
Synopsys: Sentaraus TCAD	Electronic design automation

As we can see, CUDA is widely used in the modern market. Nowadays, CUDA has many other applications among a diversified palette of utilizations.

5.3 ALTERNATIVES

Microsoft in its HPC Whitepaper (Microsoft, 2010) indicated the AccelerEyes, BrookGPU, DirectCompute, OpenCL and PGI as alternatives to CUDA. To this list we can also add PathScal's ENZO. Their specifications are beyond the scope of this thesis but we can briefly describe them.

OpenCL is by far the closest to CUDA and is also more popular than the other alternatives. Khronos Group is the developer of this open standard platform (Khronos Group, 2012). OpenCL is a framework for heterogeneous systems which supports Apple, AMD, Intel, NVIDIA, and ARM GPGPU boards (Khronos Group, 2012).

DirectCompute is Microsoft's product and works on top of CUDA \ OpenCL but it is not a popular solution. Even its creators do not provide much documentation to encourage programmers to use it.

ENZO (PathScale, 2011) is a small compiler in beta testing. It can be programmed in C, C++ and Fortran. It uses NVIDIA's native instructions and supports OpenHMPP (NVidia Corporation, 2012).

PGI (Portland Group, 2012) is the compiler from the group, which is a subsidiary of STMicroelectronics. PGI also proposes C, C++ and Fortran programming supports.

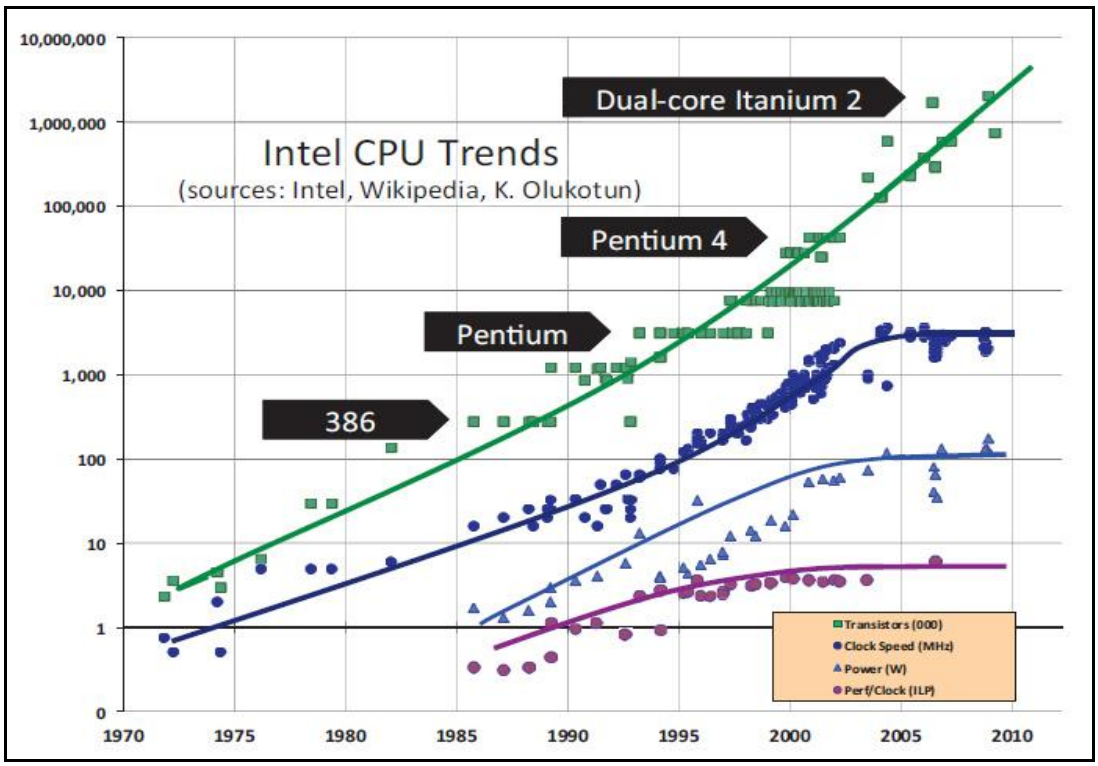
4. BFA CUDA IMPLEMENTATION

6.1 MOTIVATION

The MATLAB implementation suffered from a speed problem. This was primarily the consequence of the program targeted to being used for simulation and not real life project.

C++ implementation presented a better result but we can still improve the throughput. However; there is a limitation to how much we can advance in terms of speed in C++. The most reasonable addition we may think is to use the parallel execution.

Figure 6.1: Intel CPU Trends



Source: Sutter, 2011 *'The Free Lunch is Over': transistors per processor, clock speed, power consumption and instruction level parallelism*

The figure illustrates the physical limitation that started to appear on 2003. The effect was to contradict the Moore's Law which stipulated that the clock speed of processors would double approximately every 18 months. To counter the problem, multi-core processing was opted.

In the same logic, the problem with the C++ implementation was the fact that it relied on a single processor execution pattern. Also, we depend on the processing power of the CPU which is heavily controlled by the OS. BFA implementation in CUDA was selected because it allowed programmers to use the board as a parallel execution platform. Also the language used to execute was basically C with the addition of CUDA specific APIs that simplifies the transition from the previous C version to CUDA.

6.2 STRENGTHS & WEAKNESSES

CUDA is a GPGPU platform. It is designed to be used in blocks of data being processed simultaneously. This is translated by coding style changes and new constraints to be added.

Since our code will work on the GPU, we will not have the latency imposed by the OS which has to listen to many IO and time consuming threads. It is as if we had a huge multi-core CPU dedicated only to our job.

CUDA is dependent on the architecture of the GPU to be used on. Knowing that architecture, it may allow further optimizations.

Memory access is the limiting factor of parallel programming. We may have as many threads as we want, they may be as quick as possible but processing power is useless if the data transfer is slow. In regular processing, we had to wait for the memory to be initialized, and then we had to wait for the result to be calculated. In parallel processing, we may multiply the number of cores working and make the processing strength virtually unmatched. However, we still have to send the data to be processed to the memory and get the result from that memory.

Since the processes all run at the same time, their synchronization may also become a problem.

6.3 APPLICATION

6.3.1 Demodulator

Until now no code was given since most of the coding was trivial (The appendix will have complete codes). However, not everyone is familiar with CUDA programming. Therefore an example will be given. The following is the demodulator's code.

Here is the kernel code that will run in the device.

```
#define MAX_THREADS_PER_BLOCK 512
#define MAX_BLOCKS 512

__global__ void demodulate(const float *dev_noise, char* dev_demodulated, const int size)
{
        const int i = threadIdx.x + blockIdx.x*MAX_THREADS_PER_BLOCK;
        if(i>=size)
                return;

        if(*(dev_noise+i)<0)
                *(dev_demodulated+i) = 0;
        else
                *(dev_demodulated+i) = 1;
}
```

Notice that the code is meant to work in a succession of blocks. This is seen by the parameter blockIdx.x an API specific identifier. As we remember, the blocks were 2D arrays and the x is for that value. In this implementation, 1D blocks were used to simplify the coding.

In the same logic threadIdx.x gives the index of the thread. With these two parameters, we will be able to compute the integer i as the index of the thread.

For example, let us say we want to demodulate a block of 1000 floating point values. These can represent a typical portion of WirelessHD frames that passed through a noisy channel.

If we consider that the device can support max threads/block 512 and max blocks/device 512, and if we wish to process the demodulation in one time, we will need to be concurrently working with two blocks:

$$\lceil 1000/512 \rceil = 2\,blocks$$

From the equation we can deduce that the first block will be 100 percent occupied, while the second block will have 95 percent occupancy with 488 threads running of the possible maximum 512.

The kernel code is very straightforward but one should be aware of the structure of the kernel. The beginnings of two memory locations in the device are passed as arguments. Those are dev_noise for the input and dev_demodulated as the result of the demodulation. The index is calculated to correlate the current thread with its corresponding input and output memory locations. The return condition, even though adding minor latency from the check condition, is necessary for handling erroneous and unexpected device behaviour.

Now let us see the actual instantiation of the working threads.

```
int main()
{
        // Allocate device memory
        cudaMalloc((void**)&dev_num,memSize);

        // Copy noisy numbers to device memory
        cudaMemcpy(dev_num, num, memSize, cudaMemcpyHostToDevice);

        // Call demodulate kernel
        demodulate<<<blockCount, threadCounts>>>(dev_num, size);

        // Copy demodulated result back from device memory
        cudaMemcpy(num, dev_num, memSize, cudaMemcpyDeviceToHost);

        // Freeing memories
        cudaFree(dev_num);
}
```

As we can see from the main program there are typical changes to C coding.

i. As we remember, the device has no access to the host memory. Therefore, programmers must copy host data to the device's memory with cudaMemcpy.
ii. Actual kernel call looks like a regular C function call with the addition of <<<…,…>>> after the function name. Considering the kernel code, it is apparent that CUDA will instantiate as many as blockCount blocks with threadCounts threads each.
iii. When we wish to get the result from the threads, another memory copy call is performed from the device to the host.

6.3.2 Parallel Blocks

Several parts of the coding were adjusted to take advantage of the CUDA's proposed massively parallelism as in the case of encoder - decoder explained in 6.3.1. Using the same kind of procedure, the following blocks were changed from iterative blocks to parallel blocks to allow concurrent execution. Hence, theoretically those blocks now present a drastic performance increase.

In an initial CUDA implementation attempt of the system, it was apparent that each block could not be made parallel as easily as demodulator. A summary of used blocks are given in the table below.

Table 6-1: Initial CUDA parallelism without optimization

Block Name	Initial CUDA implementation
Modulator \ Demodulator	Parallel block
Metric calculation	Parallel block
State finder	Parallel block
Random generation	Parallel block
Channel AWGN	Parallel block
Encoder	Iterative block
Decoder	Iterative block

The blocks that could be made parallel are those that presented no memory in their coding.

Notice that the random generation code was part of a blog (Nobile, 2011) and it was used for both the generation of random input values for testing and uniform random values for applying AWGN to modulated data.

6.3.3 Iterative Blocks

Two major blocks, encoder and decoder were not successfully made parallel in the first design of the system. As shown in 4.3.2, the encoder needs to know the previous six bits to generate the next encoded value.

The Fano algorithm is a sequential decoding. Therefore, the algorithm is by design an iterative block. However, the use of those blocks in parallel will be investigated in the next chapter.

6.3.4 Decoder Structure

We can represent the code as the followings.

Table 6-2: call_decoder

void call_decoder
Inputs BFA_params, BFA_operations Purpose 1) Prepare decoder primitives and memory 2) Initialize pointers contents 3) Call __global__ void fano_decoder 4) Benchmark time spent and collect decoder's result

Table 6-3: fano_decoder

__global__ void fano_decoder
Inputs BFA_operation, BFA_params, output Purpose 1) Create and set shared memory 2) For each code word a) Get next 3 bits to decode b) Call operateOnPreconditions (calculate metrics, next move decision, decide tail bit, etc.) c) Call moveForward or moveBackward d) Call checkOverflowAndMerge e) If d) found no FD&BD merge or overflow return to a) f) Stop decoding 3) Calculate final decoder response

Table 6-4: metric_calculation

__device__ void metric_calculation
Inputs preconditions, BFA_params Purpose 1) Calculate 2 possible metrics & next state & next output 2) Choose next metric 3) Call __global__ fano_decoder 4) Benchmark time spent and collect decoder's result

Table 6-5: operateOnPreconditions

__device__ void operateOnPreconditions
Inputs preconditions Purpose 1) Decide isTail=0 or isTail=1 2) Calculate next output, next state 3) Call metric_calculation 4) Set MF 5) Decide moveForward=0 or moveForward=1

Table 6-6: moveForward

__device__ void moveForward
Inputs preconditions Purpose 1) Push current context to previousContexts stack 2) Set current state and current output bit 3) Increase depth 4) Update T, M, Flag_LFNB and visit_record

Table 6-7: moveBackward

__device__ void moveBackward
Inputs preconditions Purpose 1) Pop previousContext stack to get previous state, metric and Flag_LFNB 2) If needs tightening, T=T-delta and stop function 3) Decrease depth 4) Update M, Flag_LFNB

Table 6-8: checkOverflowAndMerge

__device__ void checkOverflowAndMerge
Inputs preconditions, preconditions Purpose 1) Check FD or BD do not overflow, stop decoding otherwise 2) Stop decoding if FD and BD merge

5. OPTIMIZATIONS

7.1 MEMORY OPTIMIZATIONS

7.1.1 Array Optimizations

The memory is the bottleneck of the algorithm. While the operation to be done at each step is trivial, the spatial-temporal requirement is huge. A typical call needed:

i. Visit_record which has the size inputSize x 64 x 1bit,
ii. State_history which has the size inputSize x sizeOfState,
iii. Flag_history which has the size inputSize x 1bit,
iv. Metric_history which has the size inputSize x sizeOfMetric.

For this memory organization, tests were conducted to find a correlation between memory usages and their actual implementations. A first test for back tracing was done. In it, for each time a move back was attempted, the final back tracing amount was recorded to find a possible maximum back tracing amount. The result was that even back tracing was very local, generally one or two steps back. Statistical analysis revealed that a significant portion of the back traces were eight or lower covering 97.72 percent of encountered traces hence the maximum possible back tracing was set to eight. This opened an improvement area. Instead of keeping all the possible metrics in metric_history, we only needed to keep track of the previous eight metrics. Since we are also pretty much using the flag for back tracing keeping the flag_history was also unnecessary.

This was found to be somehow similar to a return back of a function from a regular assembly code, in which a context was constructed. In it we had metric, flag and state. Each time a move forward is done, a new context is pushed into a circular queue that can contain at most eight items. This means that, storage of inputSize x (6+1) bits is reduced to a mere constant: 8 x (6+1) bits. Each time a move back is done, the last context is popped out rather than reading from a big array and they are both one cycle operations.

7.1.2 Shared Memory

For an optimal operation speed, special attention was given to the usage of the shared memory. Since global memory usage is a slow access, shared memory was preferred whenever possible. However, handling such memory comes with a price: it has a small size, in the order of several kilobytes per block.

The practice used was to use shared memory for every internal variable as count computation, latest contexts, input BFA_params or the preconditions structure. Shared memory was not used for big storage requirements as state_history or visit_record_history. This is due to the fact that in such event, the shared memory would saturate and the compiler would even refuse building such program.

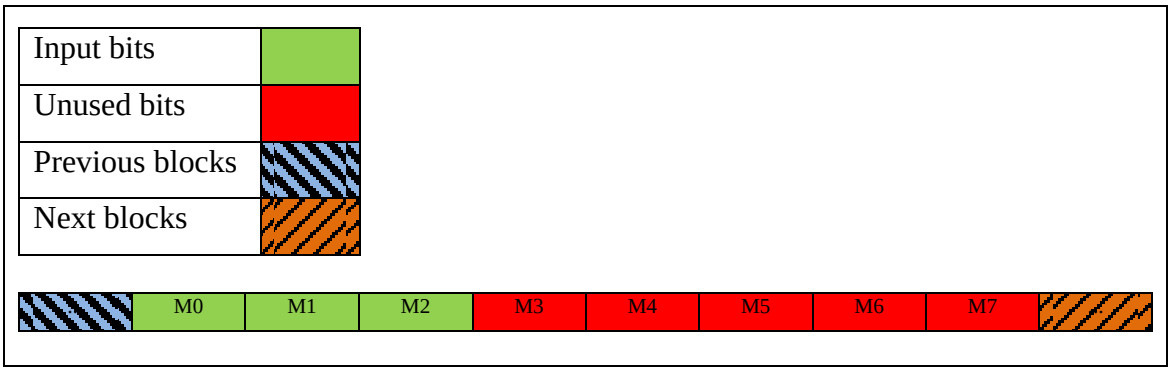
7.1.3 Inputs In The Memory

As explained earlier, 1-bit is encoded with 3-bits. Therefore, at each step of the decoding we are dealing with three bits. Considering the hardware structure of either CUDA or more generally C, we can realize that such a representation was not intended. We have Boolean or characters etc. which represent respectively one or eight bits. Boolean actually are not stored as 1-bit memory spaces but rather packed in 8-bit blocks (Microsoft). In that perspective, we can assert that the smaller possible addressable memory is a word that is an 8-bit pack which is a char type.

With the previous statement, our goal was to use char for inputs and such claim would give the following memory organization.

If we use a big-endian representation with M0...M7 and M0 is the most significant bit.

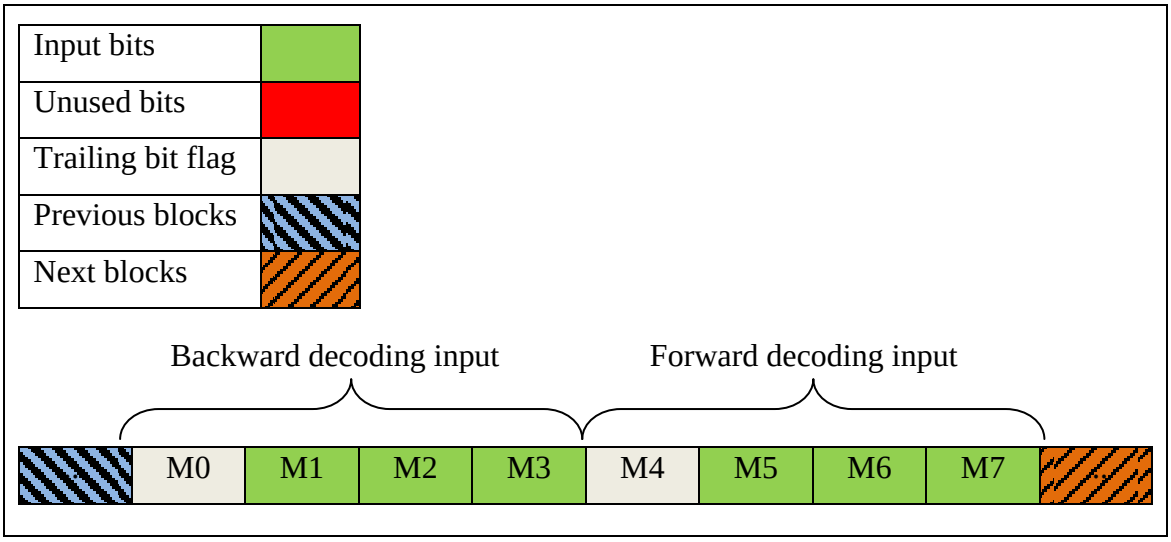
Figure 7.1: Inputs representation before optimization



This would make a 3/8 occupancy making 62.5 percent memory waste for each input.

When reading inputs, we are always carrying the depth information such that at each input read we ask for a 3-bit block to be read for a depth d where $0 \leq d < inputSize$. Considering BFA, the following was decided to be used:

Figure 7.2: Inputs representation before optimization



In such reorganization, what we achieve is the followings:

1) All bits of the memory are effectively used.
2) Additional information is stored along with the input bits. This information trailing bit is used in the algorithm such that:

$$\text{Trailing bit flag} = \begin{cases} d * 3 < inputSize - 7 & 0 \\ otherwise & 1 \end{cases} \quad \textbf{(7.1)}$$

This way, we no longer need the inputSize information in the calculation, it is already calculated and we only use depth information.

3) Forward or backward decoding has the same progress schema. Before this, the algorithm needed to check if the current decoder was a forward decoder (FD) or backward decoder (BD). This information would dictate the calculation of the index of the next input block. For example, if we move forward we would have depth=depth+1 but index=index+3 for forward decoder and index=index-3 for backward decoder. In this optimization, we no longer calculate the index of the input nor do we calculate whether or not in the trailing bit. This last information can be obtained directly from the input.

7.2 ALGORITHMIC OPTIMIZATIONS

7.2.1 Structures Usage

The code was also changed to use parameterized structures. Those structures are UFA_Memory, BFA_params, BFA_operation.

Memory allocation for a UFA was partitioned as follows. Rather than allocating and freeing memory in the device a single chunk of allocations were done in bulk. That memory contained all the memory needed for a single decoding to run. In such manner, offsets were calculated to indicate where each memory blocks started. For this, we used an initialization function that takes frame length and determines the values for offsetVisitRecord, offsetStateHistory, offsetFlagHistory, offsetOutput, offsetMetricHist, offsetStateOrder, offsetStateInput, offsetMetricOrde, sizeAllocated.

The usage of UFA_Memory would allow the reuse of the same allocated memory between calls of the same frame length bypassing the allocation procedure to ease the memory access with the use of offsets directly accessible within CUDA code.

The purpose of BFA_params was to pass the same BFA parameters between calls and avoid repetitive delta, M0, M1 calculations. The structure also presents an initialization method that takes a SNR value, delta, maximum operations, and metric precision. The function will store delta, max operations and calculate M0, M1 given the SNR, delta. Calling the initialization function again will simply update those values if need be.

Finally, the last structure is BFA_operation. In it are held the input and the length of the frame and the operation specification inputs. However, the structure is not limited to inputs; the results are also read from it. The actual output, flags, merging depth, sequential and parallel operation counts are also accessible from this object.

7.2.2 Algorithm Optimization

A problem in the BFA's algorithm was that while moving back from the initial case with depth = 0, MB was set to $-\infty$ which has no meaning in a hardware environment. The MATLAB implementation handled the problem by setting MB to -2^{20}. This is a solution but it means we have to make many looping to get the actually effectual values to allow a move back. Testing was made to find the possible minimum value of MB to

make the first move back. The result was that such number was -39, therefore $-\infty$ value was taken as -39 reducing redundant initial looping.

7.2.3 Local Optimizations

Code local optimizations were done.

For example, when tightening was needed, originally the code would tighten as long as T < MF - delta was true. Such a condition would introduce unwanted iterative code.

The iterative code was changed with its corresponding procedural form.

Table 7-1: Tightening procedural version

Before	After
`while(T_f<=MF_f-delta)` `  T_f=T_f+delta;`	`if(T_f<=MF_f-delta)` `  T_f+=((MF_f-T_f)/delta)*delta;`

This way, that portion of the code was no longer iterative. Therefore, this section was no longer a limitation to parallelism.

7.2.4 Code Organization Rearrangement

After the 10^{th} iteration of the code, the program started to present serious abnormalities. This was simply a memory usage problem. In the dereferencing, referencing of pointers and handling complex pointer mechanism, the end result was a code that was prone to too many mistakes and made the advancement difficult.

For this purpose three general methodologies were used.

i. The usage of typedef was done whenever possible. This allowed both a dynamic code and a good counter measure to erroneous types that are mostly clouded by the compiler. This is the result of its casting values if such implicit conversion exists. However, if this was not the programmer's intend, such initiative can and did end up with hardly detectable errors. This small change made a long time coding improvement.

ii. The algorithm was divided into sub-sections:

a. The first section is “operate on preconditions”. In this section state input, state next, state next output, metric calculations, move forward or backward and detect if it is a tail situation or not is done.

b. The next section is “move forward”. In it, in the case of the memory optimization, the current context is pushed into the queue. Also, an actual output is generated. Depth, M and flag are calculated. Moreover a T value tightening is done in the case it is the first time visiting as required from the algorithm.

c. “Move backward” is the third section. The part is characterized by MB updating and in case of successful trace back (pop in the case of the memory optimization), change to that previously visited context. Naturally, the index of the latest pop index is decreased for each time we enter this section.

d. Lastly, “check overflow and merge” operation takes place. As the name suggest, in it both an operation overflow and merging conditions are checked.

iii. Finally a structure called preconditions was created. The purpose of this structure was to allow regrouping all values necessary before a forward or backward movement. The object contained inputs as isForwardDecoder, Flag_LFNB, Depth, received_bits, State_current, M, T. Also outputs as isTail, Metric_order, Metric_tail, State_order, State_tail, State_input, State_input_tail,State_next_output, State_next, MF, moveForward.

7.3 PARALLELISM OPTIMIZATIONS

7.3.1 Look-Up Tables

In order to handle the encoder problem, look-up tables were imagined. A somehow similar idea was also found in another paper (Ran Xu, T. K., 2010) about BFA. In it, authors used LUT for merged states operations. In this thesis, the motivation behind the

usage of a LUT was that we have seven successive stages all represented by a logical 0 or 1. Hence each stage including the previous ones only constitutes a 7-bit string.

Total number of combination is:

$2^7 = 128\ combinations$

Thus, the total size of the LUT is:

$128 \times 7 = 896\ bits$

Such memory is negligible when we take into account that constant memory has more than 70 times that amount in reserve.

If this optimization is actually done, even though latency is added for getting the first encoded value, in the end a big optimization is done.

The latency is in the order of calculation of 128 encoded inputs and their storage to the memory. However once done, the initial LUT is generated, the encoding process is reduced to a memory read access. Let us consider that a WirelessHD frame length is 300 bits, even in a single frame we can see an improvement while the remaining frames to be encoded are just constant memory readings. This is in the order of a register read speed (Kirk & Hwu, 2010).

Also a crucial point is that transforming the encoder's iterative operation to a LUT makes the encoding a massively parallel mechanism which is even quicker than a modulation\demodulation.

The same logic was used to optimize state calculations to avoid unnecessary calculations to a simple memory block reading from a LUT.

6. RESULTS AND ANALYSIS

8.1 TEST ENVIRONMENT

Table 8-1: Host specifications

MODEL	N \ A
CPU	Pentium Dual-Core E6800 @ 3.33Ghz
RAM	4GB RAM
GPU	GeForce GTX 650
OS	Windows 7 x64

Source: Microsoft

Table 8-2: Device specifications

MODEL	GeForce GTX 650
CORES	384 (48MPx8Cores/MP)
GPU CLOCK	1058Mhz
MEMORY CLOCK	5000Mhz
CONSTANT MEMORY	65536 bytes
SHARED MEMORY \ BLOCK	65536 bytes
THREADS \ BLOCK	1024
CUDA CAPABILITY	3.0
CUDA RUNTIME	5.0

Source: NVIDIA Corporation

8.2 SPEED COMPARISON BETWEEN IMPLEMENTATIONS

To measure the time difference between MATLAB, C++ and CUDA implementation, the same encoded symbols were used as inputs, thus making equal requirements comparison.

The following are the results of 10 test vectors for SNR = 4. For the purpose of analysis simplification, delta was chosen to be 8, max computational was 10000 which virtually made the search continuing indefinitely.

Table 8-3: MATLAB, C++, CUDA time comparison with SNR=4, delta=8, R=1/3

Tested lengths	MATLAB delay in seconds	C++ delay in seconds	CUDA *** delay in seconds
L = 2	0.000019993	**0.000051886**	0.000147
L = 4	0.000031512	**0.000103189**	0.000196
L = 8	0.000051350	**0.000205401**	0.000314
L = 16	0.000089938	**0.000409390**	0.000508
L = 32	0.000140053	0.000816677	**0.000749**
L = 64	0.000219301	0.001629516	**0.001153**
L = 128	0.000474368	0.003232794	**0.001701**
L = 256	0.000708974	0.006415299	**0.003022**
L = 512	0.001232765	0.012683948	**0.004701**
L = 1024	0.001913129	0.025045202	**0.006820**

*** The original CUDA implementation was used for fair comparison by putting any optimization effects.

8.3 NOI AND BER ANALYSIS

The concept of Number of Iterations was introduced in the simulation results of the original BFA paper (Ran Xu, 2009). In that paper, it is defined as:

$$NoI_{total} = \max(NoI_{FD}, NoI_{BD}) \tag{8.1}$$

The delay of the BFA operation is then assumed in the same paper as:

$$Delay_{BFA} = \max(NoI_{FD}, NoI_{BD}) \tag{8.2}$$

The analysis was done using the latest CUDA implementations of UFA\BFA code with 1000 codewords. As in a previous paper (Ahmet Kakacak, 2012), the analysis of NoI was done by excluding overflow cases while calculating the delays. However, they were taken into account for BER.

8.3.1 Experimental Results For Delta = 4

Figure 8.1: Throughput at delta=4

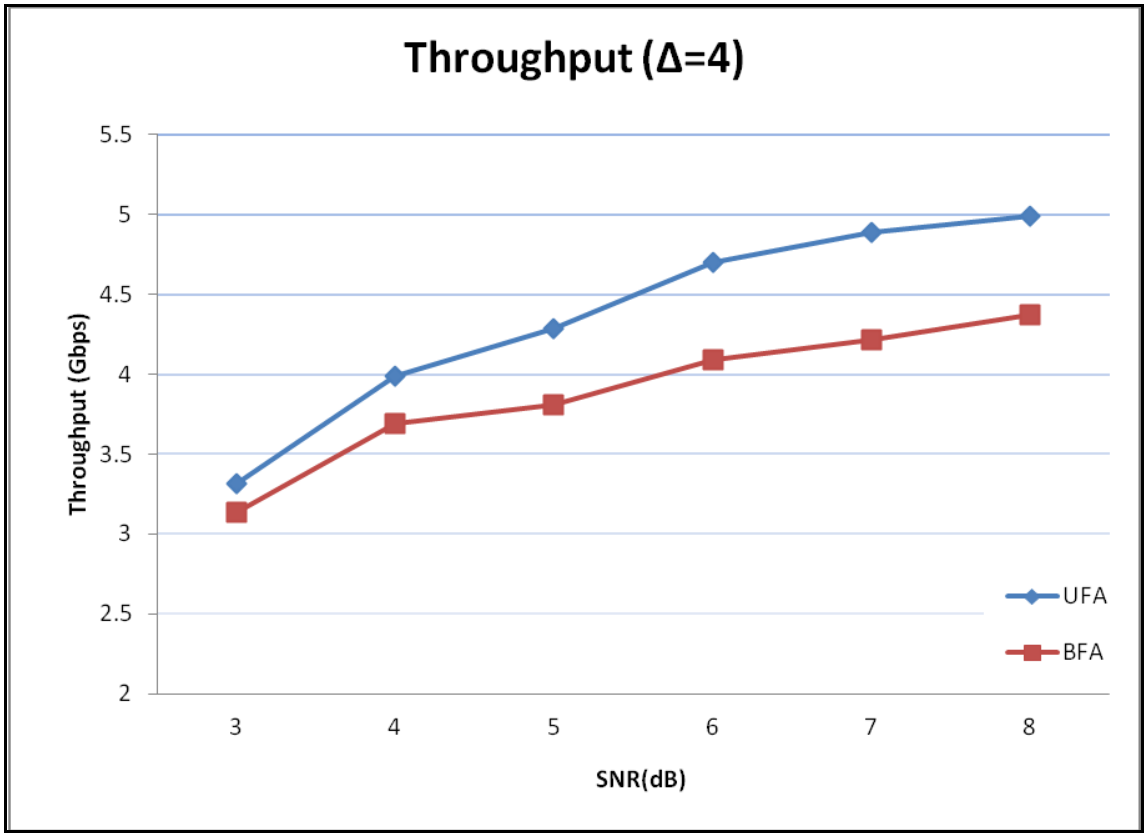
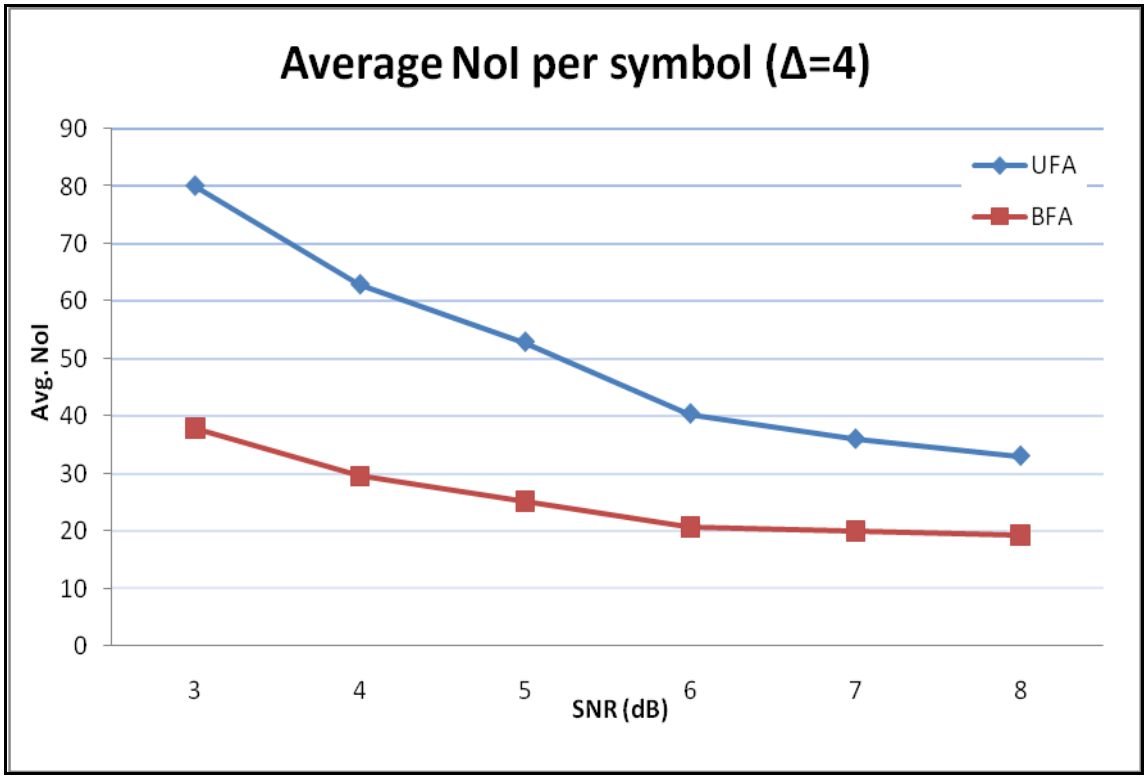


Figure 8.2: Avg. throughput at delta=4

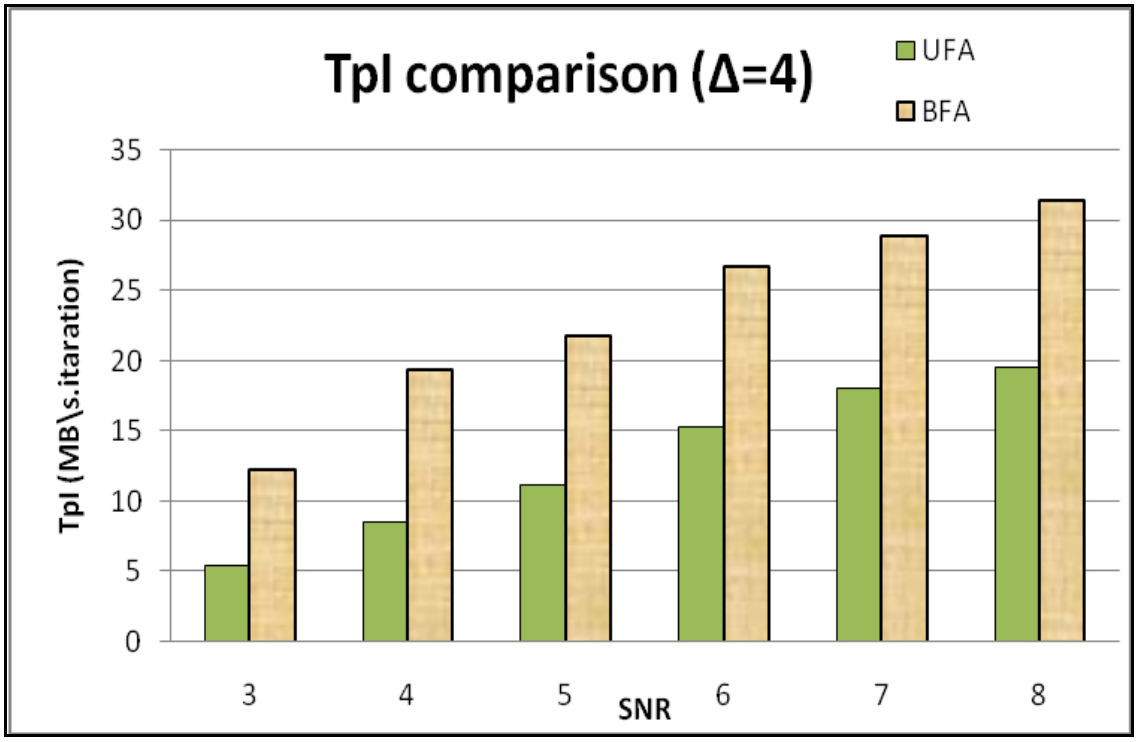


In the same logic, we can effectively compare UFA and BFA with a new metric that we can express as the throughput per iteration (TpI) defined as below.

$$TpI = \frac{Throughput}{Iterations} \quad \textbf{(8.3)}$$

The NoI analysis allowed measuring the effectiveness of the Fano decoder in terms of iterations made per symbol. Following TpI, we are able to measure the speed of each iteration.

Figure 8.3: TpI comparison at delta=4

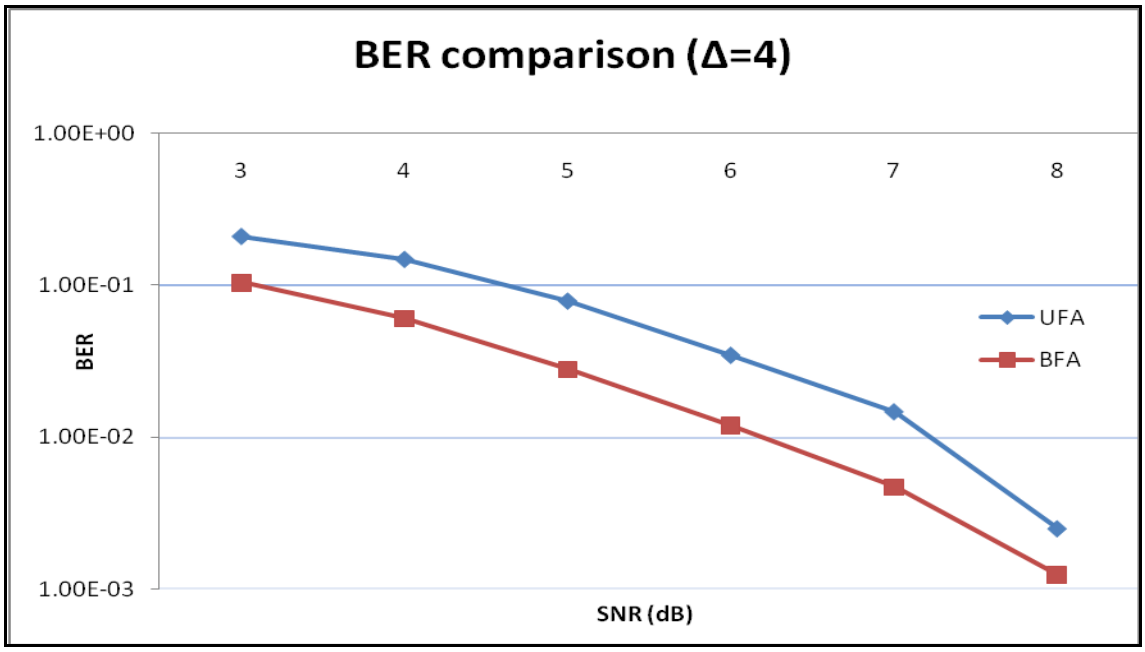


The graph shows us that each iteration in BFA is run with a higher throughput than its corresponding UFA for the same SNR. As SNR increases the improvement decreases but we always have a gain for using a BFA over UFA.

Table 8-4: TpI gain comparison at delta=4

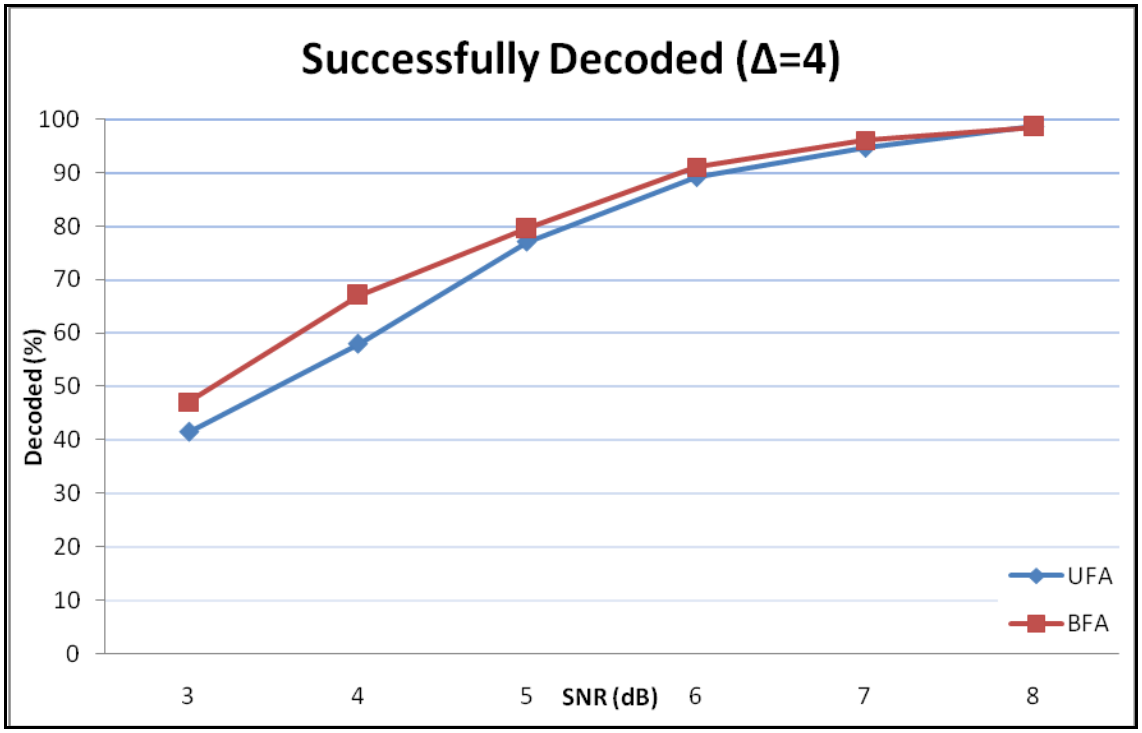
SNR	3	4	5	6	7	8
Gain percent	127.46	127.69	95.38	74.81	60.11	61.60

Figure 8.4: BER at delta=4



The BER figure above shows that UFA has higher error rates compared to BFA regardless of the SNR.

Figure 8.5: Decoded percentages at delta=4

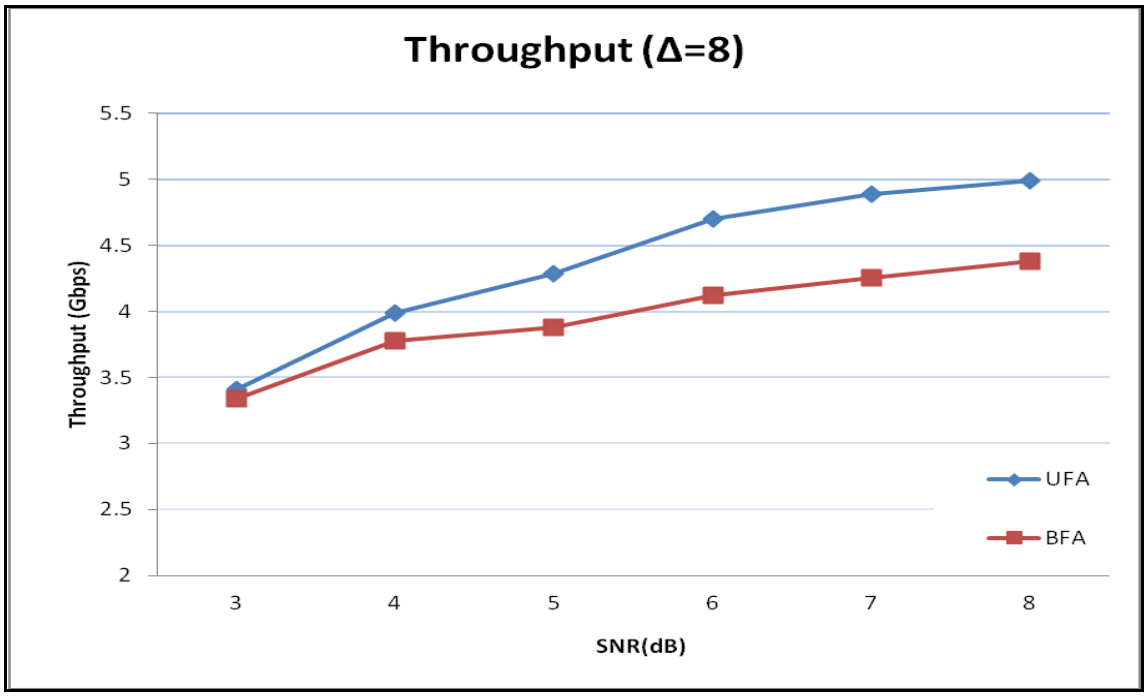


The decoded percentage is also bigger regardless of the dB used.

8.3.2 Experimental Results For Delta = 8

In this experiment, changing delta 4 to 8 comes with the following assumption. Since, delta influences directly the tightening rate, the higher it is, the greater the tightening.

Figure 8.6: Throughput at delta=8



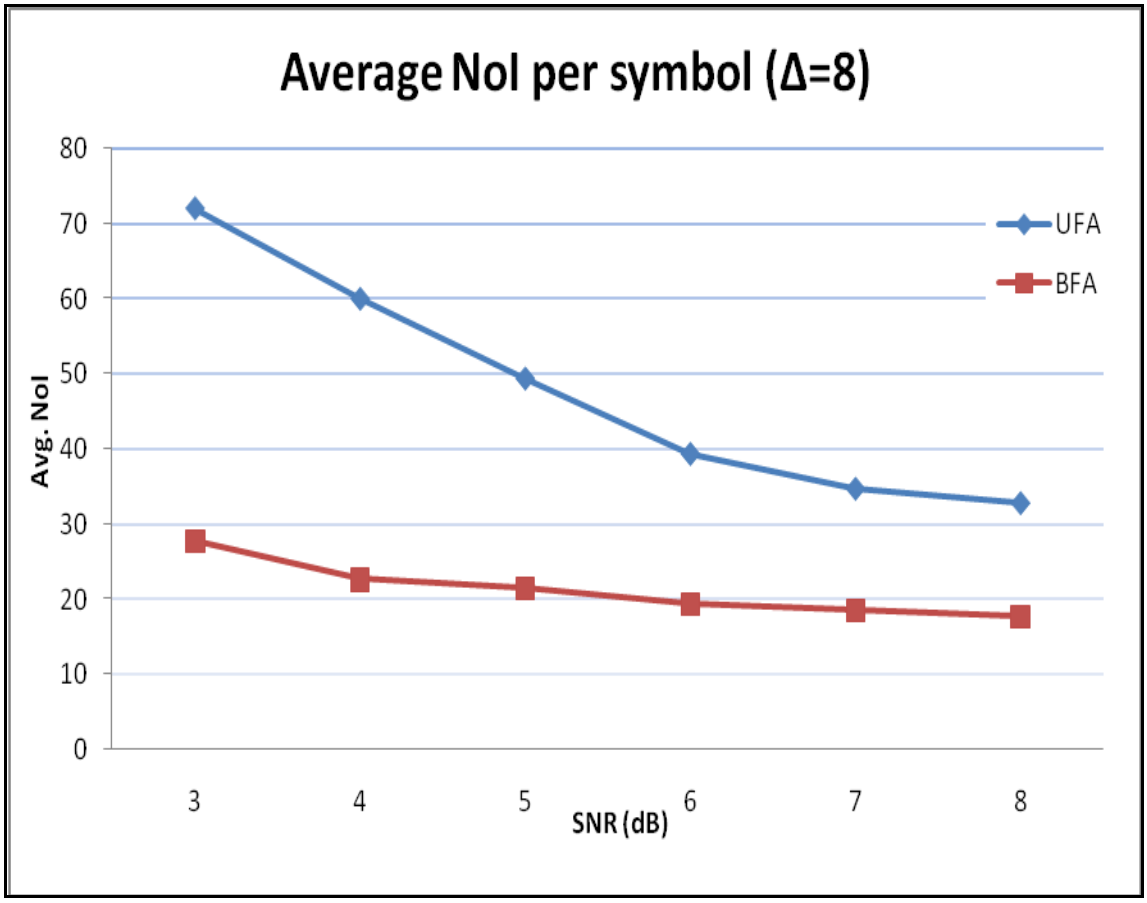
We can note that the throughputs are comparable at lower SNR. In previous researches, the conclusion was that BFA is quicker than UFA for noisy channels while presenting worse throughputs for higher SNR. The results obtained may be explained by two points.

Firstly, the additional merge checks for BFA add an overhead of computation.

Secondly, the workload coming from the increased secondary decoders translates by a drop in occupancy.

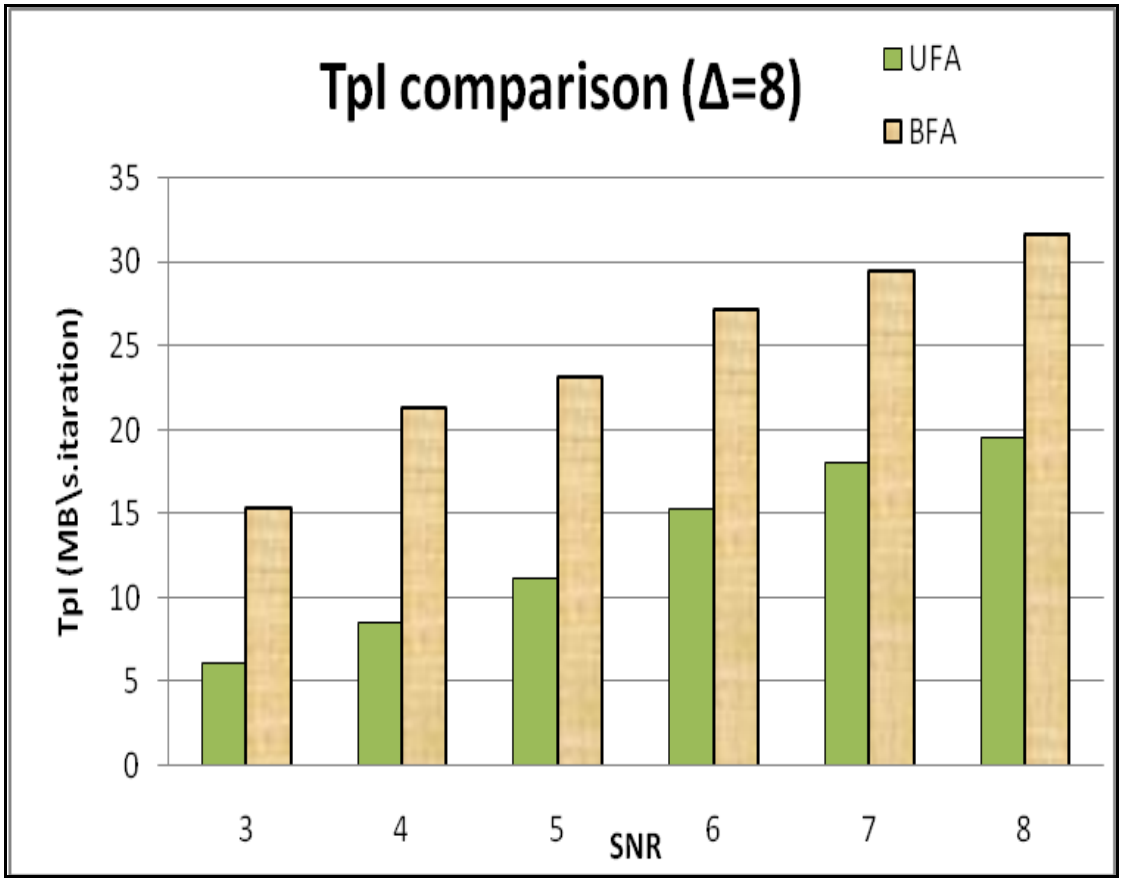
Let us consider the perfect situation where forward and backward decoders merge at the middle of their decoding. In such case, the allocated resource beyond their merge presents a sub optimal memory usage. This situation is actually not an exceptional case problem but a general problem for CUDA implementation of BFA.

Figure 8.7: Avg. NoI per symbol at delta=8



The NoI analysis shows a better result before 6dB, at which UFA and BFA are relatively comparable.

Figure 8.8: TpI comparison at delta=8

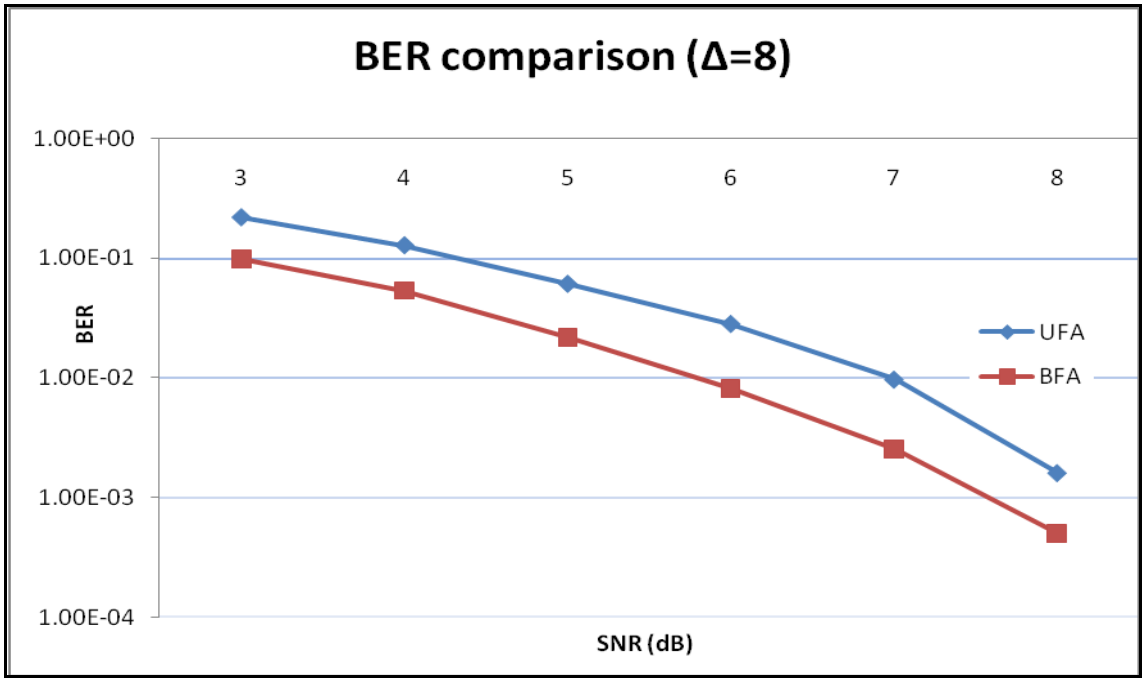


The graph shows us that each iteration in BFA is run with a higher throughput than its corresponding UFA for the same SNR. As SNR increases the improvement decreases but we always have a gain for using a BFA over UFA.

Table 8-5: TpI gain comparison at delta=8

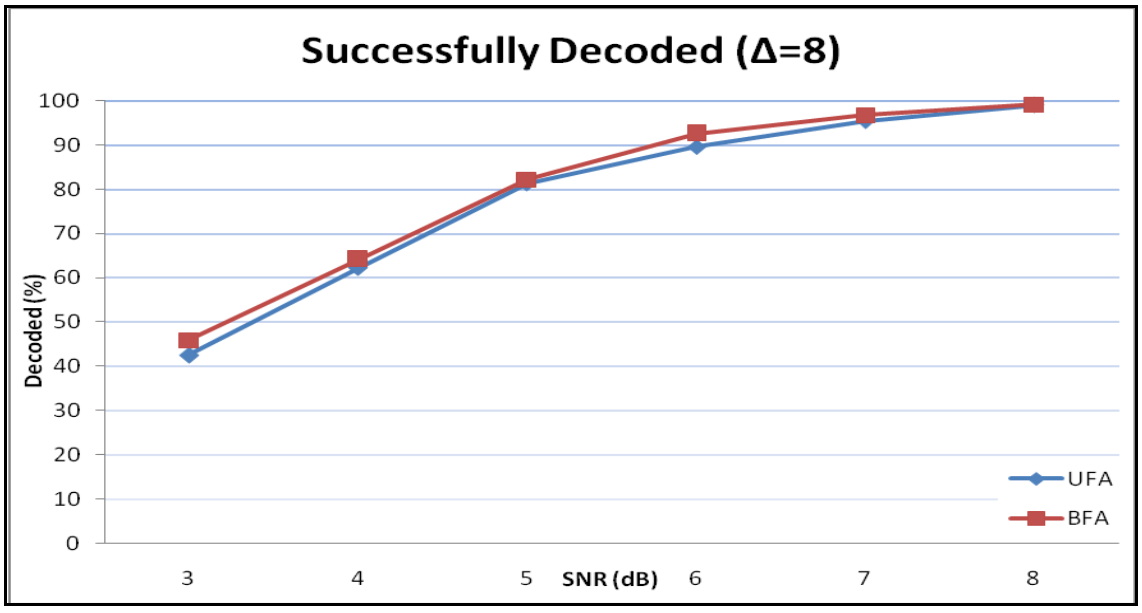
SNR	3	4	5	6	7	8
Gain percent	154.12	150.49	108.45	77.81	63.55	62.51

Figure 8.9: BER at delta=8



While the BER for UFA follows the same pattern in delta 4 or 8, we can notice that the error rate is even improved for higher SNR values of BFA.

Figure 8.10: Decoded percentages at delta=8



The decoded percentage of the same sample is again higher for BFA compared to UFA regardless of the SNR.

8.4 FINAL CUDA CODE

8.4.1 Chosen Parameters

FRAME_SIZE = 32

INPUT_SIZE = 96

The size of the frame was selected as 32, making it equal to the size of an integer, which eases the merge check explained in the previous problem.

MAX_OPERATIONS = 200

OVERLAP_LENGTH = 2

CHECK_FREQ_MASK = 3

KERNEL_LOOPS = 100

This means, overflow was considered when 200 iterations were taken. Also, we check the mask every 3 iterations rather than making a systematic check. Finally, the merge check is done over 2 merging outputs.

Used batch was decoded over 1D grids with 64 blocks and 64 threads. So, the batch size was 64x64x100= 409600 codewords.

Also, notice that the kernels are reused. In practice, it means a single kernel is responsible for consecutive codewords to be decoded. This mitigates the kernel calls computing overhead. For example, we can note a progress of 3 percent when we change the number of codewords from 10 to 100.

8.4.2 Handling Merge Problem

To minimize the effect of systematic merge check of BFA, the following implementation was adopted. While the original paper stated that merge was done in respect to FD and BD's state and depth, the final code checks output and depth. This annuls the need for a final state history array of 16 byte per BFA.

Furthermore, the merge check becomes a simple logic operation rather than a comparison of state, depth values among decoders.

8.4.3 Queue Usage Optimization

Array optimization mentioned in 7.1.1 had the advantage of transforming the history arrays into a smaller array with merely 8 blocks in it instead of INPUT_SIZE blocks. The optimization in memory size at 7.1.1 allowed the following gain:

Table 8-6: History size before array optimization

Array name	Size	Total size
Visit_record	inputSize x 64 x 1bit	6144
State_history	inputSize x stateSize	576
Flag_history	inputSize x 1bit	96
Metric_history	inputSize x metricSize	768

This would mean these arrays would occupy 7584 bits in total per decoder, just a little less than 1 KB \ decoder. However, the optimization changed the required memory to 8x(stateSize+1+metericSize+depthSize) = 8x(6+1+8+5) = 184. Considering that each memory is byte addressable, we have to consider that the actual memory is 192 bits.

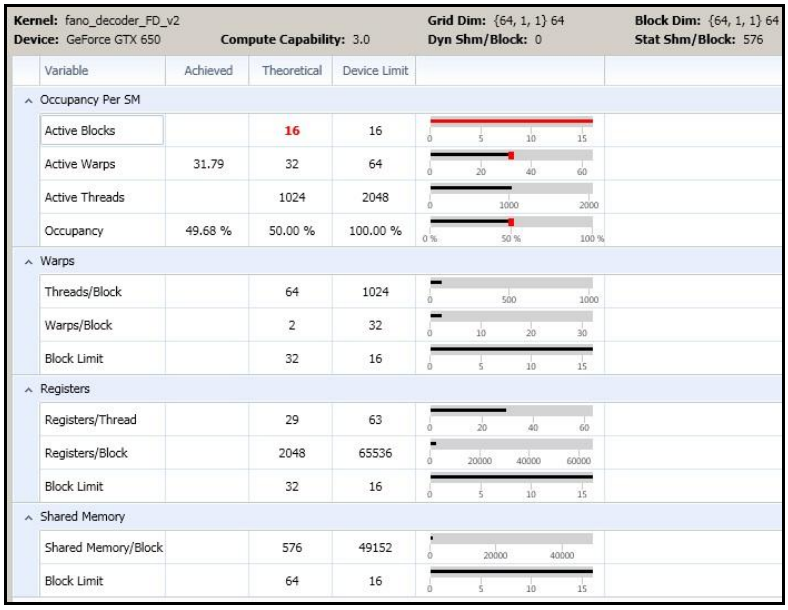
This optimization can be said to allow a swift reduction of 7584/192=39.5.

On the other hand, what this optimization brought was that each time we check whether we are visiting the same {depth, state} pair again, we need to look to the entire queue. Although this is a small constraint, it definitively adds computational overhead to both forward and backward moves. In other words, we changed move to a O(n) operation.

To overcome this problem, the following was considered. At each time in the queue one may only have 8 distinctive depths. We can actually use this feat to use the queue as an indexed queue. This would make the queue a bit like a database where we use the last 3 bits of the depth as key and that makes the move forward or backward O(1) operation.

8.4.4 CUDA Analysis

Figure 8.11: UFA CUDA occupancy analysis

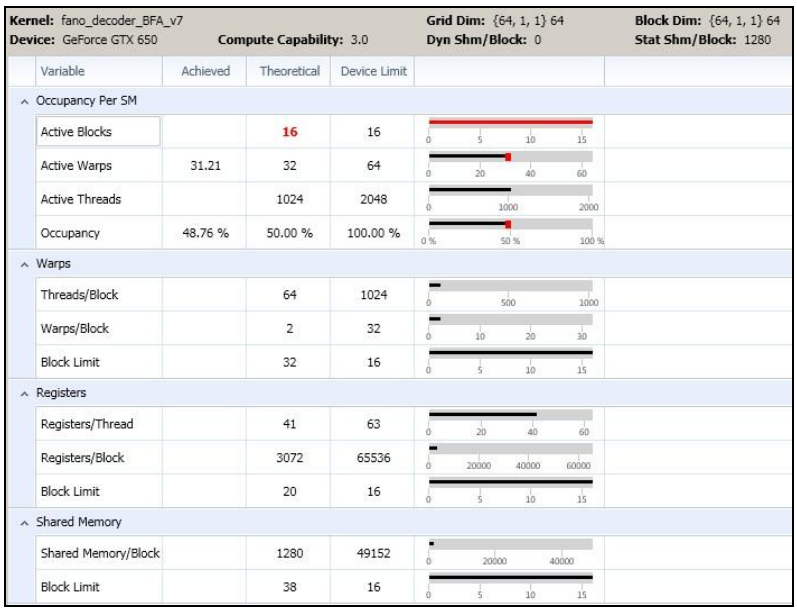


Kernel: fano_decoder_FD_v2 **Grid Dim:** {64, 1, 1} 64 **Block Dim:** {64, 1, 1} 64
Device: GeForce GTX 650 **Compute Capability:** 3.0 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 576

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		16	16
Active Warps	31.79	32	64
Active Threads		1024	2048
Occupancy	49.68 %	50.00 %	100.00 %
Warps			
Threads/Block		64	1024
Warps/Block		2	32
Block Limit		32	16
Registers			
Registers/Thread		29	63
Registers/Block		2048	65536
Block Limit		32	16
Shared Memory			
Shared Memory/Block		576	49152
Block Limit		64	16

Source: NVIDIA Nsight, 2013

Figure 8.12: BFA CUDA occupancy analysis



Kernel: fano_decoder_BFA_v7 **Grid Dim:** {64, 1, 1} 64 **Block Dim:** {64, 1, 1} 64
Device: GeForce GTX 650 **Compute Capability:** 3.0 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 1280

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		16	16
Active Warps	31.21	32	64
Active Threads		1024	2048
Occupancy	48.76 %	50.00 %	100.00 %
Warps			
Threads/Block		64	1024
Warps/Block		2	32
Block Limit		32	16
Registers			
Registers/Thread		41	63
Registers/Block		3072	65536
Block Limit		20	16
Shared Memory			
Shared Memory/Block		1280	49152
Block Limit		38	16

Source: NVIDIA Nsight, 2013

The analysis shows that we are reaching the theoretical limit in terms of occupancy but not the device limit. The speeds are respectively 650MB\s and 550MB\s for UFA and BFA.

8.4.5 Final CUDA Communication Schema

In this version of the code, a particular attention was given to the use of the fastest possible busses. In that intend, both global and local memory usages were avoided.

Since look up tables are by definition unchanged, they were placed in the constant memory for the fastest possible reach. Parameters were also put in the constant memory. This is possible since CUDA's constant memory is a memory that will not change in the scope of a kernel call. However, we may change that memory between calls, for instance by changing the parameters for the next call.

Until now, the same input was sent to every kernel for testing. This allowed the programmer to send the input into constant memory for fast access. It had a relatively small size which makes simultaneous tests for thousands of codewords impossible. However, using the texture memory as linear memory provides a solution to this problem. This type of memory is optimized for reading one block of memory and caching the adjacent blocks. It was an adequate solution to our algorithm since the algorithm only moves to neighbouring blocks. This allows all inputs to be sent to memory hence all kernels will be able to process different frames at a time.

Registers kept queues while shared memory kept different temporary variables.

Figure 8.13: Memory transfers for 2 blocks with 2 threads per block

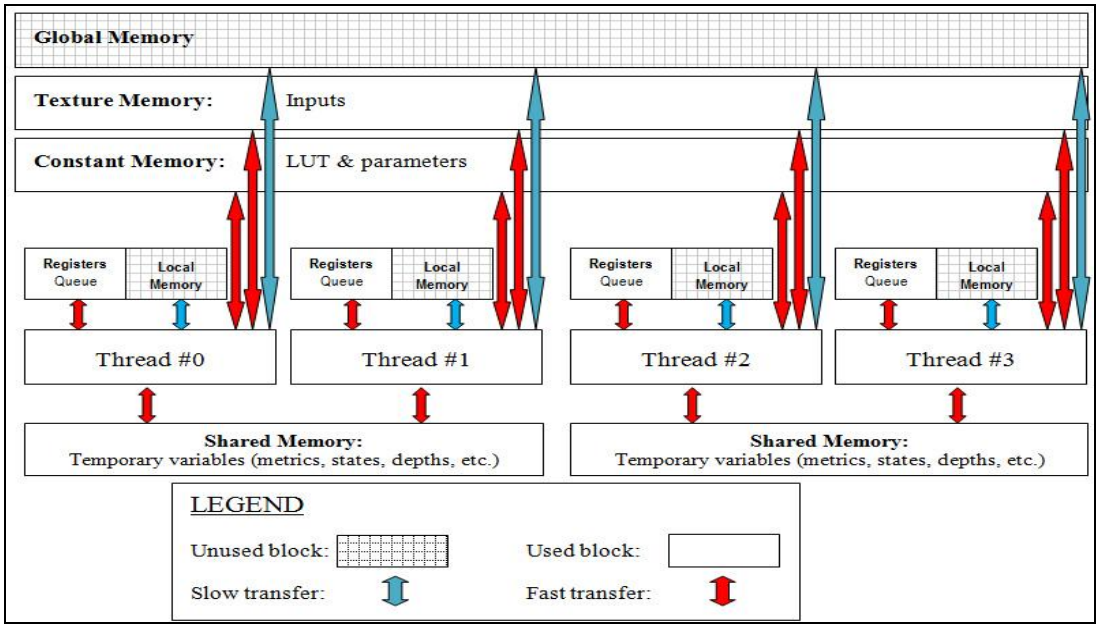
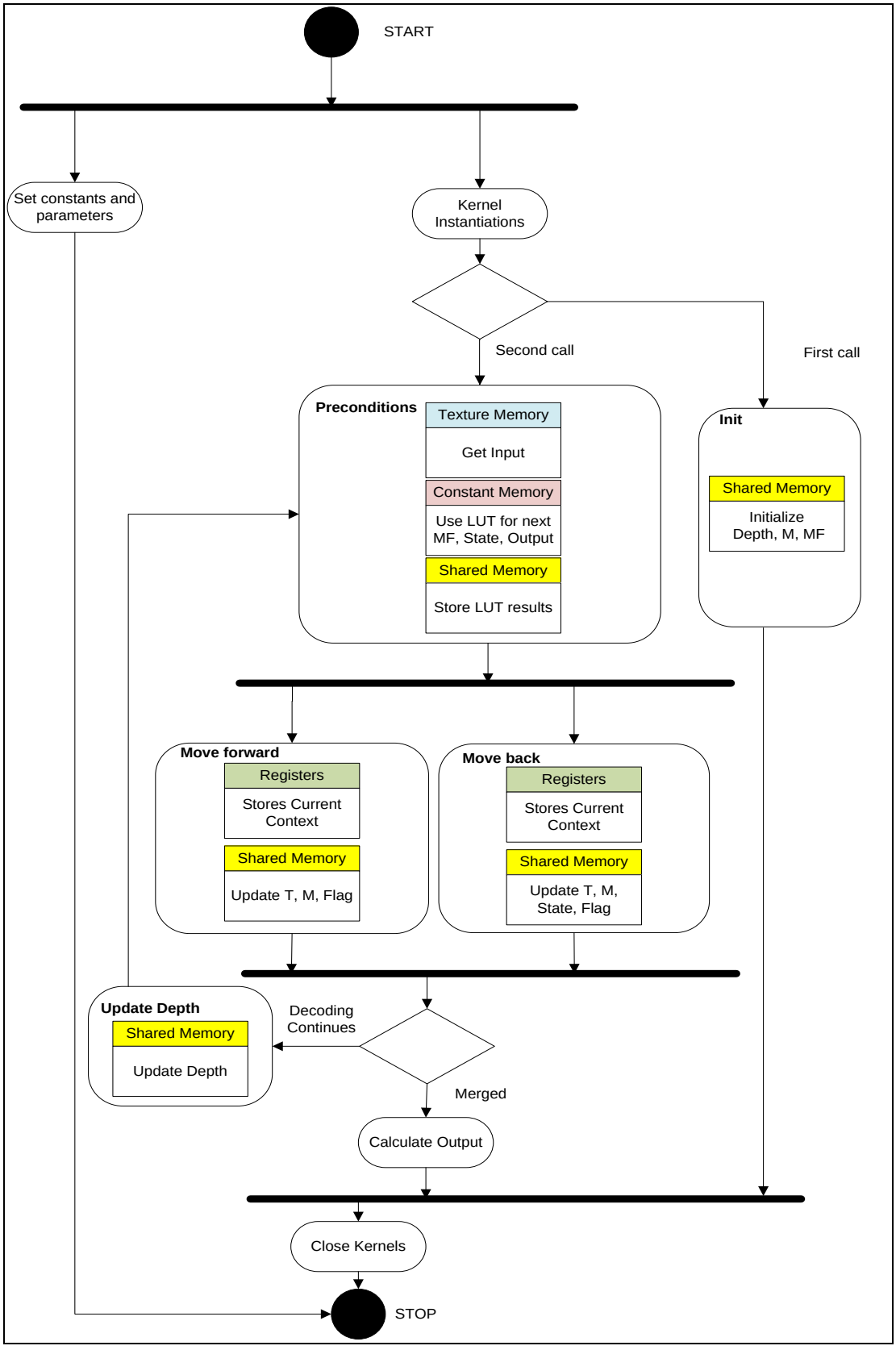


Figure 8.14: Functionality - Board parts mapping



8.4.6 Improvements Effects

Table 8-7: Improvements gains

Optimizations	Throughput (in MB/s)	Version Improvement
Version 1: 7.1.1 & 7.1.3 & 7.2.1	49	
Version 2: Length sent as constants	120	145 percent
Version 3: Reuse of variables for loops	148	23 percent
Version 4: Frame size made a macro instead of being variable	220	49 percent
Version 5: 8.4.1&0&8.4.3	550	150 percent

Version 1 is simply the result of optimizations in Chapter 7.

Version 2 & 4 indicate that CUDA processes at a greater rate when the kernels are deterministic and have little or no variable involved.

Version 3 allows us to determine that each variable allocation induces processing overhead. In this optimization, loop counter was reused when leaving that loop by setting it to 0 for first visit and 1 to 9 for otherwise, which made the presence of a Boolean flag allocation for this purpose redundant.

In Version 5, most of the improvement was due to the indexed queue which again added more deterministic characteristic to the computation.

8.4.7 BFA With Dual Threads Per Codeword Implementation

The algorithm was transformed into parallel code with each decoder working in synchronization with its pairing decoder. The following are updated results with this addition.

Figure 8.15: BFA dual threads throughput

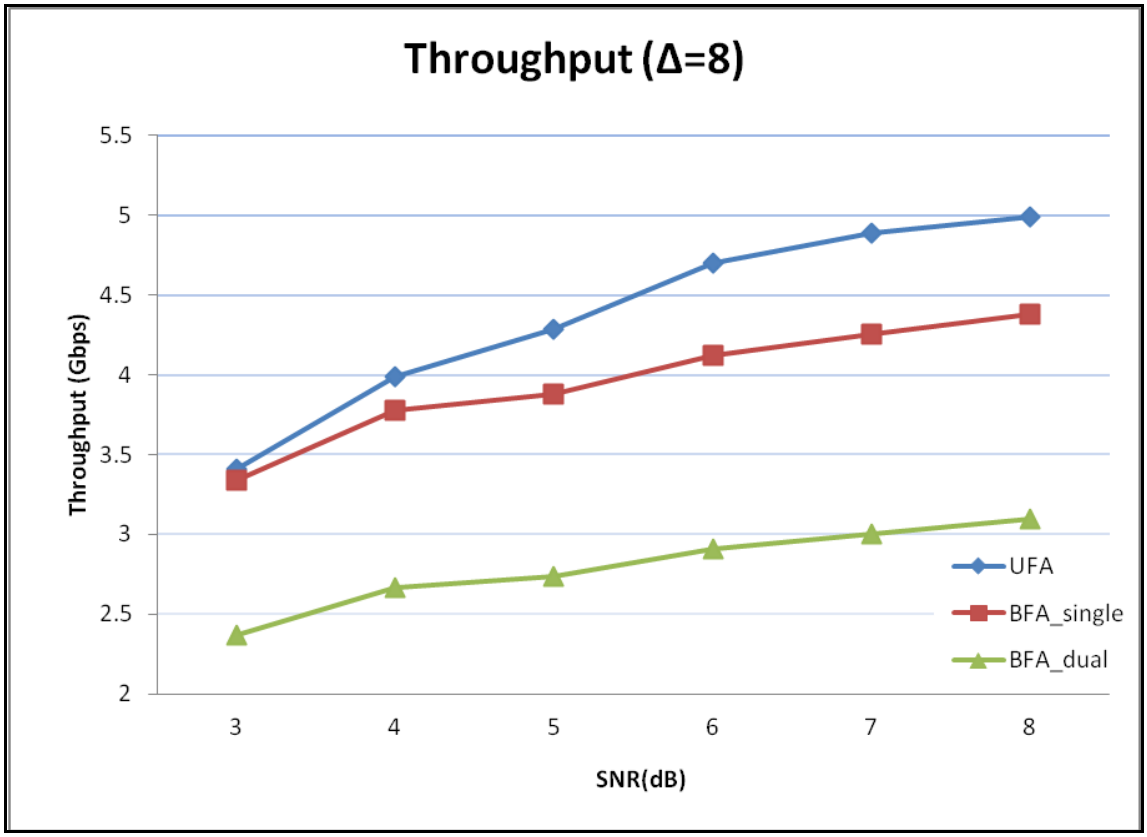
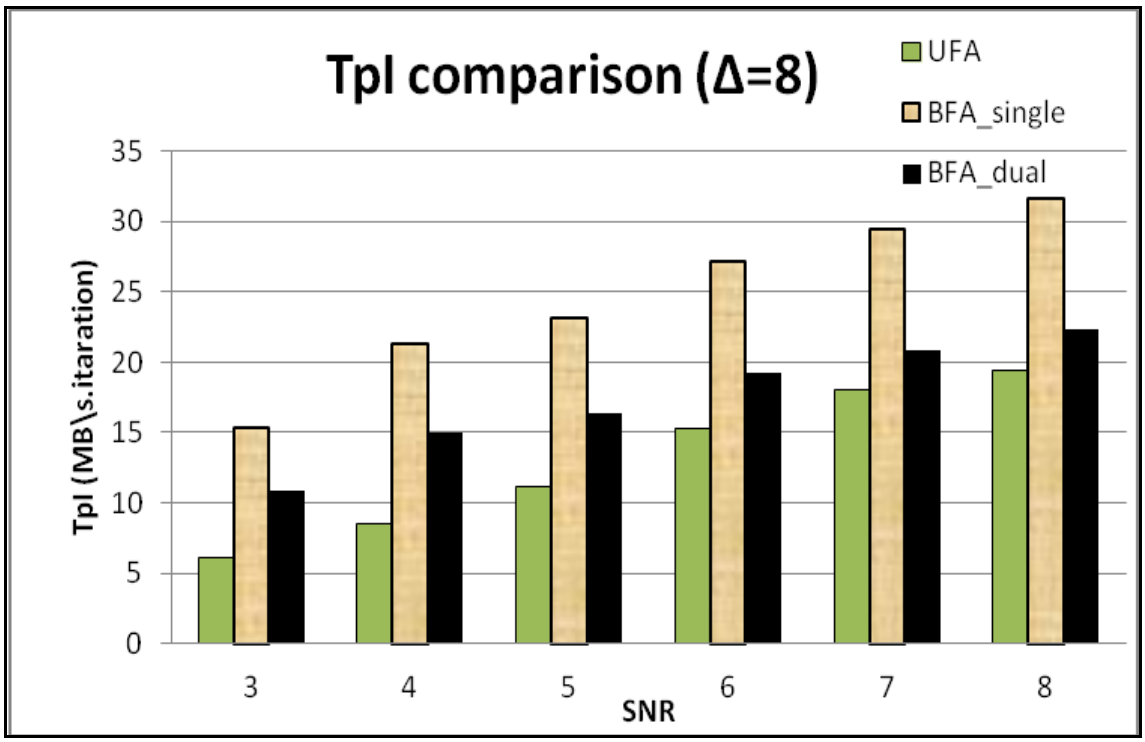


Figure 8.16: BFA dual threads TpI comparison



8.4.8 BFA Result Discussion

In the following console output, we can see that in C++ the same codeword (SNR=8 and delta=8) was decoded for both UFA and BFA. The time spent to decode is greater for BFA.

We can explain this by two factors. A new check is added for merging. This has a processing overhead. In addition, FD and BD are decoded iteratively in this BFA code.

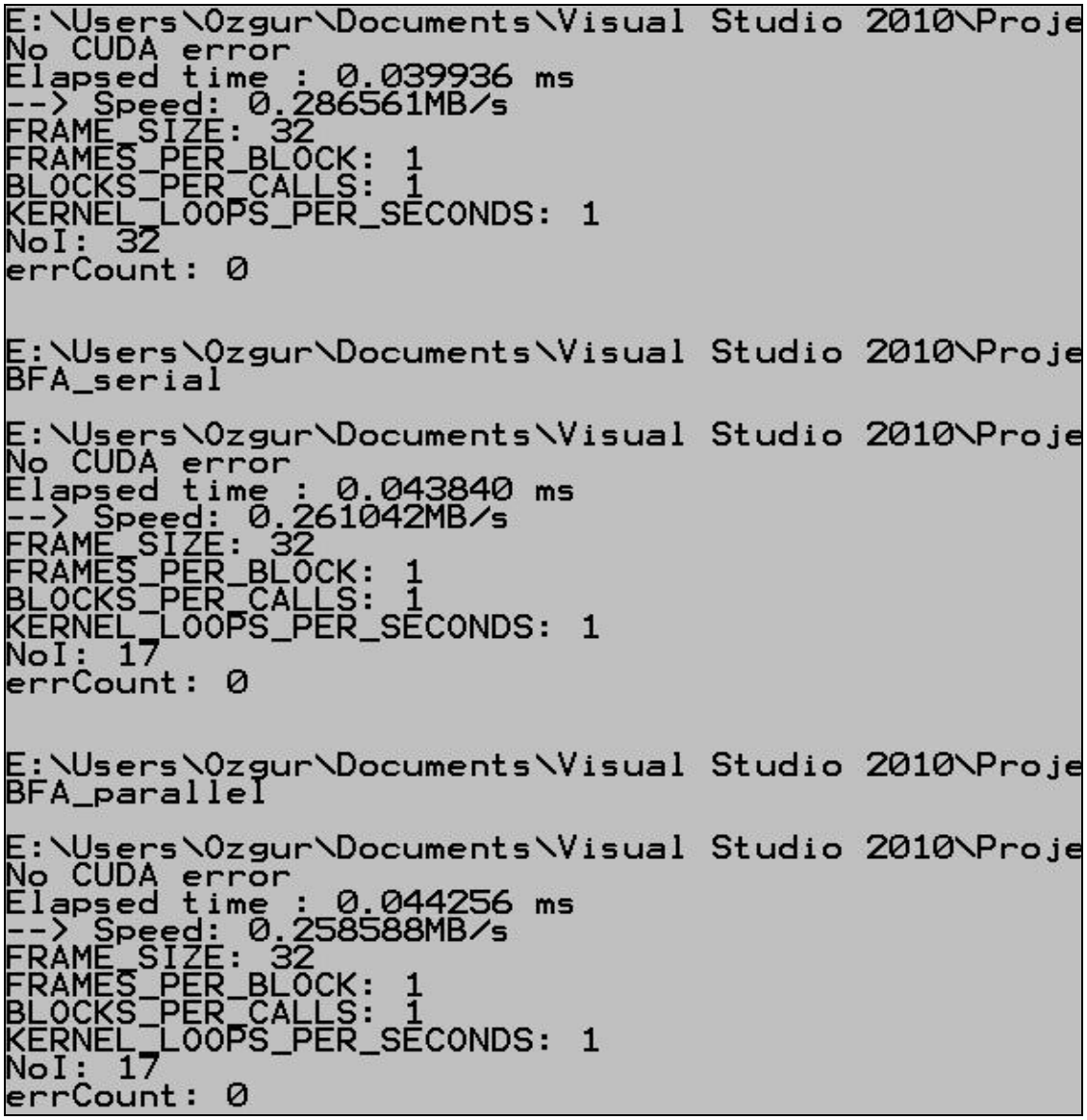
Figure 8.17: C++ UFA and BFA comparison

```
ozgur@ozgur-Satellite-A660:~$ g++ UFA_shorten_test.cpp
ozgur@ozgur-Satellite-A660:~$ ./a.out
difference is 33 microseconds
Count computation: 32
Sample size: 26
Error count: 0

ozgur@ozgur-Satellite-A660:~$ g++ BFA_shorten_test.cpp
ozgur@ozgur-Satellite-A660:~$ ./a.out
difference is 40 microseconds
Count computation: 18
Sample size: 26
Error count: 0
```

In the same logic, the CUDA version may be discussed. The following console is the output of another codeword first using UFA, then using BFA iteratively (like C++ version) and finally in parallel. In the test, only a single thread was used for the UFA. In the same manner, a single thread was used for BFA_single. As for BFA_dual, it uses one thread for FD and another thread for BD.

Figure 8.18: CUDA UFA and BFA comparison



In the above output, we first see the duration for UFA, and then we see the duration for BFA_single and finally BFA_dual.

We note that UFA is once again quicker than BFA. It seemed rather surprising at first since it goes against the assumption that we should see a drastic improvement for duo threading CUDA implementation compared to single threaded BFA. However, we have to keep in mind that the algorithm cannot be divided into pieces. It is due to its serial nature, thus the definition of the Fano algorithm as a sequential decoding algorithm.

This means that the dependencies involved prevent effective data parallelism. Data parallelism meaning several threads collaborating to work on the same data. Only task parallelism can be done both in UFA and BFA. This type of parallelism is when threads are working in parallel in different set of data or input codeword in our case.

As noted from the improvement effect discussion, CUDA delivers better results when dealing with deterministic datasets. When FD and BD check whether or not they have already merged, it imposes an intrinsic dependency. It is precisely this requirement of BFA -which UFA does not have- that adds latency to these final results, as can be seen here-above.

Table 8-8: Cuda analysis comparison

	UFA	BFA_single	BFA_dual
Active Blocks	16	16	16
Active Warps	32	32	64
Active Threads	1024	1024	2048
Occupancy percent	49	48	86
Threads/Block	64	64	128
Warps/Block	2	2	4
Block Limit	32	32	16
Registers/Thread	29	41	32
Registers/Block	2048	3072	4096
Block Limit	32	20	27
Shared Mem/Block	576	1280	1664

As a final comment, we note in the above table that the active blocks are always limited to the same value. However, the number of registers and shared memory used per block increases from UFA to BFA when using a single thread per codeword, then increases further when BFA uses two threads per codeword. Therefore, the overhead from memory transactions increases in that same order, hence the speed drop demonstrated by the throughputs.

7. CONCLUSIONS

In this thesis, both C++ and CUDA implementations were done with the latter showing the best throughput.

UFA and BFA algorithms have a relatively small memory requirement but still too high to be effectively and properly parallelized as can be seen in the CUDA analysis. For this reason, the board's limitation prevented the verification of Xu et al.'s research in which the authors used dynamic scheduling for 8 decoders in 2011.

Improvements were introduced by using look-up tables, which avoided calculating the metrics and other temporary parameters on the fly. Also, we note that instead of using complete historic records, 8 previous records queue were used to drastically reduce the memory requirement.

655MB\s (5.0Gbps) was reached for CUDA's UFA implementation (output throughput) while 12.75MB\s (100Mbps) was previously achieved in a FPGA implementation (Kakacak, 2012). 550 MB\s (4.4Gbps) was reached for CUDA's BFA. The results were in pair with the result of the previous BFA related researches by Ran Xu in which, BFA offers better BER compared to UFA regardless of the SNR. In the same way, BFA completes decoding quicker for lower SNR. However, at highest SNR, the UFA is preferred. The reason for CUDA's UFA showing higher speed was due to the additional memory transactions and merge check from BFA.

The following difficulties were met in this work. Firstly, this thesis presented several alternatives: The line of codes written should be between 10000 and 20000 lines of codes in total. Cross-checking their respective speed was difficult especially in terms of CUDA benchmarking. C++ implementation had around 20+ versions, 40+ different codes for CUDA.

Also, the fact that the resource of the CUDA board is relatively small dictates the upper boundary we can reach for historical keeping algorithms such as Fano algorithms. In such case, we are confronted with the well known problem of parallelism, that is, the memory being a limiting factor. To overcome this, we need to perform the same

operation over many threads, meaning decoding many codewords at the same time, which in turn requires as many temporary variables to be kept in memory as codewords.

As future work, we may try to use several CUDA compliant boards on the same machine. A possible implementation could be assigning two different NVIDIA units each one decoding a different set of codewords. Another direction in which this thesis could be broadened in the future would be in the field of shortening the memory even further, as this is a critical issue in current implementations.

One other future work would be to study the reasons why BER in both UFA and BFA is slightly higher as compared to the original Ran Xu researches and the algorithm in MATLAB.

Finally, even though testing has already been conducted in 1000 distinctive codewords for each SNR, it would still be a good idea to make a real-time testing in real environment, for example using WirelessHD.

REFERENCES

Books

John B. Anderson, S.M., 1983. *Source and Channel Coding: An Algorithmic Approach*. New Jersey: Prentice-Hall.

Kirk, D. & Hwu, W.-m., 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Burlington: Morgan Kaufmann.

Peter A. Beerel, R.O.O.M.F., 2010. *A Designer's Guide to Asynchronous VLSI*. Cambridge: Cambridge University Press.

Periodicals

D. Haccoun, M.F., 1975. Generalized stack algorithms for decoding convolutional codes. *Information Theory, IEEE Transactions on Nov*, **21**(6) pp638-651.

Elias, P., 1955. Coding for noisy channels. *IRE Convention Record*, **3**(4):37-46.

Fano, R., 1963. A heuristic discussion of probabilistic decoding. *Information Theory, IEEE Transactions on*, **9(**2) pp64-74.

Forney, G.D..J., 1973. The viterbi algorithm. *Proceedings of the IEEE*, **61**(3) pp268-278.

Forney, G. & Bower, E., 1971. A High-Speed Sequential Decoder: Prototype Design and Test. *Communication Technology, IEEE Transactions on*, **19**(5) pp821-835.

Hamming, R.W., 1950. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, **29**: pp.147-160.

Jelinek, F., 1969. A fast sequential decoding algorithm using a stack. *IBM J. Res. and Dev.*, **13**, pp. 675-685.

Li, K., 1991. Bidirectional sequential decoding for convolutional codes. In *Communications, Computers and Signal Processing, 1991., IEEE Pacific Rim Conference on*. Vancouver, 1991.

Omura, J., 1969. On the Viterbi decoding algorithm. *Information Theory, IEEE Transactions on*, **15**(1) pp:177- 179.

Ran Xu, T.K.G.W.K.M.C.D., 2009. Bidirectional Fano algorithm for high throughput sequential decoding. In *Personal, Indoor and Mobile Radio Communications, 2009 IEEE 20th International Symposium*. Tokyo, 2009.

Ran Xu, T.K.G.W., 2010. Throughput improvement on bidirectional Fano algorithm. In *IWCMC '10 Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*. New York, 2010. ACM.

Shannon, C.E., 1948. A mathematical theory of communication. *Bell System Technical Journal*, pp.**27**:379-423, 623-656.

Sutter, H., 2005. The Concurrency Revolution. *C/C++ Users Journal*, p.23 (2).

Sutter, H., 2011. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, p.**30**(3).

Viterbi, A., 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Trans*, **13**(2) pp:260-269.

Xu, R. et al., 2011. High Throughput Parallel Fano Decoding. *Communications, IEEE Transactions on*, **59**(9) pp2394-2405.

Zigangirow, K.S., 1966. Some sequential decoding procedures. *Probl. Peredachi Inf.*,.**2**(4) pp 13-25.

Others

Ahmet Kakacak, T.K., 2012. *Design and implementation of high throughput bidirectional Fano decoding*. Istanbul.

ArrayFire, 2012. *ArrayFire CUDA Python*. [Internet] (Published 2012) http://www.accelereyes.com/arrayfire/python/ [accessed 1 July 2012].

Anon., 2012. *Java bindings for CUDA*. [Internet] (Published 2012) http://www.jcuda.de/ [accessed 10 December 2013].

G. D. Forney, J., 1967. *Final report on a coding system design for advanced solar missions*. Moffett Field: NASA Ames Res. Ctr.

HDMI, 2012. *Knowledge Base*. [Internet] (Published 2012) http://www.hdmi.org/learningcenter/kb.aspx [accessed 18 September 2012]

ISO, 2011. *ISO/IEC 14882:2011 (TR1)*.

Khronos Group, 2012. *Conformant Products*. [Internet] (Published 2012) http://www.khronos.org/conformance/adopters/conformant-products/ [accessed 29 November 2012]

Khronos Group, 2012. *OpenCL - The open standard for parallel programming of heterogeneous systems*. [Internet] (Published 2012) http://www.khronos.org/opencl/ [accessed 19 October 2012]

Khronos, 2011. *OpenGL: the industry's foundation for high performance graphics.* [Internet] (Published 2011) http://www.opengl.org/registry/ [accessed 19 October 2012]

Microsoft, 2010. *GPGPU Computing Horizons: Developing and Deploying for Microsoft Windows*. [Internet] (Published 2010) http://download.microsoft.com/download/A/6/C/A6C0223B-9460-4346-8DC0-B6BCFD9269B4/MSFT_GPGPU_whitepaper_FINAL.PDF [accessed 19 October 2012]

Nobile, M., 2011. *CUDA, random numbers inside kernels*. [Internet] (Published 2011) http://aresio.blogspot.com/2011/05/cuda-random-numbers-inside-kernels.html [accessed 19 October 2012]

NVIDIA Corporation, 2009. *NVIDIA GeForce 310*. [Internet] (Published 2009) http://www.nvidia.co.uk/object/product_geforce_310_uk.html [accessed at 19 October 2012]

NVIDIA Corporation, 2012. *CUDA C Programming Guide*.[Internet] (Published 2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf [accessed 7 March 2012]

NVIDIA Corporation, 2012. *GTC2012: Programming Heterogeneous*. [Internet] (Published 2012) http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0631_Monday-Programming-Heterogeneous.pdf [accessed 7 March 2012]

NVIDIA Corportation, 2012. *TESLA GPU COMPUTING SOLUTIONS FOR SERVERS*. [Internet] (Published 2012) http://www.nvidia.com/object/tesla-servers.html [accessed at 19 December 2012]

NVIDIA Corporation, 2012. *CUDA-Accelerated Applications*. [Internet] (Published 2012) http://www.nvidia.co.uk/object/cuda_app_tesla_uk.html [accessed 11 June 2012]

PathScale, 2011. *ENZO 2011 GPGPU Solution | PathScale*. [Internet] (Published 2011) http://www.pathscale.com/enzo [accessed 18 September 2012]
Portland Group, 2012. *Portland Group*. [Internet] (Published 2012) http://www.pgroup.com/ [accessed at 19 October 2012]
Sailer, T., 1999. *randn() modified from*. [Internet] (Published 2012) http://pastebin.com/uxYpZJYE [accessed 7 March 2012].
WHDI, 2012. *WHDI About*. [Internet] (Published 2012) http://www.whdi.org/About [accessed 11 Feburary 2013]

APPENDICES

APPENDIX A.1: Test Vectors

MATLAB, C++, CUDA time comparison with SNR=4, delta=8, R=1/3 used the following test vectors:

111011111010001000111000
Sample size: 2

000000100111011111110011110111
Sample size: 4

00000000011101111111000110011100000000000
Sample size: 8

1110110000100101001111001100001011001000011111011011010001010111111
Sample size: 16

1111000111000111000111101111100001100110010100011000110001101011000010101110011111110000101000101010001000011111000
Sample size: 32

1110110001010011101011100000000100000100100111011100011010111111011101110011101111011111011110111011001000000000100011110001010000100111101111110000010100101001111000011011100110011111111100011100110011111
Sample size: 64

1000110110010101110010000110111010100111111101001000110111111011011010000100000001100101000111111011101000000000101011110010011001000000001111001010000111011101000000110111000110001011100101010100011100010100100000000001111011101000101010100111000111011001101010000011001001101100001010000010101001011
Sample size: 128

1000000001110110010100100010111000001001111000101011001011010101101100011110011110100101101111001100000101001111001111011011100110110111100001000010110100010011110011101111111001001000111111000011100010011000100011111110000000000110001000000110000010001101000010011110100000010001100100101100100000111001001010010100100111010011011101110000011000010010011000011011101010100000110011010000000011011111011100001010011001001011100000010101000100001010100101011111000111110011010110010010001101010001111001110111111100000101110011111100111101011110100101101111001100000101000111010100101111010011101000000010101101110000001010001010111011110001011101111000011010110100110001100011001011111000101000111010100101111010011101001110010000010101101110100101010011011100000000010101101110000001010001011101111000101110111000011010110100110001100011001011111000101000111010100101111010011100
Sample size: 256

1001111000110110001000010111011100111110110101001001111100010110110110010000 1
11110000000001010011110010011001001101111000111100011010111001000001101001000 1
1110011010110010011010001101110111000110110110101001110100110111000011111010 1
10000101000010011010010100110111001001101101110001010110000101011000011100000
0111100100110010100000111111111000110011101101110011100111110001100111110100
0110101010110101111010100011101011000000000110011111011011110100000000011011
1100100001101110111010100111110000101111001000010110011000101111101100000110 0
1100110100110000010110010011001001011100111110001100000011000110101101111111 1
1011001001111001001100100101110000010100110111011001010111100100000101010000 1
10000111100001111010010010000100011100010010110011110000000100111110010000010
1111011000001000111101000000000010001100101011111011100010100100111001101110 0
0000011101101111100110110110111001011000001100110110011000100011100110110011 1
0011101000111111110100100001101111010110001001000010110011001000000101010101 0
0110110011111011011110001010100111011001110000011100111100000001101000111011 0
1010001111101111010101000110001111110010111001100001010011111101100110110010 1
0100000010110011010111011100011110101110010110111110101010101110001011000110 1
1100111000110001101011100001010001101111001101000001110001100010010111111110 0
0001111001110011010010010010010001101010101101000001111001100000001101100101 1
1010100011000000111101010011001000100100001000000110010011110001011011100011 0
0000000010110111011010110100111001010101101101110100110110011111010011110111 1
10111101011101
Sample size: 512

1111001111101100100100000000011111001100101111100000010010001110010110001110 0
1101111100001011000110010110100101111001001011101101000101100100000110101101 1
1100001110110101110011011010000011101110000100101100110011010001001000001110 1
1111100010011101001101011100001010001111111111000011000011011001101011111100 0
1100010000000100110011100101001010001101011111011110010100000101101011000111 10
1101111011100101000101000010110001110000111110110001100011001100110101110111 0
0011010101101010110000100010101010001010000110001111110000101111101001100100 1
1001001000010011000010010110010111001110011111011111001111010001110010101111 1
1001110110101010011010111101110100001101010110111100100101010000011100101110 0
0010000000011010001000000001000101111111001110011100010010111110000010100001 01
1100101000011110111111001010100111111001001100100110000101001001001101001110 1
0001101001011100101111010000000111110001011111011111000100110101011110011101 10
1110000000101010101111010000000011101010101000000010101111101001101100100100 0111
1111100111100111111110001001101111110001011010001000111011111101111111110010 01
1110010100101010101001010101111001111000110101110100001000110100101001110011 0
1010000000011000100011001010010110100110011100110100000000101101100100111000 01
1010111111110111101100011011110111111010001111011101111011100011001010000101 0
1100010010000111110101100010110011010100100011101100001111000110111110100010 0
0011000011101110000001010100001010110111011010111111111101101101011000000111 0110
1000000011111110101100100101000110111100110000010001100111110000101000011010 0
1111100000000000110110000101100010111110110111100101011111111101010011011101 10
1100101010000100000000010101000100011101011101111001100001010001000011000110 1
0000100010000100010000000011111100100100011111010101101001001000000111001011 101
1111010111000001111001011111001110101110111100110001010001011100110011111010 0
1100011111111010011001111111100101100001100011100010011000100110111001001111 0
0100000001110101011000110011000110111100000000100100011001011001010111001000 1
1100010011000000111100010101000011001001110001111000010100010010011111000001 1
0100100111000111100000101111111001001111010101101011001111101011111010110100 00
1000001111100000110100110101000110111010111011001100001000010100000111000100 1
1101000001011100001010000000111010001100110001111000100111110010010001110010 01

00011111001011100100011111111100111010011111110110100011111110101100100110110
10001010010100110101110101110100110110000011100111111001100100010100010110101
01000111000110001101100011110001111001001100010010010111001100011010010111000
00100000000000010110101111011000001001101101010011000111111011010100101111000
11001110011001100110000011010000101010011101000001110000100100101111110010000
00110110000100010111111110101100110000000000010010001101111011010101111010011
10010010111101111010010100110110000000010001001000001011110110001010011010010
11101101110101010001100001110101101111000111110100010011101100101010000011100
01100111010100000000000001100111100011010000100001100001110000011110011001101
01101001001011110100110011000011001100101000100011110001000010100001101001111
1001000000
Sample size: 1024

APPENDIX A.2: Code Dependencies

Table 1: Dependencies for BFA

Variable	Size (bits)	Copies \ decoder	Variable changed in function	Depending variables
isTail	1	1	operateOnPreconditions	Depth, length_frame
state_next	6	2	operateOnPreconditions	isForwardDecoder, LUT_stateNextFinder, State_current
state_next_output	6	1	operateOnPreconditions	metric
MF	8	1	operateOnPreconditions	isTail, metric
metric	8	2	operateOnPreconditions	receive_bit,state_next_output, M0, M, M
state_order	2	1	operateOnPreconditions	receive_bit,state_next_output, M0, M, M
state_input	1	2	operateOnPreconditions	receive_bit,state_next_output, M0, M, M
moveForward	1	1	operateOnPreconditions	MF, T
previousContexts	24	8	moveForward, moveBackward	M, State_current, Flag_LFNBi contextIndex
contextIndex	3	1	moveForward, moveBackward	contextIndex
state_current	6	1	moveForward, moveBackward	State_order, previousContext
output	2	1	moveForward	moveForward, State_input, Flag_LFNB
depth	8	1	moveForward, moveBackward	moveForward, State_input, Flag_LFNB
receive_bit	3	1	At each iteration	moveForward, State_input, Flag_LFNB
visit_record	1	InputSize	moveForward	Depth
T	8	1	moveForward	T, MF, delta
M	8	1	moveForward	MF
Flag_LFNB	1	1	moveForward, moveBackward	previousContext
Flag_overflow	1	1	checkOverflowOrMerge	previousContext
Merging_depth	8	1	checkOverflowOrMerge	Depth, Overlap_count, Overlap_length, Merging_depth
Overlap_count	3	1	checkOverflowOrMerge	State_history, Depth, length_frame

APPENDIX A.3: Calls Analysis

The following analysis was taken in account while optimizing. Here, we can see that most of the time is spent either when calculating metrics or moving. This is the reason behind the extensive LUT usage.

Table 1: Call counts

Function	Call count	Note
fano_decoder	1	Kernel instantiations
initBFA	1	Kernel initiation
getMergedOutputIntoFD	1	Kernel termination. The function makes simple logic arithmetic to put the merged output to forward decoder.
isTail	28	This function simply checks Depth < InputSize-6 which was rendered useless with the input reorganization.
checkOverflowAndMerge	30	The check first check overflow then check merge. This was a major time consuming time when benchmarked. The optimization done for merge solved this problem.
moveBackward	34	A queue iteration until the appropriate depth and tightening if needed.
moveForward	84	We can note that move forward was more often called than moveBack.
setOutputBit	84	This function is called systematically each time moveForward was called.
operateOnPreconditions	117	Metric and temporary variables calculation. This function is called for each iteration.

APPENDIX A.4: Move Back Analysis

The following analysis was taken in account while optimizing back tracing.
In the below figure, the distribution of back traces over 100000 random vectors at SNR=3 and delta=4 are displayed. In the following table are shown the back traces and their respective percentages in those random vectors.

Figure 1: Back trace distributions

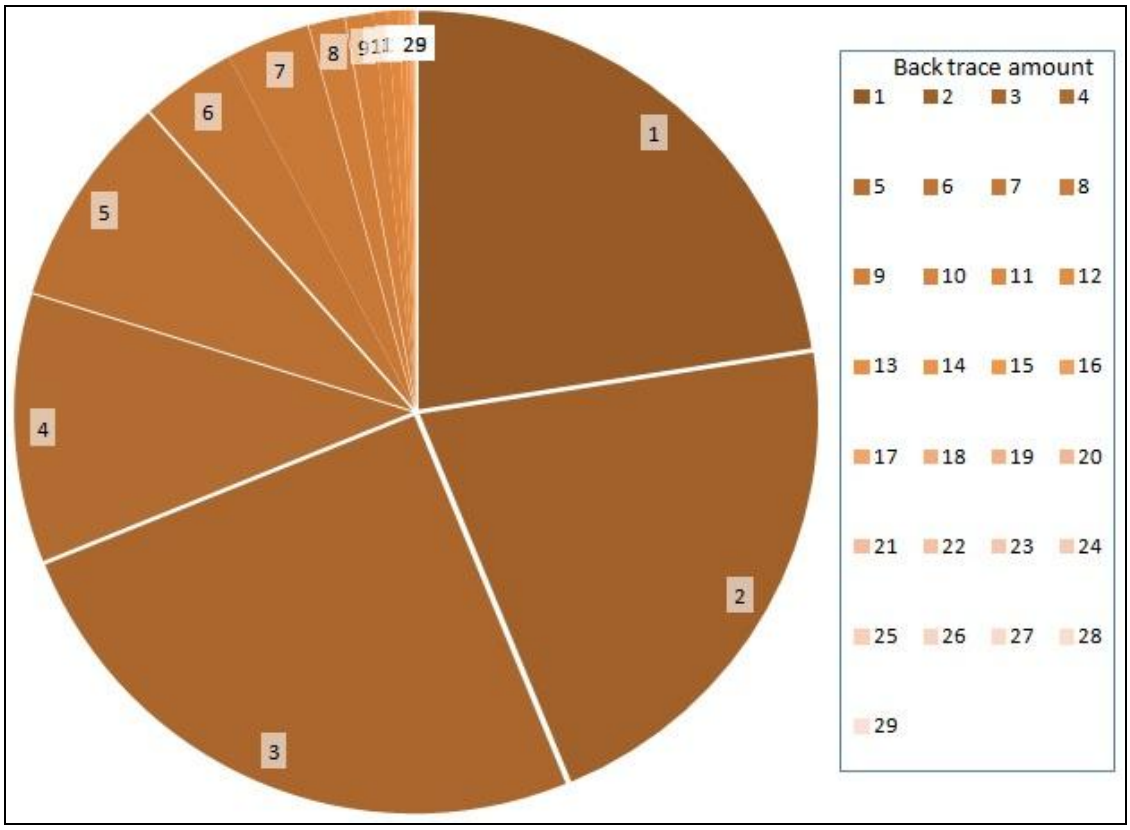


Table 1: Back trace cover

Current back trace	Occurred	percent Back trace
1	45620	45.62
2	42984	42.984
3	50857	50.857
4	22028	22.028
5	17505	17.505
6	7767	7.767
7	6897	6.897
8	2958	2.958
9	2279	2.279
10	1187	1.187
11	942	0.942
12	518	0.518
13	364	0.364
14	190	0.19
15	110	0.11
16	84	0.084
17	48	0.048
18	23	0.023
19	21	0.021
20	11	0.011
21	4	0.004
22	5	0.005
23	0	0
24	0	0
25	2	0.002
26	0	0
27	0	0
28	0	0
29	0	0