Hardware Architectures for Lossless Compression

By

SATYABRATA SARANGI
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Bevan M. Baas, Chair

Venkatesh Akella

Shyhtsun Felix Wu

Committee in Charge

2022

Abstract

Demands for storing huge volumes of data and limited communication network bandwidth call for effective data compression with high performance and energy efficiency to reduce high storage and communication costs for a wide range of systems and applications. Data compression consists of two types: lossy and lossless. Canonical Huffman encoding and Gzip are two popular lossless compression techniques. Hardware implementation of such lossless compression techniques in many-core processors and other hardware platforms is crucial in achieving optimized performance and energy-efficiency. This dissertation analyzes static and dynamic canonical Huffman codec algorithms, Gzip compression techniques, and presents high throughput and energy-efficient hardware and software implementations with good compression ratios.

Canonical Huffman encoding is naturally sequential which consists of several tasks: finding a histogram of symbol-frequencies, sorting of symbols, Huffman tree creation, building canonical code tables, and encoding symbols. This dissertation demonstrates energy-efficient canonical Huffman encoder architectures exploiting task-level parallelism for the above tasks and introduces a concurrent approach to execute sorting, Huffman tree creation, and code length computation tasks that yields better memory efficiency and performance than the conventional approach. The proposed architectures are implemented on a many-core array, Intel i7, Nvidia GT 750M, Intel FPGA, and 45-nm ASIC.

The many-core encoder implementations achieve a scaled throughput per chip area that is 89.2× and 4.7× greater on average and 44.7× and 8.2× greater in terms of scaled energy efficiency (compressed bits encoded per energy) than the Intel i7 and Nvidia GT 750M, respectively executing the common *Corpus* benchmarks for data compression. Encoder implementations on the many-core processor array yield scaled throughput per chip area and scaled energy efficiency that is 58× and 4.8× greater on average than the state-of-the-art efficient canonical Huffman encoder implementation on Tesla V100 GPUs executing the *enwik8* dataset.

Scaled synthesis results from a proposed pipelined canonical Huffman encoder 45-nm

ASIC results in 2.44× greater on throughput, 3.5× lower on total power dissipation, and 7.6× lower energy dissipation over Intel FPGA implementations.

Next, this dissertation presents bit-parallel static and dynamic canonical Huffman decoder implementations using an optimized lookup table approach on a fine-grain many-core array, Intel i7, Nvidia GT 750M, Intel FPGA, and 45-nm ASIC. The many-core implementations achieve a scaled throughput per chip area that is 891× and 7× greater on average and scaled energy efficiency (compressed bits decoded per energy) that is 149.5× and 3.9× greater on average than the i7 and GT 750M, respectively.

The 45-nm ASIC synthesis results show that the pipelined and memory-efficient static decoder yields a 5.1× throughput improvement and 13.4× energy efficiency improvement over the FPGA implementation.

Furthermore, this dissertation presents energy-efficient and high throughput Gzip hardware architectures using a chained hash bank memory design of depth three for LZ77 encoder and synthesis results of the Gzip compression engines implemented in a 45-nm ASIC. The proposed encoder architectures exploit both static and dynamic canonical Huffman encoders along with a pipelined LZ77 encoder. The pipelined Gzip engine using a static canonical Huffman encoder with a parallel window size (PWS) of 16 bytes per clock cycle, achieves a maximum input throughput of 2.53 GB/s, while the dynamic canonical Huffman encoder-based Gzip compressor achieves a maximum input throughput of 0.52 GB/s. To the best of our knowledge, this Gzip compressor offers the highest reported compression ratio in the literature of 2.47 for the Calgary Corpus benchmark.

Finally, this dissertation presents *DeepScaleTool,* an open-source tool for the accurate estimation of deep-submicron technology scaling by modeling and curve fitting published data by a leading commercial fabrication company for silicon fabrication technology generations from 130 nm to 7 nm for the key parameters of area, delay, and energy.

I dedicate this dissertation to my entire family.

Acknowledgments

My PhD journey and the work presented in this dissertation is the product of years of my hard work and enormous support and encouragement given by many generous and kind people along the way. I would be remiss if I did not take time to acknowledge the people that have helped and encouraged me throughout this journey.

First of all, I would like to express my appreciation and gratitude to my advisor and mentor, Prof. Bevan Baas. Thank you for the opportunity to have worked with you and learned from your expertise, and for the advice, work ethics, support and encouragement during difficult times, and valuable research insights that helped me grow through these past few years and come out as a stronger and wiser person. I would also like to thank you for encouraging us to do independent research and teaching us how to present our research work in a lucid manner.

I would like to thank Prof. Venkatesh Akella and Prof. Shyhtsun Felix Wu for serving in my dissertation and qualifying exam committee and providing invaluable feedback on my research. I would also like to thank Prof. Hussain Al-Asaad and Prof. Khaled Abdel-Ghaffar for serving in my qualification committee and giving insightful advice on my research proposal.

I would like to thank Prof. Bevan Baas, Prof. Venkatesh Akella, and Prof. Rajeevan Amirtharajah for their outstanding lectures that helped me build a solid foundation in computer architecture and digital VLSI design domains. I would also like to thank Prof. Bevan Baas, Prof. Venkatesh Akella, Prof. Hussain Al-Asaad, Prof. Omeed Momeni, Prof. Weijian Yang, Prof. Hooman Rashtian, Prof. Robert Redinbo, Prof. Diego Yankelevich, for whom I have had the privilege to serve as a teaching assistant for several digital design, computer architecture, and circuits classes for over 20+ quarters. Moreover, many thanks to all my amazing undergraduate and graduate students, whose technical interactions and quarter-end TA feedback made me a better teaching assistant over these years.

I have been extremely fortunate to have had the pleasure of working alongside several amazing graduate students at the VLSI Computation Lab. In particular, I am thankful to have worked with Dr. Shifu Wu, Timothy Andreas, Dr. Mark Hildebrand, Dr. Brent Bohnenstiehl, Dr. Aaron Stillmaker, Dr. Jon Pimentel, Dr. Bin Liu, and Emmanuel Adeagbo, whose perseverance and enthusiasm made the research group's environment very welcoming and astute exchange of great ideas. I would also like to thank all the previous and current VCL members, Michael Braly, Peiyao

Shi, Filipe Borges, Christi Tain, Dylan Finch, Delvin Huynh, Sarvagya Singh, Yuanyuan Xiang, Renjie Chen, Arthur Hlaing, Zhangfan Zhao, Sharmila Kulkarni, Yushan Wu, Yikai Mao, Ziyuan Dong, Aidan Callahan, Ryan Atkins, Jiayu Wang, Haotian Wu, Tony Tsoi, Christian Lum, and Benjamin Moore.

I would like to thank my amazing friends and colleagues, Dr. Shifu Wu and Timothy Andreas for their endless support throughout this journey and for all those fun trips, attending events, and exploring restaurants around together. Thank you again for offering your great insights on my papers and dissertation. I would also like to thank some of my friends and colleagues from other research groups, Terry O'Neill, Dr. Mohammad Motamedi, Dr. Dan Fong, Dr. Son Nguyen, and Anirban Samanta for their tireless humor as well as sparking many fun and interesting discussions.

I would like to thank the former Dean of College of Engineering, Prof. Jennifer Curtis, ECE department's former chair Prof. Saif Islam, ECE's graduate program advisors - Prof. Erkin Seker, Prof. Omeed Momeni, and many other campus administrators, for whom I have had the privilege to serve as a Graduate Student Advisory Committee Member to Dean of College of Engineering, Student Representative in the ECE Graduate Studies Committee, President/Vice-President of the ECE Graduate Student Association, and President of the Indian Graduate Student Association at UC Davis. I would also like to thank all student officers who worked with me for those student organizations to serve the graduate student community.

I have been lucky to work with very professional and caring ECE department's administrative staff during my time at UC Davis so far, Kyle Westbrook, Sacksith Ekkaphanh, Michelle Theresa Walker, Carole Bustamante, and others helped me navigate various processes. I would also like to thank Philip Young and Lance Halsted for their swift response and help related to IT and lab logistics.

I would like to thank my colleagues during my two internships at Intel with whom I had the pleasure of discussing my research and working with, Chung-lun Chan, Peter Ly, Dr. Bhavya Daya, Dr. Jesus Corbal, Dennis Bradford, Dr. Jon Pimentel, Rohan Sharma, and others in the Silicon Engineering Group (SEG), Charles Li, Sagar Pravin, Abhishek Munjal, Ajay Kumar Adha, Mahesh Kumboju, Aniketh Esamudra, Animesh Banerjee, and others in the Programmable Solutions Group (PSG). Thank you Charles and Chung for giving me an opportunity to work on some of the exciting projects at Intel.

A special thanks to Vice-Chancellor Prof. Prasant Mohapatra and his wonderful family for being so warm and welcoming all these years. You always make me feel just like family.

I would like to thank my friends, Abhishek Dhiman, The Gundus (Sree Balaji, Sandeep Rasoori, Sachin Kumawat, Harshvardhan Singh Chauhan), and Yagnik Suchak for their unconditional support and encouragement throughout this journey. Thanks to many other friends, housemates, and even some of their families, who kept me in good spirits and supported me throughout the duration of graduate school. It's impossible to list you all, but here's a start: Ismail, Akhil, Ajinkya, Dr. Sumeet, Dr. Subodh, Abhilash, Praveen, Prashant, Kirtanpal (KP), Sonam, Surender, Simerjeet, Geet, Bikram, Anmol, Jenny, Katherine, Angelica, Subhadeep, Kaustav, Girish, Sumanta, Smruti, and Nirmal.

A special thanks to my friends, Parnya Baradaran and Dr. Jon Pimentel, who were instrumental in helping me with the decision process for the full-time role post graduation. I would also like to thank my former colleagues at IIT Kharagpur, Prof. Swapna Banerjee, Dr. Pradyut Sanki, and Dr. Pallab Nath for their regular encouragement.

I am blessed to have a supportive and wonderful family. A special thanks goes to my parents, brothers, sister-in-law, and nephew for providing unconditional love and unwavering support and keeping me sane over these years, without which I wouldn't have been able to reach this milestone in my life. Thank you for inspiring me to seek out every opportunity to grow and for being in touch every single day amidst thousands of miles away physically. I also want to thank all my relatives for their continuous support and blessings.

Finally, a big thanks to the SISS staff at UC Davis, UC Davis Graduate Studies, and Davis community, for providing a welcoming environment to the international students. I will fondly reflect upon my time at UC Davis as I venture forward in the professional journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents a brief overview of lossless compression techniques and lays out the research motivation and organization of this dissertation.

1.1 Data Compression

In today's information age, global data production continues to increase significantly. Recently, certain external factors, such as the COVID-19 pandemic, fueled the volumes of data to greater heights. In the first quarter of 2020 alone, global data usage increased by 47% [1].

To accommodate such scenarios and future surges, a promising way is to opt for data compression. Data compression reduces data storage space and therefore data compression is pivotal in alleviating the cost of data storage and movement in data centers and other industries with a limited budget for capacity and I/O bandwidth. The vast volumes of data get compressed, which in-turn demands fewer computing resources to store, process, and transmit data.

From an economic point of view, due to data surge the global data compression market is expected to reach $4.51 billion by the year 2026 to cater to several industries that generate and store massive amounts of user data and online activity [2], for example, finance and banking, media and entertainment, health care, social media, communications, retail and e-commerce. Consequently, data compression and decompression techniques are becoming increasingly valuable.

Data compression consists of two types: lossy and lossless. In lossy compression, the compressed data doesn't need to translate into its exact original contents after decompression, which is shown in Figure 1.1. Therefore, lossy compression is suitable for the applications where loss of

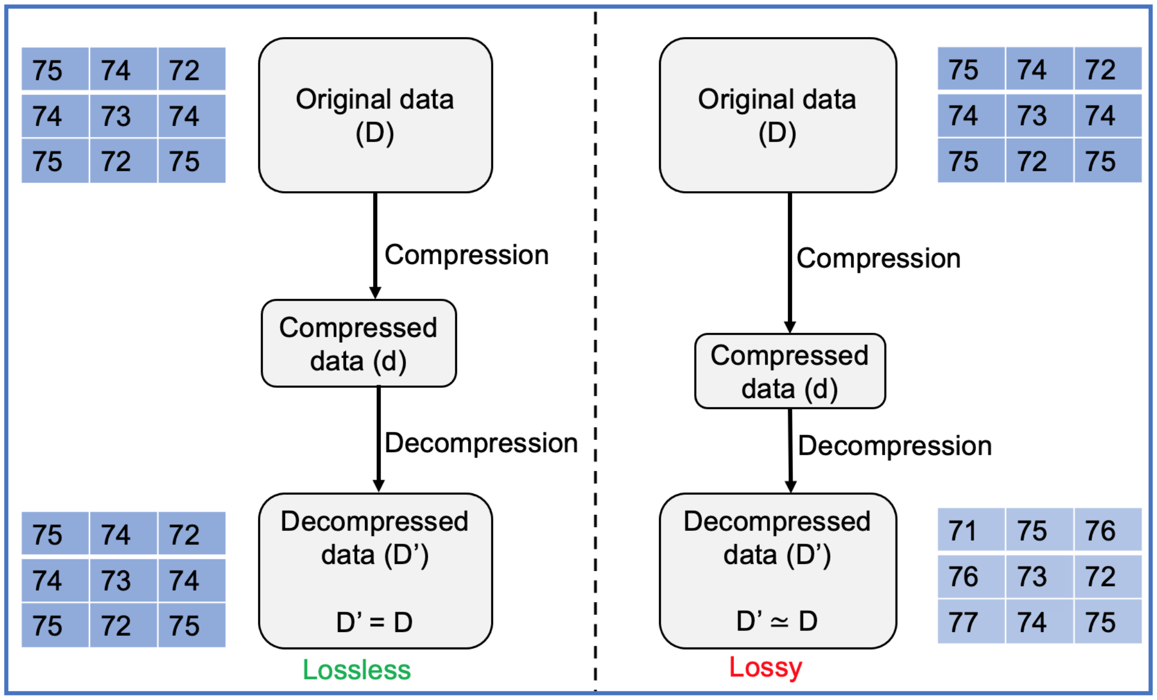


Figure 1.1: Example of lossless vs. lossy compression.

information is acceptable. For example, visual and multimedia data (audio, video, and images) can be compressed by degrading the contents to an acceptable extent, especially in streaming media to improve data bandwidth. However, lossless compression is typically required for text and data files, such as bank records and text articles, software codes, and genomic data, where the compressed contents must match with the original contents as any loss of data consistency and accuracy are intolerable for the above data sets.

The efficacy of data compression is measured by a metric called *compression ratio*. The compression ratio is the ratio between the size of the uncompressed or original data and the size of the compressed data.

$$Compression\ ratio = \ Size\ of\ uncompressed\ data\ /\ Size\ of\ compressed\ data \tag{1.1}$$

For example, if the size of the uncompressed data is 100 KB, a compression ratio of 2 yields compressed data of size 50 KB. Lossy compression algorithms achieve higher compression ratios than lossless compression, but at the cost of quality loss.

The data compression model can be fixed (*static*) or adaptive (*dynamic*). With static data compression, both the sender and receiver know the probability of data distribution beforehand. Therefore, a fixed code table with code words for corresponding symbols in the data set can be generated. However, in the case of dynamic compression, the frequency distribution of the symbols in a data set is not known. Therefore, the compressor determines the statistical distribution of all symbols in the data set first, creates a code table, and then proceeds to encode the symbols.

The compression ratio of dynamic data compression is better than the static data compression techniques as the earlier one is not tuned to specific distribution of symbols in advance. Dynamic data compression is preferred over static for streaming data sets. However, dynamic data compression results in lower compression speed. The dynamic compressor generates the symbol and frequency distribution from the data stream, encodes the symbols, and the compressed stream is passed to the receiver along with the required information to reconstruct the code table.

1.2 Lossless Compression Techniques

Lossless compression techniques are primarily divided into two categories: statistical compression and dictionary-based compression. Huffman compression, arithmetic compression, and Shannon-Fano compression are well-known statistical or entropy-based lossless compression techniques. Similarly, Lempel-Ziv (LZ77, LZ78) compression techniques are frequently-used dictionary-based lossless compression techniques.

The statistical compression techniques rely on statistical models to encode single symbols to bit strings that use fewer bits. However, the dictionary-based compression techniques don't encode single symbols into bit strings; rather, they encode variable length sub-strings by short codewords.

1.2.1 Shannon-Fano Compression

Shannon-Fano compression [3,4] is a variable-length encoding scheme that assigns codewords to each symbol based on their probability of occurrence. The compression scheme starts with laying out a list of probability or frequency of a given set of symbols. Next, the symbols are sorted in decreasing order of their frequencies. The next task is to split up the list into two parts while making sure that the total probability of both parts being as close to each other. Next, the left sub-group is assigned as binary "0" and the right sub-group is assigned as binary "1". The above process continues until each symbol is separated into left and right sub-groups. Finally, the least-frequent symbol gets assigned to the longest codeword and the most-frequent symbol gets assigned to the shortest codeword, which sets the average bits per symbol in a data set lower than the traditional ASCII encoding. However, Shannon-Fano codes are sub-optimal, which means that they don't always result in the best possible code length.

1.2.2 Huffman Compression

Huffman compression [5] is one of the most popular lossless compression techniques that is slightly different than the Shannon-Fano compression algorithm. The first step in Huffman compression is the same as for Shannon-Fano compression, i.e., to sort the symbols in a given data set per their frequencies of occurrences. Next, a Huffman tree is created by first taking the two nodes with the smallest probability as the leaf nodes and generating an internal node with a frequency of the sum of the frequencies associated with the two child nodes. The first extracted node is the left child and the other extracted node is the right child. The new node is added to the list of symbols. This process is repeated until all symbols in a data set are placed in the Huffman tree and the tree reaches the root node. After the Huffman tree is created, a binary "0" is assigned to each left nodes and a binary "1" is assigned to every right nodes of the tree. Finally, codewords are assigned to symbols by traversing the tree from the root node to the corresponding leaf nodes and appending the resulting binary bits. In general, Huffman compression always results in at least the same average code length as of Shannon-Fano compression and the resulting codewords are always optimal. Therefore, Huffman compression is preferred over the Shannon-Fano compression technique. However, the inefficiency with the Huffman codes are that they are always integer-length codes.

1.2.3 Canonical Huffman Compression

Canonical Huffman compression is a type of Huffman compression where code assignment is canonical instead of the the tree traversal-based approach in the case of the regular Huffman method. In regular Huffman encoding, the encoder passes the complete Huffman tree to the decoder so that the decoder traverses the tree to decode every encoded symbol. However, in the case of the canonical Huffman encoding, code length for each symbol is enough for the decoder to decode each encoded symbol and the tree traversal step is eliminated from the decoding process. Therefore, canonical Huffman coding reduces the memory footprint and executes faster than the regular Huffman decoder, while preserving the same compression ratio.

1.2.4 Arithmetic Compression

Arithmetic compression is a data compression technique that tackles the inefficiency of Huffman's representation of the compressed data by assigning the entire message into a single number, which is a fraction between 0.0 and 1.0. Therefore, a better compression ratio is achieved than the Huffman compression by lowering the number of bits to represent the code lengths at the cost of a more complex algorithm. However, arithmetic compression is slower than the Huffman compression. Therefore, Huffman compression is used frequently as the critical step in several lossless compression standards.

1.2.5 Lempel-Ziv Compression

Lempel-Ziv compression [6] methods don't consider the individual symbols' frequencies to do the compression. Instead, they look for repeated sequential patterns of characters in a streaming message. Therefore, they are effective at compressing long strings of repeated characters. The basic idea behind Lempel-Ziv (LZ) compression methods is to process the input data stream sequentially and find the best string match at any current location to that of previous occurrences. If any match is found, the compressor points to that previous occurrence, or else, outputs the symbol and continues this process to find the best possible string match.

LZ77 compression [6] is a lossless compression technique that replaces the repeated occurrences of data with a reference to the earlier presence in the data stream. The compressor encodes a string match by a length-distance pair, where length denotes the match length, and distance or offset depicts the exact distance to the previous occurrence. The compressor uses a fixed-length sliding-window over previously seen characters to keep track of the past occurrences, and increasing the size of the sliding-window increases the overall computations. Therefore, to alleviate this situation, *LZ78* compression was proposed [7], where, instead of a sliding window, the encoder maintains a dictionary of unlimited collection of previously-seen phrases. Instead of outputting the best possible string match, just an index to the dictionary is sufficient. Also, LZ78 doesn't need to pass on the length of the phrases to the decoder as the decoder already is aware of this information.

1.2.6 Deflate Compression

Deflate [8, 9] is one of the most widely used lossless compression standards that uses a combination of LZ77 and Huffman compression techniques to achieve better compression efficiency than standalone LZ77 or Huffman compression. A deflate stream consists of a series of blocks that holds the header information. Based on the header information, static or dynamic Huffman compression is used.

The gzip [10] file format is based on deflate compression. The gzip implementation allows end-users to select a compression ratio and intended speed of encoding as they are inversely related, which is discussed in Chapter 8. Acceleration of the gzip compression results in lower compression ratio. Therefore, a good trade-off between compression ratio and encoding speed is decided for any application.

1.2.7 Zstandard

Zstandard (zstd) is a lossless compression algorithm designed by Facebook [11, 12]. Zstandard is a combination of entropy coders, Huffman Coding and finite state entropy (FSE) [13] which performs well on modern CPUs and improves upon the trade-offs made by other compression algorithms. Zstandard combines the dictionary-matching from LZ77 with a large search window and entropy coding from FSE and Huffman. Zstandard provides a higher compression ratio than the deflate compression at a similar or faster compression speed.

1.3 Research Motivation

The demands for storing a huge volume of data and limited communication network bandwidth calls for effective data compression with high performance and energy efficiency. Moreover, energy efficiency in large data centers is an important concern, where a large chunk of the overall global power dissipation is used. To mitigate such concerns, data compression plays a key role as storage, network, and computing power of processing elements benefit due to the resulting lower data footprint. Along with increased system performance, data compression reduces the overall storage cost as well.

Compression ratio is one of the critical parameters for several applications that adopt data compression techniques. For example, in SSD applications that utilize expensive Flash memory, higher compression ratio is crucial, due to the architecture involved in writing the data where a flash cell must be erased before it can be reprogrammed, and the nature of the media with the limited number of program and erase cycles available. In Flash operations, erasure can only occur on a block level, whereas programming is done on page level, leading to a larger portion of the flash getting erased and rewritten than needed for any new data. Lossless compression with higher compression ratio enables the need of fewer program and erase cycles, extending the SSD lifetime [14].

With the diminishing returns from the Moore's law and Dennard scaling, clock frequency can no longer be significantly boosted up due to the limitations on power budgets. Therefore, acceleration of any computing workload primarily depends on parallel execution per clock cycle. Several hardware accelerators for lossless data compression using Field Programmable Gate Arrays (FPGAs), massively parallel Graphical Processing Unit (GPUs), and Application Specific Integrated Circuits (ASICs) were proposed that outperformed the software implementations on Central Processing Unit (CPUs)-based implementations. While the throughput and compression ratio of those compression and decompression hardware implementations were extensively studied in the literature, energy-efficiency of both compression and decompression hardware implementations were not discussed often with proper benchmarks. That gives us the first research question: **How do performance, area utilization, and energy-efficiency of various lossless compression and decompression techniques vary with computing platforms?**

Huffman encoding is sequential in nature. Therefore, to attain improved encoding throughput most of the research work focused on splitting up the input data stream into blocks of data which can be encoded and decoded independently. However, this results in degradation of compression ratio and may not be suitable for several compression standards. Moreover, very little research has been done on the efficient hardware implementations for canonical Huffman encoders and decoders. Therefore, we investigate the second research question: **How can the canonical Huffman encoding be accelerated despite having such serial bottleneck in its pipeline, while preserving the compression ratio?**

Similarly, hardware implementations of regular/canonical Huffman decoders were proposed either by splitting up the original data block into several chunks or exploring the synchronization property of Huffman codes. However, not all Huffman codes are synchronized. Moreover, independent encoding/decoding of the data chunks degrades compression ratio as discussed in the previous paragraph. This brings us the next research question that needs to be investigated: **how can the canonical Huffman decoder architectures be optimized for performance and energy-efficiency?**

In the case of the gzip compression pipeline, LZ77 compression plays a critical role, where the string matching block is a primary factor for the overall timing critical path. The string matching block can be accelerated by using proper hashing method per the benchmarks. Moreover, in gzip there is a trade-off between compression efficiency and compression throughput. Most of the literature on Deflate compression employed a static Huffman encoder to speed-up the overall compression. However, to obtain high compression ratio a dynamic canonical Huffman encoder is required. Therefore, **a proper study of the compression throughput vs. compression ratio along with designing the optimized architectures for LZ77, both static and dynamic canonical Huffman compression is the next research question that this dissertation addresses**.

To answer all the above research questions, this dissertation first investigates the design of efficient canonical Huffman encoder architectures leveraging task-level parallelism and concurrent execution of symbol-frequency sorting, Huffman tree creation, and code length computation. The proposed encoder architectures have been implemented on ASIC [15], FPGA [16], and a many-core processor array [17–19]. Second, an optimized lookup table (LUT)-based decoding architecture has been proposed for both static and dynamic canonical Huffman decoders. The proposed decoder architectures have been implemented on ASIC, FPGA, and a many-core processor array. Third, to optimize the throughput and study the throughput vs. compression ratio of the gzip compression, optimized hashing-based string matching techniques have been investigated and hardware implementation on ASIC for the gzip pipeline is proposed. The performance evaluation of all the designs have been done using standard lossless compression benchmarks. Metrics such as throughput per area and bytes encoded per unit energy are compared to results with optimized software implementations on an Intel i7 and massively parallel GPU.

Finally, this dissertation presents DeepScaleTool [20], a tool for the accurate estimation of technology scaling in the deep-submicron era as the traditional Dennard scaling factors go obsolete. DeepScaleTool is designed by modeling and curve fitting published data by a leading commercial fabrication company for silicon fabrication technology generations from, 130 nm to 7 nm for the key parameters of area, delay, and energy.

1.4 Computing Platforms

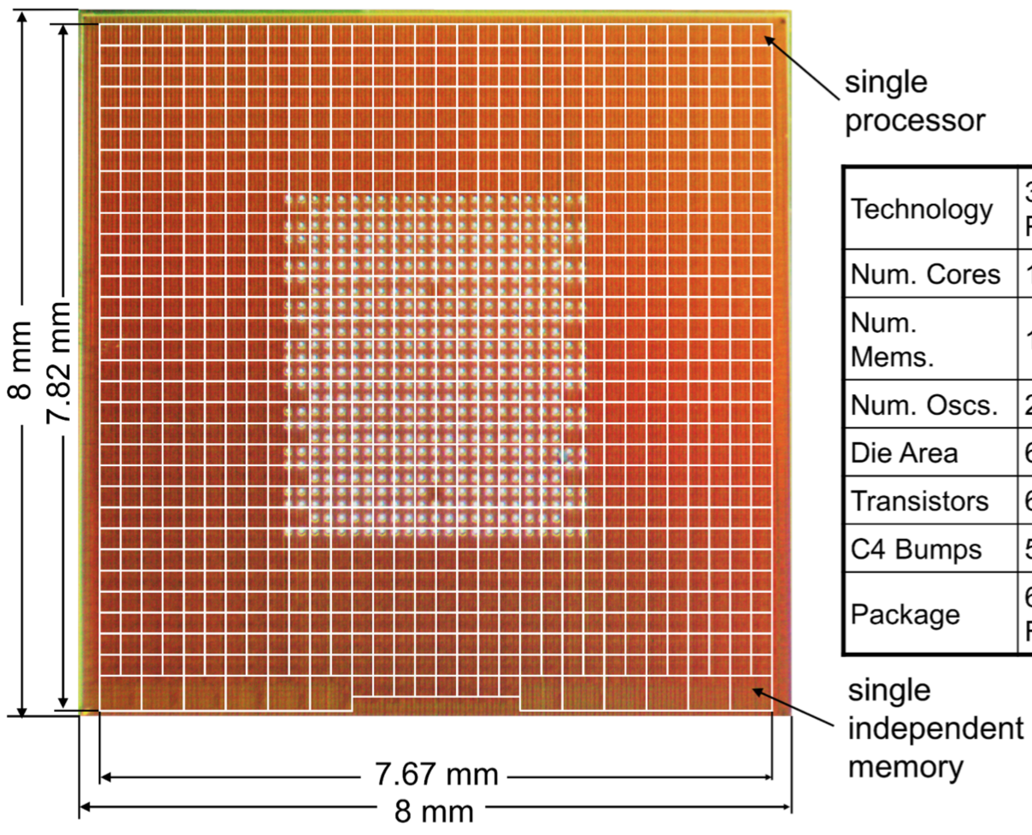
The lossless compression and decompression designs are evaluated on the following hardware platforms:

- A many-core processor array (32 nm) running at 1.2 GHz at 0.9 V [19]
- Intel MAX 10 FPGA (55 nm) with a base clock frequency of 50 MHz [16]
- NanGate 45 nm ASIC Library [15]
- Intel i7 CPU (22 nm) with a base clock frequency of 2.3 GHz [21]
- Nvidia GT 750M (28 nm) GPU with 384 shader cores running at a base clock of 941 MHz [22]

1.4.1 The Targeted Many-Core Processor Array

The use of many-core processor array platform is to provide fine-grained or task-level parallelism, which differs from single instruction multiple data (SIMD) architecture that offers coarse-level parallelism. GPUs are the prime examples of SIMD-based computing machines [22, 23]. The targeted many-core processor array platform in this dissertation is the KiloCore chip, a 1000-processor chip [17–19], which is based an asynchronous array of simple processors design paradigm. There are two other chips based on such many-core processor array paradigm, a 36-processor AsAP [24–26] and a 167-processor AsAP2 [27, 28].

Over the years, fine-grained many-core processor array-based computing platforms have demonstrated significant energy-efficiency and throughput per area improvements over general purpose processors and SIMD-based GPUs across several signal processing applications. A few of those applications are as follows: display stream compression (DSC) decoder [29, 30], low-density parity check (LDPC) decoder [31], parallel Advanced Encryption Standard (AES) computing engines [32, 33], H.264/AVC baseline residual encoder [34, 35], Context-Based Adaptive Binary Arithmetic Coding(CABAC) encoder [36], pattern matching [37,38], and sparse matrix multiplication engines [39].



Technology	32nm IBM PDSOI CMOS
Num. Cores	1000
Num. Mems.	12
Num. Oscs.	2012
Die Area	64 mm^2
Transistors	621 M
C4 Bumps	564 (162 I/O)
Package	676 Pad Flip-Chip BGA

Figure 1.2: Micrograph of the KiloCore chip, with key statistics [40].

Figure 1.2 shows the micrograph of the 32 nm KiloCore chip with key details of the chip. The KiloCore chip consists of 1000 RISC-style processors and 12 64-KB independent memory modules. Each processor has an instruction memory of 128x40-bits, a data memory of 256x16-bits, and two 32x16-bits input FIFOs [19]. Processors are connected by a 2D circuit-switched mesh network, allowing for nearest-neighbor communication to adjacent processors, as well as long-distance communication to connect non-adjacent cores. Each processor occupies 0.055 mm^2 and has an average maximum clock frequency of 1.78 GHz at 1.1 V.

The applications are implemented on the KiloCore chip by writing individual tasks in C++ or assembly language and the applications are simulated and verified using a cycle-accurate simulator.

1.5 Benchmarks and Performance Evaluation

We consider various corpus lossless text compression benchmarks [41] as follows to validate the compression and decompression designs. The following benchmarks cover a wide range of files of different sizes and symbol distributions.

- Artificial corpus [41]
- Calgary corpus [41]
- Canterbury corpus [41]
- Large corpus [41]

We verify the functionality of each design first before conducting the performance evaluation. The performance in terms of throughput per area and compressed bits encoded or decoded per unit energy of the computing platforms is evaluated for each implementation.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides a comprehensive background of the lossless compression and decompression techniques. Chapter 3 presents architectures for the optimized canonical Huffman encoding and implementations on a many-core processor array. Chapter 4 presents the proposed architectures for canonical Huffman decoders and their implementations on a many-core processor array. Chapter 5 discusses the design of the canonical Huffman encoders on ASIC and FPGA. Chapter 6 discusses the design of the canonical Huffman decoders on ASIC and FPGA. Chapter 7 covers the detailed hardware implementation of the gzip compression pipeline on ASIC. Chapter 8 presents the DeepScaleTool, a tool for the accurate estimation of technology scaling in the deep-submicron era. Finally, Chapter 9 concludes the dissertation and proposes future work.

Chapter 2

Background

This chapter overviews background of several lossless compression and decompression techniques which are the primary research interests of this dissertation, such as canonical Huffman encoder, canonical Huffman decoder, LZ77 encoder, and gzip compression, covering the corresponding algorithms, examples, and hardware or software implementations. A detailed literature survey of the encoders and decoders is illustrated.

2.1 Huffman Encoder

Huffman encoding is a variable-length and prefix-free encoding that encodes the symbols based on statistical redundancy of symbols in the data sets. The more common symbols are encoded using fewer bits and the less frequent symbols get assigned to longer code words. The prefix-free property of Huffman coding ensures that no code word is a prefix of any other code word. Huffman encoding works by creating a binary tree, called a Huffman tree, based on the symbol-frequency distribution, and then encoding symbols that are present at leaf nodes by traversing the tree from root to leaf nodes and assigning bit “0” to the left child node and “1” to the right child nodes.

Figure 2.1 details the Huffman compression pipeline. First, the frequency of each symbol in the uncompressed data set is computed. Next, the symbol-frequency list is sorted in non-decreasing order per corresponding frequencies. Next, the Huffman tree building process begins with the two nodes with smallest probability taking the leaf nodes, which creates a new internal node having these two nodes as children. The weight of the new node is set to the sum of the weights of the children. The first extracted node is the left child and the other extracted node is the right child. The left,

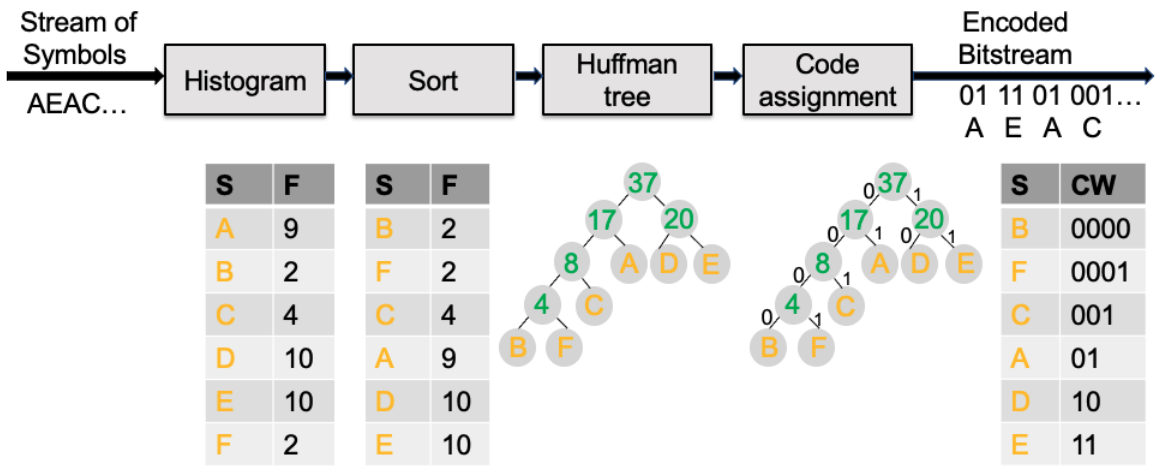


Figure 2.1: Example of Huffman compression. S: Symbol, F: Frequency, CW: Codeword.

right, and new nodes form a sub-tree. The new node is added to the list of symbols. The above steps are repeated on the new internal node and on the remaining symbols until only one node remains, which is the root of the Huffman tree. After the Huffman tree is built, each symbol is encoded with bits by traversing the tree from root to corresponding leaf nodes and assigning assigning bit "0" to the left child node and "1" to the right child nodes. For example, in Figure 2.1 symbol "B" gets "0000" and symbol "F" gets "0001". Similarly, symbol "A" gets encoded to "01" and so on. Next, a code book is created with symbols and corresponding code words, which helps in generating the encoded bitstream.

Using ASCII encoding, the source data shown in Figure 2.1 consumes 37*8 = 296 bits of space. The total number of symbols is 37 in the source data. However, Huffman compression on the same data set results in 9*2 + 2*4 + 4*3 + 10*2 + 10*2 + 2*4 = 86 bits of storage, achieving a compression ratio of 3.44.

2.2 Canonical Huffman Coding

Canonical Huffman encoding utilizes canonical code assignment process. Once the Huffman tree is built like in regular Huffman encoding, the idea is to assign code words to a list of symbols sorted lexicographically or alphabetically, based on starting code word and then increment by one for the rest of the symbols sequentially. This canonizing code assignment process helps the decoder

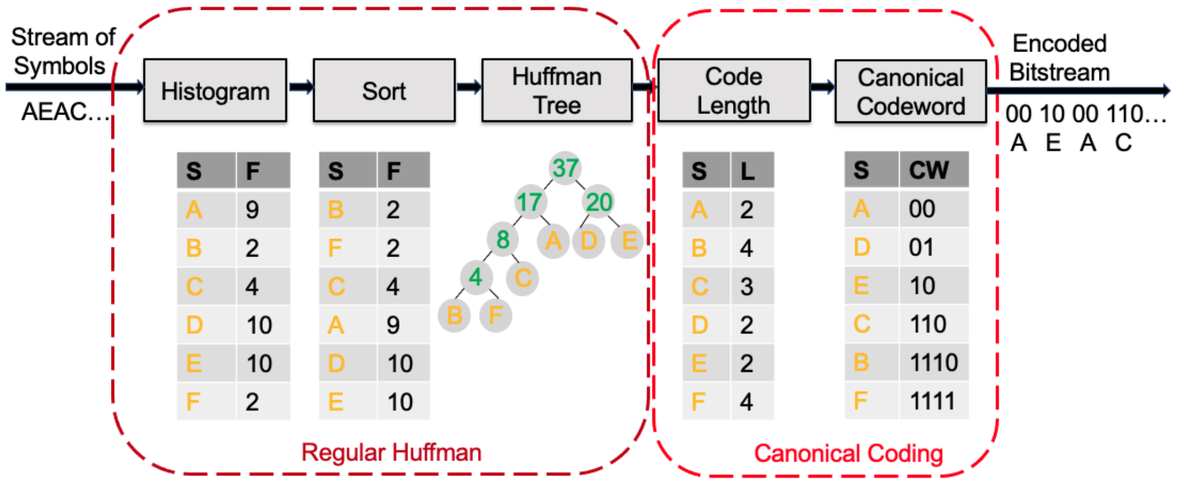


Figure 2.2: Example of canonical Huffman compression. Canonical Huffman expression differs from regular Huffman compression only in the code assignment stage.

not to require the Huffman tree to be sent by the encoder; rather, only code length information for each symbol is required to rebuild the code book. Therefore, canonical Huffman encoding is memory-efficient. Moreover, canonical Huffman encoding results in the same compression ratio as with regular Huffman encoding.

Canonical Huffman encoding starts with finding the symbol frequency histogram from the stream of uncompressed symbols. Figure 2.2 shows that once the symbol-frequency information is computed, the next task is to sort the symbol-frequency list and build a binary Huffman tree. Next, in the case of regular Huffman encoding, the code word generation module traverses the Huffman tree from root to leaf nodes and assigns bit “0” to the left child node and “1” to the right child nodes to create code words for symbols. However, in case of canonical Huffman encoding process, the first step after creating a Huffman tree is to find the code length for each symbol. In the example shown in Figure 2.2, symbols “B” and “F” have code lengths of 4-bits, symbol “C” gets a code length of 3-bits and so on. Next, the starting code word for each code length is computed per Algorithm 1, where “MAX_BITS” denotes maximum code length and “bl_count[ ]” gives the number of symbols having the same code length.

Therefore, symbol “A” gets assigned to code word “00”, symbol “C” codes to “110” and so on. Next, the rest of the symbols in the same code length group are encoded by subsequently incrementing the starting code by one. Therefore, symbol “D” gets assigned to code word “01”,

Algorithm 1 Computation of starting codeword for each code length [8]

```
code = 0;
bl_count[0] = 0;
For ($bt = 1;\ bt <= MAX\_BITS;\ bt++$)
code = (code + bl_count[bt-1]) << 1;
next_code[bt] = code;
```

symbol “F” gets encoded to “1111” and so on. Finally, the code book is constructed with symbols and corresponding code words, which is used to encode all symbols, and thereby creating the encoded bit stream.

Canonical Huffman encoding on the source data shown in Figure 2.2 results in 86 bits of storage, achieving the same compression ratio of 3.44 with regular Huffman encoding. In the case of the canonical Huffman encoder, along with the encoded bit stream, header data with code length information is sent to the decoder to reconstruct the code book.

2.2.1 Static and Dynamic Canonical Huffman Coding

The canonical Huffman encoding can be static or dynamic. The static version of the encoder uses a static code [8] table to encode the symbols. Table 2.1 depicts the static code table that is used for Deflate compression. The static code tables are predefined for various compression standards. However, in the case of the dynamic canonical Huffman encoder, Huffman trees are created dynamically for blocks of uncompressed data. Dynamic canonical Huffman encoding method achieves better compression ratio than the regular canonical Huffman encoding at the cost of compression speed. Deflate supports both static and dynamic canonical Huffman coding, providing a range of compression speed vs. compression efficiency.

2.2.2 Related Work

There has been extensive work on designing Huffman encoders for various applications. Mukherjee et al. [42] proposed parallel Huffman encoders that generated the codewords for a symbol in one clock cycle. However, their implementation was limited to static Huffman codes. Liu et al. [43] proposed VLSI implementation of variable-length codec using efficient memory architecture for accessing Huffman trees. Park et al. [44] demonstrated ASIC implementation of Huffman coding

Table 2.1: Static Code Table for Deflate. This table is adopted from [8].

Literal Value	Bits	Codes
0 - 143	8	00110000 through 10111111
144 - 255	9	110010000 through 111111111
256 - 279	7	0000000 through 0010111
280 - 287	8	11000000 through 11000111

for 8-bit symbols using a single processing element to reduce area overhead. Rigler et al. [45] designed dynamic Huffman encoders for GZIP on FPGA, where two memory blocks were used to store the Huffman tree nodes to track the leaf and parent nodes.

Matai et al. [46] proposed the first hardware accelerator for complete pipelined stages of canonical Huffman encoders on a FPGA using high-level synthesis tools. The authors proposed several optimization techniques to accelerate each stage of the encoder with input sizes 256, 536, and 704 symbol frequencies per their requirement from LZ77 stage. Overall, a detailed benchmarking and analysis is missing, which is critical as performance and energy-efficiency of the encoder strongly depends on the data sets and corresponding symbol distribution.

There were some previous work that focused on designing Huffman encoder ASICs for image or video compression standards. Patrick et al. [47] proposed a variation of Huffman encoding to perform JPEG encoding using a DSP processor. However, the details regarding architecture and hardware implementation results were not discussed for the Huffman encoder. Later, various other work also proposed Huffman encoder designs for JPEG [48–50].

Satpathy et al. [51] designed a canonical Huffman encoder for DEFLATE compression in ASIC where they eliminated the serial dependency of the encoder by using concurrent computation of sorting and tree construction and other optimizations. Recently, a high-throughput ASIC architecture was proposed by Shao et al. [52] for a canonical Huffman encoder. The authors demonstrated a parallel computing architecture for sorting and code length computation blocks, which results in improved throughput for JPEG data sets. However, the encoding process relies on a sorting operation every time a new node is generated in the Huffman tree. Also, energy-efficiency results for various benchmarks were not reported.

In the past, several GPU-based parallel Huffman encoders were proposed, especially for the field of high performance computing. Rahmani et al. [53] designed parallel Huffman encoders in the CUDA framework where a parallel prefix-sum based method was used. However, this method is unsuitable for shorter code words and degrades memory bandwidth of GPUs. Sitaridi et al. [54] proposed a specific file format for compression where the input is split into chunks and exploits both intra and inter-block parallelism. Funsaka et al. [55] proposed a dictionary-based compression for GPUs without closely following Huffman's method. Similarly, Angulo et al. [56] implemented Huffman compression of seismic data on GPU by altering the original Huffman encoding scheme.

Patel et al. [57] used the Burrows-Wheeler transformation to accelerate GPU compression; however, the implementation didn't yield better results than the single-core CPU implementations due to the sorting stage. Recently, Tian et al. [58] proposed a parallel implementation of Huffman encoding algorithm on GPUs. The authors proposed a novel reduction-based encoding scheme that results in higher GPU memory bandwidth utilization. In this design, the code word encoding scheme follows the regular Huffman's method.

In this dissertation, we propose energy-efficient mapping of the canonical Huffman encoders on a many-core processor array, FPGA, and ASIC. We deploy task-level parallelism for the entire encoding pipeline, strictly follow Huffman's original method, and preserve the compression efficiency. We also propose a method of concurrent tree building and code length computation design to bypass the storage of the complete Huffman tree with references for each of the child nodes to their parents nodes. This mechanism makes the encoder more efficient in terms of speed, area, and memory usage.

2.3 Huffman Decoder

In Huffman coding, the Huffman tree that is generated during the encoding process is passed to the decoder in the header information. The decoder traverses the tree to decode each symbol [46] after parsing the encoded bitstream. This slows down the decoding process and adds complexity to the compressed bitstream header.

Figure 2.3 shows the Huffman decoder that uses tree traversal-based approach. The decoder parses each bit from the encoded bitstream and traverses the tree starting from the root node. For example, “01” routes to symbol “A”. Similarly, the next iteration of “11” results in the symbol “E”, and so on. This process continues until the last bit in the encoded bitstream.

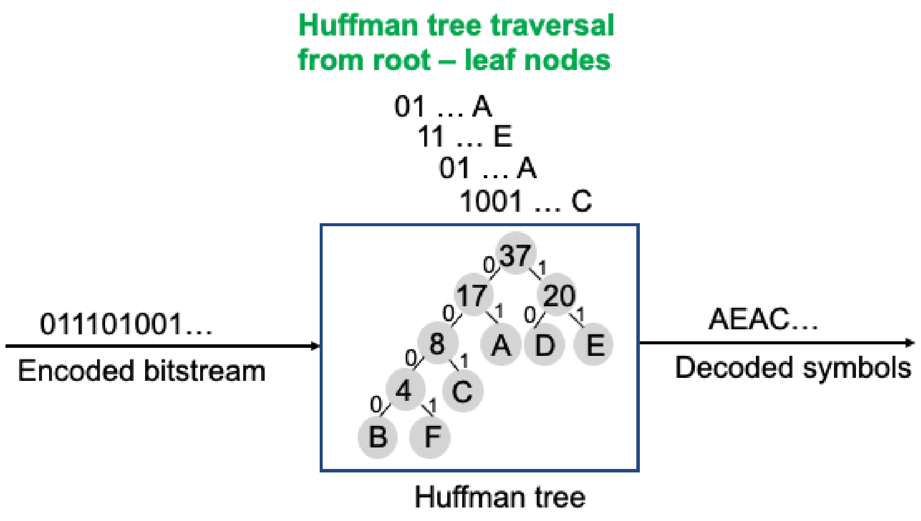


Figure 2.3: Huffman tree-traversal based decoding.

2.4 Canonical Huffman Decoder

In the case of canonical Huffman coding, the encoder assigns the codes based on canonical coding assignment: the code for the first symbol with a specific code length is computed based on both code for the first symbol in the previous code length group and the number of symbols for the previous code length [8]. The details of canonical code assignment steps are discussed in the deflate standard [8]. Subsequently, the codewords for the rest of the symbols with the same code length can be computed by incrementing the codeword of the predecessor symbol by one [59]. The code assignment technique in canonical Huffman coding enables the decoding of the compressed bitstream without a Huffman tree. Instead, the canonical Huffman encoder sends the number of symbols per code length and the corresponding lexicographically sorted symbols in the header to the decoder, where the codes can be easily regenerated.

A canonical Huffman decoder can be static or dynamic. The static decoder ensures that the code table containing symbols and corresponding code lengths is indexed correctly by the encoded bitstream. The code table is fixed in this case, hence it is called a static decoder. In the case of the dynamic canonical Huffman decoder, a code table is created for each encoded data block based on

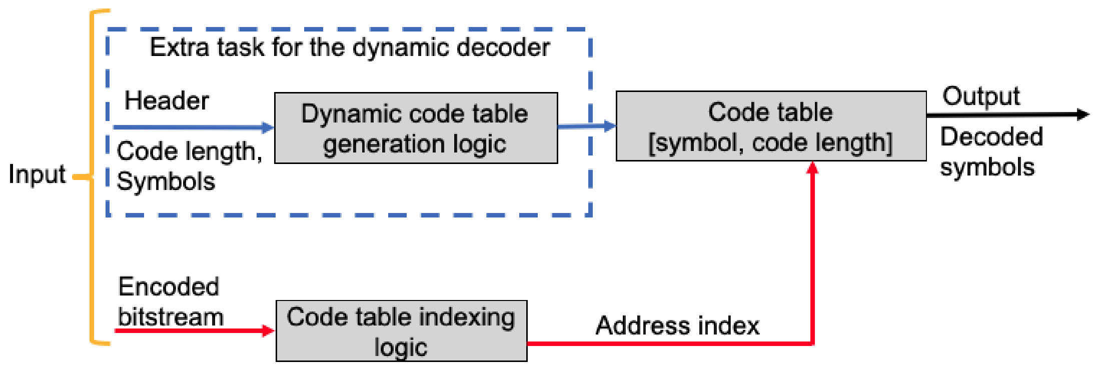

Extra task for the dynamic decoder
Header
Code length, Symbols
Input
Dynamic code table generation logic
Code table [symbol, code length]
Output
Decoded symbols
Encoded bitstream
Code table indexing logic
Address index

Figure 2.4: Static and dynamic canonical Huffman decoder block diagram. The dynamic code table generation task is only necessary in a dynamic canonical decoder.

the header information. The canonical code assignment process enables the decoder to self-generate a code table at the receiver based on the code length information in the header.

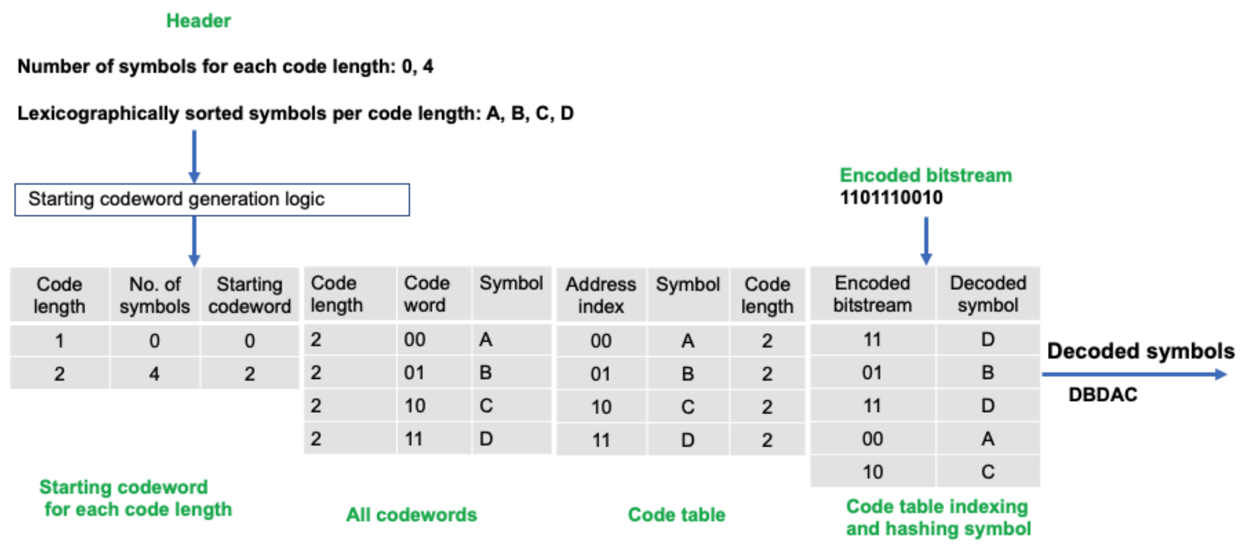
Figure 2.4 shows a block diagram representation of the static and dynamic canonical Huffman decoder, which shows the extra task in case of a dynamic canonical Huffman decoder of forming a code table.

2.4.1 Dynamic Canonical Huffman Decoder

The dynamic canonical Huffman decoder first recreates a dynamic code table by using the code length sequences and symbols from the header, which is described in the following Section 2.5. Next, the compressed bitstream is indexed into the dynamic code table to decode the symbols. In Deflate format each block is coded independently by creating dynamic code tables for each block per header information, making the decoding process slower than the static decoder. However, the speed up in the case of the static decoder comes at the cost of reduced compression ratio.

2.4.2 Algorithm and Example of Canonical Huffman Decoder

Figure 2.5 depicts the canonical Huffman decoding process for a given encoded bitstream and header. The number of symbols for each code length from the header is used to generate the starting codeword for each code length per the starting codeword generation logic [8]. The next step is to read the starting symbol for each code length and assign the corresponding starting codeword to it [59]. The rest of the symbols with the same code length are assigned by incrementing the codeword of the predecessor symbol by one.


Header
Number of symbols for each code length: 0, 4
Lexicographically sorted symbols per code length: A, B, C, D
Starting codeword generation logic
Encoded bitstream
1101110010
Decoded symbols
DBDAC
Starting codeword for each code length
All codewords
Code table
Code table indexing and hashing symbol

Code length	No. of symbols	Starting codeword
1	0	0
2	4	2

Code length	Code word	Symbol
2	00	A
2	01	B
2	10	C
2	11	D

Address index	Symbol	Code length
00	A	2
01	B	2
10	C	2
11	D	2

Encoded bitstream	Decoded symbol
11	D
01	B
11	D
00	A
10	C

Figure 2.5: Example of a dynamic canonical Huffman decoding process which consists of the following tasks: computing the starting cdoeword for each code length, mapping of symbols to codewords, code table containing symbols and code lengths, and decoding of symbols by parsing the encoded bitstream.

For example, symbol “A” gets the starting codeword which is “00”. Therefore, symbol “B”, “C”, and “D” get the subsequent codewords which are “01”, “10”, and “11” respectively. The maximum code length is two; therefore, a code table of four words is created and each word stores both symbols and code lengths. Each codeword acts as an address index to the code table. When the number of symbols is less than the depth of the LUT, the contents of the LUT are populated based on the prefix of the codeword for each symbol as shown in the earlier work [59].

Once the code table is constructed, the index decoding task performs the following subtasks: first, read 2 bits (same as MAX_BITS for this example) at a time from the encoded bitstream, index it to the code table, and extract the corresponding symbol as the output, which results in “DBDAC” as the decoded symbols for this example. For the static canonical Huffman decoder, the code table is pre-defined and the bitstream indexing task follows the same process as for the dynamic canonical Huffman decoder.

2.4.3 Related Work

The serial dependency of the Huffman decoder results due to the intrinsic dependency between locations of adjacent symbols in the bitstream [60]. Therefore, to parallelize the decoding process many prior attempts alter Huffman's original method by splitting up the input data into

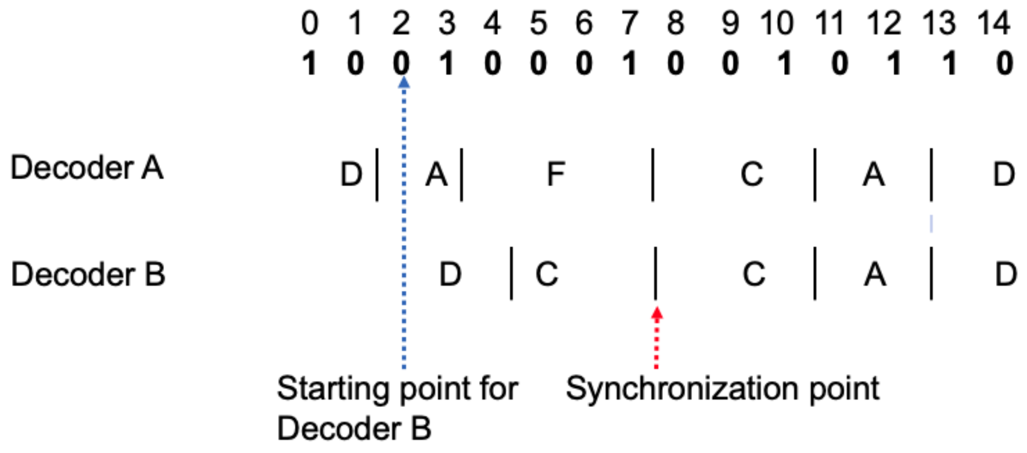


Figure 2.6: Self-synchronization property of the Huffman Codes that enables parallel decoding.

independent chunks which can be compressed and decompressed separately [59]. However, this method is unsuitable for several file formats [61].

Mukherjee et al. [42] were among the first to propose parallel and adaptive Huffman encoders that generated the codewords for a symbol in one clock cycle. However, the proposed Huffman decoder was limited to static Huffman codes and the performance was limited due to the Huffman tree traversal approach. Seong et al. proposed a variable length decoder using a LUT partitioning approach to reduce the power dissipation based on code frequency [62]. However, details regarding LUT generation from the encoded bitstream and header information were not provided. Also, the hardware implementation was specific to MPEG-2 applications only.

One of the first real-time and variable length decoder implementations was proposed on FPGAs using an optimized lookup table (LUT) [63]. The static lookup table decodes one symbol per clock cycle. However, due to a maximum limit in the length of the data input stream of 8 bits, more than one clock cycle is necessary to decode a codeword of longer than 8 bits. The authors in [64] investigated a speculative parallel approach for Huffman decoder and used that in Gzip decompression. The parallel scanning of the end of each block can be accomplished; however, the probability of false-positive boundaries is the bottleneck in achieving high performance. Most recently, Ledwon et al. proposed a high throughput hardware accelerator for DEFLATE compression and decompression using high-level synthesis [65] and an optimized LUT architecture from the paper [66]. The authors [65] reported the static and dynamic throughput for the Calgary corpus files; however, the detailed architecture to generate the optimized LUT from the header information and energy efficiency (compressed bytes per unit energy) results were not reported.

There has been some work on accelerating canonical Huffman decoders by using GPUs. Angulo et al. [67] implemented Huffman compression and decompression of seismic data on GPU by altering the original Huffman encoding scheme. Also, the proposed design was limited to seismic datsets. The Burrows-Wheeler transformation used by the authors Patel et al. [68] to accelerate GPU decompression speed didn't work well when compared to CPU implementation due to the Huffman tree traversal approach. Ozsoy et al. [69] implemented the Lempel-Ziv-Storer-Szymanski (LZSS) lossless data compression algorithm and showed a significant improvement of compression speed over CPU platforms. However, the decompression performance resulted in a lower speed up due to the memory intensive workload creating a bottleneck for the CUDA implementation [69]. Reference [70] presented an efficient implementation of decompression using GPUs without closely following Huffman's method [61].

The authors [61] investigated the self-synchronizing property of the Huffman code that enabled massively parallel implementations on Nvidia GPUs following Huffman's original method. The self-synchronization property of Huffman codes enables the Huffman decoder to correct itself even after initial erroneous output [71, 72]. For example, if a decoder starts decoding from an intermediate bit of a codeword sequence, it results in a wrong decoded symbol as the intermediate bit position and start of codeword sequence may be different. However, once there is a correct decoded symbol, subsequent decoded symbols are all correct. Figure 2.6 demonstrates the self-synchronization property of Huffman codes. Decoder A and decoder B start concurrent decoding, but starting at different bits of the encoded bitstream. Therefore, decoder B results in wrong outputs at first. However, once the synchronization point is achieved, subsequent outputs are all correct. In terms of hardware complexity, finding synchronizing points across data chunks adds significant computational complexity to the decoding module. Moreover, the authors [73] demonstrated that some of the Huffman codes would never be self-synchronized.

The authors [73] proposed a new technique called Huffman coding with a gap array to eliminate the self-synchronization decoder's performance bottleneck. Using gap array enables finding the correct beginning of the encoded data sequence; however, it adds area and encoding time overhead. Moreover, compression efficiency gets affected due to addition of gap array to the encoded bitstream. Most recently, the authors [71] analyzed both self-synchronization and gap array Huffman decoding methods and proposed error-bounded lossy compression and decompression of scientific data.

In this dissertation, we propose detailed energy-efficient architectures for canonical Huffman decoders and mapping of them on many-core processor array, FPGA, and ASIC. We show an optimized look-up table structure for the decoder that results in less area and code table construction time than the traditional architectures.

2.5 Gzip Compression

Gzip compression is based on Deflate [8], which is a lossless compression algorithm. Gzip was originally created by Philip Katz to develop PKZIP [74]. The Deflate format was later utilized by Jeanloup Gailly and Mark Adler in their open-source software application, gzip [8], and the software library, zlib [9], which was created in order to facilitate the usage of the PNG lossless compressed image format.

Deflate is now commonly used by many different compression and decompression applications, such as Hypertext Transfer Protocol (HTTP) standard [75], .zip [76], .gz [77], and .png [78] file formats etc [79]. Each of these compressed file formats includes a header and footer wrapper that encloses the binary Deflate compressed data payload.

2.5.1 Deflate Compression Algorithm

Deflate is a lossless data compression technique comprising both LZ77 and Huffman coding. Since the LZ77 encoding format is patented, Deflate describes a similar but more general algorithm for replacing duplicated strings [79]. Figure 2.7 shows the two-stage Deflate encoding steps, where LZ77 encoding is the first level of compression at byte-level and then canonical Huffman compression accomplishes the second stage of compression at bit-level.

2.5.2 LZ77 Encoding

LZ77 compression replaces repeated occurrences of a string with references to a single copy of that string existing earlier in the uncompressed string stream. A match is encoded by a pair of numbers called a *length-distance pair*. As each byte of the file is read, it is recorded in a history

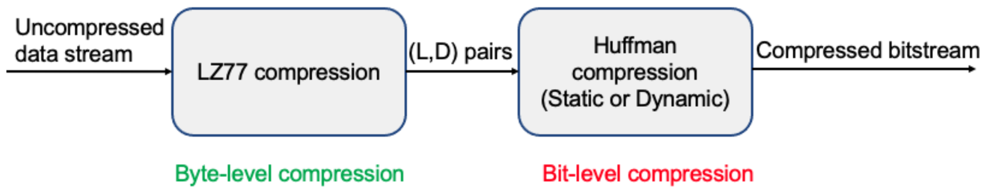


Figure 2.7: Block diagram representation of Deflate compression showing that first stage of LZ77 Compression does byte-level compression based on string match and the second stage does bit-level compression.

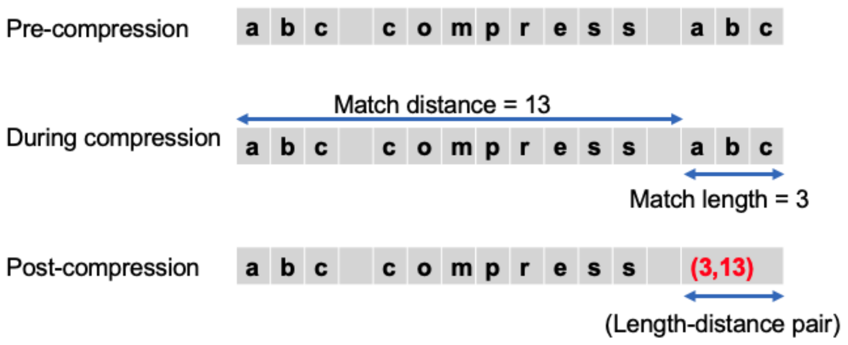


Figure 2.8: Example of a LZ77 encoding process which consists of the following two tasks: find the best possible string match and encoding the length-distance pair from the string match process using Huffman encoding.

buffer and any potential previous matches are looked up in that buffer. The length-distance pairs correspond to the length of a match and the distance back within the history buffer. In the Deflate format, matching strings of lengths of up to 256 bytes long and match distances of up to 32,768 bytes are allowed.

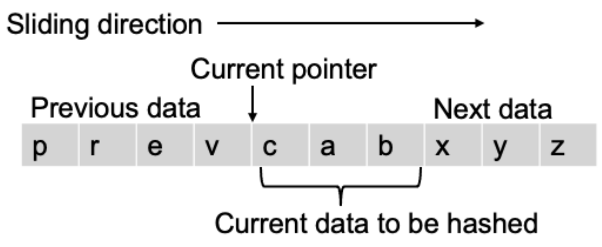


Figure 2.9: Sliding window mechanism.

Typically, a fixed-width sliding window is used, which slides across the sequence of input bytes, as shown in Figure 2.9. Based on the current pointer, a fixed number of bytes of input are hashed, which is basically turning the current bytes to a fixed-length hash signature. Next, the hash

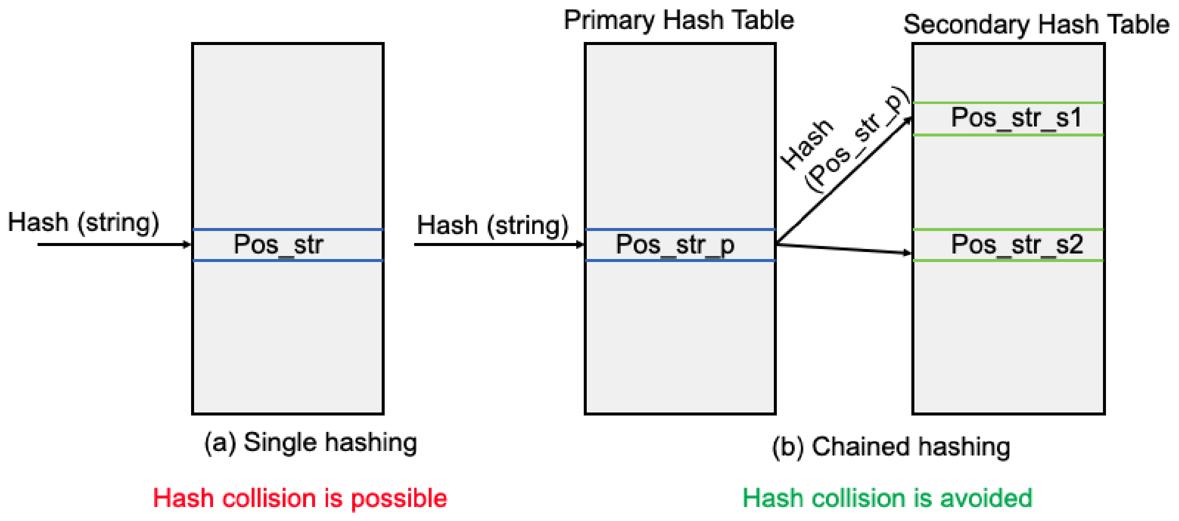


Figure 2.10: Single vs. chained hashing to find the matched strings.

signature is looked up in the hash table. The hash tables can be single or multi-staged. Single stage hashing may result in collision due to multiple input bytes translating to the same hash signature. The Deflate specification recommends using chained hash tables to find potential matching strings, though any match-finding method can be used [79] as shown in Figure 2.10.

The hash signature is looked up in a primary hash table, which contains the positions of all of the most recently hashed and stored strings. If more than one string maps to the same hash signature, the table will contain a link chaining from that signature's position in the primary hash table to a secondary hash table. A chain of such links can be traversed to find all the positions of matching strings that have the same hash value [79].

Figure 2.10 shows the single vs. chained hash tables. To avoid a hash collision, chained hash tables are used that adopt indirect indexing based on a primary hash signature. The next task is to find the best match length. The potential matches from the compare windows are then compared to the current window and the longest match keyword is used to replace the input bytes with a length-distance pair. Figure 2.11 depicts the best string match length computation process. To attain the best possible compression ratio, the LZ77 compressor tries to find the best possible match. However, string matching logic is a part of the critical path affecting the overall compression speed. Therefore, a trade-off between compression speed and compression ratio must be attained.

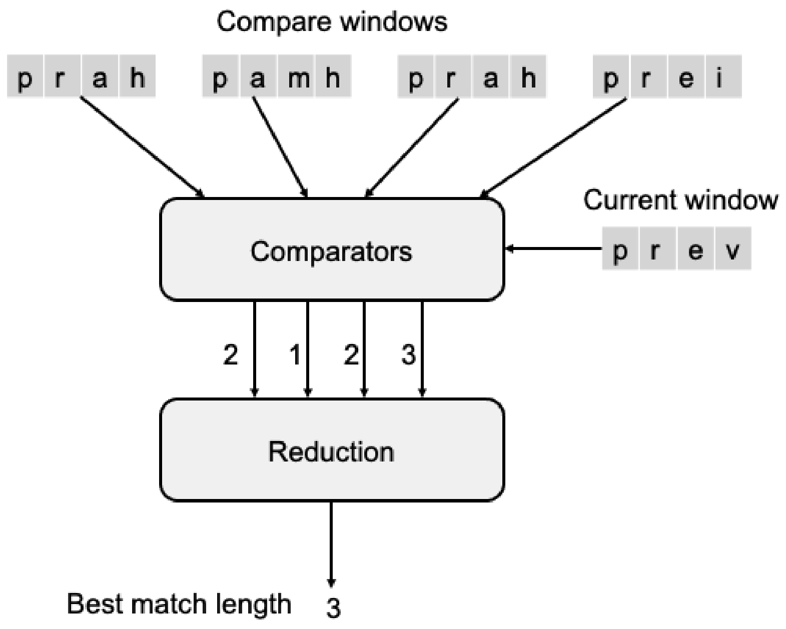
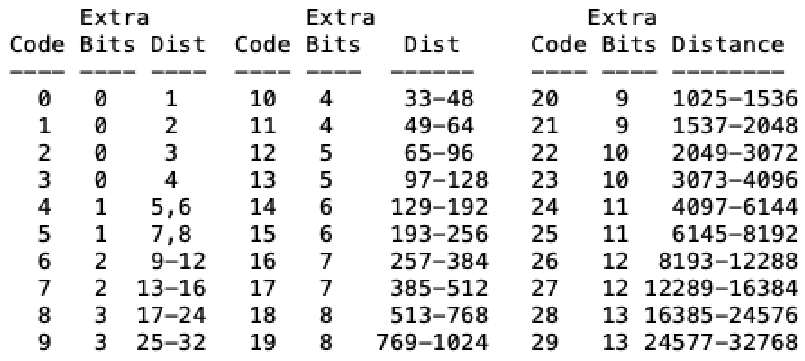


Figure 2.11: Best string match length computation.

2.5.3 Canonical Huffman Compression for Deflate

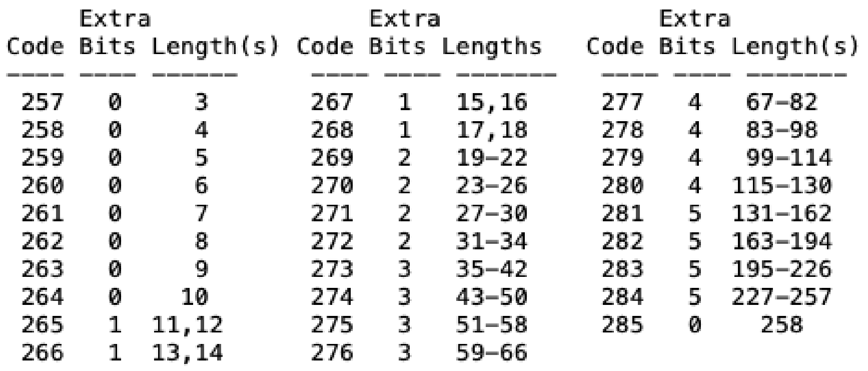
In Deflate compression, canonical Huffman compression does the bit-level compression after LZ77 encoding. Deflate supports both static and dynamic canonical Huffman compression. Normally, two pre-defined code tables are used in the case of the static canonical Huffman compression, one for length and the other for distance. Figure 2.13 shows the static code table for length, where symbols 257 to 285 are the length symbols. Symbols 286 and 287 are not actually used for encoding, but are, rather, used in the construction of the Huffman code [79]. Figure 2.12 shows the static code table for distance that contains 30 different distance symbols from 0 to 29.

The extra bits that are shown in Figure 2.12 and Figure 2.13 correspond to a binary offset that is added to the base length or distance value after decoding. In Deflate static Huffman encoding, the literal/length codes are from 7 to 9 bits long and the distance codes are fixed-length 5-bit codes that are just the binary values of their symbols (e.g., the static code for distance symbol 5 is "5'b00101" [79]. In dynamic encoding, both the literal/length and distance codes can be from 1 to 15 bits long. In both static and dynamic encoding, the length codes may be followed by 0 to 5 extra bits and distance codes may be followed by 0 to 13 extra bits [79].



	Extra			Extra			Extra	
Code	Bits	Dist	Code	Bits	Dist	Code	Bits	Distance
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Figure 2.12: Static code table for distance [8].



	Extra			Extra			Extra	
Code	Bits	Length(s)	Code	Bits	Lengths	Code	Bits	Length(s)
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Figure 2.13: Static code table for length [8].

2.5.4 Details of Deflate Compressed Format

Each block of compressed data begins with 3 header bits containing the following data: first bit, BFINAL, and next 2 bits, BTYPE. BFINAL is set if and only if this is the last block of the data set. BTYPE specifies how the data are compressed, as follows: 00 - no compression, 01 - compressed with fixed Huffman codes, 10 - compressed with dynamic Huffman codes, 11 - reserved (error). The only difference between the two compressed cases is how the Huffman codes for the literal/length and distance alphabets are defined [8].

Figure 2.14 shows the configuration of the header for various modes of Deflate compression. The length of a Deflate block is unknown until the EOB code has been found. LEN denotes code lengths and NLEN denotes number of symbols with each code length. In the Deflate format, all of the data elements (i.e., Huffman codes, block headers, code lengths, etc.) are packed into bytes starting with the least significant bit of the byte [79].

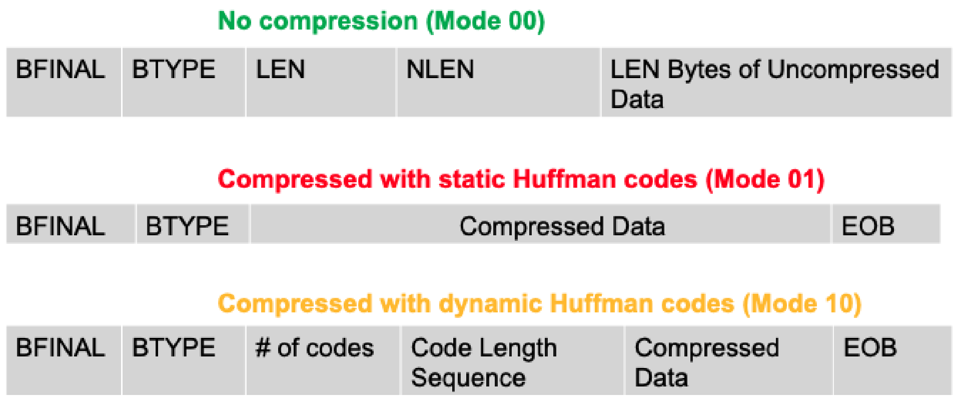


Figure 2.14: Deflate header configuration [8].

2.5.5 Related Work

Over the years several Gzip-based encoder designs have been proposed on CPUs, GPUs, FPGAs, and ASICs. Normally, the best string matching algorithm on software for LZ77 uses a sliding window size of 32 KB. However, processing speed of such a window is very slow due to extensive string search. The sliding window size determines the speed as well as the compression ratio. Reducing the sliding window size can increase the performance; however, compression ratio gets degraded.

Most of the work on LZ77 compression focuses on parallel sliding widow design and intelligent hash function for better string match output. The use of parallel window size (PWS) parameter is very common with the recent Gzip designs. Overall performance of the compressor primarily depends on efficient string match logic and how the history buffer is stored. The second stage of the Gzip is canonical Huffman-based bit-level encoding and it can be either static or dynamic based on performance vs. compression ratio trade-offs.

Mohamed et al. [64] proposed the first FPGA-based Gzip compressor using Open Computing Language (OpenCL). The design uses a fully-connected hash dictionary and a static canonical Huffman encoder. The proposed design also uses duplicated hash dictionaries to avoid any possible conflicts. They reported a compression ratio of 2.17 for the Calgary corpus.

Researchers from Microsoft [80] proposed a scalable fully pipelined FPGA accelerator that performs LZ77 compression and static Huffman encoding. They employed a number of hash tables with a parameter called hash table depth (HTD), which is equivalent to the number of hash chain walks allowed in the original DEFLATE algorithm. To improve overall compression, the algorithm doesn't commit the matches immediately, but instead searches for another match at the next position. If a longer match is found, the algorithm truncates the previous match to a literal and repeats this process of lazy evaluation until it encounters a worse match. Otherwise, it emits the previous match and skips forward by the match length; this repeats until it covers the entire input data set.

Ledwon et al. [79] proposed a Gzip compressor on Xilinx Virtex UltraScale+ FPGA using high-level synthesis (HLS). They used a PWS of 16 bytes and only static Huffman encoding is performed, providing a fixed input compression rate of 16 bytes per clock cycle. The hash bank architecture consists of 32 banks, with 3 depth levels and 512 indexes per bank. The static canonical Huffman encoder was used to make the design simpler. The compressor achieved a geometric mean compression ratio of 1.92 across Calgary corpus.

In this dissertation, we investigate the effect of both static and dynamic canonical Huffman encoders on overall Deflate compression speed and compression ratio by implementing a Deflate compressor on ASIC and analyzing the performance, energy-efficiency, and compression ratio results on standard lossless compression benchmarks.

Chapter 3

Canonical Huffman Encoders on Many-core Processor Arrays

This chapter presents canonical Huffman encoder designs on a many-core processor array. The encoder designs exploit task-level parallelism for the entire encoder pipeline and details out a concurrent sorting, Huffman tree build, and code length computation technique for optimized performance, area, and energy-efficiency.

3.1 Design Space Exploration

The canonical Huffman encoder executes the following tasks before encoding the uncompressed data set.

- *Histogram* module parses the serial stream of symbols and computes the corresponding symbol-frequency distribution for the complete data stream.
- *Sorting* module receives the symbol-frequency distribution for the data set and sorts the symbols in non-decreasing order per symbol frequencies.
- *Huffman tree building* module receives the sorted symbol-frequency list and build the Huffman tree following the regular Huffman's algorithm.
- *Code length computation* module traverses the Huffman tree from root to leaf nodes and outputs the code length for each symbol.

- *Canonical code assignment* module receives the code length information for each symbol and generates canonical codewords following canonical code assignment algorithm.

Conventionally, all the above tasks are done serially. However, some of the recent work [46, 51, 52] tried to alleviate this issue by proposing concurrent operations across the encoder pipeline.

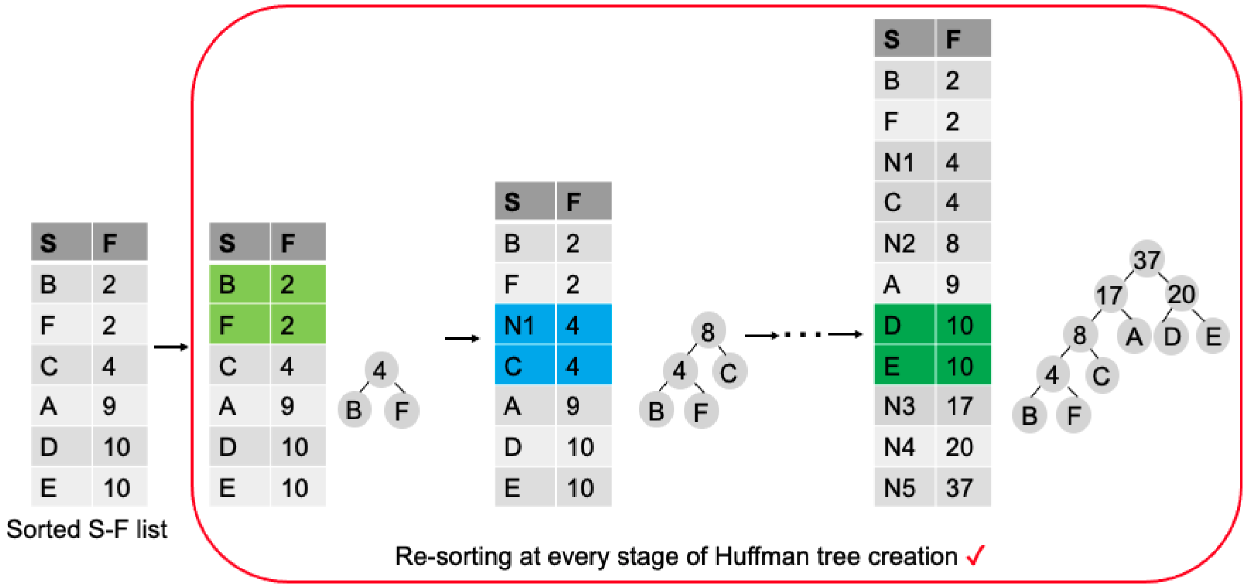


Figure 3.1: The sorted symbol-frequency list re-sorts after every intermediate node generation during Huffman tree creation.

3.1.1 Re-sorting of the Symbol - Frequency List

One of the computationally-intensive task is to do re-sorting of the symbol-frequency list after every new node generation during the Huffman tree creation process. This process needs several cycles of computation and hurts the encoder pipeline's latency. Therefore, the authors [46, 51] proposed a technique where in new intermediate nodes are stored in a separate list and those nodes are always sorted in non-decreasing order. This approach guarantees correct child nodes selection for every sub-tree creation and the same technique has been used in our designs.

Figure 3.1 shows the conventional steps of re-sorting the symbol-frequency list with every intermediate node generation. Figure 3.2 demonstrates with the addition of a new list for the intermediate nodes, re-sorting is avoided, while both approaches use the same sized lists in total.

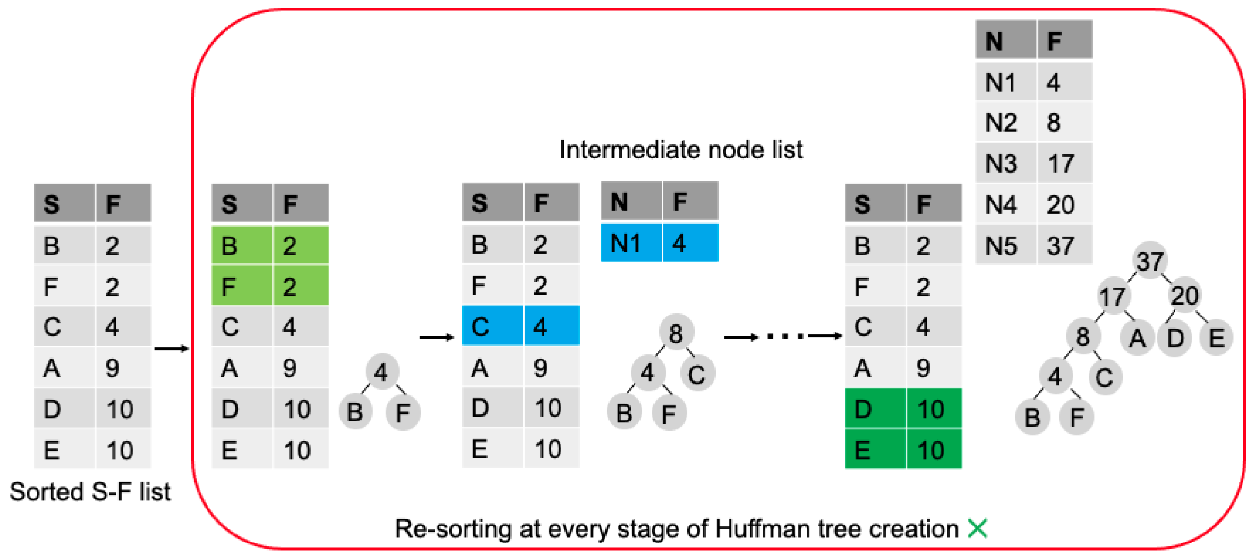


Figure 3.2: The sorted symbol-frequency list doesn't need to re-sort as there is an additional list for intermediate nodes, which makes sure correct child nodes are chosen for each sub-tree creation.

3.1.2 Concurrent Sorting, Huffman Tree Creation, and Code length Computation

Traditionally, sorting, Huffman tree building, and code length computation modules run sequentially. The researchers from Intel labs [51] proposed a concurrent sorting and Huffman tree creation method, where they demonstrated a design that initiates Huffman tree creation before complete sorting of symbol-frequency occurs. The idea behind the proposed design was originated from the necessity of only two symbols with smallest frequencies at any cycle of sub-tree creation.

Figure 3.3 shows iterative approach for depth computation, where pointer to parent's node traversal until the root node subsequently gives number of hops and hence, the code length for each symbol. The depth measurement task in this case starts only after the complete Huffman tree is created and highly iterative hurting the performance.

However, none of the works so far show a concurrent approach for the above three tasks. Figure 3.4 demonstrates an non-iterative approach for depth computation, where traversal through pointer to parent's node until the root node is not required and code length computation can concurrently start along with sorting and tree building process. The idea stem from the observation that sub-tree creation in case of Huffman tree can be either in horizontal or vertical direction, which

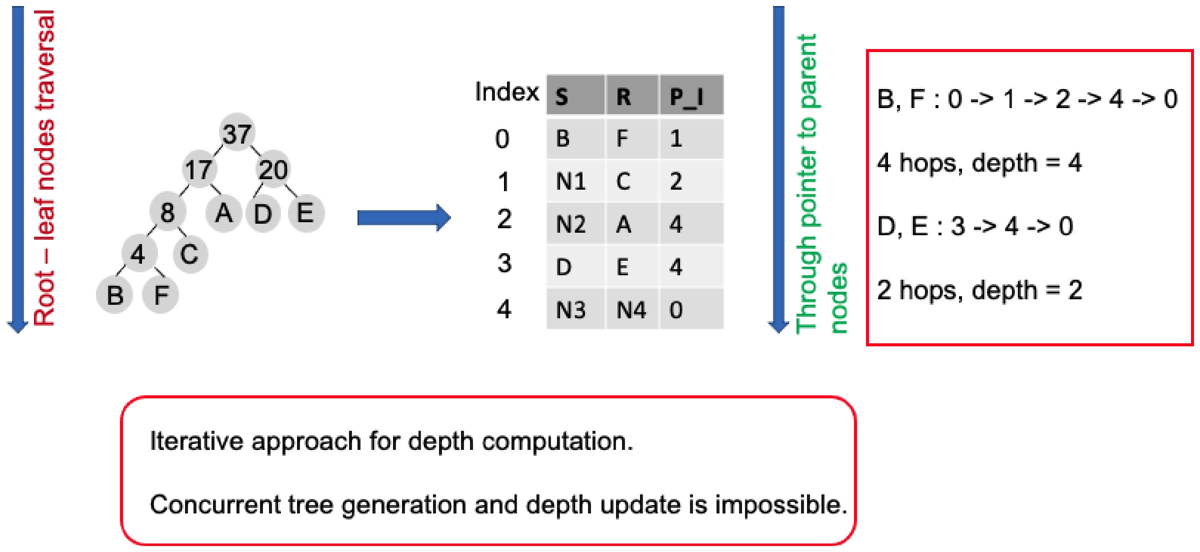

Root – leaf nodes traversal
37
17
20
8
A
D
E
4
C
B
F
Index
S
R
P_I
0 B F 1
1 N1 C 2
2 N2 A 4
3 D E 4
4 N3 N4 0
Through pointer to parent nodes
B, F : 0 -> 1 -> 2 -> 4 -> 0
4 hops, depth = 4
D, E : 3 -> 4 -> 0
2 hops, depth = 2
Iterative approach for depth computation.
Concurrent tree generation and depth update is impossible.

Figure 3.3: Depth measurement starts after creation of Huffman tree by traversing successive parent-children pointers until it reaches at root node and counting the hops in between.

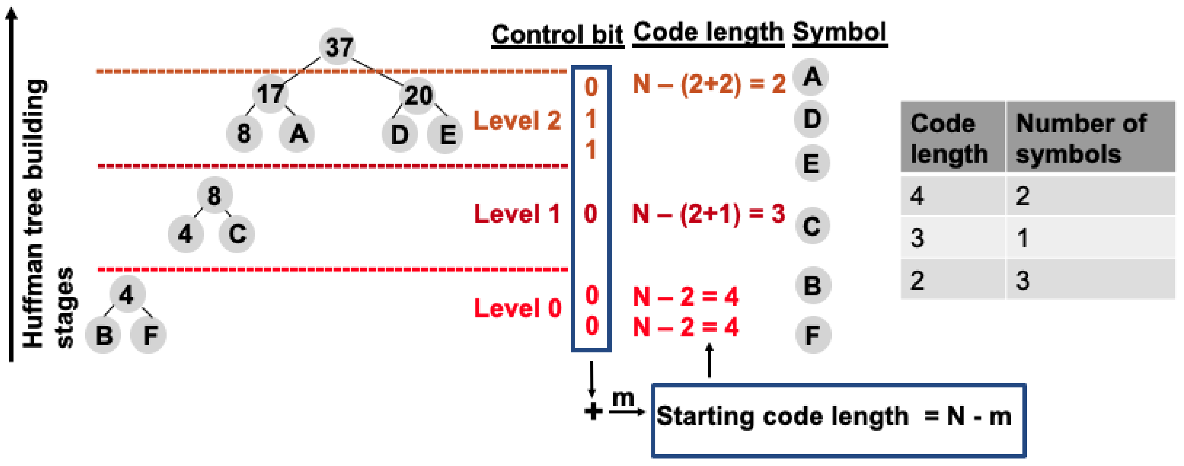


Figure 3.4: Huffman tree creation drives concurrent execution of depth measurement by observing the selection of nodes to create a sub-tree, enabling a non-iterative process.

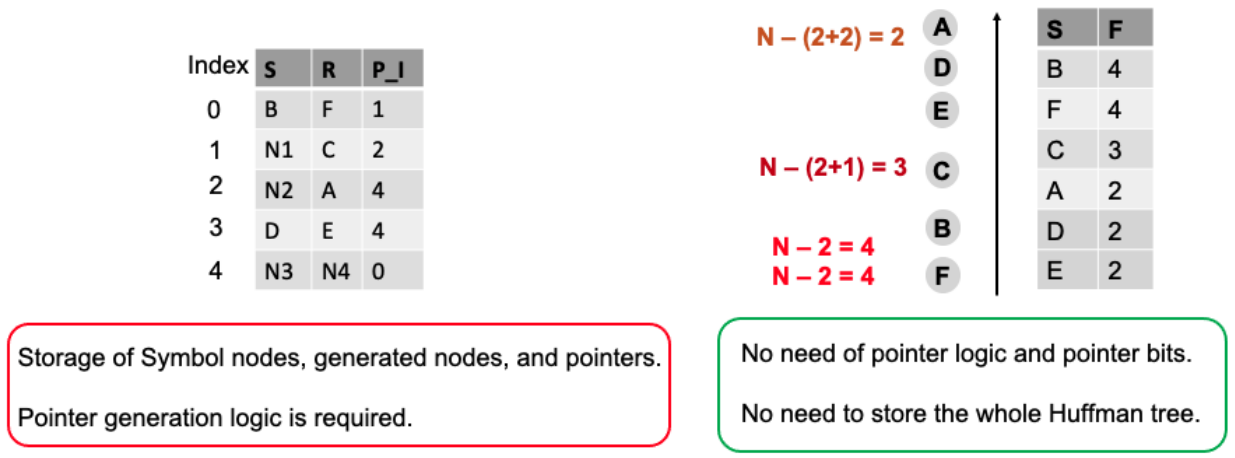
can be tracked with a single bit at every stage. Also, code length for every symbol in the same level or depth of the tree is same, and the maximum depth depends on total number of symbols in the data set.

This method doesn't apply to two special cases: a data set with all equi-probable symbols, where maximum code length can be computed as $\lceil \log_2 N \rceil$, where N denotes number of symbols in a data set. Another corner case with Huffman tree creation is when symbol frequencies are in the format of a Fibonacci series. In that case, whole tree grows in vertical direction, making maximum code length as $N - 1$. Both special cases have been considered in our design. However, with real compression benchmarks we haven't encountered such special cases.

3.1.3 Elimination of Storage of Complete Huffman Tree

Conventionally, whole Huffman tree is stored in memory along with a pointer for each parent node, which is useful or depth measurement. We investigate whether it's necessary to store complete Huffman tree when canonical Huffman encoder relies on code length and whole tree doesn't need to be sent out to decoder.

Figure 3.5 demonstrates that by adopting the proposed method of code length computation shown in the previous subsection, there is no need to store the complete tree and compute reference pointers for each node. Rather, the same symbol-frequency list, resulting after sorting can be updated with symbol-code length list.


Storage of Symbol nodes, generated nodes, and pointers.
Pointer generation logic is required.
N – (2+2) = 2
N – (2+1) = 3
N – 2 = 4
N – 2 = 4
No need of pointer logic and pointer bits.
No need to store the whole Huffman tree.

Index	S	R	P_I
0	B	F	1
1	N1	C	2
2	N2	A	4
3	D	E	4
4	N3	N4	0

S	F
B	4
F	4
C	3
A	2
D	2
E	2

Figure 3.5: Eliminating the storage of the complete Huffman tree enables non-requirement of creating pointer for each parent node.

3.2 Mapping of the Canonical Huffman Encoder Tasks

The mapping of the canonical Huffman encoders on a many-core processor array chip comprises of the following tasks.

Figure 3.6 shows the data flow diagram showing mapping of the tasks of the canonical Huffman encoder on a many-core processor array by using conventional approach using 68 processors and 4 SRAM modules.

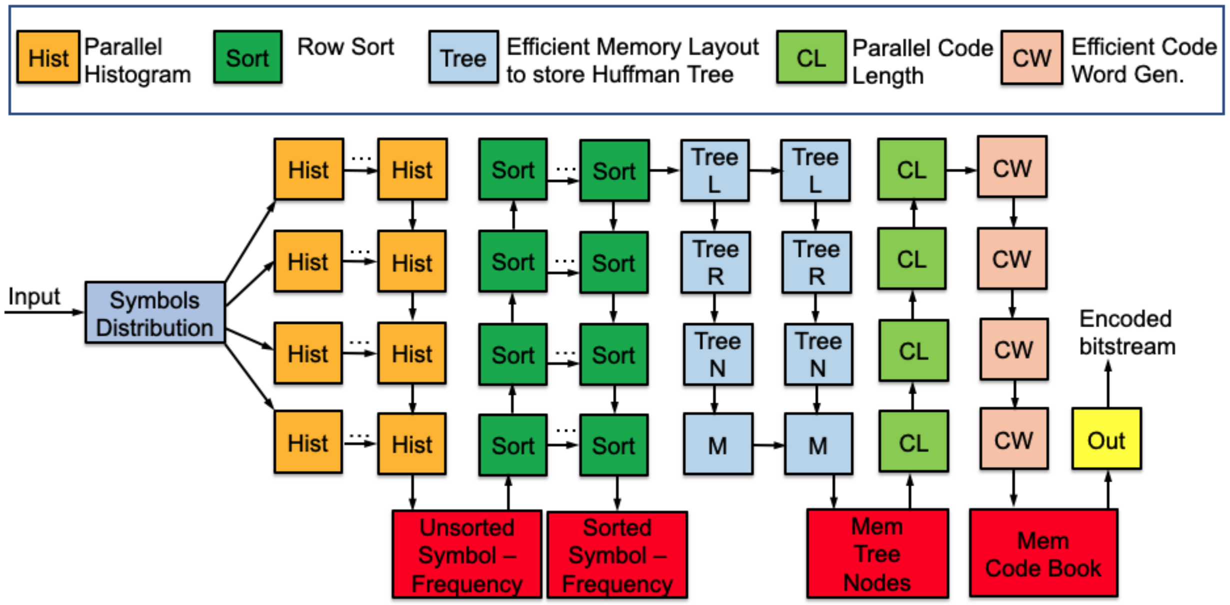


Figure 3.6: Data flow diagram showing mapping of the tasks of the canonical Huffman encoder on a many-core processor array.

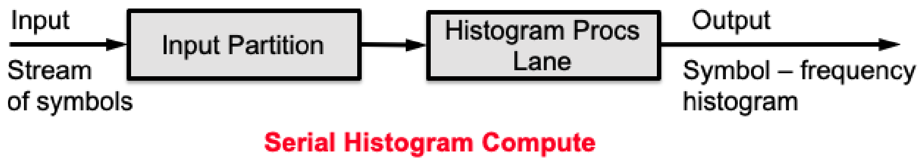


Figure 3.7: Serial histogram computation data flow diagram. The processors in a single lane are designed to obtain the symbol frequency pairs in a serial fashion.

3.2.1 Histogram Computation

The histogram computation block serially receives the stream of symbols and computes the corresponding frequency for each symbol in an uncompressed data set. The straightforward serial approach to compute symbol-frequency on the many-core processor array is to serially streaming the symbols through a lane of processors that are programmed to increment the symbol-frequency and

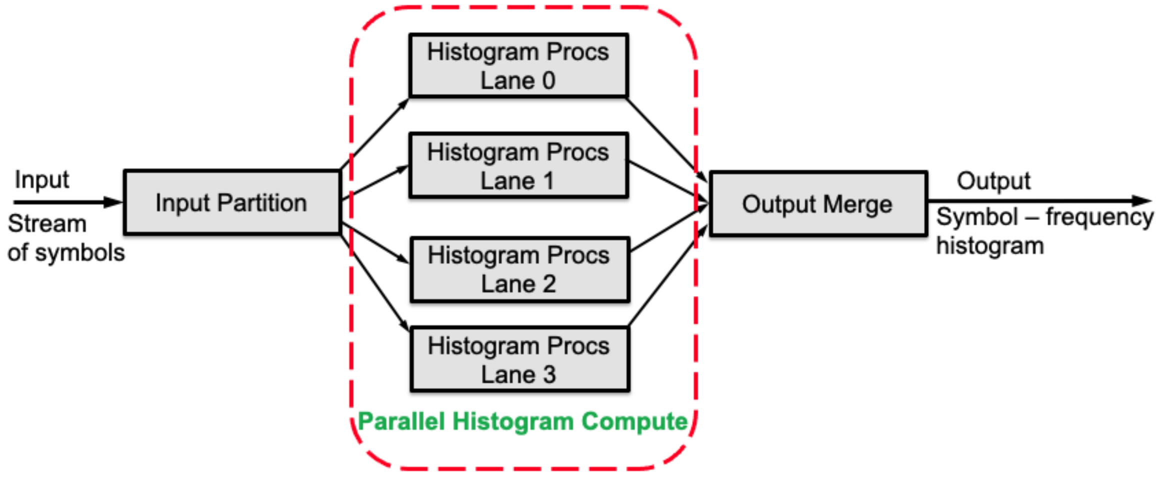


Figure 3.8: Parallel histogram computation for throughput improvement. The intermediate results from individual processor lanes are merged to obtain the final symbol frequency pairs.

then pass on the computed histogram through communication processors to store them in a SRAM block for the next task of encoder. For example, Figure 3.7 shows that a lane of 16 processors can be used to find the histogram data of 512 symbols by adopting serial partitioning approach, where each processor is assigned to count up frequency of 32 symbols.

We achieve better throughput for the histogram block by adopting data level parallelism. Data parallelism approach divides the input stream of symbols into different chunks of data that are processed through lane of processors and finally, histogram output per lane are merged to provide the total histogram of symbol-frequency. The maximum frequency for each symbol is limited to 65535.

The input partition processor pass on the stream of symbols to one of the four lanes as shown in Figure 3.8. The processors in the same row assigned for partial histogram task will buffer the symbols, compare them with the address index and increment the corresponding counter by one for any match. In that way each lane of processors will compute per-lane histogram information. The lane of processors in each row then downstreams the data to merge processors. Merging processors will accumulate partial histogram data for each symbol and finally, compute the total histogram. In total, 37 processors are used for parallel histogram computation, that achieves $3.7\times$better throughput than the serial implementation at the cost of extra 28 processors.

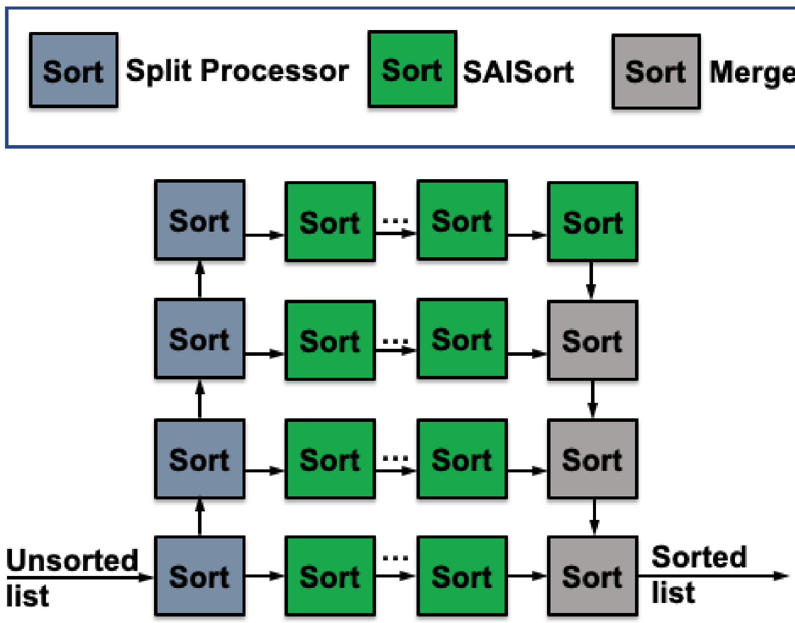


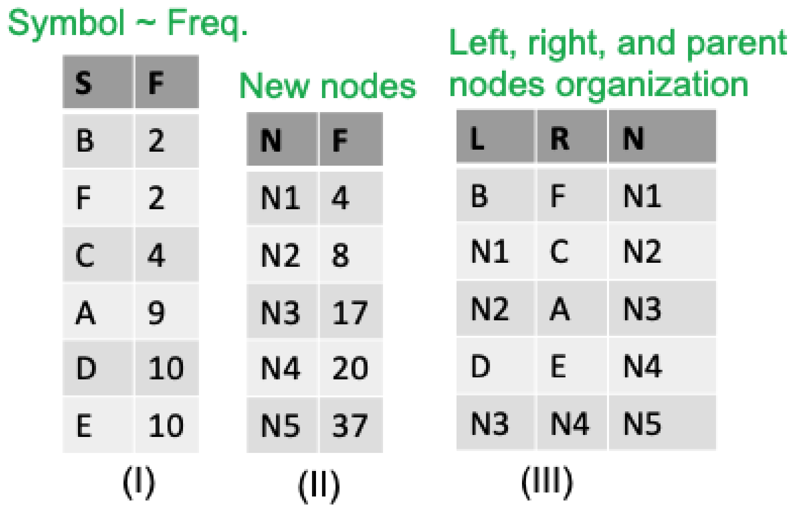
Figure 3.9: Mapping of the sorting kernel using Row sort. Row sort kernel uses split processors to split up the input list and distributes them to the row of SAISort processors. Finally, the output from the SAISort processors are merged through the merge processors to result the sorted list.

3.2.2 Sorting

The sorting module sorts the symbols per increasing order of their frequencies per symbol-frequency histogram and results a list of sorted symbol-frequency. We adopt the proposed row sort architecture for mapping of the sorting operation into the many-core array that utilizes simple kernels and fit within the target many-core architecture [40].

In row sort, when stream of data enters the processor array, a set of processors first split up the data and distribute the input record into each of the rows in the array. Next, each row sorts the given input and results in intermediate sort output. The intermediate sort output from each row are then merged using merge processors to result the final sorted list. The two-stage mapping of the row sort operation enables parallel sorting operation and speeds up the data path for any given list.

First, the four processors in the first column splits up the unsorted list and pass them to four rows, which perform the first pass sorting based on serial array of insertion sorts, or SAIsort, using eight processors [40]. The results from lane of processors doing first pass sorting gets passed to the merge processors. Four merge processors are used to merge intermediate sort data to produce the final sorted list. Next, the sorted list is written into the sorted symbol-frequency memory.


Symbol ~ Freq.

S	F
B	2
F	2
C	4
A	9
D	10
E	10

(I)

New nodes

N	F
N1	4
N2	8
N3	17
N4	20
N5	37

(II)

Left, right, and parent nodes organization

L	R	N
B	F	N1
N1	C	N2
N2	A	N3
D	E	N4
N3	N4	N5

(III)

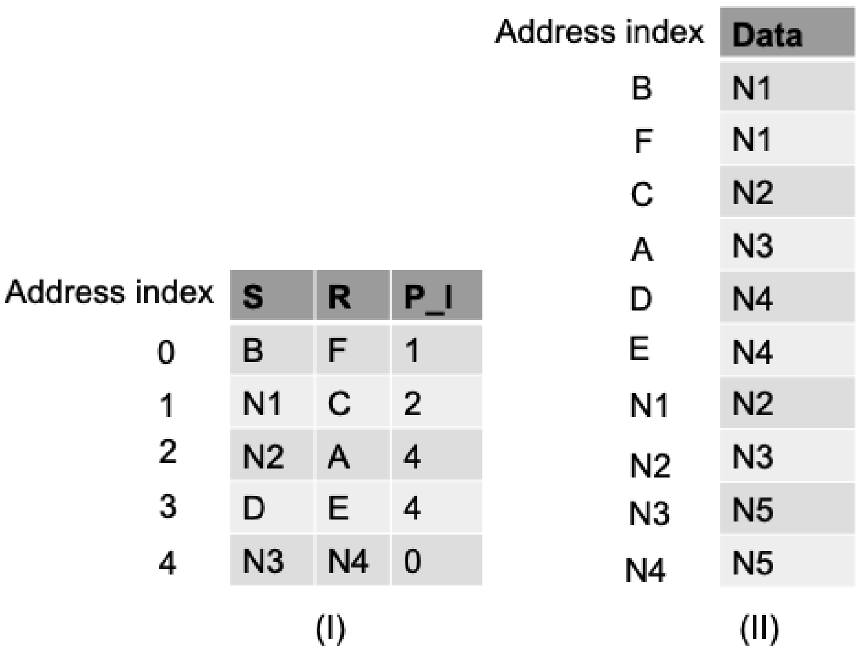
Figure 3.10: Storage of the Huffman tree's nodes in a list-based memory layout. (I) Symbol frequency distribution table (II) New nodes frequency table (III) Lists to store symbols, their left, right, and parents nodes

3.2.3 Huffman Tree Generation

The Huffman tree kernel works in parallel with sorting to speed up execution and improve throughput. The idea of concurrent execution of both tasks stem from the idea of the sorting results in creating a new node only once at a time. Therefore, the tree node creation can be initiated once first pass sorting is completed.

There are three separate lists that are mapped to lane of processors to store all nodes of Huffman tree. It supports any distributing of Huffman tree creation. Figure 3.16(I) shows the first list that contains symbols and their frequencies, all of them are sorted per frequency, resulting from the *sorting* module. Figure 3.16(II) shows the second list that contains new nodes and their respective frequencies.

The way we have designed the second list of new nodes is that we can avoid sorting of the whole list every time a new node is generated. This architecture follows the proposed algorithm [46], where in first two minimum elements from the sorted symbol-frequency list is first chosen. Next, an intermediate node *N1* is added to the new node-frequency list. Next, the algorithm selects the next element *E1* from the sorted symbol list and compares *E1* and *N1*. If the frequency of *E1* is smaller



Address index	S	R	P_I
0	B	F	1
1	N1	C	2
2	N2	A	4
3	D	E	4
4	N3	N4	0

(I)

Address index	Data
B	N1
F	N1
C	N2
A	N3
D	N4
E	N4
N1	N2
N2	N3
N3	N5
N4	N5

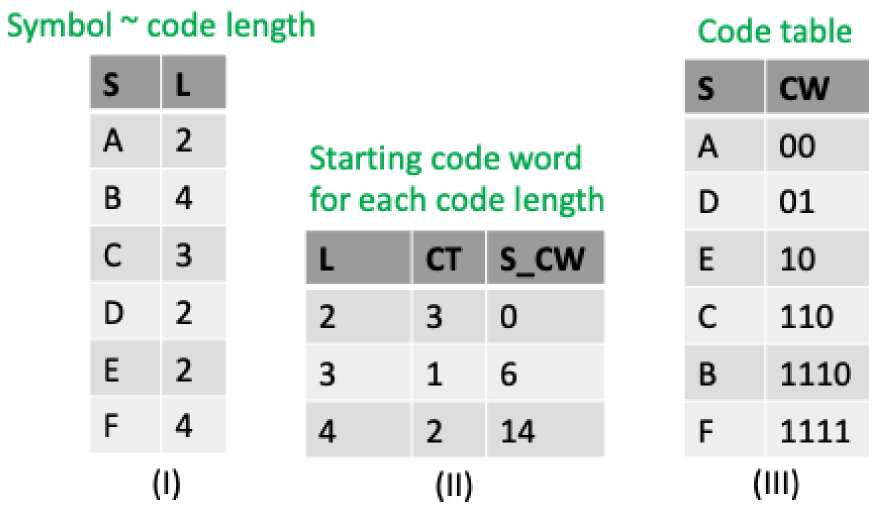
(II)

Figure 3.11: Efficient storage of the Huffman tree's nodes along with information on parental nodes. (I) Architecture with a pointer to each node that stores the corresponding parent nodes [46]. (II) Direct address index access using the child node that hashes out the corresponding parent node.

than frequency of *N1*, *E1* is the left child. Otherwise, *N1* is the right child. If there is no element left in the intermediate list after selecting the left child, next element from the sorted symbol list is chosen and made as right child. This process continues until all the elements in the sorted symbol list are processed. Therefore, this tree building process with a separate list for the intermediate nodes avoids re-sorting every time an intermediate node is generated. Symbol-frequency list and new node lists are stored in 4 processors.

3.2.4 Code Length Generation

The primary task of the code length computation unit is to first obtain the depth of every node in the Huffman tree. Normally, the steps include traversing the whole Huffman tree from root to any nodes and thereby, the critical path delay becomes huge. In the architecture shown in Figure 3.11(I), for each node we need to find parent's index address *P_I* and thereby, we can subsequently have the total depth by finally reaching at the parent index address of 0. For example, symbols "B" and "F" points to address index 1, address index 1 points to address index 4, and


Symbol ~ code length

S	L
A	2
B	4
C	3
D	2
E	2
F	4

(I)

Starting code word for each code length

L	CT	S_CW
2	3	0
3	1	6
4	2	14

(II)

Code table

S	CW
A	00
D	01
E	10
C	110
B	1110
F	1111

(III)

Figure 3.12: Code length generation process: (I) Symbols vs. code length (II) Number of symbols for each code length (III) Starting code word for each code length

finally, address index 4 points to address index 0. Therefore, code length of symbols “B” and “F” is 4.

However, this process requires to store pointer to parent’s address index, along with both child nodes as a memory word, which may not fit within the memory word size constraints for some computing platforms. Therefore, we have proposed a memory layout that needs to store only the parent nodes and nodes can be directly used to index the memory to hash out parent node, which can be subsequently used to reach at the final node of the tree, which is the root node. In this case, each memory access will be incremented by one starting for any symbol and final value of the counter gives the code length for any symbol once the final memory access gives out the root node. We need to store the tree nodes in maximum 1023 memory words for 512 symbols. Figure 3.11(II) shows the memory layout which is mapped to a single SRAM block of the many-core processor array.

3.2.5 Canonical Code Assignment and Encoding

Figure 3.12 describes the code word generation process. This includes first finding code length for each symbol from depth kernels. Next, a code length processor acts as a counter to store total number of symbols for each code length.

Finally, using the algorithm by deflate [8] we can find starting code word for each symbol.

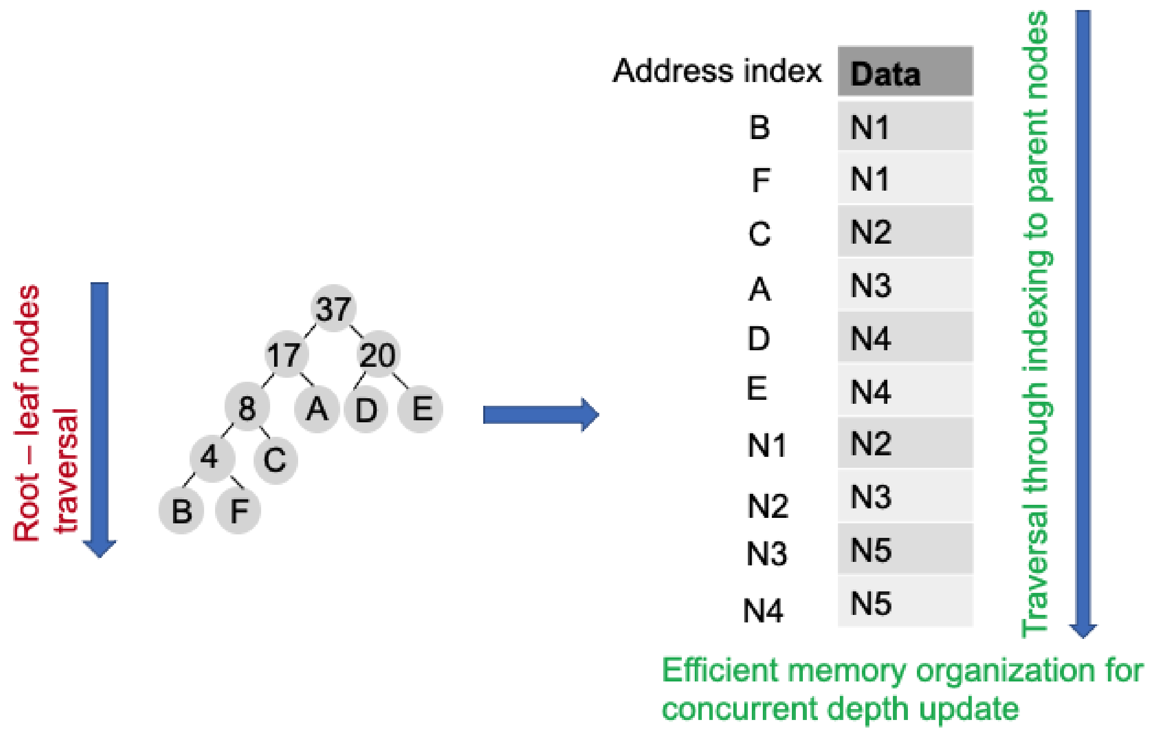

Root – leaf nodes traversal
37
17
20
8
A
D
E
4
C
B
F
Address index
Data
B N1
F N1
C N2
A N3
D N4
E N4
N1 N2
N2 N3
N3 N5
N4 N5
Traversal through indexing to parent nodes
Efficient memory organization for concurrent depth update

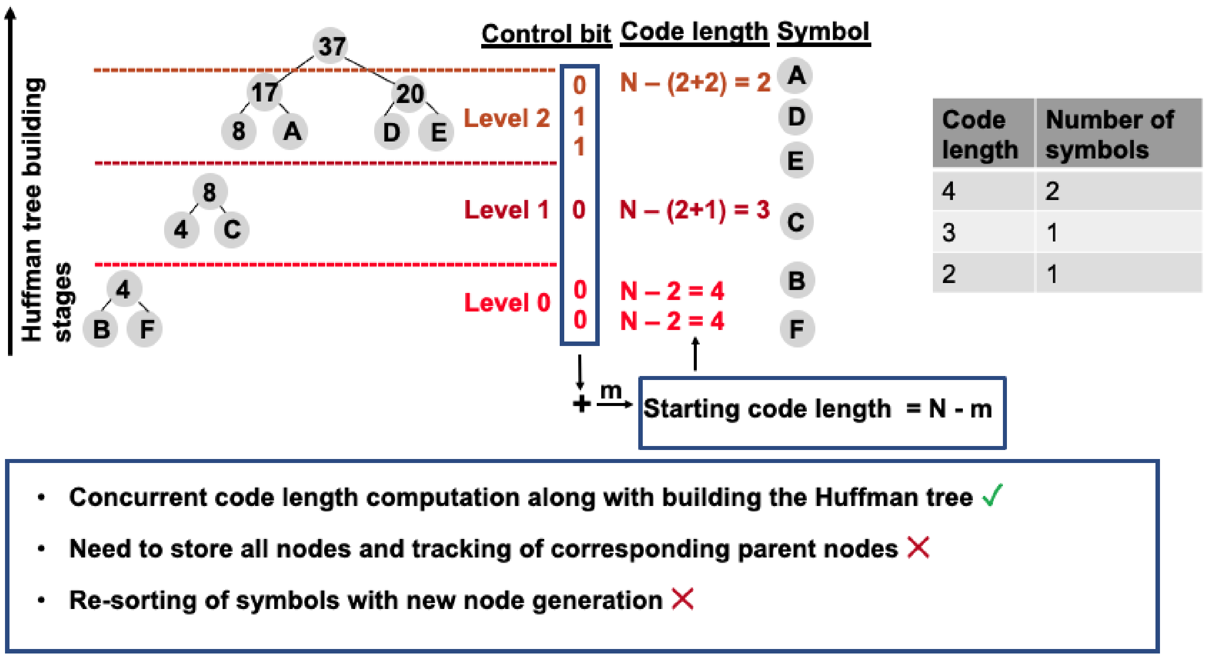
Figure 3.13: Serial vs. parallel depth measurement. In parallel approach, depth of any node can be obtained by iterating through parent node of each node via address indexing. However, in the case of serial approach complete traversal of root-nodes is required, which slows down the process.

The efficient codeword generation processors are responsible for finding the starting codeword for each code length first, and then, incrementing codeword by one subsequently for rest of the symbols with same code length.

Once the code words for each symbol gets computed, the processors at the downstream direction writes the codeword to the linked big memory for each symbol, which is the address index for corresponding write operation. Once codewords for all symbols are written into the code book memory, input symbols are read and encoding of each symbol happens by writing out the corresponding codeword from each processor.

3.2.6 Concurrent Sorting, Huffman Tree Building, and Code Length Computation Architecture

Conventional canonical Huffman encoder architecture requires complete Huffman tree creation, storing all child and corresponding parent nodes, and subsequently, traversing through



Code length	Number of symbols
4	2
3	1
2	1

- Concurrent code length computation along with building the Huffman tree ✓
- Need to store all nodes and tracking of corresponding parent nodes ✗
- Re-sorting of symbols with new node generation ✗

Figure 3.14: Proposed architecture for concurrent Huffman tree building and code length computation.

the memory through multiple iteration for every parent node address to find the code length. We proposed an efficient approach to execute one pass sorting, both Huffman tree building and code length computation together, which is discussed in Section 3.1.

The proposed architecture stems from the idea that canonical Huffman encoder doesn't need to send the Huffman tree to the decoder and the codeword computation unit only needs code length information to complete the canonical coding step. Therefore, it may not be necessary to store the parent node address index for all nodes and execute multiple iteration-based code length computation.

Figure 3.14 shows the proposed architecture for concurrent Huffman tree building and code length computation. In the proposed architecture, a bit vector is updated with the bit "0" as long as the current sub-tree is at the same level, which is denoted by accessing either two symbols from the symbol list or new node list. With the level increasing up, the bit vector is updated with the bit "1". This process is adopted to keep track of the symbols that cater to Huffman tree growing in horizontal direction. Finally, adding the bit map contents and subtracting from the number of symbols gives us the starting code length (i), which is the code length for the two symbols with the smallest probabilities. Rest of the code lengths can be computed in parallel by doing the computation as shown in Figure 3.14 using both total number of symbols and number of extra symbols present at each level.

The data flow mapping of the proposed architecture on the many-core processor array is similar to the baseline implementation except, the Huffman tree processors computing tree nodes and pointer references are no longer required. Instead, two processors are used to store the bit map and pass along to code length computation processor column to compute code length in parallel. The proposed architecture also reduces the big memory requirement, which is used to store tree nodes.

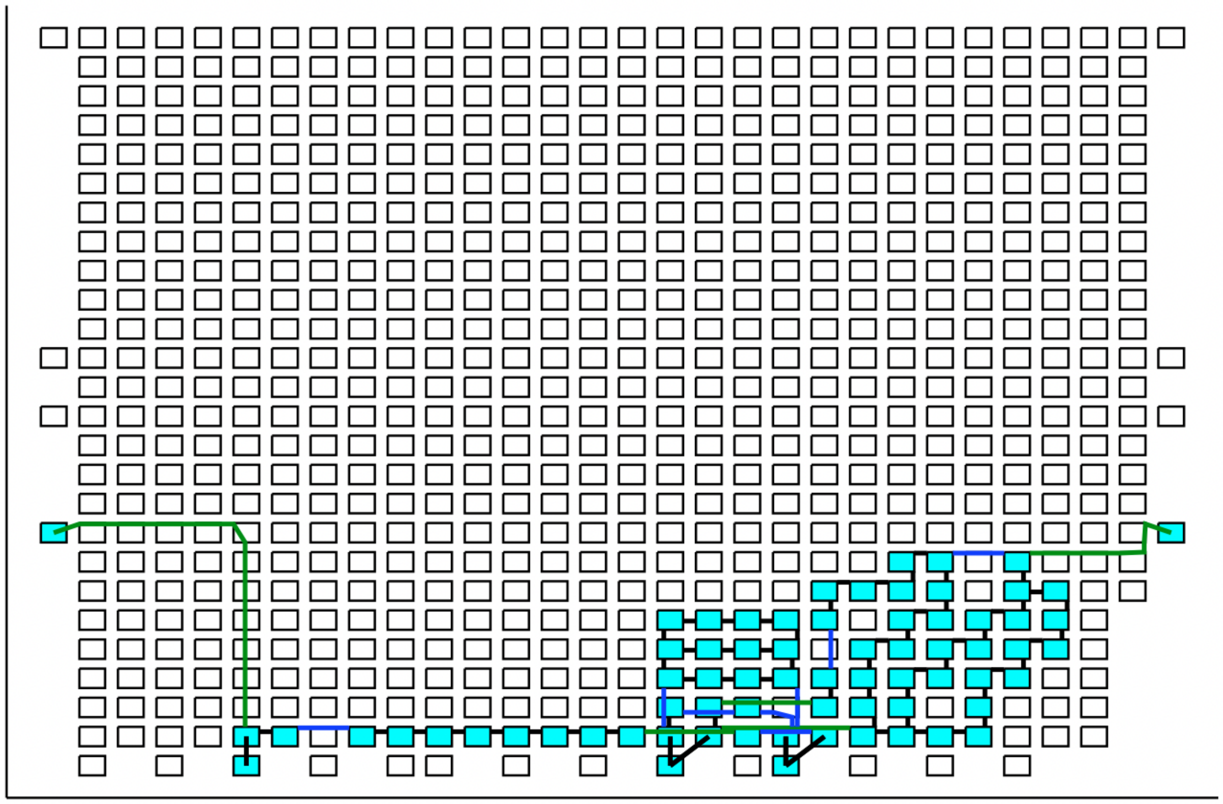


Figure 3.15: The entire optimized encoder application is mapped to 66 processors, three 64 KB memory (on the bottom of the array), and I/O ports on the edge of the array.

Table 3.1: Throughput per area and Compressed bits encoded per energy data for KiloCore implementations at supply voltages of 0.56 V and 1.1 V.

Benchmark	VDD (V)	Throughput/Area (Mbps/mm^2)	Compressed Bits/Energy (Bits/µJ)
Artificial Corpus	0.56	8.5	202.3
	1.1	13.3	97.1
Calgary Corpus	0.56	22.3	489.8
	1.1	35.5	238.8
Canterbury Corpus	0.56	11.1	263.6
	1.1	18.2	129.9
Large	0.56	2.3	46.6
	1.1	3.6	22.9

3.2.7 Application Task Mapping into Many-Core Array

For the optimized encoder design using concurrent sorting, Huffman tree building, and code length computation architecture to execute on the processor array, it must first be mapped onto the array given its particular constraints in processors, memory, and I/O. A specifically-designed CAD tool for the many-core processor array performs a one-to-one mapping of the tasks onto physical processors. In addition, the mapping tool allocates chip interconnects to task graph communication links. Figure 3.15 shows the completed mapping of the optimized decoder application onto a many-core processor array using a total of 66 processors and three memory modules.

3.3 Results and Discussions

This section discusses the benchmarks, simulation set up, results of many-core implementations, and presents comparisons with the optimized Intel i7 and GPU software implementations.

3.3.1 Benchmarks and Computing Platforms

We evaluate the canonical Huffman encoder designs on the following computing platforms.

- Intel i7-4850HQ containing four cores and eight threads with a base clock frequency of 2.3 GHz [21].

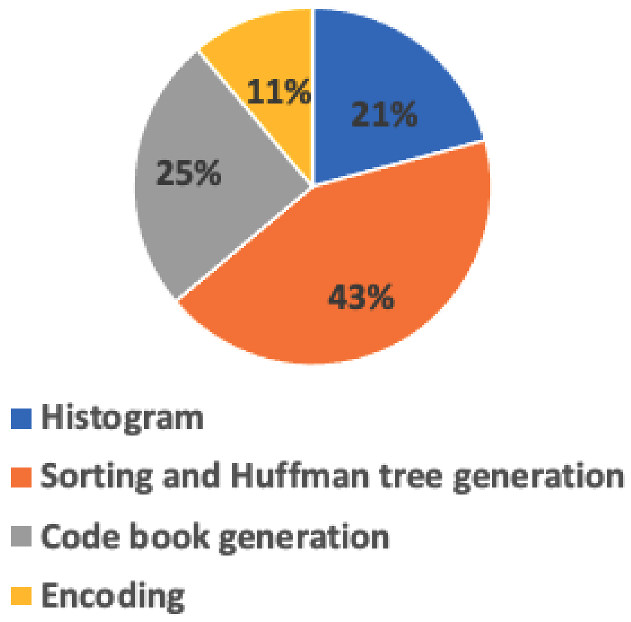


Figure 3.16: The execution time distribution of canonical Huffman encoding, where critical path lies in the sorting and Huffman tree creation modules.

- Nvidia GeForce GT 750M with 384 shader cores running at a base clock of 941 MHz [22].
- KiloCore a 1000-processor array with each processor running at 1.78 GHz [17].

We consider various lossless text compression benchmarks (Artificial corpus, Calgary corpus, Canterbury corpus, and Large corpus) to validate the canonical Huffman enocder designs and compare the performance and compressed bits decoded per energy of the computing platforms. We verify the functionality of each design first before conducting the performance evaluation. The benchmarks cover a wide range of files of different sizes and symbol distribution.

3.3.2 Simulation Setup

We simulate the KiloCore designs using a cycle-accurate simulator which is modeled based on the fabricated chip data [17]. First, the simulations are conducted with the base supply voltage of 0.9 V to analyze the key parameters of throughput, area, power, and workload per unit energy. Later, we scale the supply voltage to 1.1 V to see the effect of voltage scaling on performance and energy efficiency while preserving the functionality of each design.

We report the simulation results for Intel i7 and Nvidia GT 750M GPU after speeding up the execution time by triggering the –O3 flag for both gcc and nvcc compilers respectively. We have referred to the source code for canonical Huffman encoder implementation [81] for Intel i7-4850HQ implementation. The source code has been modified for bit-parallel LUT-based decoding. For power calculations for the Intel i7-4850HQ chip, TDP/2 = 23.5 W is used since the actual power dissipation is not known [59]. The die area of the Intel i7-4850HQ is reported as 260 mm^2 [21].

We have validated the encoder [61] on a CUDA-enabled GT 750M GPU with compute capability 3.0 and a TDP ratings of 50 W. Therefore, we consider a TDP/2 = 25 W for power calculations since the actual power dissipation is unknown. The die area of the GT 750M GPU is reported as 118 mm^2 [22].

We have also compared many-core implementations with Nvidia Tesla V100 GPU-based two most recent work using enwik8 data set. The Tesla V100 GPU has a TDP ratings of 250 W. Therefore, we consider a TDP/2 = 125 W for power calculations since the actual power dissipation is unknown. The die area of the GT 750M GPU is reported as 37045.5 mm^2 [22].

3.3.3 Canonical Huffman Encoders on KiloCore Array

The resulting area, throughput, throughput per area, and energy efficiency data are reported in Table 4.2 for the optimized implementations. The optimized implementation achieves a throughput that is **18%** greater on average than the baseline implementation. Similarly, the optimized implementation depicts an energy efficiency improvement of 33% greater and area utilization reduction of **5%** on average than the baseline implementation.

3.3.4 Comparisons of Results for Intel i7, Nvidia GT 750M, and KiloCore Array Implementations

Table 4.2 compares the performance of the canonical Huffman decoder designs on a variety of programmable hardware: general-purpose Intel i7 CPU, Nvidia GT 750M GPU, and KiloCore processor array. The i7, FPGA, and GPU values are scaled to 32 nm (technology for KiloCore) to account for differences in technology generations using scaling data by the DeepScaleTool [20].

The GPU implementation its huge die area results an average throughput improvement of **8.5**× and **5**× when compared to optimized KiloCore and Intel i7 implementations respectively. However, in terms of both throughput per chip area and energy efficiency the proposed many-core processor array implementations have clear advantages—the optimized KiloCore design gives an

Table 3.2: Throughput per area and Compressed bits encoded per energy data for Intel i7, Nvidia GPU, and Many-Core (KiloCore) implementations. The i7, and GPU values are scaled to 32 nm (technology for KiloCore) to account for differences in technology generations using scaling data by the DeepScaleTool [20].

Benchmark	Platform	Throughput (Mbps)	Area (mm^2)	Throughput/Area ($Mbps/mm^2$)	Bits/Energy (Bits/µJ)
Artificial Corpus	i7-4850HQ [81]	98.5	548.5	0.2	4.2
	GT 750M [61]	**525.3**	154.9	3.4	21.1
	Many-Core	65.6	**4.1**	**16.1**	**202.3**
Calgary Corpus	i7-4850HQ [81]	263.5	548.5	0.5	11.2
	GT 750M [61]	**1405.6**	154.9	9.1	56.2
	Many-Core	175.7	**4.1**	**42.6**	**489.8**
Canterbury Corpus	i7-4850HQ [81]	135.4	548.5	0.2	5.8
	GT 750M [61]	**722.4**	154.8	4.7	28.9
	Many-Core	92.3	**4.1**	**22.5**	**263.6**
Large	i7-4850HQ [81]	26.7	548.5	0.1	1.1
	GT 750M [61]	**141.9**	154.8	0.9	5.7
	Many-Core	19.7	**4.1**	**4.8**	**46.6**
enwik8	Tesla V100 [82]	7400	37045.5	0.2	19.7
	Tesla V100 [58]	**34800**	37045.5	0.1	92.7
	Many-Core	224.5	**4.1**	**40.9**	**131.5**

throughput per area improvement of **4.7**× and **89.2**× over GT 750M GPU and Intel i7 respectively and an energy-efficiency improvement of **8.9**× and **44.7**× over GT 750M GPU and Intel i7 respectively. Figure 4.8 shows the throughput per area improvement for KiloCore and GT 750M GPU when normalized to Intel i7 Similarly, Figure 8.2 compares the gain in energy efficiency for KiloCore and GT 750M GPU when normalized to Intel i7.

We have also validated the encoder implementations on KiloCore using enwik8 data set and throughput per area and Compressed bits encoded per energy data comparisons with an optimized GPU implementation is shown in Table 4.2. The KiloCore implementations achieve throughput per area improvement of **58**× and **272**× than the most-recent work [58] and [82], respectively. Similarly, the KiloCore implementations achieve energy-efficiency improvement of **4.8**× and **22.7**× than the most-recent work [58] and [82], respectively. Figure 3.19 shows the scatter plot comparing

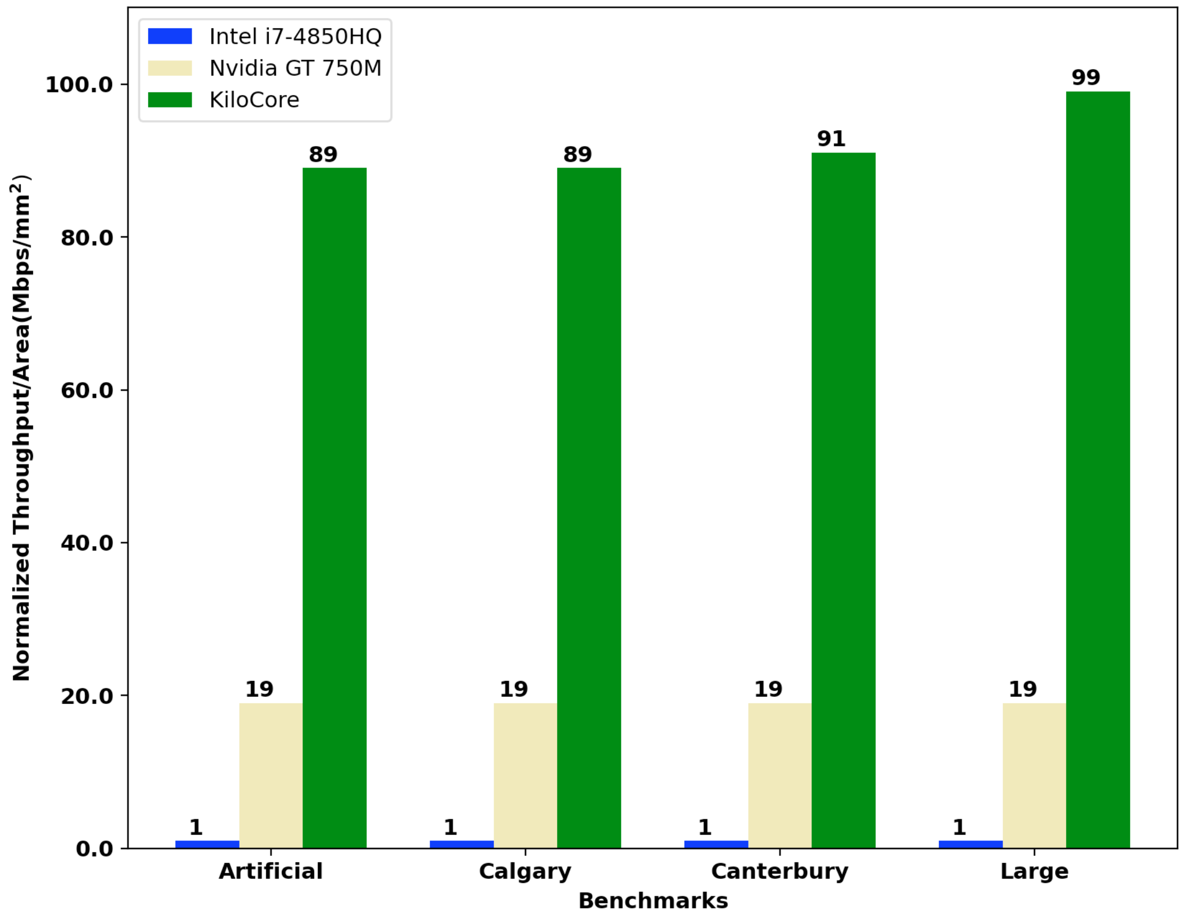


Figure 3.17: Comparison of scaled throughput per area (Mbps/mm^2) for the many-core array (KiloCore) and Nvidia GeForce GT 750M GPU, normalized to Intel I7-4850HQ.

the encoder results for the enwik8 data set.

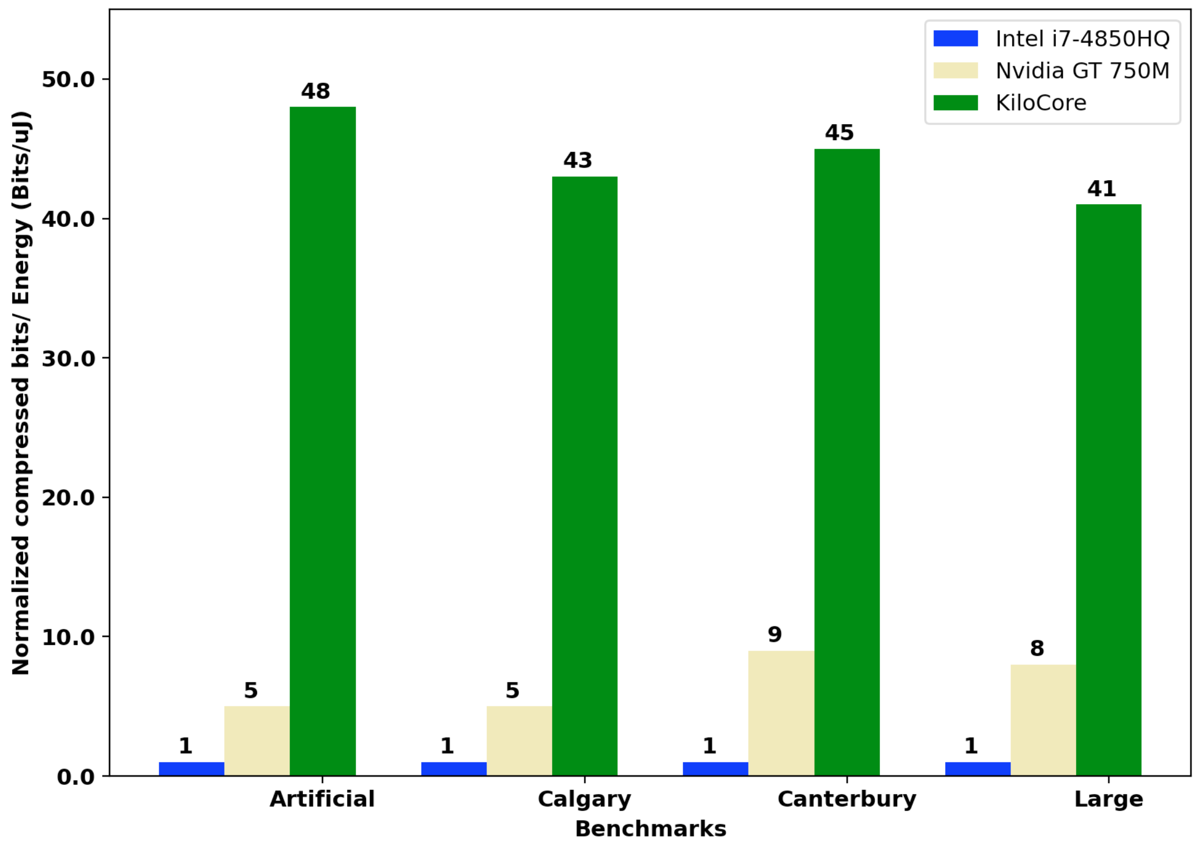


Figure 3.18: Comparison of scaled energy efficiency (Compressed Bits/µJ) for the Nvidia GeForce GT 750M GPU and many-core array (KiloCore), both normalized to the Intel i7-4850HQ.

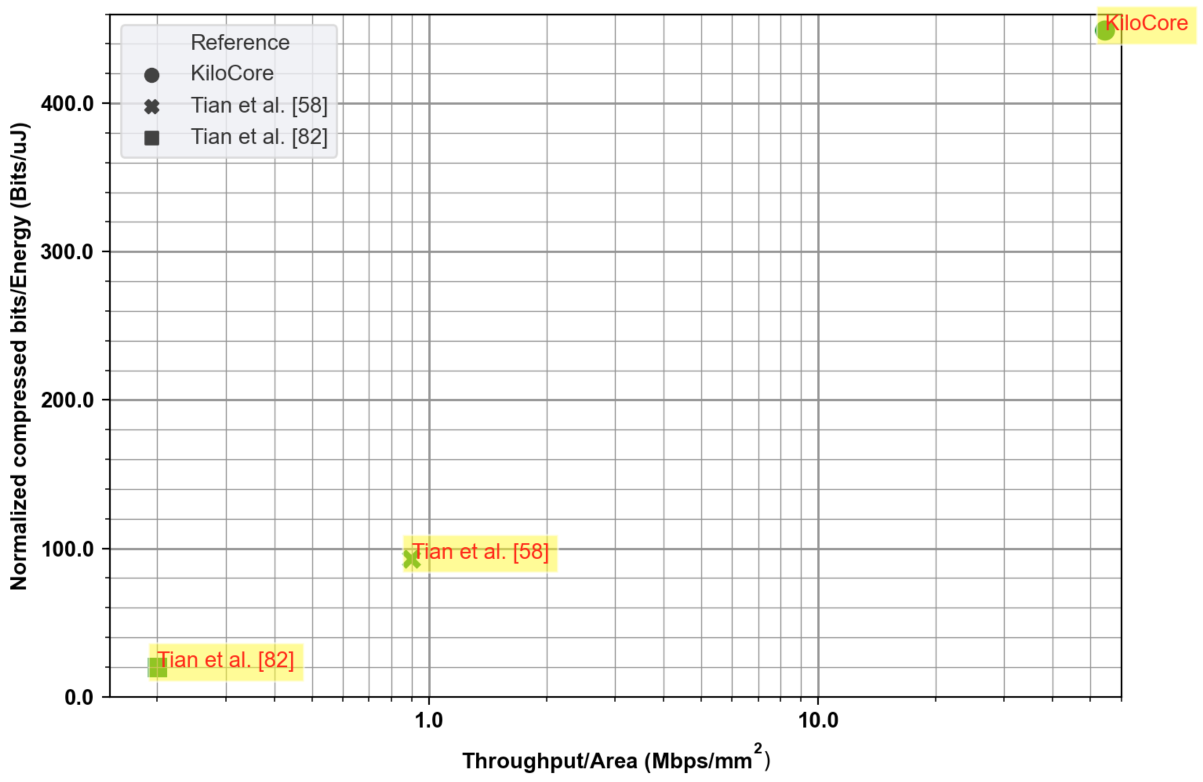


Figure 3.19: Comparison of scaled energy efficiency (Compressed Bits/µJ) and scaled throughput per area (Mbps/mm^2) for the canonical Huffman encoder implementations on enwik8 data set.

Chapter 4

Canonical Huffman Decoders on Many-core Processor Arrays

This chapter presents canonical Huffman decoder designs on a many-core processor array. The decoder designs use serial task partitioning-based code table construction and an optimized LUT-based indexing architecture.

4.1 LUT-based Indexing Architecture

The canonical Huffman decoder architecture primarily comprises of the two tasks: creating a code table and indexing the code table to lookup a symbol. For the static canonical Huffman decoder, the code table is pre-defined and the bitstream indexing task follows the same process as of the dynamic canonical Huffman decoder. In case of dynamic decoder the code table needs to be dynamically generated per bitstream header, which slows down the overall execution.

Generally, canonical Huffman decoding is performed by indexing encoded bits to hash out a symbol from a LUT. LUT-based indexing is faster than any other approach. The LUT can be of single or multi-level based on area and performance trade-offs. A single-level LUT can hash out a decoded symbol in a clock cycle, while multi-level LUT takes multiple clock cycles; although, in case of later, memory footprint to store the LUT is lesser than the earlier.

4.1.1 Baseline Architecture

The baseline architecture [59] follows a big LUT indexing approach, where input codewords are directly indexed into a LUT of depth of $(1 << maximumcodelength)$. Figure 4.1 shows the baseline architecture for dynamic canonical Huffman decoder. First, the starting code word for each code length is computed using shifters and adders. Next, the rest of the symbols with same code length gets assigned codewords by incrementing the codeword of the predecessor symbol by one due to the contiguous codeword assignment property of the canonical Huffman code. This is done in parallel as the number of symbols per code length is provided from the header. The codewords are then indexed into the lookup table that consists of the words with symbols and code length information. The symbol and code length sets are replicated for the corresponding LUT index as shown in the work [59]. The final task is to index the lookup table to decode the corresponding symbol.

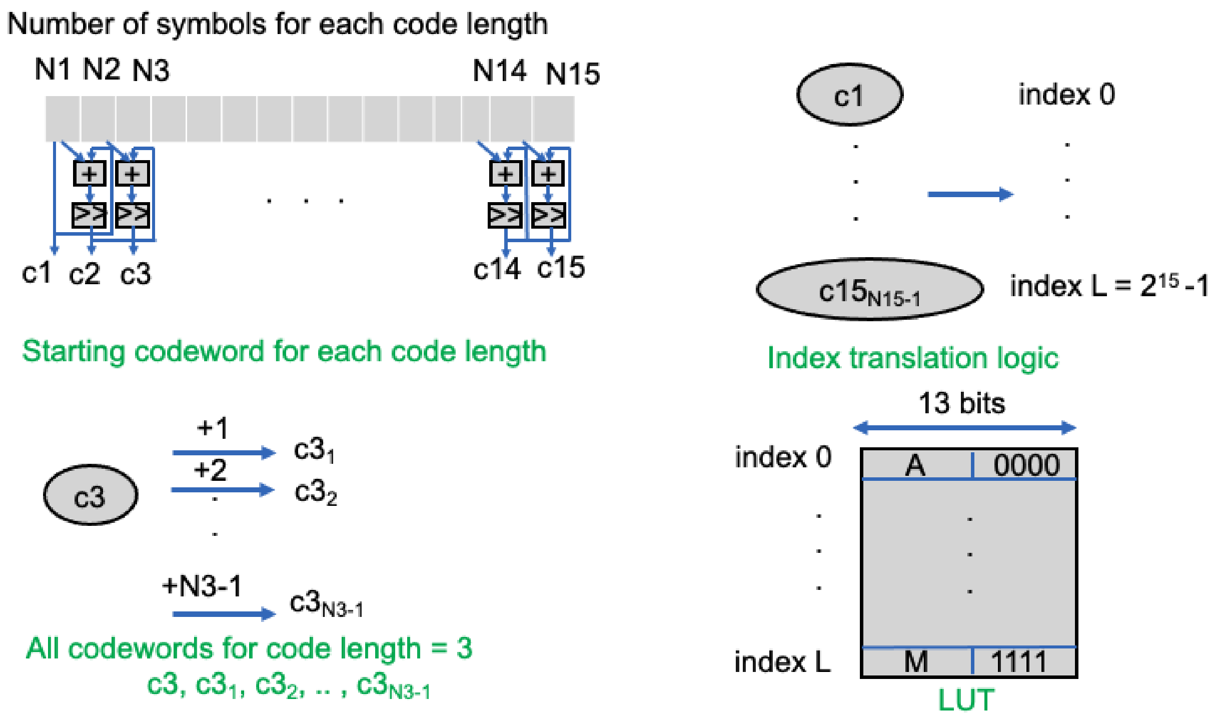


Figure 4.1: Baseline architecture for direct codeword indexing into a LUT which covers the following tasks: finding starting codeword for each code length and subsequently, all codewords, translating codewords to address indices, and finally writing the symbol-code length pairs into the LUT.

Algorithm 2 Optimized LUT Indexing Method

```
1. D ← Number of encoded symbols; // Depth of LUT
2. SB ← Symbol size in bits
3. LB ← Maximum code length (L) in bits
4. W ← SB + LB; // Width of LUT
5. Find starting codeword (ci) for each code length c1..cL [8]
6. Count[i] ← Number of symbols with code length i
7. Find address indices for starting codewords
add_is[1] ← 0
for i = 2 .. L
add_is[i] ← add_is[i-1] + Count[i-1]
end for
8. Find address indices for all codewords
for each parsed bitstream sequence B
do
len ← [1 .. L] per range of B w.r.t starting codewords
add_i ← add_is[len] + [B - ci]
Decoded symbol ← LUT[add_i]
end for
```

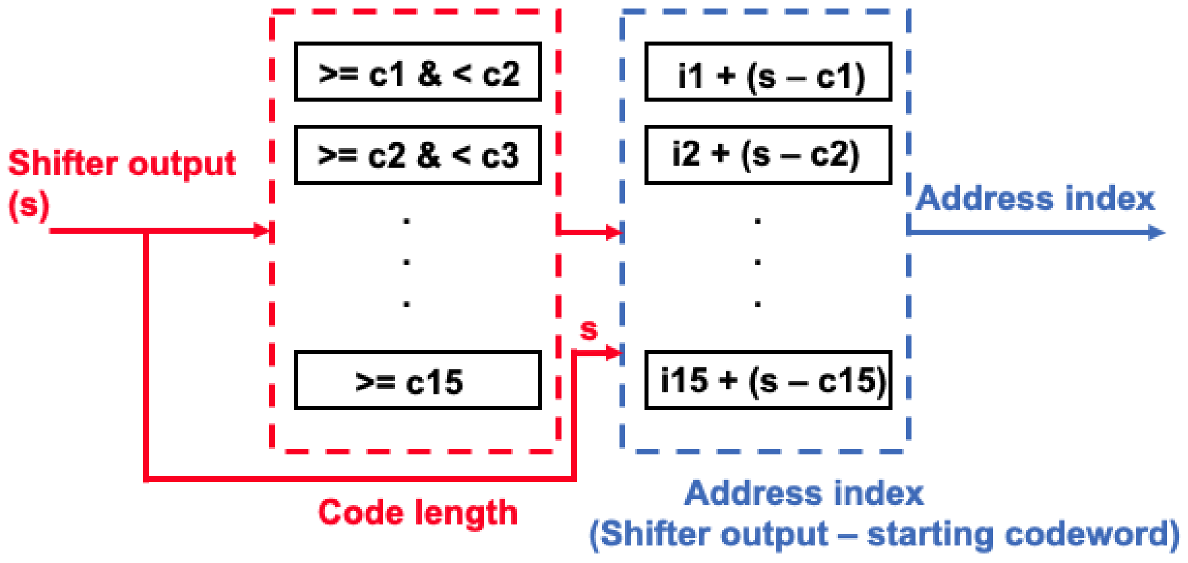


Figure 4.2: Code length computation and indexing architecture, where i1, i2, .., i15 denote the address indices for the starting codewords and c1, c2, .., c15 denote the starting codewords.

4.1.2 Optimized Architecture

The biggest drawback in the baseline architecture is that the size of the LUT adds both space and time complexity to the design as the table grows bigger for an input bitstream encoded with longer code lengths. For example, in case of DEFLATE the LUT should be designed with a depth of 32768 due to maximum code length of 15 in the dynamic Huffman decompression. We overcome this bottleneck in the optimized architecture by adopting an optimized LUT and efficient LUT indexing mechanisms. The optimized LUT indexing algorithm is shown in Algorithm 2. The optimized lookup table where the maximum depth is same as the available number of symbols, and each word is comprised of symbol (9-bits) and code length (4-bits) set of 13-bits considering a 512 symbols alphabet and a maximum code length of 15. This LUT architecture provides a scalable platform for different compression standards that supports wide range of alphabet size and code lengths.

Moreover, we propose an efficient architecture for LUT indexing in Figure 4.2 per Algorithm 2. The starting codeword computation task is conducted by the architecture shown in Figure 4.1. The comparators will decide the code length of the buffered bitstream data by comparing them with the ranges of starting codewords. For example, a codeword that is greater than equal to “c1” and less than “c2” should be of code length 1. The starting codewords take the address indices based on number of symbols for each code length. For example, a compressed bitstream with the number of symbols, “1”, “2”, and “4” for code length of “1”, “2”, and “3” respectively, are assigned as follows: starting codewords of code lengths “1”, “2”, and “3” get address indices of “0”, “1”, and “3” respectively. For rest of the symbols the address index of LUT can be calculated in the following steps: by subtracting the buffered bitstream from the starting codeword, then adding to the address index of corresponding starting codeword as the symbols of the same code length are stored in contiguous memory space of LUT, which is shown in Figure 4.3.

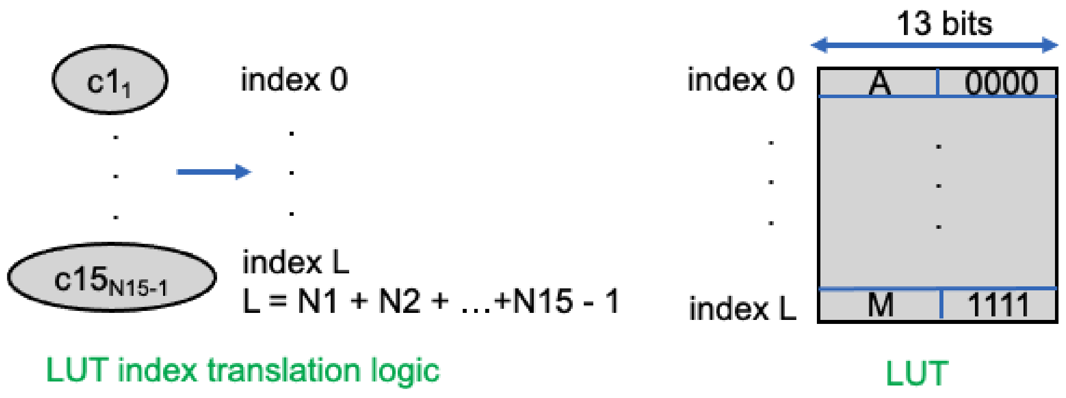


Figure 4.3: Optimized LUT architecture for decoder where depth of LUT depends on the maximum number of symbols that are present in the compressed bitstream instead of maximum code length.

4.2 Mapping of the Baseline and Optimized Architectures on a Many-Core Processor Array

This section lays out the mapping of the baseline and optimized architectures for dynamic canonical Huffman decoders on a many-core processor array.

4.2.1 Baseline Implementation

The baseline implementation is same as the optimized kernel implementation from our earlier work [59]. In this implementation, starting codeword computation task is sped up by exploiting loop unrolling and distributing the task across set of processors. Next, the processors

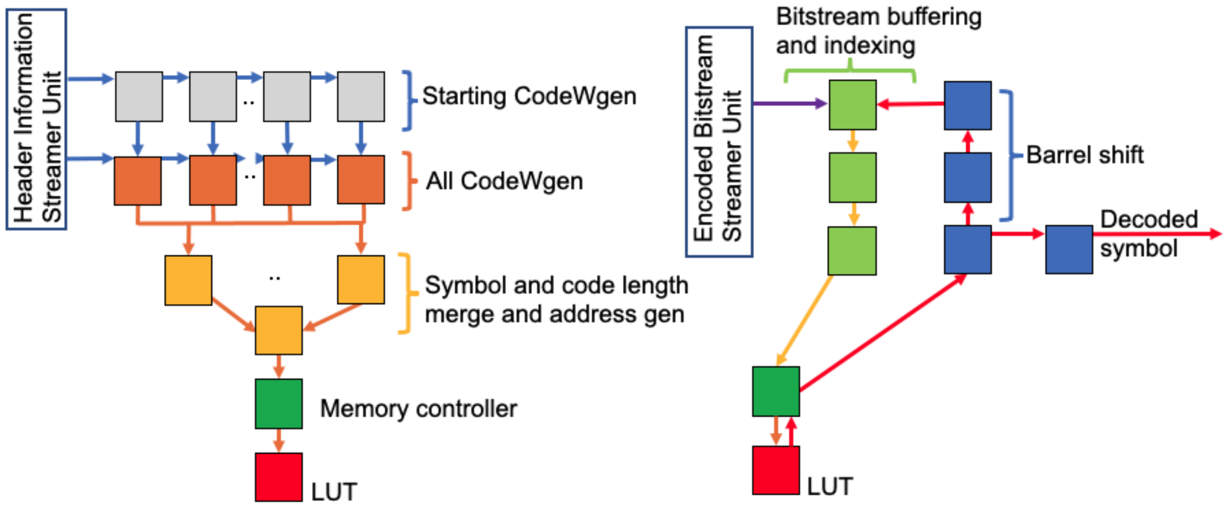

Header Information Streamer Unit
Starting CodeWgen
All CodeWgen
Symbol and code length merge and address gen
Memory controller
LUT
Encoded Bitstream Streamer Unit
Bitstream buffering and indexing
Barrel shift
Decoded symbol
LUT

Figure 4.4: Mapping of the header to LUT conversion logic and bitstream decoding: (a) this architecture involves starting codeword generation (CodeWgen) for each code length, subsequently assigns codewords to all symbols, and loads the symbol and corresponding code length data into a LUT. (b) this architecture uses the bitstream chunk as hash index, access the LUT, and read the corresponding symbol and code length. The corresponding symbol is decoded as the output and code length is used to decide the next chunk of the encoded bitstream [59].

downstream the starting codeword data to the processor in south, where each processor dedicated for a code length then maps the symbols with the corresponding codewords by canonical code assignment process [59]. Next, a set of processors combines code length and symbol set as a word to be written into the LUT.

The LUT indexing task is executed by assigning codewords as direct addresses for the memory module and then the symbol is looked up from the SRAM module which is mapped as the LUT shown in Figure 4.4. The next task is to buffer the encoded bitstream data, which is sent to the processors to the south, who are responsible to generate the read address and index the two bytes chunk to the memory module through the memory controller processor to lookup the symbol. The corresponding code length feeds back to the barrel shifter, which in turn decides the next chunk of data to be indexed. The baseline implementation is mapped onto the physical array using 52 processors and a memory module.

4.2.2 Optimized Implementation

In this section, we propose an optimized version of the decoder which utilizes an improved LUT indexing method. This improved method retains the same starting codeword computation

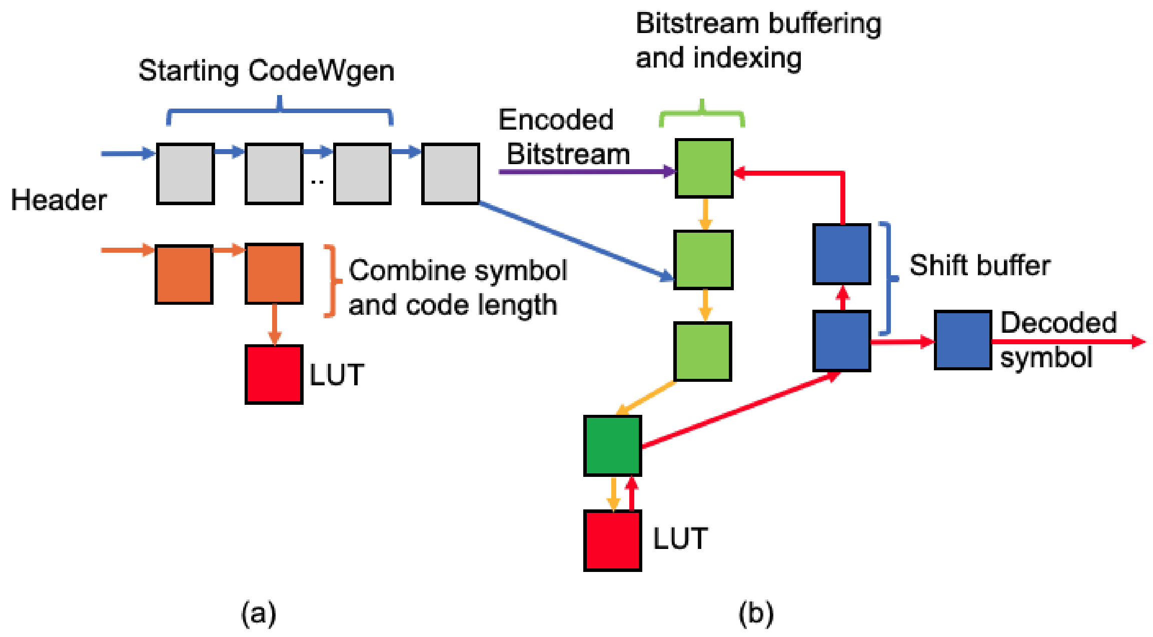


Figure 4.5: Mapping of the optimized architecture for: (a) LUT generation and (b) bitstream indexing.

task mapping as in the baseline implementation, i.e., loop unrolling the task across 14 processors to speed up the execution. To reduce the execution time for the dynamically created LUT for every input bitstream, we optimize both LUT size and data write operation into it as discussed in Section 4.2. The header data consisting of the encoded symbols that are sorted per code length are combined with the corresponding code lengths as the words to be written into the LUT. In this implementation, there is no need to find all possible codewords which helps in reducing the number of processors unlike the baseline implementation. Moreover, the LUT write address generator task is simpler than the baseline implementation as discussed in Section 4.1. The task graph of the optimized implementation is shown in Figure 4.5.

The shift buffer task comprises shifting the buffer processor data by the code length of the previously decoded symbol, which is mapped into one processor. The LUT word gets partitioned into symbols and code lengths, and the decoded symbol is sent to the output stream. In this implementation, the bitstream buffering and indexing task requires two processors to do both code length computation and index address computation tasks. In total, the optimized implementation requires 25 processors and a memory module, which reduces the required area by over **2**× compared to the baseline implementation.

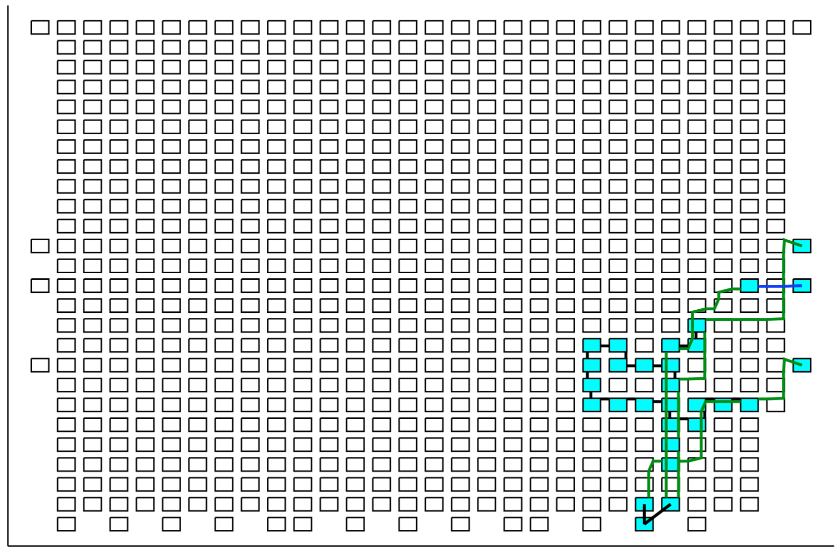


Figure 4.6: The entire optimized decoder application described in Section 4.2.2 is mapped to 25 processors, a single 64 KB memory (on the bottom of the array), and I/O ports on the right edge of the array.

4.2.3 Application Task Mapping into Many-Core Array

For the unmapped task graph shown in Figure 4.5 to execute on the processor array, it must first be mapped onto the array given its particular constraints in processors, memory, and I/O. A specifically-designed CAD tool for the many-core processor array performs a one-to-one mapping of the tasks onto physical processors. In addition, the mapping tool allocates chip interconnects to task graph communication links. Figure 4.6 shows the completed mapping of the optimized decoder application onto a many-core processor array using a total of 25 processors and a memory module.

4.3 Results and Analysis

This section discusses the benchmarks, simulation set up, results of the many-core implementations, and presents comparisons with the optimized Intel i7 and GPU software implementations.

4.3.1 Benchmarks and Computing Platforms

We evaluate the canonical Huffman decoder designs on the following computing platforms.

Table 4.1: Throughput per area and Compressed bits decoded per energy data for KiloCore implementations at supply voltages of 0.9 V and 1.1 V.

Benchmark	Design	VDD (V)	Throughput (Mbps)	Area (mm^2)	Throughput/Area (Mbps/mm^2)	Bits/Energy (Bits/µJ)
Artificial Corpus	Baseline	0.9	334.17	3.02	110.51	697.18
		1.1	520.01	3.02	171.96	538.21
	Optimized	0.9	432.27	1.54	280.88	**1436.90**
		1.1	**678.54**	**1.54**	**440.90**	1100.01
Calgary Corpus	Baseline	0.9	867.68	3.02	286.93	1602.31
		1.1	1355.38	3.02	448.21	1289.33
	Optimized	0.9	1168.07	1.54	758.98	**3448.84**
		1.1	**1855.69**	**1.54**	**1205.78**	2845.84
Canterbury Corpus	Baseline	0.9	432.25	3.02	201.21	857.00
		1.1	693.76	3.02	229.42	698.22
	Optimized	0.9	539.36	1.54	350.46	**1737.50**
		1.1	**884.54**	**1.54**	**574.75**	1442.05
Large	Baseline	0.9	74.86	3.02	297.59	137.47
		1.1	119.39	3.02	39.48	108.20
	Optimized	0.9	103.72	1.54	67.40	**306.67**
		1.1	**166.47**	**1.54**	**108.17**	241.37

- Intel i7-4850HQ containing four cores and eight threads with a base clock frequency of 2.3 GHz [21].

- Nvidia GeForce GT 750M with 384 shader cores running at a base clock of 941 MHz [22].

- KiloCore a 1000-processor array with each processor running at 1.78 GHz [17].

We consider various lossless text compression benchmarks (Artificial corpus, Calgary corpus, Canterbury corpus, and Large corpus) to validate the canonical Huffman decoder designs and compare the performance and compressed bits decoded per energy of the computing platforms. We verify the functionality of each design first before conducting the performance evaluation. The benchmarks cover a wide range of files of different sizes and symbol distribution.

4.3.2 Simulation Setup

We simulate the KiloCore designs using a cycle-accurate simulator which is modeled based on the fabricated chip data [17]. First, the simulations are conducted with the base supply voltage of 0.9 V to analyze the key parameters of throughput, area, power, and workload per unit energy. Later, we scale the supply voltage to 1.1 V to see the effect of voltage scaling on performance and energy efficiency while preserving the functionality of each design.

We report the simulation results for Intel i7 and Nvidia GT 750M GPU after speeding up the execution time by triggering the –O3 flag for both gcc and nvcc compilers respectively. We have referred to the source code for canonical Huffman decoder implementation [81] for Intel i7-4850HQ implementation [59]. The source code has been modified for bit-parallel LUT-based decoding. For power calculations for the Intel i7-4850HQ chip, TDP/2 = 23.5 W is used since the actual power dissipation is not known [59]. The die area of the Intel i7-4850HQ is reported as 260 mm^2 [21].

The massively parallel decoder [61] has been implemented on a CUDA-enabled GT 750M GPU with compute capability 3.0 and a TDP ratings of 50 W. Therefore, we consider a TDP/2 = 25 W for power calculations since the actual power dissipation is unknown. The die area of the GT 750M GPU is reported as 118 mm^2 [22].

4.3.3 Implementation Results

The resulting area, throughput, throughput per area, and energy efficiency data are reported in Table 4.1 for both baseline and optimized implementations. The optimized implementation achieves a throughput per chip area that is **2.62**× greater on average than the baseline implementation. Similarly, the optimized implementation depicts an energy efficiency improvement of **2.13**× greater and area utilization reduction of **50%** on average than the baseline implementation.

4.3.4 Comparisons of Results for Intel i7, Nvidia GT 750M, and KiloCore Array Implementations

Table 4.2 compares the performance of the canonical Huffman decoder designs on a variety of programmable hardware: general-purpose Intel i7 CPU, Nvidia GT 750M GPU, and KiloCore processor array. The i7 and GPU values are scaled to 32 nm (technology for KiloCore) to account for differences in technology generations using scaling data by the DeepScaleTool [20]. KiloCore

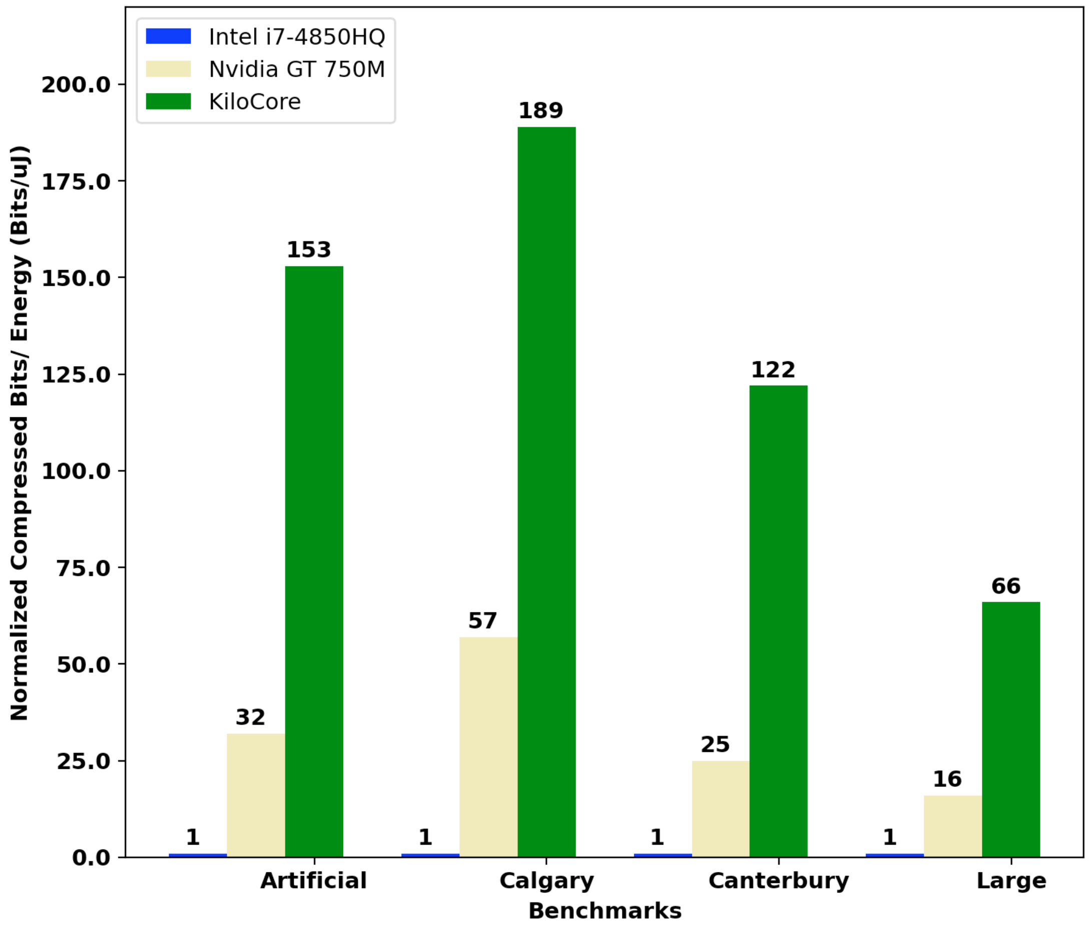


Figure 4.7: Comparison of scaled energy efficiency (Compressed Bits/µJ) for the many-core array (KiloCore) and Nvidia GeForce GT 750M GPU, all normalized to the Intel i7-4850HQ.

implementations yield a reduction in power dissipation of **77**× and **70**× when compared to Intel i7 and GPU implementations respectively.

The GPU implementation using a massively parallel decoding algorithm and exploiting its huge die area results an average throughput improvement of **14**× and **35**× when compared to optimized KiloCore and Intel i7 implementations respectively. However, in terms of both throughput per chip area and energy efficiency the proposed many-core processor array implementations have clear advantages—the optimized KiloCore design give an throughput per area improvement of **7**× and **891**× over GT 750M GPU and Intel i7 respectively. Figure 8.2 compares the gain in energy efficiency for KiloCore and GT 750M GPU when normalized to Intel i7. Similarly, Figure 4.8 shows the throughput per area improvement for KiloCore and GT 750M GPU when normalized to Intel i7.

To summarize different canonical Huffman decoder algorithms: the decoder eliminates bit-serial decoding approach that uses tree traversal method. A big LUT-based approach [59]

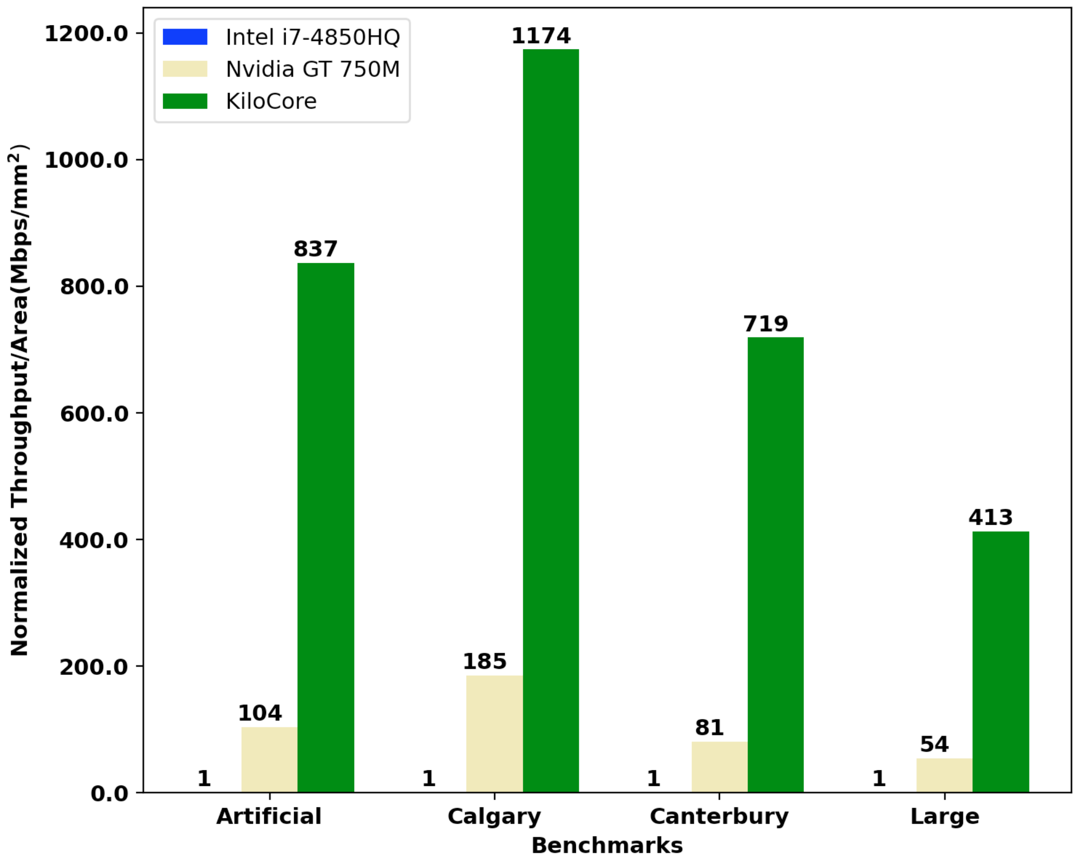


Figure 4.8: Comparison of scaled throughput per area (Mbps/mm^2) for the many-core array (KiloCore) and Nvidia GeForce GT 750M GPU, normalized to Intel I7-4850HQ implementations.

speeds up the decoding process at the cost of big area overhead per maximum code length. The optimized LUT-based decoding approach that is presented in this work cuts down the significant delay in code table construction and helps in achieving better performance and energy efficiency. Self-synchronization based decoding enables parallel decoding; however, the performance gain is highly statistical.

Table 4.2: Throughput per area and Compressed bits decoded per energy data for Intel i7, Nvidia GPU, and Many-Core (KiloCore) implementations. The i7 and GPU values are scaled to 32 nm (technology for KiloCore) to account for differences in technology generations using scaling data by the DeepScaleTool [20].

Benchmark	Platform	Throughput	Area	Throughput/Area	Bits/Energy
		(Mbps)	(mm^2)	($Mbps/mm^2$)	(Bits/µJ)
Artificial Corpus	i7-4850HQ [81]	288.6	548.52	0.53	9.33
	GT 750M [61]	**8475.45**	154.86	54.73	301.72
	Many-Core	678.54	**1.54**	**440.90**	**1436.90**
Calgary Corpus	i7-4850HQ [81]	563.36	548.52	1.03	18.22
	GT 750M [61]	**29,478.94**	154.86	190.36	1049.45
	Many-Core	1855.69	**1.54**	**1205.78**	**3448.84**
Canterbury Corpus	i7-4850HQ [81]	437.99	548.52	0.80	14.17
	GT 750M [61]	**10,053.12**	154.86	64.92	357.89
	Many-Core	884.54	**1.54**	**574.75**	**1737.50**
Large	i7-4850HQ [81]	143.66	548.52	0.26	4.65
	GT 750M [61]	**2195.51**	154.86	14.18	78.16
	Many-Core	166.47	**1.54**	**108.17**	**306.67**

Chapter 5

Design of Canonical Huffman Encoder Hardware Accelerators

This chapter presents hardware architectures for both static and dynamic canonical Huffman encoders. The proposed architectures are implemented on MAX10 FPGA and 45 nm ASIC, and the synthesis results on performance and energy-efficiency are discussed.

5.1 Static Canonical Huffman Encoder Design

The static canonical Huffman encoder design is based on indexing a static code table. For any compression standard that uses canonical Huffman encoding as a part of the compressor stage, fixed or static code tables are provided. Conventionally, next task is to serially index a symbol from the uncompressed data set and hash out the encoded codeword. In terms of hardware implementation, static code table is primarily designed as a ROM and direct address indexing by symbol hashes out the encoded codeword, which is shown in Figure 5.1. If the code table is not provided, for a specific data stream a fixed code table can be created based on given symbol-frequency distribution by adopting the rest of the tasks in the case of the dynamic canonical Huffman encoder. We will discuss each of these tasks in details in the following section and performance for a static encoder based on LZ77 compressed data using static code tables for the Deflate compression is discussed in Chapter 8.

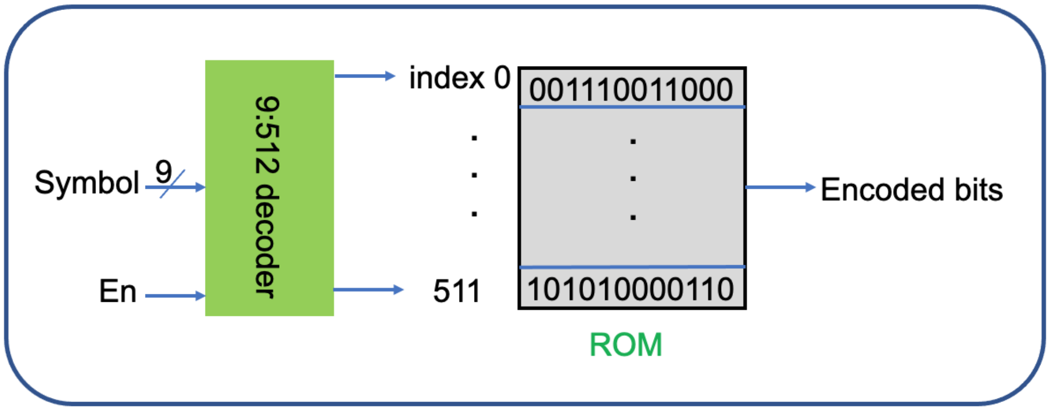


Figure 5.1: Realization of the static canonical Huffman encoder using a ROM.

5.2 Dynamic Canonical Huffman Encoder Design

The dynamic canonical Huffman encoder hardware builds a dynamic code table, which starts with accumulating symbol-frequencies, builds a Huffman tree, assigns code lengths, and stores symbol-codeword pairs in a code table. The proposed optimized canonical Huffman encoder architecture that is presented in Chapter 3, is used to implement the hardware designs.

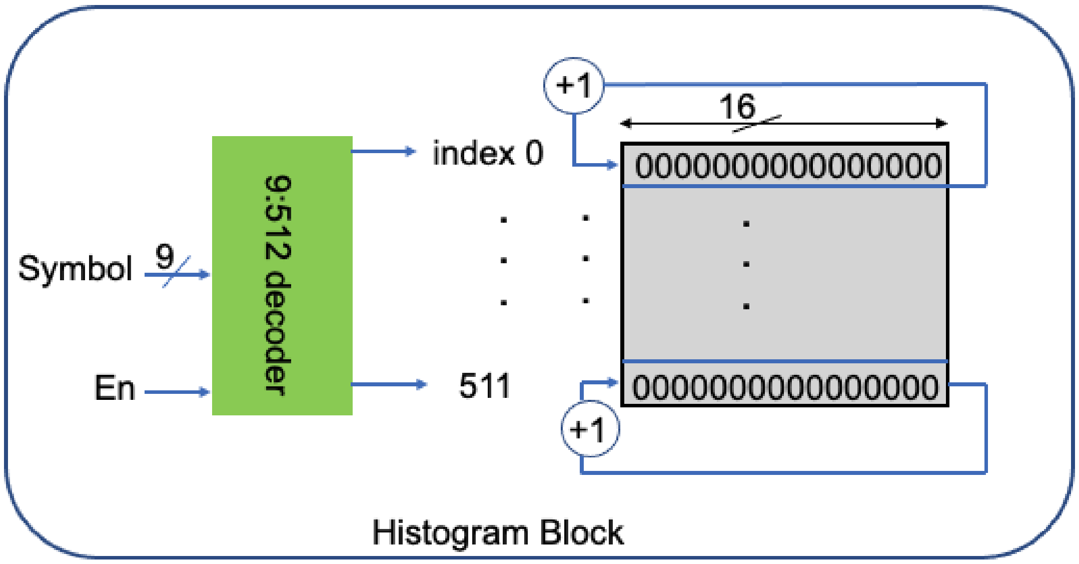


Figure 5.2: Histogram computation hardware executing frequency accumulation for each symbol using 512-word X 16-bits accumulators.

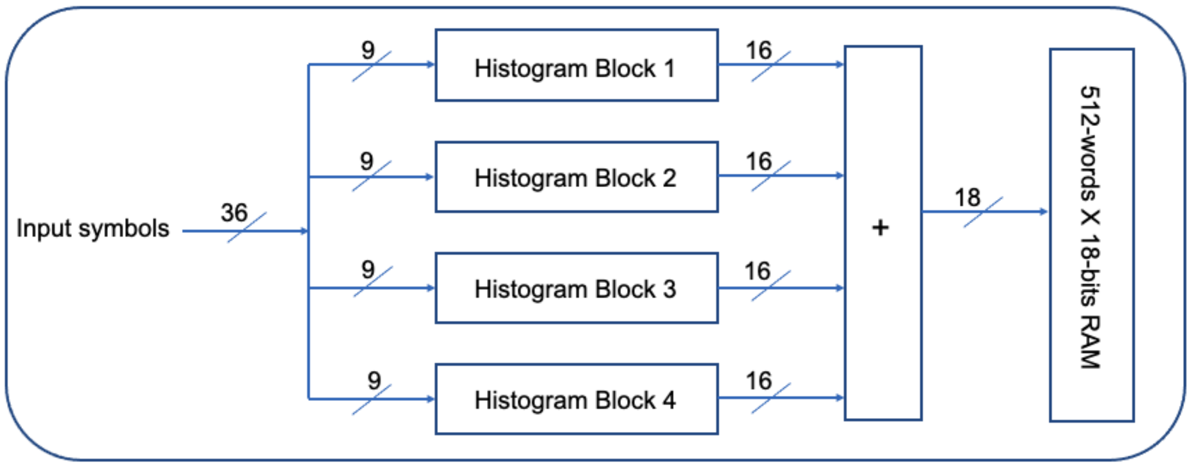


Figure 5.3: Parallel histogram computation unit using 4 histogram blocks and associated logic.

5.2.1 Histogram Block

The histogram computation block parses a symbol serially from the input bitstream and sends it to a decoder for address indexing. The 5:912 decoder enables one of the accumulators based on decoder output and its contents gets incremented by one. Figure 5.2 shows the hardware for serial histogram computation.

To speed-up serial operation, four histogram blocks are executed in parallel. Every clock cycle, four symbols are read from the input bitstream and pass on to four individual histogram blocks. Next, output from each histogram block is merged and written to a RAM memory of 512-words X 18-bits. Figure 5.3 shows the hardware for parallel histogram computation that uses 4 histogram blocks, adders, and a 512-word X 18-bits RAM to store final symbol-frequency histogram for all symbols.

5.2.2 Sorting Block

The sorting module sorts the symbol-frequencies in non-decreasing order, that are stored in histogram memory.

Instead of sorting the whole 512-symbols list in one pass, we have employed two-staged sorting - intra-block and inter-block sorting, which is similar to the state-of-the-art design [51]. In this method, during intra-block sorting, corresponding addresses are selected to access individual

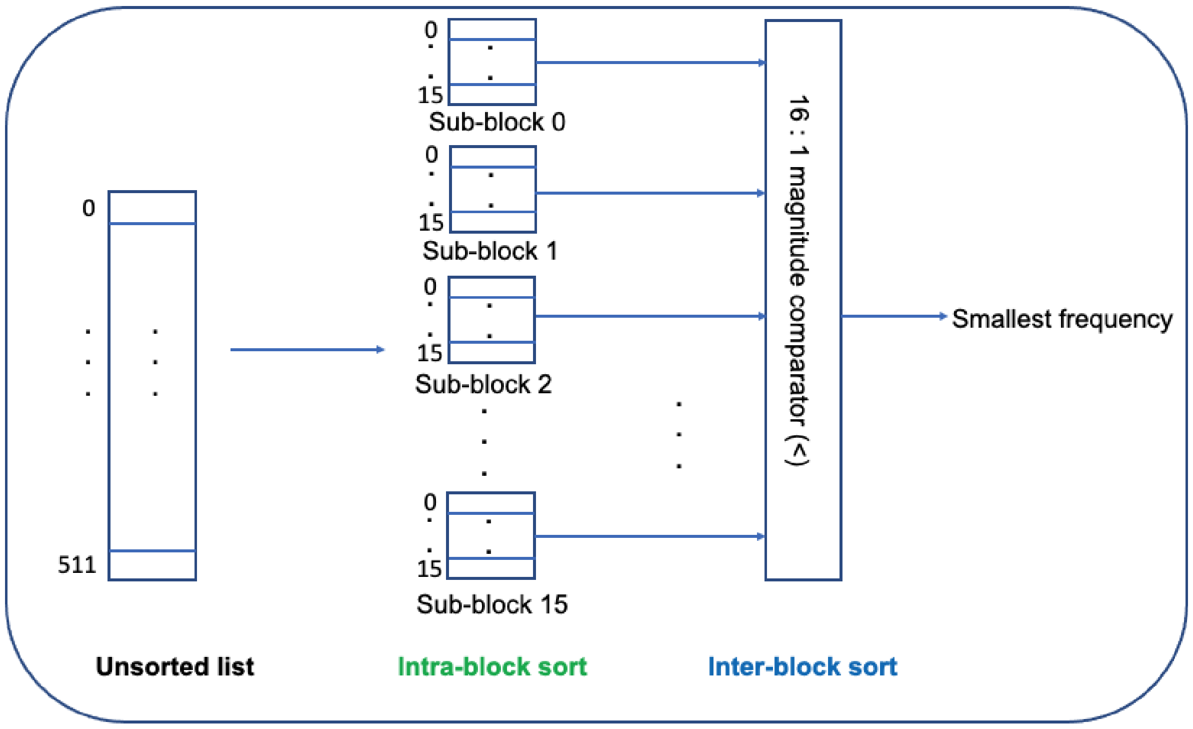


Figure 5.4: Sorting of the symbol-frequency list using intra-block sorting followed by inter-block sorting to improve speed and area utilization.

frequencies and swap them based on comparator output, i.e. smaller frequency goes to lower address indices. This process continues until smallest frequency in each sub-block are computed. Figure 5.4 demonstrates the sorting architecture.

In inter-block sorting, a 16:1 magnitude comparator is used to select the smallest among all that results from intra-block sorting. To optimize latency, a multi-stage magnitude comparator is designed. Figure 5.5 shows both single-stage and multi-stage comparator for inter-block sorting. Once the first two smallest frequencies are generated, a control signal triggers the Huffman tree building process.

5.2.3 Huffman Tree Building and Code length Computation Block

The proposed concurrent sorting, Huffman tree creation, and code length computation architecture as shown in Figure 3.14 of Chapter 3, is used to implement the three tasks in hardware.

A counter is used to count number of symbols per code length along with the a bitmap to store growing pattern of sub-tree. The end of Huffman tree creation process triggers a control signal to assign code length to all symbols in parallel, which is shown in Figure 3.14.

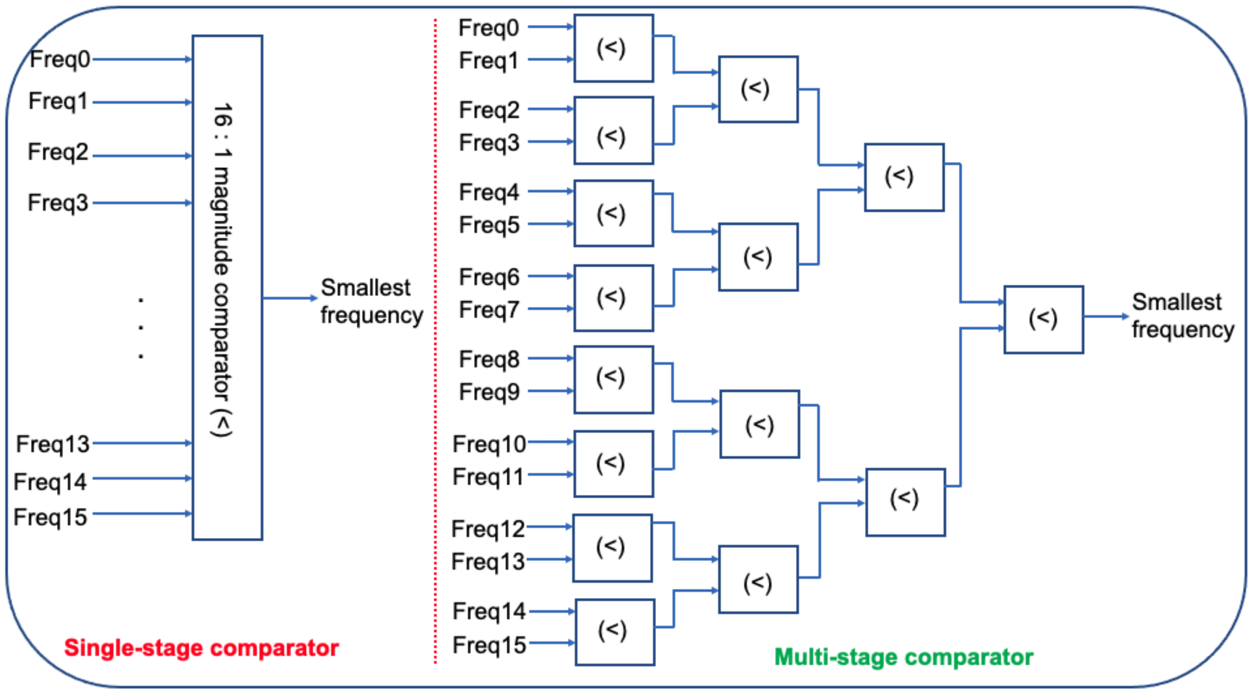


Figure 5.5: Single-stage vs. multi-stage magnitude comparator. Multi-stage comparator is used to minimize latency.

Another counter is used to count number of symbols that exceeds maximum code length. In such case, instead of re-assigning frequency to each symbol iteratively and executing the whole encoder pipeline again, an efficient method is adopted from the work [51]. This design uses an alternate approach where a symbol with shorter code length is replaced with code length of 15 and with a code length of 15, a large space is created to accommodate non-compliant symbols.

5.2.4 Canonical Code Assignment Block

The canonical code assignment task first creates starting codeword based on number of symbols for each code length. Based on number of symbols per code length, rest of the codewords are generated in parallel as they are incremented by one subsequently for the same code length group.

Figure 5.6 demonstrates the starting codeword generation process for each code length and all codewords generation logic. The codewords are stored in a RAM of 512-word and width of 19-bits (15-bits for codeword and 4-bits for code length).

To encode each symbol, a symbol is used to decode and decoder output is used as address index for the RAM. The read content is subsequently passed through a shifter based on the code

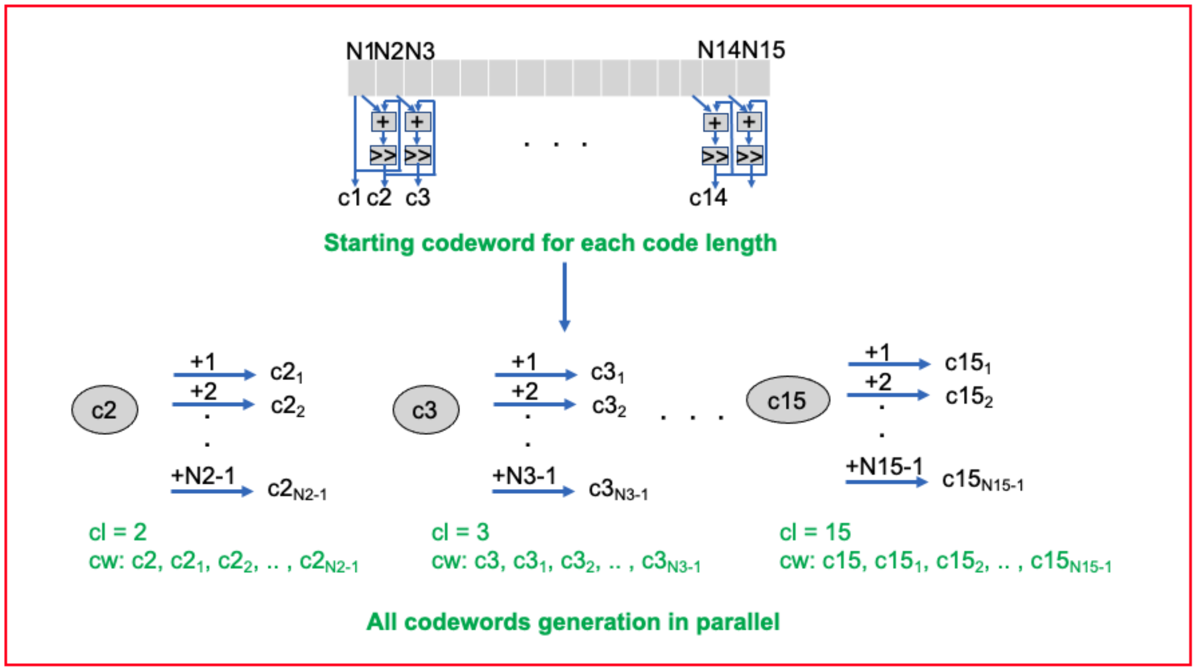

N1N2N3
N14N15
c1 c2 c3
c14
Starting codeword for each code length
c2
c3
c15
+1
+2
+N2-1
+N3-1
+N15-1
$c2_1$
$c2_2$
$c2_{N2-1}$
$c3_1$
$c3_2$
$c3_{N3-1}$
$c15_1$
$c15_2$
$c15_{N15-1}$
cl = 2
cw: c2, $c2_1$, $c2_2$, .. , $c2_{N2-1}$
cl = 3
cw: c3, $c3_1$, $c3_2$, .. , $c3_{N3-1}$
cl = 15
cw: c15, $c15_1$, $c15_2$, .. , $c15_{N15-1}$
All codewords generation in parallel

Figure 5.6: Starting codeword generation for each code length and rest all codewords generation in parallel.

length information and OR logic to generate the final encoded bitstream.

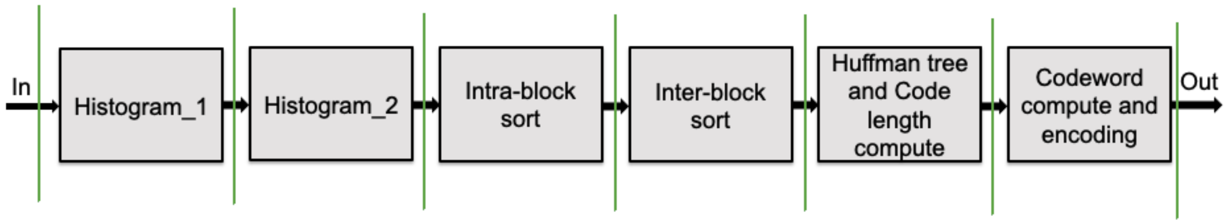

In
Histogram_1
Histogram_2
Intra-block sort
Inter-block sort
Huffman tree and Code length compute
Codeword compute and encoding
Out

Figure 5.7: Pipelined canonical Huffman encoder.

5.2.5 Pipelined Block Diagram of Canonical Huffman Encoder

A pipelined block diagram of the canonical Huffman encoder design is shown in Figure 5.7 to speed-up the datapath. The histogram computation task is divided into two sub-tasks, where in the first task executes a single histogram and second sub-task consists of a ripple carry adder and memory write operation. Similarly, intra-block sorting and inter-block sorting are partitioned.

Table 5.1: Throughput per area and Compressed bits decoded per energy data for MAX10 FPGA Dynamic Canonical Huffman Encoder Implementation.

Benchmark	Throughput (Mbps)	Compressed Bits/Energy (Bits/ µJ)
Artificial Corpus	494.7	313.5
Calgary Corpus	521.7	358.5
Canterbury Corpus	484.2	238.6
Large	344.5	129.7

5.3 Implementation Results

The proposed pipelined with task-level parallelism dynamic canonical Huffman encoder design is evaluated on both MAX 10 FPGA and 45-nm ASIC and performance and energy-efficiency results are discussed.

5.3.1 Results of Canonical Huffman Encoders on MAX 10 FPGA

We evaluate the proposed dynamic canonical Huffman encoder architectures on the MAX 10 FPGA. The timing and power(total thermal power dissipation) results are obtained from the Quartus Timing Analyzer and PowerPlay Power Analyzer respectively.

The MAX 10 FPGA implementation uses five out of 1638 available M9K memory block and 16.3% of the available logic elements for the dynamic canonical Huffman encoder, while a recent work [65] on Xilinx Virtex Ultrascale+ series FPGA for the DEFLATE compliant static Huffman encoder has the lookup tables and BRAM tiles utilization of 1.75% and 2.85% respectively.

The scaled performance and energy-efficiency results are discussed in Table 5.1.

5.3.2 Results of Canonical Huffman Encoders on 45 nm ASIC

We evaluate the proposed static and dynamic canonical Huffman decoder architectures on the 45 nm ASIC at 1.25 V. The designs are functionally verified for all benchmarks and the throughput and energy efficiency results are posted in Table 5.2. Each design is synthesized into a gate-level netlist using a NanGate 45 nm Open Cell Library by Synopsys Design Compiler.

The ASIC synthesis results show that the pipelined canonical Huffman encoder occupies

Table 5.2: Throughput per area and Compressed bits decoded per energy data for ASIC Dynamic Canonical Huffman Encoder Implementation.

Benchmark	Throughput (Mbps)	Compressed Bits/Energy (Bits/nJ)
Artificial Corpus	1156.2	1.9
Calgary Corpus	1374.4	3.1
Canterbury Corpus	1237.4	1.7
Large	1059.9	1.2

total area of 215,501.8 μm^2.

Table 5.3: Summary of the FPGA and ASIC implementations of the canonical Huffman encoder (both scaled to 32 nm).

Benchmark	Platform	Technology (nm)	Throughput (Mbps)	Voltage (V)	Power (W)	Area (µm^2)	Compressed Bits/Energy (Bits/nJ)
Artificial Corpus	ASIC	45	**1344.5**	1.25	**0.5**	**107750.9**	**2.7**
	FPGA	55	575.2	1.25	1.3	NA	0.4
Calgary Corpus	ASIC	45	**1598.2**	1.25	**0.4**	**107750.9**	**4.5**
	FPGA	55	606.5	1.25	1.1	NA	0.5
Canterbury Corpus	ASIC	45	**1237.4**	1.25	**0.5**	**107750.9**	**2.4**
	FPGA	55	563.0	1.25	1.7	NA	0.3
Large	ASIC	45	**1059.9**	1.25	**0.6**	**107750.9**	**1.7**
	FPGA	55	400.6	1.25	2.1	NA	0.2

Table 5.3 summarizes the performance and energy-efficiency of the canonical Huffman decoder implementations on both ASIC and FPGA. On an average, scaled results on ASIC shows 2.44× improvement on throughput, 3.5× improvement on total power dissipation, and 7.6× energy efficiency improvement over FPGA implementations.

Chapter 6

Design of Canonical Huffman Decoder Hardware Accelerators

This chapter presents hardware architectures for both static and dynamic canonical Huffman decoders using optimized LUT-based indexing methods. The proposed architectures are implemented on MAX10 FPGA and 45 nm ASIC, and the synthesis results on performance and energy-efficiency are discussed.

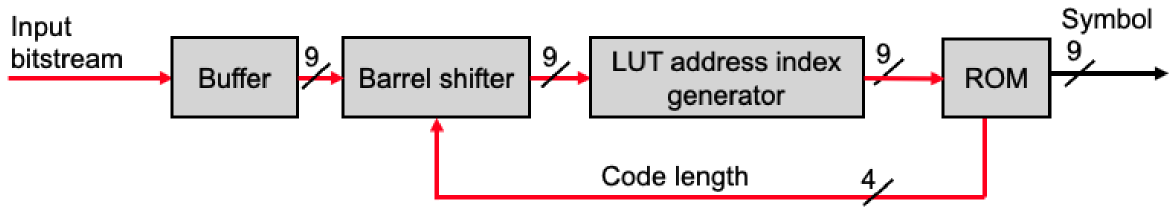


Figure 6.1: Static canonical Huffman decoder architecture. The architecture uses an optimized ROM-based fixed-LUT that enables faster hashing than the conventional architecture.

6.1 Baseline LUT-based Indexing Architecture

Generally, canonical Huffman decoding is performed by indexing encoded bits to hash out a symbol from a LUT. The LUT can be of single or multi-level based on area and performance trade-offs as discussed in Chapter 4.

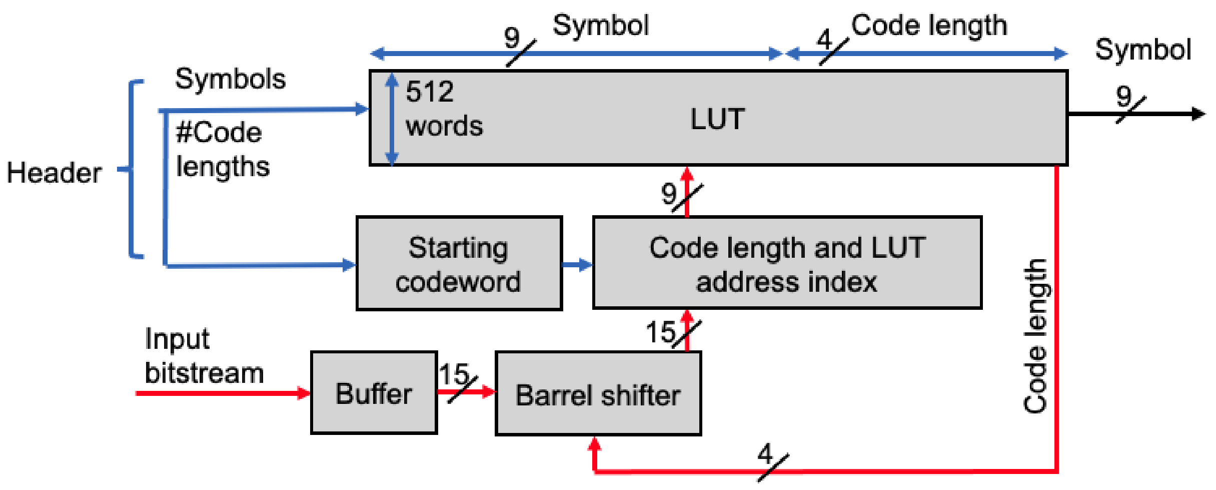


Figure 6.2: Dynamic canonical Huffman decoder architecture. The architecture first creates an optimized LUT by executing the following tasks: finding starting codeword for each code length and subsequently, all codewords, translating codewords to address indices, and finally writing the symbol-code length pairs into the LUT. Next, LUT-based indexing occurs to decode a symbol at every cycle.

The canonical Huffman decoder architecture primarily comprises of the two tasks: creating a code table and indexing the code table to lookup a symbol. For the static canonical Huffman decoder, the code table is pre-defined and the bitstream indexing task follows the same process as of the dynamic canonical Huffman decoder. In case of dynamic decoder the code table needs to be dynamically generated per bitstream header, which slows down the overall execution than a static decoder.

6.1.1 Static Canonical Huffman Decoder

At first the static code table shown in Table 2.1 is implemented using a ROM of depth 512. The static canonical Huffman decoder reads the encoded bitstream into a buffer which acts as the accumulator. The barrel shifter shifts the buffer data based on the code length of the last decoded symbol. Next, the output from the barrel shifter gets translated to the index address of ROM and the corresponding symbol is looked up. Figure 6.1 shows the block diagram representation of the static canonical Huffman decoder design.

The address indexing logic and configuration of the ROM for the LUT are same as of the designs discussed in Chapter 4.

6.1.2 Dynamic Canonical Huffman Decoder

In dynamic canonical Huffman decoder, the first task is to read the symbols that are sorted per code lengths from the header. These symbols are then written into a RAM memory along with the corresponding code lengths, which makes the LUT of size 512 * 13-bits. The LUT index address logic is implemented using a 9-bit up counter.

The configuration of the header which consists of the symbols that are sorted per code length makes the LUT write process faster than the header format in which unsorted symbols are present. The bitstream indexing process is little different than the static decoding process. First, the shifted buffer output is compared to find the code length following the architecture shown in Figure 4.2 in Chapter 4 and then, the code length is used to compute the LUT index based on an offset from the starting codeword. In this case, single LUT is indexed and there by, it makes the indexing datapath faster than the multi-lookup table approach.

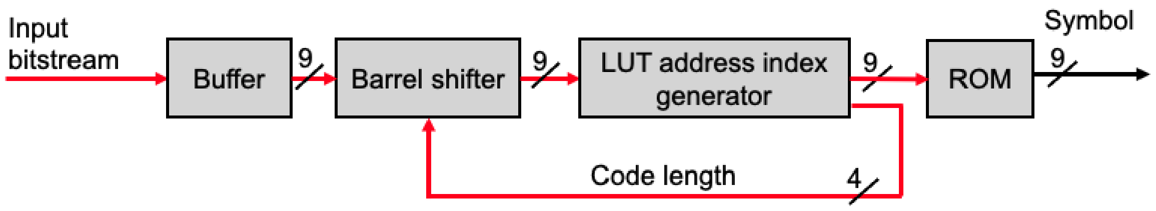


Figure 6.3: Optimized static canonical Huffman decoder architecture. The architecture proposes an optimized LUT that doesn't need to store code length information and exploits datapath resource-sharing.

6.2 Memory-efficient Canonical Huffman Decoder Architectures

The proposed baseline architecture needs to store both symbol and corresponding code length in the LUT memory. However, code length required to barrel shift the next chunk of encoded bits can also be found out from the code length and LUT address index logic. Therefore, we propose memory-efficient architectures that needs to store only symbols in LUT as the corresponding code length can be found out from the address index logic. A LUT of size 512 * 9-bits is used in this architecture.

6.2.1 Static Canonical Huffman Decoder

The static canonical Huffman decoder with a memory-efficient LUT is shown in Figure 6.3. The address index logic that is shown in Chapter 4 finds the range in which the current chunk of encoded bits fall in and based on that range it tries to find either the starting codeword or offset from the starting codeword. Therefore, based on the starting codeword or offset, code length can be found out as all symbols that are mapped to either the starting codeword or offset from the starting codeword share the same code length.

The proposed memory-efficient static canonical Huffman decoder reduces the memory size by 44% than the baseline architecture.

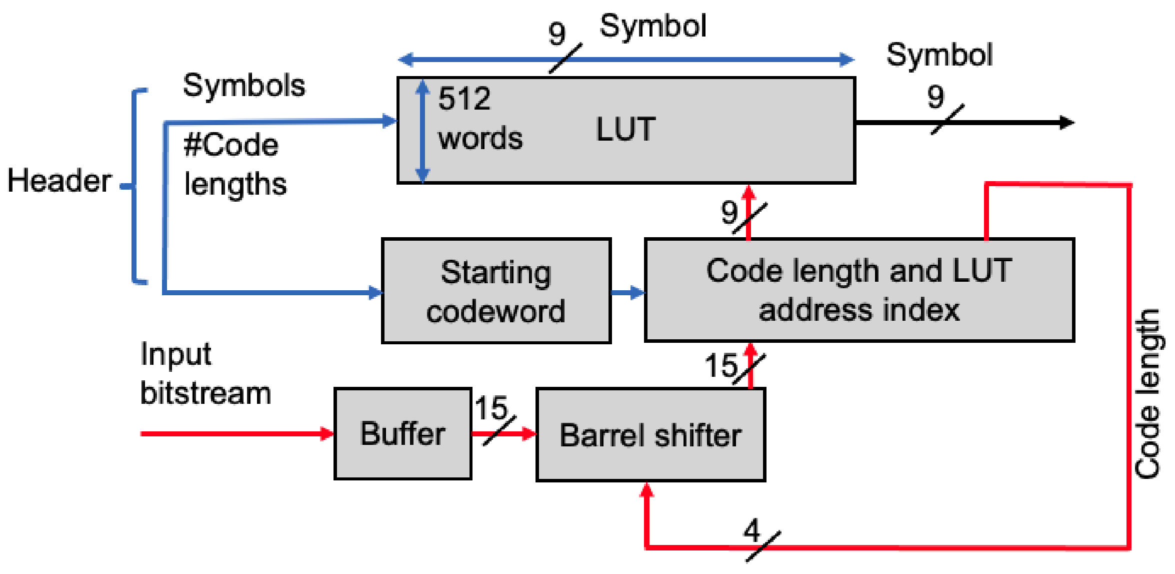


Figure 6.4: Optimized dynamic canonical Huffman decoder architecture. The architecture proposes an optimized LUT that doesn't need to store code length information and exploits datapath resource-sharing.

6.2.2 Dynamic Canonical Huffman Decoder

The dynamic canonical Huffman decoder with a memory-efficient LUT is shown in Figure 6.3. The code table creation stage from the header is same as of the baseline architecture, except configuration of the LUT now doesn't need to store code length. Also, the address index logic yields the code length information in this architecture as discussed in the previous sub-section. Therefore, code length information doesn't need to get processed for barrel shifter logic until the LUT indexing happens.

Similar to the static decoder, memory-efficient version of the dynamic decoder reduces the memory size requirement by 44% than the baseline architecture.

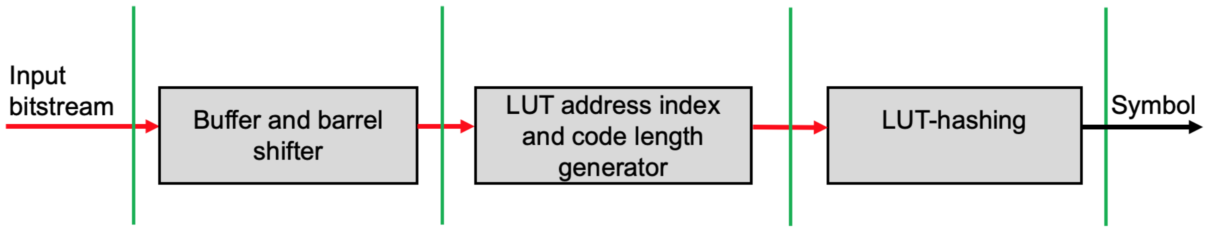

Input bitstream
Buffer and barrel shifter
LUT address index and code length generator
LUT-hashing
Symbol

Figure 6.5: Pipelined canonical Huffman decoder architecture.

6.3 Pipelined and Memory-Efficient Canonical Huffman Decoder Architectures

To speed-up the critical path and further improve throughput, a pipelined architecture is proposed for the memory-efficient canonical Huffman decoder design. Figure 6.5 shows the proposed pipelined architecture for the canonical Huffman decoder design. The pipelined architecture has been implemented for both static and dynamic decoders.

6.4 Results of Canonical Huffman Decoders on MAX 10 FPGA

We evaluate the proposed static and dynamic canonical Huffman decoder architectures on the MAX 10 FPGA. The timing and power(total thermal power dissipation) results are obtained from the Quartus Timing Analyzer and PowerPlay Power Analyzer respectively.

Table 6.3 shows the performance and energy efficiency of the static and dynamic canonical Huffman decoders for various benchmarks. The static canonical Huffman decoder implementation results in an performance improvement of **2.1**× on an average over the dynamic decoder. Similarly, the static canonical Huffman decoder yields an energy efficiency improvement of **3.68**× on an average than the dynamic decoder due to the fixed LUT architecture. However, the speed-up in case of static canonical Huffman decoder comes at the cost of compression ratio.

Table 6.1: Throughput per area and Compressed bits decoded per energy data for MAX 10 FPGA Static and Dynamic Canonical Huffman Decoder Implementations based on Pipelined and Memory-Efficient Architectures.

Benchmark	Decoder	Throughput (Gbps)	Compressed Bits/Energy (Bits/nJ)
Artificial Corpus	Static	**2.0**	**2.5**
	Dynamic	0.7	0.6
Calgary Corpus	Static	**4.1**	**5.0**
	Dynamic	1.9	1.7
Canterbury Corpus	Static	**2.3**	**3.2**
	Dynamic	0.9	0.7
Large	Static	**0.4**	**0.5**
	Dynamic	0.2	0.1

The MAX 10 FPGA implementation uses one out of 1638 available M9K memory block and 2.7% of the available logic elements for the dynamic canonical Huffman decoder, while a recent work [65] on Xilinx Virtex Ultrascale+ series FPGA for the DEFLATE compliant Huffman decoder has the lookup tables and BRAM tiles utilization of 3.04% and 0.28% respectively. Moreover, the proposed FPGA implementation results in an scaled throughput improvement of **59**% when compared to the DEFLATE based decompressor [65].

6.5 Results of Canonical Huffman Decoders on 45 nm ASIC

We evaluate the proposed static and dynamic canonical Huffman decoder architectures on the 45 nm ASIC at 1.25 V. The designs are functionally verified for all benchmarks and the throughput and energy efficiency results are posted in Table 6.3. Each design is synthesized into a gate-level netlist using a NanGate 45 nm Open Cell Library by Synopsys Design Compiler.

The ASIC synthesis results show that the pipelined and memory-efficient static decoder occupies total area of 24,308.7 μm^2, while the dynamic version of the design occupies a total area of 37,274.4 μm^2. On an average, scaled results on ASIC shows 5.1× improvement on throughput and 13.4× energy efficiency improvement over FPGA implementations.

Table 6.3 summarizes ASIC and FPGA implementation results for various benchmarks

Table 6.2: Throughput per area and Compressed bits decoded per energy data for ASIC Static and Dynamic Canonical Huffman Decoder Implementations based on Pipelined and Memory-Efficient Architectures.

Benchmark	Decoder	Throughput (Gbps)	Compressed Bits/Energy (Bits/nJ)
Artificial Corpus	Static	**13.0**	**41.3**
	Dynamic	4.4	9.2
Calgary Corpus	Static	**22.1**	**57.7**
	Dynamic	9.9	27.7
Canterbury Corpus	Static	**10.4**	**29.9**
	Dynamic	5.2	13.1
Large	Static	**2.1**	**7.0**
	Dynamic	0.9	1.6

executing both static and dynamic canonical Huffman decoder.

Table 6.3: Summary of the FPGA and ASIC implementations of the canonical Huffman decoder (both scaled to 32 nm).

Benchmark	Decoder	Platform	Technology	Throughput	Voltage	Power	Area	Compressed Bits/Energy
			(nm)	(Gbps)	(V)	(W)	(μm^2)	(Bits/nJ)
Artificial Corpus	Static	ASIC	45	**15.1**	1.25	**0.25**	**12154.3**	**59.9**
		FPGA	55	2.5	1.25	0.63	NA	3.9
	Dynamic	ASIC	45	**5.1**	1.25	**0.36**	**18627.2**	**13.4**
		FPGA	55	0.9	1.25	0.98	NA	0.9
Calgary Corpus	Static	ASIC	45	**26.6**	1.25	**0.32**	**12154.3**	**83.7**
		FPGA	55	5.1	1.25	0.7	NA	7.7
	Dynamic	ASIC	45	**11.9**	1.25	**0.34**	**18627.2**	**35.3**
		FPGA	55	2.5	1.25	0.92	NA	2.7
Canterbury Corpus	Static	ASIC	45	**12.5**	1.25	**0.29**	**12154.3**	**43.4**
		FPGA	55	2.9	1.25	0.6	NA	4.9
	Dynamic	ASIC	45	**6.3**	1.25	**0.33**	**18627.2**	**18.7**
		FPGA	55	1.2	1.25	1.2	NA	1.0
Large	Static	ASIC	45	**2.5**	1.25	**0.24**	**12154.3**	**10.2**
		FPGA	55	0.4	1.25	0.6	NA	0.8
	Dynamic	ASIC	45	**1.1**	1.25	**0.49**	**18627.2**	**2.3**
		FPGA	55	0.2	1.25	1.54	NA	0.1

Chapter 7

Design of Gzip Hardware Accelerators

This chapter presents hardware architectures for Gzip compression that uses both static and dynamic canonical Huffman encoders. The proposed architectures are implemented on 45 nm ASIC and the synthesis results on performance and energy-efficiency are discussed. We also discuss the effect of static or dynamic canonical Huffman encoding along with hash bank architecture on compression ratio.

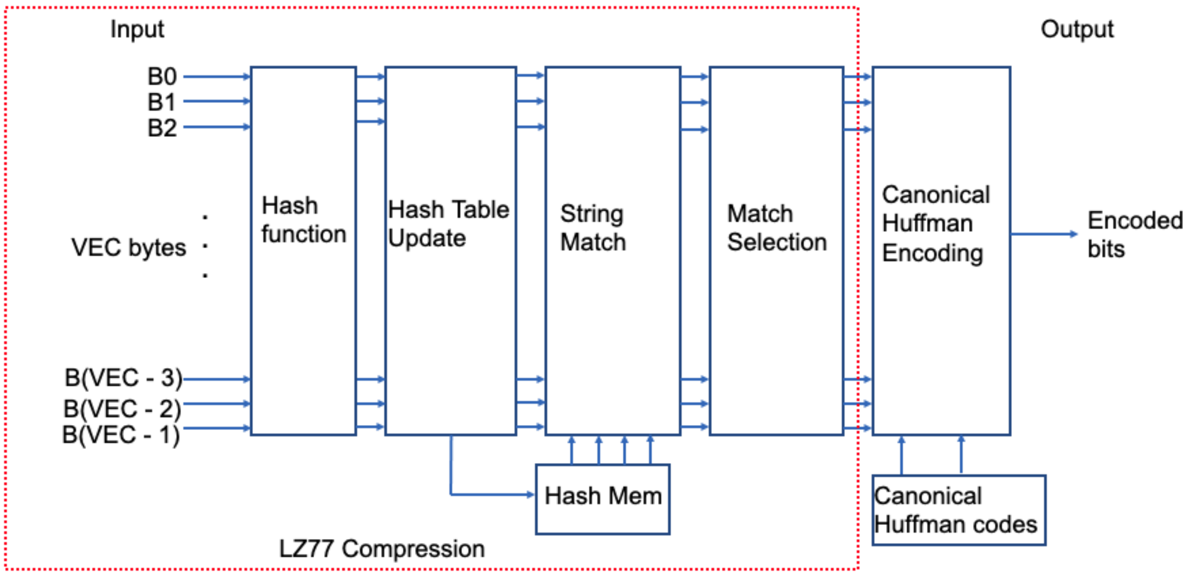


Figure 7.1: Gzip pipelined block diagram employing LZ77 and static canonical Huffman compression [83].

7.1 Overview of Gzip Hardware Design

Gzip compression is a two-stage process, where an uncompressed data stream first gets compressed by LZ77 encoding that generates a length-distance pair. Next, a bit-level compression occurs by a static or dynamic canonical Huffman encoder.

Figure 7.1 shows the pipelined block diagram of the Gzip compression demonstrating LZ77 pipeline and a canonical Huffman code module directly generating static codes. In the case of the dynamic canonical Huffman encoder for Gzip, codes need to be generated based on output from the LZ77 stage. The architectures and hardware implementations for canonical Huffman encoders are discussed in the previous chapters. We will discuss LZ77 compression in detail in the following section.

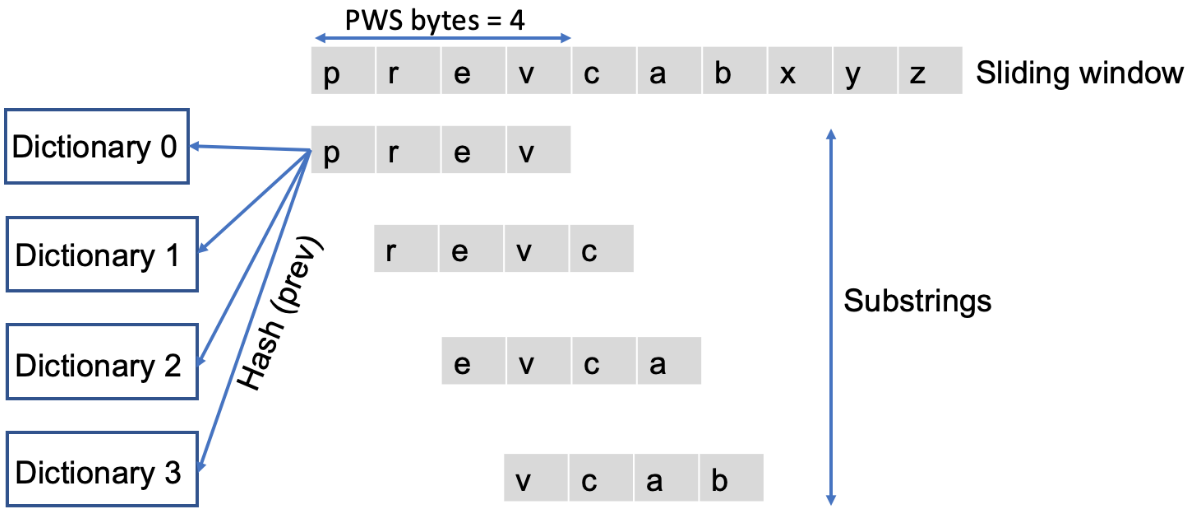


Figure 7.2: Sliding window structure for PWS = 4 bytes [64].

7.2 LZ77 Compression

We have followed the state-of-the-art LZ77 pipelined architecture, which is shown in Figure 7.1. We have considered a PWS of 16 bytes, by which every clock cycle 16 sub-strings, each 16 bytes long, will look up the dictionaries to result the best possible string match, which can be encoded to a single length-distance pair [79]. PWS of 16 bytes denotes every clock cycle 16 bytes can be compressed, which also demonstrates the input compression throughput and maximum possible string match length.

Figure 7.2 shows how PWS of four bytes sub-strings are hashed into the dictionaries to find the best possible string match with the previous strings.

7.2.1 Hashing

Overview

Hashing is the critical logic in LZ77 compression that decides both compression ratio and performance. First, every cycle 16 sub-strings are fetched from the sliding window and they perform hash operations.

Second, hash architecture compares the input sub-strings with the strings that are stored in each dictionary or memory bank and replaces the old sub-strings with the new sub-strings, which is called hash memory update.

Third, PWS bytes from memory banks are matched in parallel with sub-strings and a length-distance pair is generated per each match. If a match length is rejected for not being the best match, then length is still the byte value and distance is denoted as “0” [83].

Finally, the matches in the current window could stretch to the next window, which can lead to match overlapping between windows. This issue is addressed in the pipelined stages.

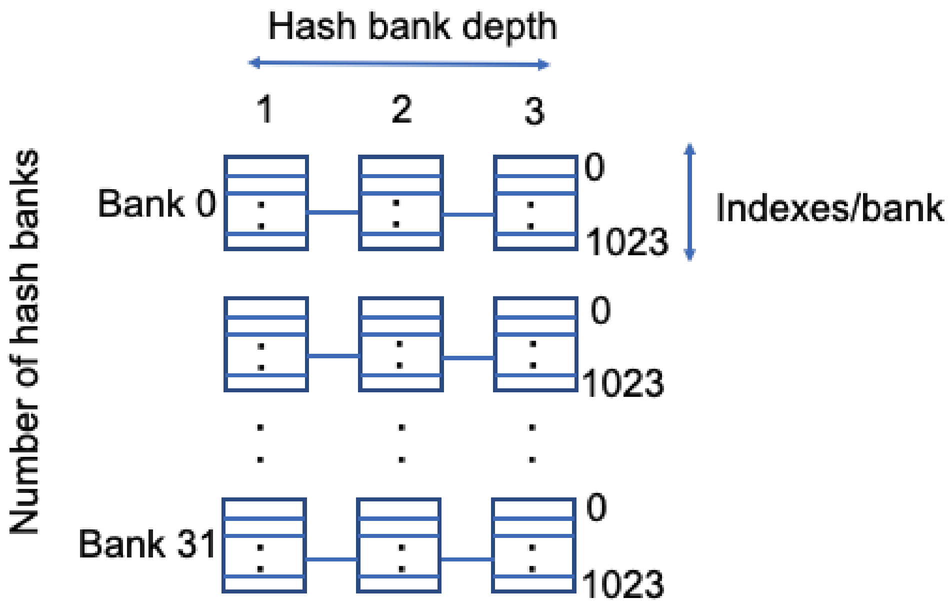


Figure 7.3: Hash bank memory organization architecture for LZ77 [79, 83].

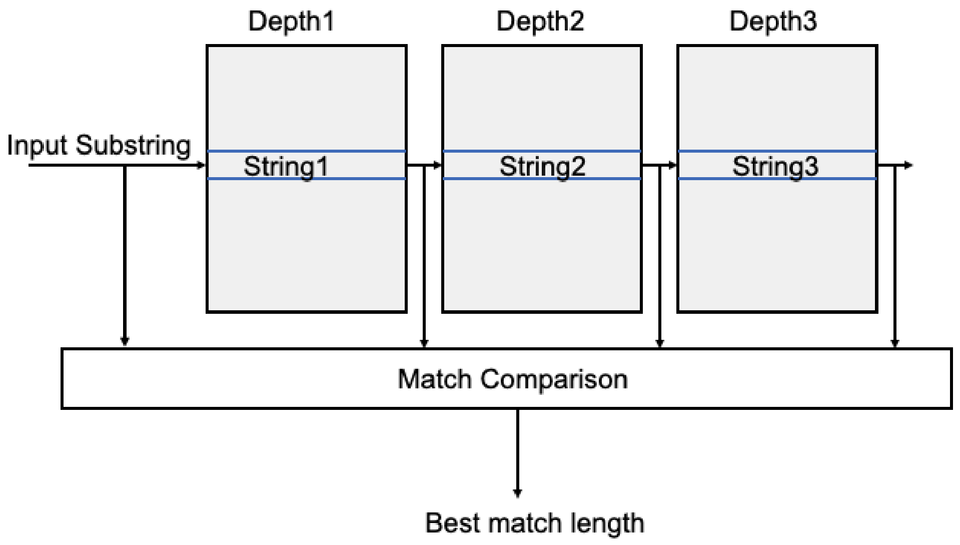


Figure 7.4: Hash bank match comparisons for LZ77 [79, 83].

Hash Chain Organization

Instead of a single hash memory, conventionally, a hash memory chain is used to avoid string match collision. We have adopted the state-of-the-art architecture for a hash chain design [79, 83]. The idea behind a chained hash structure is that every clock cycle at different depths of the chained memory all return a candidate string as the read output, and the current depth's read output is the input written into its next depth's memory. The sub-string in the current window will be stored into the memory in the first depth of the chain [83].

Figure 7.4 shows that *String1*, *String2*, and *String3* are the output of each memory at different depths of the memory chain, and all of them will be compared to the input substring to find the best match. After that, the current input substring will be stored into chain depth 1, *String1* will be stored into chain depth 2, and so on.

Basically, as the substrings are recorded in the dictionary banks, new strings are recorded in the top depth bank while each previously stored string is written from one depth to the next, effectively shifting each of the strings down a level with the oldest string being discarded. At the same time, the previous strings from each depth level are read and compared to the input substrings to find potential matching strings. That's why the hash bank design is connected, which is shown in Figure 7.3.

In our hash bank design, which is shown in Figure 7.3, we have used 1024 indexes per bank and 32 banks for hash bank structure. The reasoning behind that comes from the fact that increasing hash indexes per bank further has diminishing return on compression ratio, which is one of the primary metrics. The overall area and memory requirement further increases with an overall impact on datapath speed.

Hash Function

Hash function is critical in deciding the compression ratio along with the PWS value. The other precaution with the chosen hash function is to ensure minimum hash conflicts, so that compression ratio doesn't get affected much.

In our design, the hash bank architecture uses 32 banks and each bank has 1024 indexes. Therefore, we need a 15-bit (5 bits for bank selection and 10 bits for addressing indexes) hash value.

Algorithm 3 Hash Function Used In Our Design

```
Input: $curr\_window[2 \text{ X } PWS]$, array of 8-bit integers
Output: $hash[PWS]$, array of 14-bit integers
for $i$ = 0 to $PWS$ - $1$ do
    $hash[i] = ((curr\_window[i]) \ll 7)\ xor\ ((curr\_window[i+1]) \ll 6)\ xor$
        $((curr\_window[i+2]) \ll 5)\ xor\ (curr\_window[i+3]) \ll 4\ \ xor$
        $((curr\_window[i+4]) \ll 3)\ xor\ (curr\_window[i+5])\ xor$
        $(curr\_window[i+6])\ ...\ xor\ (curr\_window[i+14])\ xor\ (curr\_window[i+15]);$
        $//Here\ the\ first\ 5\ bytes\ are\ shifted\ and\ then\ all\ gets\ XORed.$
        $//PWS\ of\ 16\ is\ used\ in\ our\ design.$
end for
```

We referred to the hash functions that were designed for 10-bit hash [64] and 14-bit hash [79]. The primary aim with the hash function for banked memory architecture is to make sure hash function doesn't produce more conflicts. Therefore, the first byte is shifted to the left by 7 and the next 2 bytes of the current window are shifted left to at least by 5 to make sure more randomness is there with the left most 5-bits of the hash function, which decides a bank out of 32 available banks.

String Match Comparison and Match Length Computation

To do match comparisons efficiently, match comparisons are done at the output of the banks so that the best outcome from the string match comparison is sent to a substring multiplexer [83]. The match comparison logic is executed by comparing each byte simultaneously and storing the results in a bitmap. This parallel approach reduces the critical path than the sequential comparison. With a PWS of 16 bytes, match length ranges from 0 to 16, therefore, we need 16 numbers of 32:1 5-bit wide MUXes after the match comparison stage. Next, the match with the longest match length is MUXed with the substring window.

To filter out conflicting matches, we adopt a last-fit match selection heuristic that is used in which the last match in the window is always kept as it is more likely that later matches will be longer than earlier ones. One consequence of allowing matches to span from the front half of the sliding window to the second half is that matches found in the current iteration may conflict with matches in the following iteration. The last match selected is used to calculate the First Valid Position (FVP) in the window for the next iteration, where the window bytes have not already been replaced with a length-distance pair [64, 79].

Data Output Format

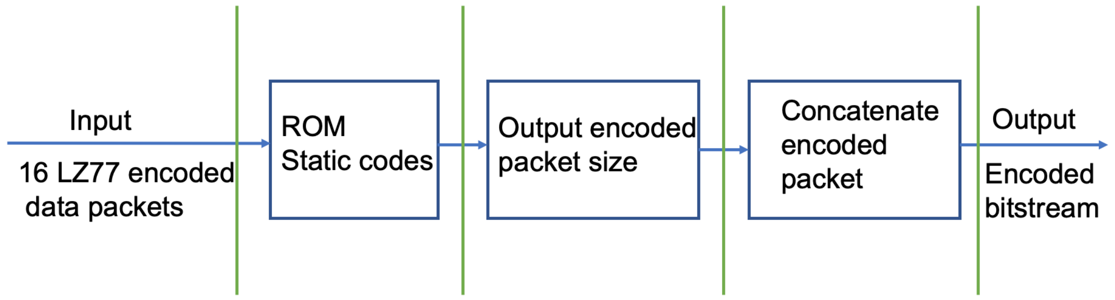
The output from LZ77 can be of three types: matched literals, unmatched literals, and matches. We have adopted the architecture for data packing from a recent work [79] that offers simultaneous Canonical Huffman encoding as the next step.

Each cycle output of the LZ77 compression gives 16 128-bit data packets or boxes [79]. There are also 2-bit flags that determines whether the corresponding data packet is for a matched literal (2'b00) or unmatched literals (2'b01) or matches (2'b10), All of the boxes following a match, up to the match length, are the matched literals and they are removed from the compressed data stream. The boxes containing matches are filled with the length and distance of the match while unmatched literal boxes are filled with the literal bytes from the sliding window.

In a length-distance pair, the length requires at least 4 bits (for values 3 to 16) and the distance requires at least 15 bits (for values 1 to 32768), together occupying 19 bits. This means that each output box needs to be 3 bytes (24 bits) wide in order to fit a length-distance pair, requiring a total of 48 bytes (384 bits) for all 16 boxes. We do this so that each output box can be classified and filled simultaneously [79].

7.3 Canonical Huffman Encoding

The output from LZ77 compression goes for the canonical Huffman encoder module for bit-level compression. We will discuss separate pipelined architecture for both static and dynamic canonical Huffman encoder below.



Input
16 LZ77 encoded data packets
ROM Static codes
Output encoded packet size
Concatenate encoded packet
Output
Encoded bitstream

Figure 7.5: Pipelined static canonical Huffman encoder block diagram.

7.3.1 Static Canonical Huffman Encoder

The literals and length-distance pairs that are resulted from the LZ77 compression uses a ROM-based static code table to get compressed.

Each data packet from the LZ77 compression is passed through a symbol coder that decides whether the symbol is a matched literal, unmatched literal, or length-distance pair and encodes them. Unmatched literals and length-distance pairs are encoded using ROMs and extra bits are appended and sent as output.

A matched literal is removed from the encoded bitstream. Figure 7.5 shows the pipelined block diagram for the static canonical Huffman encoding. The longest possible encoded size of a 16-box input window is 15 9-bit literal codes followed by a 26-bit length-distance code (7-bit length, 1 extra length bit, 5-bit distance, 13 extra distance bits) for a total of 161 bits [79].

7.3.2 Dynamic Canonical Huffman Encoder

In the case of the dynamic canonical Huffman encoder, based upon LZ77 output pairs, first the encoder creates a code table per canonical Huffman encoder architecture, that is discussed in Chapter 5. Therefore, a synchronous FIFO as a buffer is used to store the intermediate data from LZ77 first until the code table is produced.

We have adopted the symbol Integration with flag insertion scheme (SIFI) [45, 84] stores literals, length-distance pairs, and metadata in a single data FIFO buffer. If the literal and length-distance are stored in the same buffer, separate metadata is required to indicate whether each tuple is encoded. We define the metadata generated by SIFI as a compression flag. If a tuple is literal, the data buffer stores a compression flag "0" after literal. On the other hand, if a tuple is a length-distance pair, the length is first stored in the data buffer followed by a compression flag of "1" and the distance.

The Huffman encoder reads the compression flag stored in the data buffer to determine whether the tuple currently being processed is a literal or a length-distance pair. When the compression flag is "0", the Huffman encoder determines that the corresponding tuple is original data and recognizes 8 bits from the current pointer as a literal. When the compression flag is "1", the Huffman encoder handles the corresponding tuple as encoded data. Accordingly, 8 bits from the current pointer are recognized as a length, and 14 bits from the 10th bit are recognized as a distance [84].

Let the number of literals and length-distance pairs in all the intermediate data be denoted as N and M, respectively. SIFI utilizes memory space of $(8 \times (N + M) + 14 \times M + N + M)$ bits to store the intermediate data, which is adopted for the synchronous FIFO design in our work [45, 84].

Table 7.1: ASIC implementations results for the Gzip compressor using both static and dynamic canonical Huffman encoding.

Benchmark	Encoder	Compressed Throughput (Mbps)	Voltage (V)	Power (W)	Area (μm^2)	Compressed Bits/ Energy (Bits/µJ)	Compression Ratio
Calgary Corpus	Static	**1095.1**	1.25	**0.5**	**230,656.8**	**2190.3**	2.01
	Dynamic	223.5	1.25	0.9	426,577.7	248.3	**2.47**
Canterbury Corpus	Static	**977.1**	1.25	**0.3**	**230,656.8**	**3257.1**	2.37
	Dynamic	207.9	1.25	0.8	426,577.7	259.8	**3.03**
Artificial Corpus	Static	**1252.4**	1.25	**0.7**	**230,656.8**	**1789.2**	1.63
	Dynamic	298.2	1.25	1.2	426,577.7	248.5	**2.04**
Large Corpus	Static	**321.2**	1.25	**0.9**	**230,656.8**	**356.9**	2.46
	Dynamic	74.7	1.25	1.4	426,577.7	53.3	**2.88**

7.4 ASIC Implementation Results

We evaluate the proposed Gzip encoder implementations using both static and dynamic canonical Huffman decoder architectures on the 45 nm ASIC at 1.25 V. The designs are functionally verified for all benchmarks and the compression ratio, throughput, and energy efficiency results are evaluated.

7.4.1 Area

The ASIC synthesis results show that the pipelined Gzip encoder using a static canonical Huffman encoder occupies total area of 230,656.8 µm^2, where LZ77 encoder occupies total area of 187,925.3 µm^2 and static canonical Huffman encoder occupies a total area of 42,731.5 µm^2.

The pipelined Gzip encoder employing a dynamic canonical Huffman encoder occupies total area of 426,577.7 µm^2, where LZ77 encoder occupies total area of 187,925.3 µm^2 and dynamic canonical Huffman encoder occupies a total area of 238,652.4 µm^2.

7.4.2 Throughput

The pipelined Gzip engine using a static canonical Huffman encoder runs at a maximum clock frequency of 158.1 MHz at 1.25 V. With a PWS of 16 bytes per clock cycle, the maximum input throughput is 2.53 GB/s.

The pipelined Gzip engine using a dynamic canonical Huffman encoder runs at a maximum clock frequency of 32.7 MHz at 1.25 V, achieving a maximum input throughput is 0.52 GB/s.

The output compression throughput for each benchmark is discussed in the Table 7.1. Figure 7.8 shows the throughput improvements for dynamic Gzip implementation over static Gzip.

7.4.3 Energy-efficiency

The static canonical Huffman encoder-based Gzip compressor achieves an energy-efficiency (compressed bits per energy) of 2190.3 bits/µJ, 3257.1 bits/µJ, 1789.2 bits/µJ, and 356.9 bits/µJ for the Calgary, Canterbury, Artificial, and Large Corpus, respectively.

The Gzip compression engine employing a static canonical Huffman encoder offers an energy-efficiency improvement of 9.4$\times$ than a dynamic canonical Huffman encoder-based Gzip

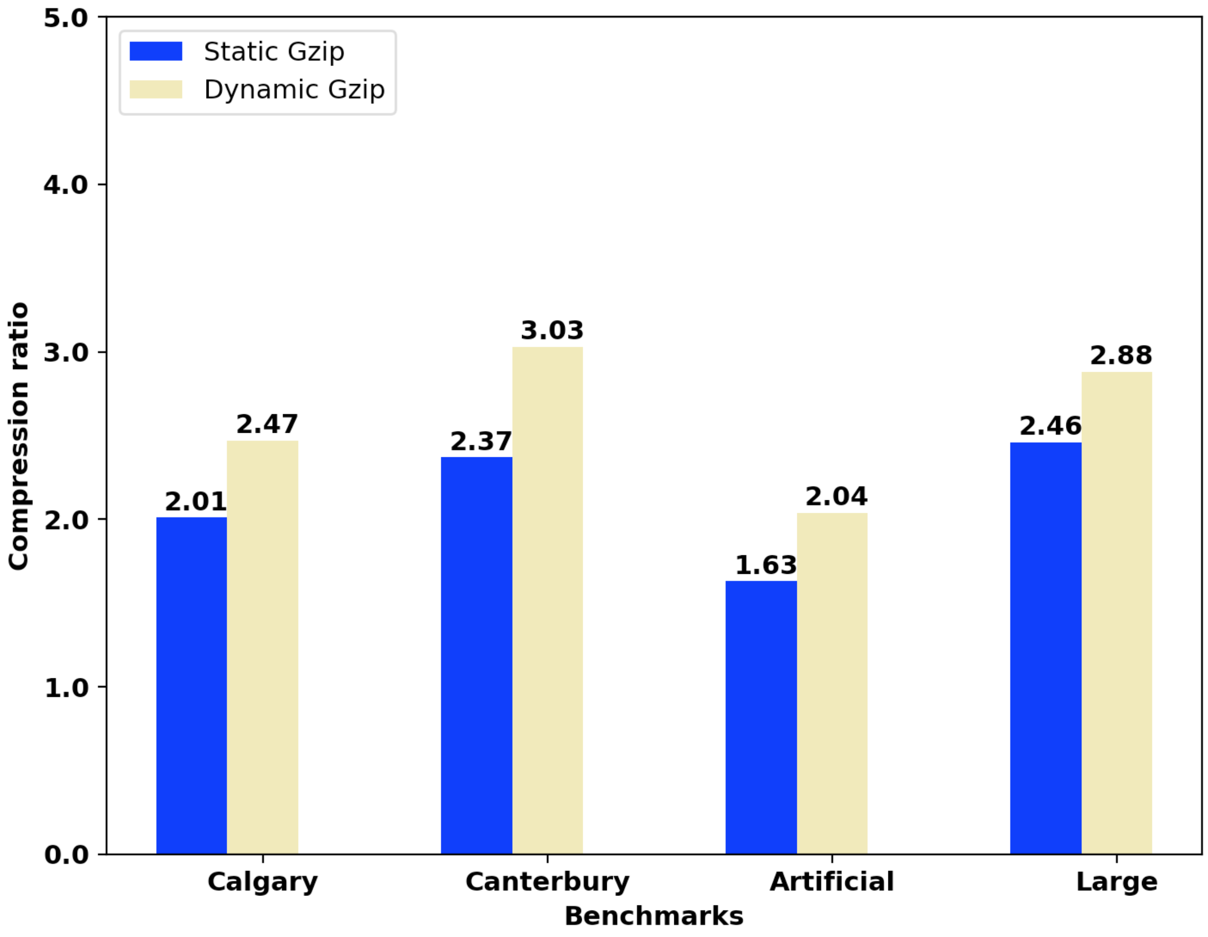


Figure 7.6: Compression ratio results for both static and dynamic Gzip.

compressor. Figure 7.8 demonstrates the energy-efficiency improvements for static Gzip over dynamic Gzip.

7.4.4 Compression Ratio

The compression ratio of the Gzip engine that uses a static canonical Huffman encoding, primarily depends on the efficacy of the hash function and string matching process. Moreover, for the Gzip engine with a dynamical canonical Huffman encoding overall compression ratio also depends on the distribution of the LZ77 output that gets compressed at bit-level.

Using 32 hash memory banks with each bank of 1024 indexes with a depth of 3 for a static encoder achieves a compression ratio of 2.01, 2.37, 1.63, and 2.46 for the Calgary, Canterbury, Artificial, and Large Corpus, respectively. By using a dynamic canonical Huffman encoder, compression ratio of Gzip compression improves by 16.05% than the static canonical-based Gzip compressor at

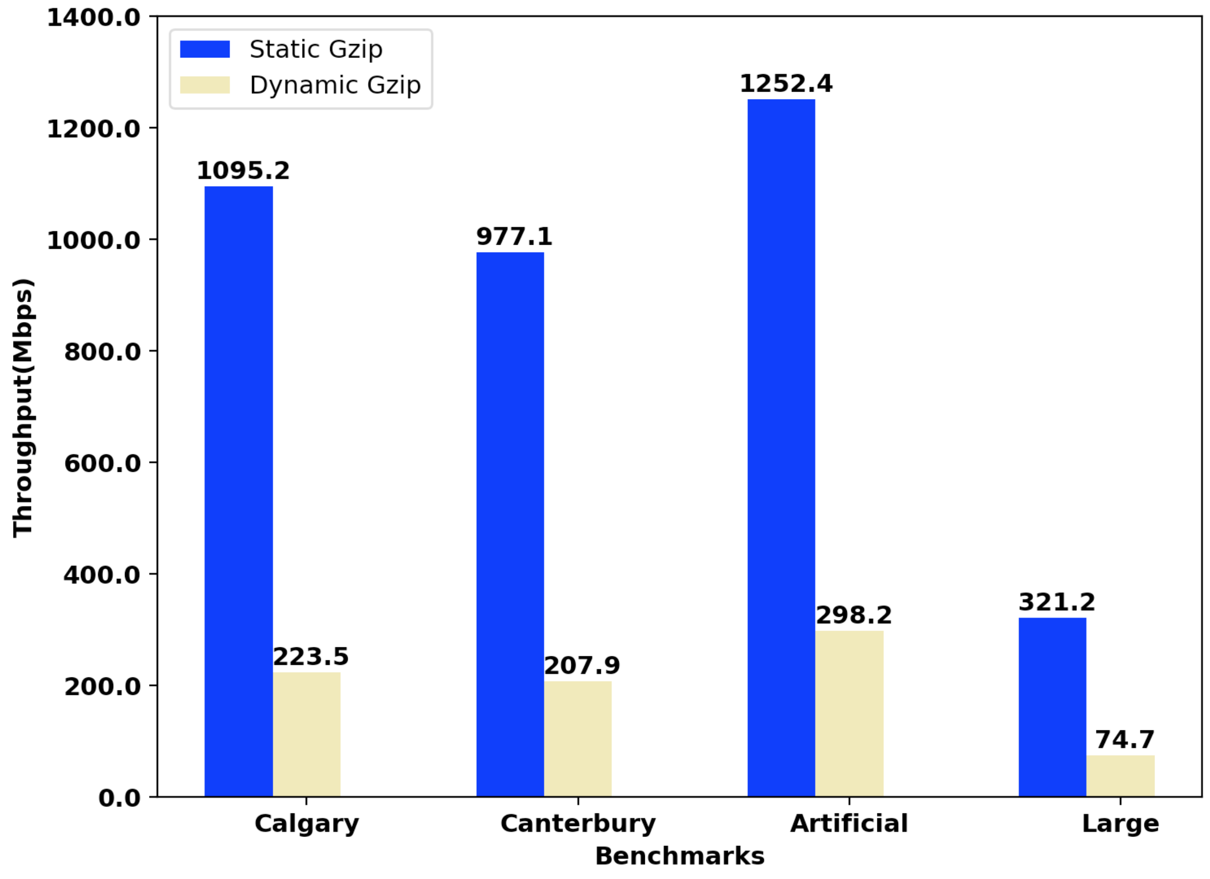


Figure 7.7: Throughput results for both static and dynamic Gzip.

the cost of performance and area.

Figure 7.6 shows the improvements in resulting compression ratios due to dynamic Gzip implementation over static Gzip.

7.5 Comparison of the Gzip Compressors for the Calgary Corpus Benchmark

Table 7.2 details out a comparison of the Gzip compressors for the Calgary Corpus benchmark.

The proposed ASIC implementation offers a compression ratio of 2.47 using a dynamic canonical Huffman encoder, while achieving an input throughput of 0.52 GBPS. The proposed architecture can be modified to have a shrinked hash bank structure to offer better throughput, which means using a memory bank architecture with lesser depth and capacity improves the

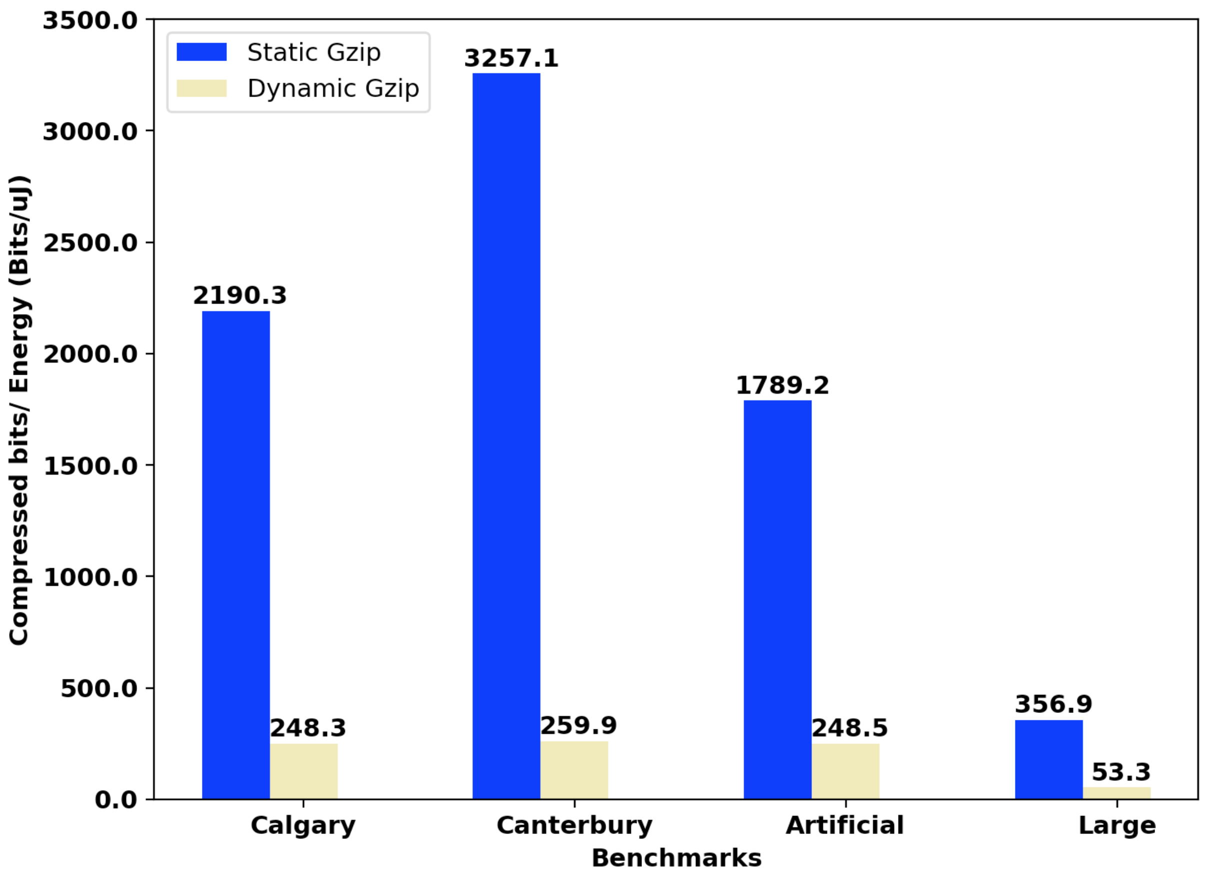


Figure 7.8: Energy-efficiency (compressed bits/µJ results for both static and dynamic Gzip.

overall compression performance. However, with a smaller memory depth, efficacy of parallel string matching, hence, the resulting compression ratio will be worse than the current setup.

Table 7.2 shows that our proposed design for hash bank memory structure with 32 hash memory banks and each bank of 1024 indexes with a depth of 3 and a 15-bits hash function results in a better trade-off of both compression ratio and compression throughput.

Ledwon et al. [79] reports the highest single-engine scaled throughput of 2.8 GBPS using a 512-index hash memory bank structure on a Xilinx Virtex Ultrascale+ FPGA and they achieve a geometric mean compression ratio of 1.92 for the Calgary Corpus.

The throughput of the overall compressor can also be improved by using multiple single-engine compressor blocks in parallel, at the cost of area, energy-efficiency, and compression ratio.

Table 7.2: Comparison of the Gzip compressors for the Calgary Corpus benchmark (scaled to 45 nm).

Reference	Platform	Encoder	Input Throughput	Voltage	Power	Area	Un-compressed Bits/ Energy	Compression Ratio
			(GBPS)	(V)	(W)	(µm^2)	(Bits/nJ)	
Ledwon et al. [79]	Virtex Ultrascale+	Static	**2.87**	NA	NA	NA	NA	1.92
Microsoft [80]	Stratix V	Static	2.33	NA	NA	NA	NA	2.05
Qiao [83]	HARP (Stratix V)	Static	2.67	NA	NA	NA	NA	2.1
This work	ASIC	Static	2.53	1.25	0.5	230,656.8	40.5	2.01
		Dynamic	0.52	1.25	0.9	426,577.7	4.6	**2.47**

Chapter 8

DeepScaleTool : A Tool for the Accurate Estimation of Technology Scaling in the Deep-Submicron Era

This chapter presents the DeepScaleTool, a tool for the accurate estimation of technology scaling in the deep-submicron era. DeepScaleTool is designed by modeling and curve fitting published data by a leading commercial fabrication company for silicon fabrication technology generations from, 130 nm to 7 nm for the key parameters of area, delay, and energy.

8.1 Introduction

Moore's law [85] has been pivotal in the advancement of the semiconductor industry for decades, which lays out a projection of doubling of the transistors on an IC every two years. Similarly, Dennard scaling [86, 87] pioneered the progress by showing scaling across physical dimensions, substrate doping, and supply voltage, which in turn results in lower area, delay, and power dissipation for MOSFETs. The resulting changes due to scaling are depicted in terms of an entity called scaling factor K, which is defined as the ratio of two technology nodes. These scaling factors are discussed in various textbooks [88,89] and the literature [90,91], and are shown in Table 8.1. The key points from the traditional scaling factors shown in Table 8.1 are the following: transistor physical dimensions shrink down by the scaling factor K, which in turn scales down the

Table 8.1: MOSFET device parameters and traditional scaling factors

Parameter	**Scaling factor**
Device dimension (W, L, tox)	$1/K$
Doping concentration Na	K
Voltage V	$1/K$
Current I	$1/K$
Capacitance $\epsilon A/t$	$1/K$
Delay time VC/I	$1/K$
Power dissipation VI	$1/K^2$
Power density VI/A	1

transistor area by a factor of K^2. Similarly, the speed of the transistor increases by a factor of K as the delay reduces by $1/K$.

The traditional scaling factors were accurate until the advent of the deep-submicron era. As the transistors get smaller in the deep-submicron regime, due to short channel effects, effect of leakage current and thermal runaway, and process variation, the traditional scaling estimations are no longer accurate. Leakage current is affected greatly by gate length, oxide thickness, and threshold voltage, so it is becoming a large issue with deep submicron processes, where these values are small, and getting smaller [91, 92]. Moreover, resulting performance gain over recent technology generations is minimal unlike the predictions by traditional scaling estimations.

The prediction of accurate scaling factors is important for a fair comparison of design performance and other metrics across different technology fabrication nodes. The inaccuracy in the traditional scaling factors can be depicted from the real silicon data from various foundries across technology fabrication generations. Notably, Holt [93] discusses transistor scaling and its effect on transistor area, gate delay, switching energy, and energy delay product in the deep-submicron regime based on Intel's data. Bohr and Young [90] describe Intel's scaling trends for area, transistor performance, and cost per transistor over the past decade.

8.2 Existing Method for Accurate Scaling in Deep-Submicron Technologies

A method to propose accurate scaling predictions has been demonstrated using data from PTM [94], ITRS [95], and simulated measurements of Fan Out 4, or FO4 circuit [91]. The primary idea behind the proposed work is that a circuit that has a delay and consumption of X number of FO4 inverter chains in a certain technology size should have roughly the same X number of FO4 inverter chains in a different technology size [96]. Therefore, the authors demonstrate simulated measurements of FO4 models in a range of different sizes and voltages to obtain approximated scaling factors for power, energy, and delay.

The authors [91] created polynomial approximations using a script based on the HSpice simulation data and formulated a third-order polynomial for the delay factor approximations, and second-order polynomials for energy and power factor approximations. Based on the delay, energy, power factor approximations, equations were proposed to scale delay, energy, and power, respectively.

Although the ITRS and PTM based simulation and modeling approach looks viable for predicting scaling factors in the deep-submicron regime, supply voltage information is not always publicly released, which is accounted for modeling the delay and power scaling equations in the article [91]. Moreover, such estimation method doesn't necessarily align with the actual silicon technology scaling trends, which are discussed in Section 7.5.2.

In academia, popular textbooks [88, 89] that cover digital VLSI design and scaling of CMOS transistors describe traditional scaling factors and reasons associated with the discontinuity in traditional scaling trends. However, an accurate estimation of scaling factors across technology fabrication nodes and correlation between traditional scaling factors and scaling factors resulting from actual silicon are usually not covered. Therefore, we propose a scaling tool whose modeling is based on the industrial technology scaling trends and polynomial based curve fitting approach for an easy and accurate estimation of scaling factors in the deep-submicron era.

The major contributions of our work are as follows:

- We demonstrate DeepScaleTool, a spreadsheet-based tool for accurate estimation of scaling factors for area, delay, and energy from 130 nm to 7 nm for educational and research purposes.
- We show and analyze the percentage errors in between classical and estimated scaling factors

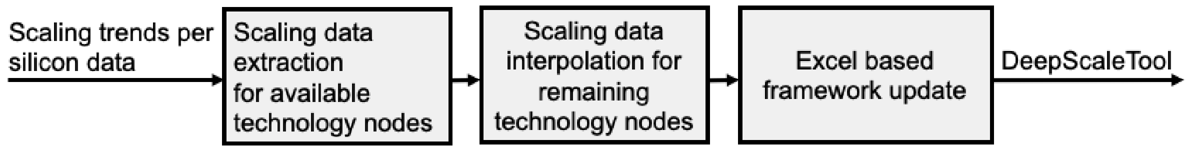

Scaling trends per silicon data
Scaling data extraction for available technology nodes
Scaling data interpolation for remaining technology nodes
Excel based framework update
DeepScaleTool

Figure 8.1: Procedure of Data Collection and Framework

from real silicon data in the deep-submicron range.

- We also illustrate examples of scaling factors estimation using DeepScaleTool and compare the results both with PTM and ITRS based modeling [91] and scaling data from TSMC [97].

8.3 Transistor Scaling Trends, Data Modeling, and DeepScaleTool Framework

Figure 8.1 provides an overview of the steps leading to the design of DeepScaleTool. We analyze published transistor scaling trends [90, 93], curve fit scaling data that are available for certain technology nodes using second-order polynomial based models, and then extrapolate scaling data for the rest of the technology nodes. Finally, we design and update the spreadsheet-based framework using modeled scaling factors for a combination of starting and target technology nodes.

8.3.1 Transistor Scaling Trends

The following two notable transistor scaling trends that are based on Intel's silicon results have been analyzed for our work. Holt [93] discusses generational technology benefits over reduction in gate delay, switching energy, and energy delay product. Figure 8.2 presents the scaling data in the article that span around 65 nm to 10 nm technology nodes and are relative to 65 nm. Moreover, the normalized transistor area across 130 nm to 14 nm technology nodes have been demonstrated. The scaling trends shown in the article over technology fabrication generations infer the following for circuits—acceleration in transistor density, higher performance, and lower power.

Similarly, Bohr and Young [90] discuss scaling logic circuit area from 45 nm to 10 nm. The key takeaway from the presented circuit area scaling data is that with the advent of newer technology nodes and transistor level innovations more aggressive scaling is possible than the traditional scaling

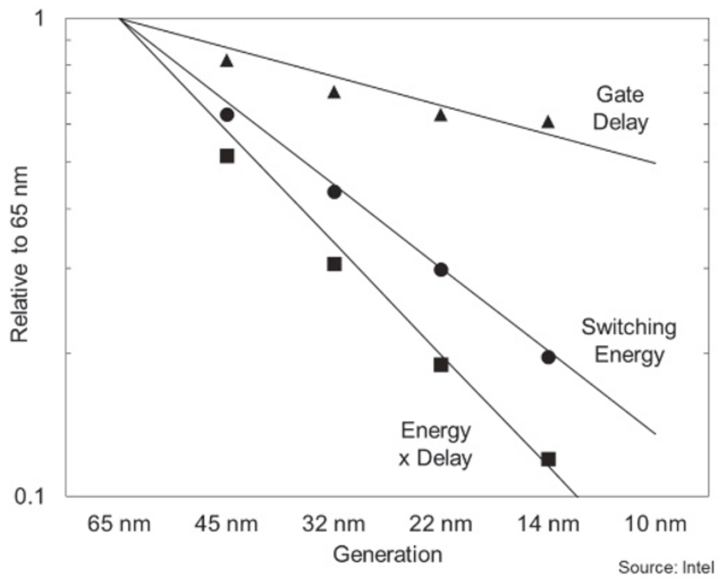


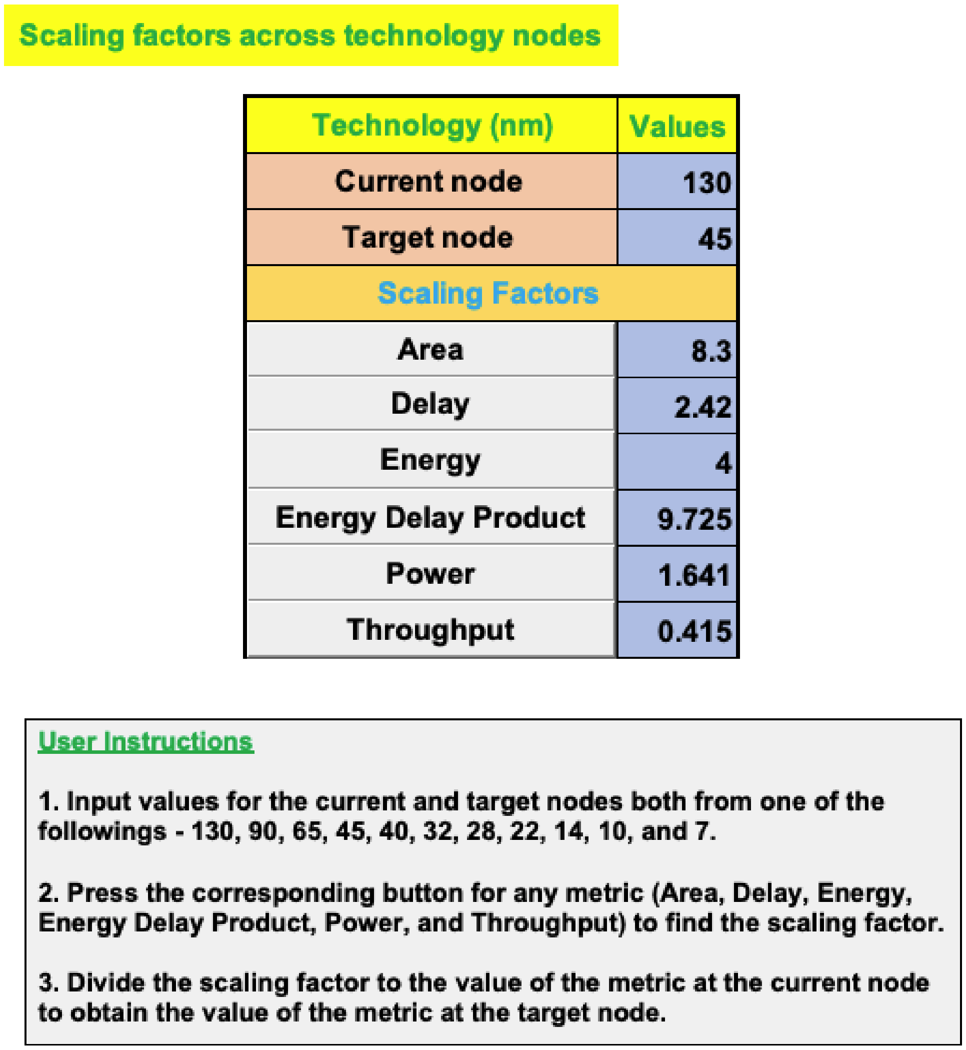
Figure 8.2: Area, delay, and energy trends from 65 nm to 10 nm technology nodes, relative to 65 nm [93]

estimation. For example, both 14 nm and 10 nm technology nodes achieve 0.37 times logic area scaling than the previous generation. This article also presents the trends in improved transistor performance, active power, and performance per watt metrics. However, due to unavailability of proper axis labeling the corresponding trends have not been considered for data modeling purposes in this work.

8.3.2 Data Extraction and Modeling

The g3data [98] tool has been used to extract the digitized data from the plots with technology scaling trends shown in the articles [90, 93]. The plots with proper axis labeling have been considered for data extraction. To obtain scaling trends from 130 nm to 7 nm for key circuit parameters like area, delay, and energy, available scaling data across technology fabrication generations have been extrapolated to obtain the scaling data of the corresponding parameter at the missing technology generations.

The polynomial based extrapolation models that are used to curve fit for various circuit parameters yield a coefficient of determination or R^2 value of equal or greater than 0.99. The small differences in area scaling factors that are obtained after modeling the scaling trends in both articles [90, 93], have been offset by taking the average of the corresponding two scaling factors for


Scaling factors across technology nodes

Technology (nm)	Values
Current node	130
Target node	45
Scaling Factors	
Area	8.3
Delay	2.42
Energy	4
Energy Delay Product	9.725
Power	1.641
Throughput	0.415

User Instructions

1. Input values for the current and target nodes both from one of the followings - 130, 90, 65, 45, 40, 32, 28, 22, 14, 10, and 7.

2. Press the corresponding button for any metric (Area, Delay, Energy, Energy Delay Product, Power, and Throughput) to find the scaling factor.

3. Divide the scaling factor to the value of the metric at the current node to obtain the value of the metric at the target node.

Figure 8.3: Screenshot of the DeepScaleTool page showing the values given for current node and target node, generated scaling factors, and user instructions.

any given starting and target technology generations.

8.3.3 DeepScaleTool Framework

Instead of providing big tables consisting of scaling factors across technology fabrication nodes for each circuit parameter, we present a spreadsheet-based framework for the automated generation of scaling factors for various circuit parameters. The framework is designed using visual basic for applications (VBA) programming language. The scaling factor values fields in the spreadsheet are programmed for any of the supported current and target technology nodes. The DeepScaleTool is available as an open source tool and it can be accessed at https://sourceforge.net/projects/deepscaletool/ [99]. Figure 8.3 depicts a screenshot of the tool. The tool can be updated easily for future nodes with the availability of future scaling trends.

8.4 Usage of DeepScaleTool and Scaling Factor Computation Examples

8.4.1 DeepScaleTool Usage

The usage of the DeepScaleTool is simple, which requires three-fold steps as shown in Figure 8.3. Currently, the tool supports scaling factors for the following fabrication nodes in units of nm: 130, 90, 65, 45, 40, 32, 28, 22, 14, 10, and 7. The user inputs one of those values for the current node and target node. Next, the user can press the corresponding button for any performance metric or parameter to display the scaling factor. Finally, the value of any metric at a target node can be found based on the value of that metric at the current node and resulting scaling factor using the following equation, where x and y denote target node and current node respectively,

$$Value_x = \ Value_y \ / \ Scalingfactor \tag{8.1}$$

8.4.2 Examples of Scaling Factor Computation

The following examples are shown to illustrate the scaling factor computation procedure. To scale a circuit from 130 nm to 45 nm, area scaling factor is 8.3 per the current version of the tool as shown in Figure 8.3. If the circuit occupies an area of 100 um^2 in 130 nm node, the resulting area in 45 nm node using equation (8.1) will be 100 / 8.3 = 12.05 um^2. Similarly, to scale a circuit from 45 nm to 32 nm, the tool displays a power scaling factor of 1.238. If the circuit dissipates 100 mW in 45 nm node, the resulting power dissipation in 32 nm node using equation (8.1) will be 100 / 1.238 = 80.775 mW.

The scaling computations for delay, energy, energy delay product, and throughput can be performed by generating the corresponding scaling factors from the tool and applying equation (8.1) like the above examples. The scaling factors for derived metrics like throughput/area and power density can also be found out using the scaling factors for the corresponding primary metrics that are generated from the tool.

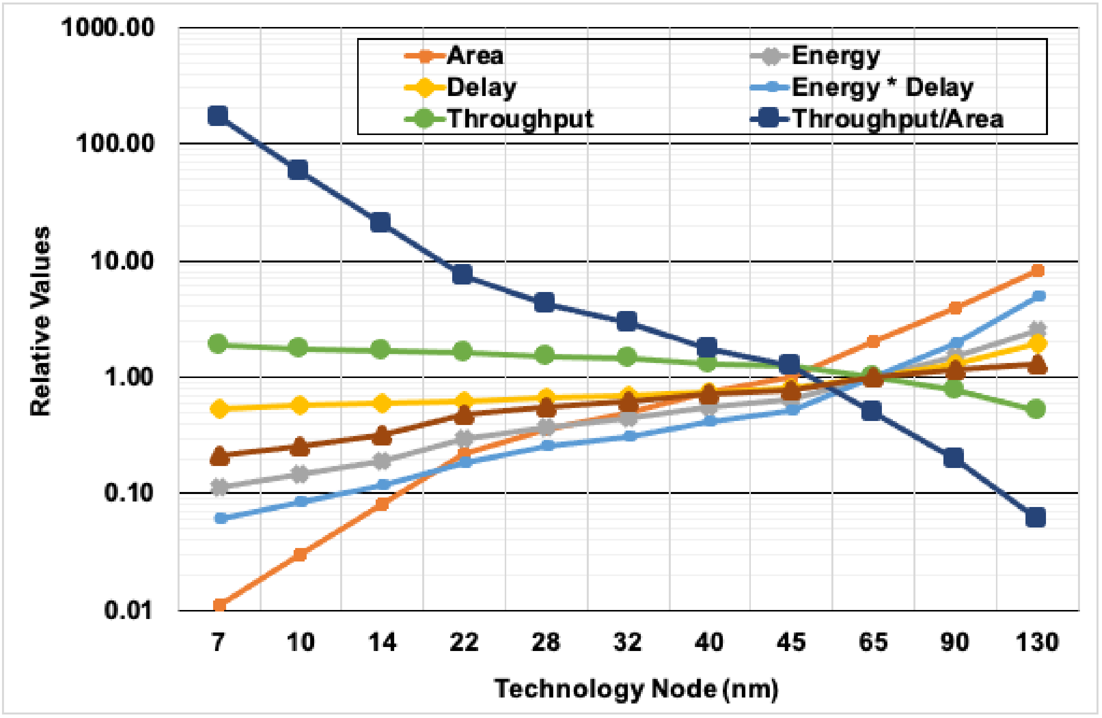


Figure 8.4: Scaling trends for area, delay, power from 130 nm to 7 nm based on the modeling presented in this work [20].

8.5 Comparison of Scaling Factors Estimation Methods and Accuracy with Traditional Scaling

8.5.1 Scaling Trends and Accuracy Analysis with Traditional Scaling

Figure 8.4 shows the scaling trends for transistor area, delay, energy, throughput, and power based on the modeling presented in this work. Among all the considered parameters transistor area achieves a remarkable scaling over the years, which is better than the traditional scaling trends. The delay and throughput achieves minimal scaling with the recent technology nodes. Figure 8.5 shows the percentage error of value of scaling factors modeled for each parameter with respect to traditional scaling factors. The percentage errors shown for each technology node are relative to the values at 130 nm. Transistor area shows the least variation among all parameters as transistor area still scales by the traditional estimation or even better with the advent of transistor innovations like high-dielectric metal gates, FinFET, and 3D FinFET structures [90]. However, delay and power dissipation trends vary significantly as compared to traditional scaling improvements of $1/K$ and $1/K^2$.

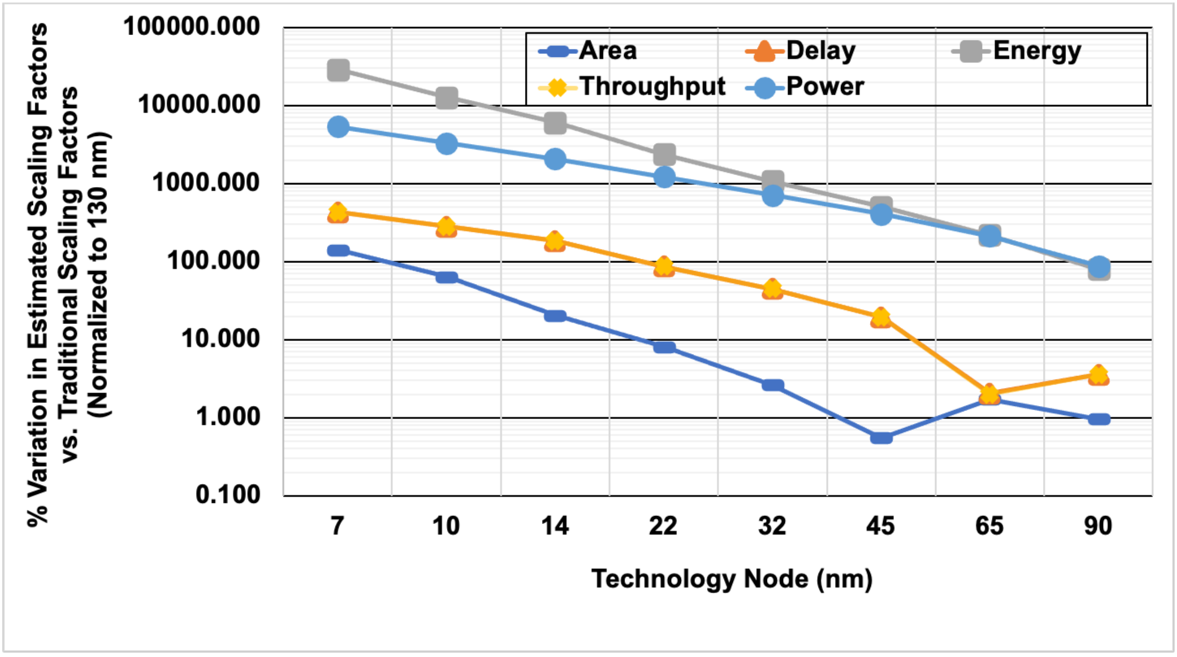


Figure 8.5: Traditional scaling factors vs. modeled scaling factors presented in this work for the deep-submicron regime [20].

Table 8.2: Comparison of area, delay, and power scaling (% reduction) factors for 10 nm to 7 nm scaling. The reference scaling data [97] corresponds to TSMC's technology scaling.

Source	**Area**	**Delay**	**Power**
Cai [97]	30 – 35	10	35
Stillmaker [91]	59	19.1	9.1
This work	**36.7**	**7.5**	**30**

Due to the effect of leakage current threshold voltage scaling doesn't happen aggressively and thus the same effect on scaling for supply voltage as well. Therefore, there is a limit on energy efficiency scaling than the traditional scaling factor of $1/K^3$. Similarly, transistor gate delay scaling is primarily limited by the slower interconnect scaling. The poor trends in gate delay and power dissipation scaling shown in Figure 8.5 affect the energy efficiency scaling and thereby a larger percentage of error is observed for scaling energy. Moreover, as we advance for the recent technology nodes we observe a greater percentage of error in relative scaling values with respect to 130 nm due to the cumulative effect of the error margins across technology nodes.

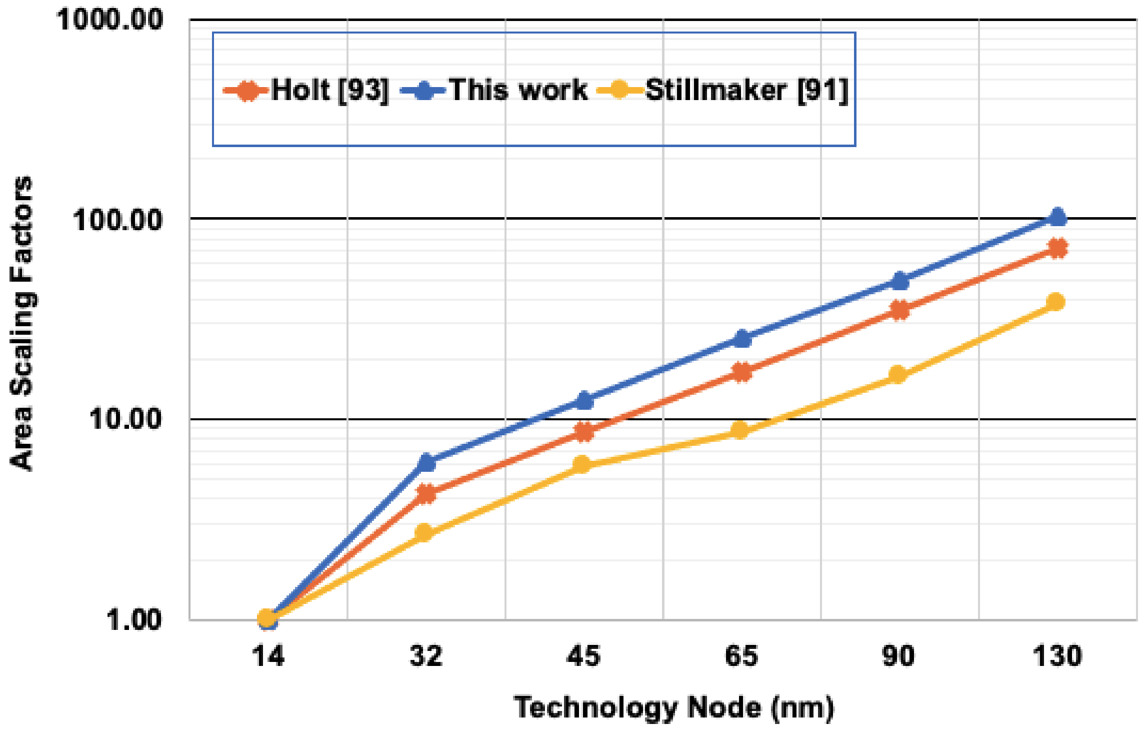


Figure 8.6: Comparison of area scaling factors between our work [20] and Stillmaker [91] for starting nodes from 130 nm to 14 nm with a target node of 14 nm, where in our work shows better correlation to the reference data by Holt [93].

8.5.2 Comparison of Scaling Factor Estimation Methods

Table 8.2 shows the variation in area, delay, and power percentage scaling from 10 nm to 7 nm. The scaling factor values modeled in this work which are based on Intel's silicon technology scaling trends [90, 93] achieve better correlation with TSMC based scaling data [97] than the ITRS and PTM based modeling approach [91] in terms of area, delay, and power. The delay and power scaling data for the article [91] have been calculated using the given corresponding coefficients, supply voltage per ITRS data, and modeling expressions. The errors in between modeled data presented in this work and TSMC's scaling data are 1%, 2.5%, and 5% for area, delay, and power respectively, which states the reliability of DeepScaleTool across major foundries. However, area, delay, and power scaling factors presented in the article [91] differ from TSMC based data by 24–29%, 9.1%, and 24.9% respectively, which is significant given the percentage scaling occurring for those parameters with the said technology nodes.

Moreover, area scaling factors presented in the article [91] varies significantly when compared to modeling based on silicon data as presented by DeepScaleTool as shown in Figure 8.6. For example, 130 nm to 7 nm technology scaling brings down the area by a factor of 110 per the article [91], while the current version of DeepScaleTool suggests the corresponding area scaling factor value of 754.55. The later scaling factor seems more accurate since area scales down by a factor of approximately 303 in general over eight generations from 130 nm to 7 nm. Moreover, the aggressive scaling than the normal rate of 0.49 makes the overall factor shoot up to 754.55. Therefore, DeepScaleTool caters to provide more accurate estimation of scaling factors for various design parameters irrespective of major foundries and it avoids the prediction inaccuracy from ITRS and PTM models.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

This dissertation discusses the hardware architectures for canonical Huffman codec and Gzip compression. The key contributions of this dissertation are as follows.

We have analyzed various canonical Huffman encoding methods and proposed an optimized task-level parallelism architecture with concurrent execution of sorting, Huffman tree generation, and code length computation tasks. The optimized architecture on many-core processor array gives an throughput per area improvement of 4.7× and 88× over GT 750M GPU and Intel i7 respectively. The GPU implementation exploiting its huge die area achieves the highest throughput at the cost of both area and power dissipation. Moreover, the KiloCore implementations achieve throughput per area improvement of 58× and 272× compared to the recent work [58] and [82], respectively. Similarly, the KiloCore implementations achieve energy-efficiency improvement of 4.8× and 22.7× compared to the most-recent work [58] and [82], respectively. We have also proposed a pipelined architecture for both ASIC and FPGA canonical Huffman encoder implementations: ASIC implementation employing the proposed architecture shows 5.1×improvement on throughput and 13.4×energy efficiency improvement over the FPGA implementations.

This dissertation presents optimized LUT-based architectures for both static and dynamic canonical Huffman decoders [100]. The proposed decoder architecture reduces the memory size requirement by 44% than the conventional architecture. The optimized KiloCore design gives an throughput per area improvement of **7**× and 891× over GT 750M GPU and Intel i7 respectively. The

ASIC synthesis results show that the pipelined and memory-efficient static decoder occupies total area of 24,308.7 um^2, while the dynamic version of the design occupies a total area of 37,274.4 um^2. On an average, scaled results on ASIC shows 5.1$\times$improvement on throughput and 13.4$\times$energy efficiency improvement over FPGA implementations.

The hardware architectures for Gzip compression are presented exploiting both static and dynamic canonical Huffman encoding [101, 102]. The proposed ASIC implementation using 32 hash memory banks with each bank of 1024 indexes with a depth of 3 for parallel string matching offers a compression ratio of 2.47 using a dynamic canonical Huffman encoder, while achieving an input throughput of 0.52 GBPS. The pipelined Gzip engine using a static canonical Huffman encoder runs at a maximum clock frequency of 158.1 MHz, resulting in a maximum input throughput of 2.53 GBPS. The pipelined Gzip encoder using a static canonical Huffman encoder occupies total area of 230,656.8 um^2, where a dynamic canonical Huffman encoder version occupies total area of 426,577.7 um^2.

Finally, the dissertation presents DeepScaleTool, an open-source tool designed to provide accurate estimation of scaling factors using published silicon trends and polynomial based curve-fitting method. Although the primary data sets considered for this work belong to Intel's published technology scaling trends, the proposed tool achieves good correlation to the TSMC based scaling trends as well. DeepScaleTool provides an easy platform to obtain reliable scaling factors for various design parameters in the deep-submicron era, understand the discrepancies with traditional scaling factors, and also helps in performing fair comparisons of circuit performance over different technology nodes.

9.2 Future Work

There are several topics related to the Gzip compression and other related lossless compression algorithms that can be investigated in future research.

9.2.1 Optimized Hash Function for LZ77 Compression

Hash function is a key parameter for the LZ77 compression, which has a big impact on the compression ratio. One of the biggest challenge for LZ77 architecture is to explore all possible hash functions for the hash memories and come up with an intelligent function that will not only result in fewer conflicts, but it will also boost-up string matching datapath.

Another challenge related to hash function is that certain hash functions are favorable for specific data sets, so it's an interesting problem to explore how hash functions can be tuned for different data sets to achieve a better performance with the string match process.

9.2.2 Effect of Intrinsic Properties of Data on Compression Speed and Compression Ratio

Different compressors offer different degrees of compression at a different speed. In this dissertation we have analyzed compression ratios, throughput, and energy-efficiency for various Corpus datasets. It will be interesting to explore how intrinsic properties of data affects both compression ratio and compression speed across a variety of compressors, so that informed decisions can be taken by using deep neural networks to result improved compression ratio and compression speed.

9.2.3 LZ77 Compression on Many-Core Processor Array

This dissertation proposes a memory-optimized architecture to result in better compression ratio for LZ77 compression. The many-core processor array-based LZ77 compressor design with parallel hash bank structure is an interesting problem statement to explore. There are certain design challenges that need to be addressed for throughput optimizations, such as efficient mapping of the hash bank memories on to the many-core array and implementation of the hash function to speed up string search mechanism.

9.2.4 Towards Better Throughput-Compression Ratio Trade-offs for the Gzip Compressor

The performance improvement of the Gzip compressor pipeline can further be explored, especially for the dynamic canonical Huffman encoder. Conventionally, most of the research spans around increasing performance of the Gzip compressor using a static canonical Huffman encoding. However, with an improved canonical Huffman encoder datapath compared to the one presented in this dissertation may result in both improved performance and compression efficiency.

Glossary

AsAP ***Asynchronous Array of simple Processors.*** A many-core MIMD-based computing platform.

ASIC ***Application-Specific Integrated Circuit.*** An integrated circuit chip dedicated for a particular application.

CMOS ***Complementary Metal–Oxide–Semiconductor.*** CMOS is the commonly used semiconductor technology in most of today's integrated circuits.

FPGA ***Field Programmable Gate Array.*** An integrated circuit that can be re-configured for different applications.

GPU ***Graphics Processing Unit.*** GPU is an integrated circuit that are efficient at manipulating computer graphics and image processing.

LSB ***Least Significant Bit.*** The rightmost bit in a binary representation.

LUT ***Look-up Table.*** A memory data structure to store information.

LZ77 ***Lempel-Ziv 77.*** A lossless data compression technique.

LZ77 ***Lempel-Ziv 78.*** A lossless data compression technique.

MAC ***Multiply-accumulate.*** A common computing operation that computes the product of two numbers and adds the product to an accumulator.

MIMD ***Multiple Instruction Multiple Data.*** A parallel computing architecture where different processors may be executing different instructions on different pieces of data.

MSB ***Most Significant Bit.*** The leftmost bit in a binary representation.

RISC ***Reduced Instruction Set Computer.*** A RISC instruction set architecture allows it to have fewer cycles per instruction (CPI) than a complex instruction set computer (CISC).

RTL ***Register-Transfer Level.*** A design paradigm that models the circuits in terms of data transfer between registers through a combinational logic.

SRAM ***Static Random Access Memory.*** SRAM is a volatile memory that uses flip-flops to store binary bits.

TDP ***Thermal Design Power.*** The TDP is the maximum amount of heat generated by a computer chip or component (often a CPU, GPU or system on a chip) that the cooling system in a computer is designed to dissipate under any workload.

VLSI ***Very Large Scale Integration.*** VLSI is the process of creating an integrated circuit (IC) by combining millions of transistors or devices into a single chip.

Bibliography

[1] Big data, big energy consumption? `https://datacentremagazine.com/technology-and-ai/big-data-big-energy-consumption`. Accessed: 07/30/22.

[2] The data compression market in 2022. `https://www.datamation.com/big-data/data-compression-market/`. Accessed: 07/30/22.

[3] C. E. Shannon. A mathematical theory of communication. Technical report, Bell System Technical Journal, 1948.

[4] Robert M. Fano. The transmission of information. Technical report, RESEARCH LABORATORY OF ELECTRONICS, MIT, 1949.

[5] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[6] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[7] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. Technical report, 1978.

[8] Deflate compressed data format specification version 1.3. `https://datatracker.ietf.org/doc/html/rfc1951`. Accessed: 07/30/20.

[9] zlib, a massively spiffy yet delicately unobtrusive compression library. `https://www.zlib.net/`. Accessed: 05/30/20.

[10] gzip file format, 2021.

[11] 5 ways facebook improved compression at scale with zstandard. `https://engineering.fb.com/core-data/zstandard/`. Accessed: 07/30/21.

[12] Smaller and faster data compression with zstandard. `https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/`. Accessed: 07/30/21.

[13] New generation entropy coders. `https://github.com/Cyan4973/FiniteStateEntropy`. Accessed: 07/30/21.

[14] Dedicated data compression engine for improved data center efficiency. `https://www.microchip.com/en-us/about/blog/learning-center/compression-benefits-for-storage-and-the-data-center`. Accessed: 07/30/22.

[15] 45nm nangate open cell library. `https://si2.org`. Accessed: 07/30/21.

[16] Intel max 10 fpga device overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/max-10/m10_overview.pdf. Accessed: 07/30/21.

[17] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[18] Brent Bohnenstiehl, Aaron Stillmaker, Jon Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. Kilocore: A 32 nm 1000-processor array. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–23, 2016.

[19] Brent Bohnenstiehl, Aaron Stillmaker, Jon Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. Kilocore: A fine-grained 1,000-processor array for task-parallel applications. *IEEE Micro*, 37(2):63–69, 2017.

[20] Satyabrata Sarangi and Bevan Baas. Deepscaletool : A tool for the accurate estimation of technology scaling in the deep-submicron era. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[21] Intel core i7-4850hq. https://www.notebookcheck.net/Intel-Core-i7-4850HQ-Notebook-Processor.90976.0.html. Accessed: July 25, 2020.

[22] Nvidia geforce gt 750m. https://www.techpowerup.com/gpu-specs/geforce-gt-750m.c2224. Accessed: 07/30/21.

[23] Nvidia tesla gpu. https://en.wikipedia.org/wiki/Nvidia_Tesla. Accessed: July 25, 2022.

[24] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Daniel Gurman, Chi Chen, Jason Cheung, Dean Truong, and Tinoosh Mohsenin. Hardware and applications of asap: An asynchronous array of simple processors. In *2006 IEEE Hot Chips 18 Symposium (HCS)*, pages 1–31, 2006.

[25] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. Asap: A fine-grained many-core platform for dsp applications. *IEEE Micro*, 27(2):34–45, 2007.

[26] Zhiyi Yu and Bevan M. Baas. High performance, energy efficiency, and scalability with gals chip multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):66–79, 2009.

[27] Dean N. Truong, Wayne H. Cheng, Tinoosh Mohsenin, Zhiyi Yu, Anthony T. Jacobson, Gouri Landge, Michael J. Meeuwsen, Christine Watnik, Anh T. Tran, Zhibin Xiao, Eric W. Work, Jeremy W. Webb, Paul V. Mejia, and Bevan M. Baas. A 167-processor computational platform in 65 nm cmos. *IEEE Journal of Solid-State Circuits*, 44(4):1130–1144, 2009.

[28] Dean Truong, Wayne Cheng, Tinoosh Mohsenin, Zhiyi Yu, Toney Jacobson, Gouri Landge, Michael Meeuwsen, Christine Watnik, Paul Mejia, Anh Tran, Jeremy Webb, Eric Work, Zhibin Xiao, and Bevan Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *2008 IEEE Symposium on VLSI Circuits*, pages 22–23, 2008.

[29] Shifu Wu. *Design of Display Stream Compression Video Codecs*. PhD thesis, University of California, Davis, CA, USA, September 2021. http://vcl.ece.ucdavis.edu/pubs/theses/2021-3.swu/.

[30] Shifu Wu and Bevan Baas. Indexed color history many-core engines for display stream compression decoders. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2020.

[31] Brent Bohnenstiehl and Bevan Baas. A software ldpc decoder implemented on a many-core array of programmable processors. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 192–196, 2015.

[32] Bin Liu. *Energy-Efficient Computing with Fine-Grained Many-Core Systems.* PhD thesis, University of California, Davis, Davis, CA, USA, September 2016. `http://vcl.ece.ucdavis.edu/pubs/theses/2016-1/`.

[33] Bin Liu and Bevan M. Baas. Parallel aes encryption engines for many-core processor arrays. *IEEE Transactions on Computers*, 62(3):536–547, 2013.

[34] Zhibin Xiao and Bevan M. Baas. A 1080p h.264/avc baseline residual encoder for a fine-grained many-core system. *IEEE Transactions on Circuits and Systems for Video Technology*, 21(7):890–902, 2011.

[35] Zhibin Xiao, Stephen Le, and Bevan Baas. A fine-grained parallel implementation of a h.264/avc encoder on a 167-processor computational platform. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 2067–2071, 2011.

[36] Sharmila Kulkarni. Implementation of context-based adaptive binary arithmetic coding on KiloCore processor arrays. Master's thesis, University of California, Davis, CA, USA, March 2021. `http://vcl.ece.ucdavis.edu/pubs/theses/2021-2.skulkarni/`.

[37] Emmanuel O. Adeagbo and Bevan M. Baas. In *Technology and Talent for the 21st Century (TECHCON 2015)), title = Energy-Efficient String Search Architectures on a Fine-Grained Many-Core Platform, year = 2015, month = sep.*

[38] Emmanuel O Adeagbo. Energy-efficient pattern matching methods on a fine-grained many-core platform. Master's thesis, University of California, Davis, Davis, CA, USA, March 2017. `http://vcl.ece.ucdavis.edu/pubs/theses/2017-1.Adeagbo/`.

[39] Peiyao Shi. Sparse matrix multiplication on a many-core platform. Master's thesis, University of California, Davis, Davis, CA, USA, December 2018. `http://vcl.ece.ucdavis.edu/pubs/theses/2018-1.pshi/`.

[40] Aaron Stillmaker. *Design of Energy-Efficient Many-Core MIMD GALS Processor Arrays in the 1000-Processor Era.* PhD thesis, University of California, Davis, Davis, CA, USA, December 2015. `http://vcl.ece.ucdavis.edu/pubs/theses/2015-1.stillmaker/`.

[41] Corpus benchmarks. https://corpus.canterbury.ac.nz/descriptions/. Accessed: July 25, 2020.

[42] A. Mukherjee, N. Ranganathan, and M. Bassiouni. Efficient vlsi designs for data transformation of tree-based codes. *IEEE Transactions on Circuits and Systems*, 38(3):306–314, 1991.

[43] Liang-Ying Liu, Jhing-Fa Wang, Ruey-Jen Wang, and Jau-Yien Lee. Cam-based vlsi architectures for dynamic huffman coding. *IEEE Transactions on Consumer Electronics*, 40(3):282–289, 1994.

[44] H. Park and V.K. Prasanna. Area efficient vlsi architectures for huffman coding. In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 437–440 vol.1, 1993.

[45] Suzanne Rigler, William Bishop, and Andrew Kennings. Fpga-based lossless data compression using huffman and lz77 algorithms. In *2007 Canadian Conference on Electrical and Computer Engineering*, pages 1235–1238, 2007.

[46] J. Matai, J. Kim, and R. Kastner. Energy efficient canonical huffman encoding. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 202–209, 2014.

[47] J.S. Patrick, J.L. Sanders, L.S. DeBrunner, V.E. DeBrunner, and S. Radharkrishnan. Jpeg compression/decompression via parallel processing. In *Conference Record of The Thirtieth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 596–600 vol.1, 1996.

[48] G. Lakhani and V. Ayyagari. Improved huffman code tables for jpeg's encoder. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):562–564, 1995.

[49] Cai Ken, Liang Xiaoying, and Liu Chuanju. Sopc based flexible architecture for jpeg enconder. In *2009 4th International Conference on Computer Science Education*, pages 1151–1154, 2009.

[50] Hao Chen, Xinyu Guo, and Ye Zhang. Implentation of onboard jpeg xr compression on a low clock frequency fpga. In *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 2811–2814, 2016.

[51] Sudhir Satpathy, Vikram Suresh, Raghavan Kumar, Vinodh Gopal, James Guilford, Kirk Yap, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, Ram Krishnamurthy, and Sanu Mathew. A 220-900mv 179mcode/s 36pj/code canonical huffman encoder for deflate compression in 14nm cmos. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, 2019.

[52] Zhenyu Shao, Zhixiong Di, Quanyuan Feng, Qiang Wu, Yibo Fan, Xulin Yu, and Wenqiang Wang. A high-throughput vlsi architecture design of canonical huffman encoder. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(1):209–213, 2022.

[53] Habibelahi Rahmani, Cihan Topal, and Cuneyt Akinlar. A parallel huffman coder on the cuda architecture. In *2014 IEEE Visual Communications and Image Processing Conference*, pages 311–314, 2014.

[54] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A. Ross. Massively-parallel lossless data decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 242–247, 2016.

[55] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. Adaptive lossless data compression method optimized for gpu decompression. *Concurrency and Computation: Practice and Experience*, 29(24):e4283, 2017.

[56] C. A. Angulo, C. D. Hernández, G. Rincón, C. A. Boada, J. Castillo, and C. A. Fajardo. Accelerating huffman decoding of seismic data on gpus. In *2015 20th Symposium on Signal Processing, Images and Computer Vision (STSIVA)*, pages 1–6, 2015.

[57] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[58] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. Revisiting huffman coding: Toward extreme performance on modern gpu architectures, 2020.

[59] Satyabrata Sarangi and Bevan Baas. Canonical huffman decoder on fine-grain many-core processor arrays. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ASPDAC '21, page 512–517, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Y. Kim, I. Hong, and H. Yoo. 18.3 a 0.5v 54w ultra-low-power recognition processor with 93.5% accuracy geometric vocabulary tree and 47.5% database compression. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, pages 1–3, 2015.

[61] André Weißenberger and Bertil Schmidt. Massively parallel huffman decoding on gpus. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Seong Hwan Cho, T. Xanthopoulos, and A. P. Chandrakasan. A low power variable length decoder for mpeg-2 based on nonuniform fine-grain table partitioning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):249–257, 1999.

[63] Z. Aspar, Z. Mohd Yusof, and I. Suleiman. Parallel huffman decoder with an optimized look up table option on fpga. In *2000 TENCON Proceedings. Intelligent Systems and Technologies for the New Millennium (Cat. No.00CH37119)*, volume 1, pages 73–76 vol.1, 2000.

[64] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, New York, NY, USA, 2014. Association for Computing Machinery.

[65] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. High-throughput fpga-based hardware accelerators for deflate compression and decompression using high-level synthesis. *IEEE Access*, 8:62207–62217, 2020.

[66] Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu, and Kehua Sheng. Fpga implementation of gzip compression and decompression for idc services. In *2010 International Conference on Field-Programmable Technology*, pages 265–268, 2010.

[67] C. A. Angulo, C. D. Hernández, G. Rincón, C. A. Boada, J. Castillo, and C. A. Fajardo. Accelerating huffman decoding of seismic data on gpus. In *2015 20th Symposium on Signal Processing, Images and Computer Vision (STSIVA)*, pages 1–6, 2015.

[68] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. Parallel lossless data compression on the gpu. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[69] A. Ozsoy and M. Swany. Culzss: Lzss lossless data compression on cuda. In *2011 IEEE International Conference on Cluster Computing*, pages 403–411, 2011.

[70] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. Adaptive lossless data compression method optimized for gpu decompression. *Concurrency and Computation: Practice and Experience*, 29(24):e4283, 2017.

[71] Cody Rivera, Sheng Di, Jiannan Tian, Xiaodong Yu, Dingwen Tao, and Franck Cappello. Optimizing huffman decoding for error-bounded lossy compression on gpus, 2022.

[72] T. Ferguson and J. Rabinowitz. Self-synchronizing huffman codes (corresp.). *IEEE Transactions on Information Theory*, 30(4):687–693, 1984.

[73] Naoya Yamamoto, Koji Nakano, Yasuaki Ito, Daisuke Takafuji, Akihiko Kasagi, and Tsuguchika Tabaru. Huffman coding with gap arrays for gpu acceleration. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[74] Pkzip:archive and transfer critical data. `https://www.pkware.com/zip/products/pkzip/`. Accessed: 07/30/22.

[75] Hypertext transfer protocol – http/1.1. `https://datatracker.ietf.org/doc/html/rfc2616`. Accessed: 07/30/22.

[76] Zip file format. `https://docs.fileformat.com/compression/zip/`. Accessed: 07/30/22.

[77] Gz file format. `https://docs.fileformat.com/compression/gz/`. Accessed: 07/30/22.

[78] Png file format. `https://en.wikipedia.org/wiki/Portable_Network_Graphics`. Accessed: 07/30/22.

[79] Morgan Ledwon. Design of fpga-based accelerators for deflate compression and decompression using high-level synthesis. Master's thesis, University of Alberta, 2019. `https://era.library.ualberta.ca/items/92b97d06-1b57-47ba-9018-5122d6a5027c/view/ad387bf9-d3b3-44a9-9e37-db6f8cabb353/Ledwon_Morgan_201909_MSc.pdf`.

[80] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, 2015.

[81] Michael Dipperstein. Huffman code discussion and implementation. https://michaeldipperstein.github.io/huffman.html, 2018. Accessed: December 27, 2018.

[82] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. cuSZ. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. ACM, sep 2020.

[83] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled cpu-fpga platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, 2018.

[84] Youngil Kim, Seungdo Choi, Joonyong Jeong, and Yong Ho Song. Data dependency reduction for high-performance fpga implementation of deflate compression algorithm. *Journal of Systems Architecture*, 98:41–52, 2019.

[85] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[86] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[87] G. Baccarani, M. R. Wordeman, and R. H. Dennard. Generalized scaling theory and its application to a 1/4 micrometer MOSFET design. *IEEE Transactions on Electron Devices*, 31(4):452–462, 1984.

[88] Jan M Rabaey, Anantha P Chandrakasan, and Borivoje Nikolić. *Digital integrated circuits: a design perspective*, volume 7. Pearson Education, 2003.

[89] J.P. Uyemura. *Introduction to VLSI Circuits and Systems*. John Wiley Sons, Inc., Hoboken, NJ, first edition, 2002.

[90] M. T. Bohr and I. A. Young. CMOS scaling trends and beyond. *IEEE Micro*, 37(6):20–29, 2017.

[91] A. Stillmaker and B. Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal*, 58:74–81, 2017. `http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/`.

[92] K. J. Kuhn. CMOS transistor scaling past 32 nm and implications on variation. In *2010 IEEE/SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*, pages 241–246, 2010.

[93] W. M. Holt. 1.1 moore's law: A path going forward. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 8–13, 2016.

[94] Predictive technology model. `"http://ptm.asu.edu/"`, October 2015. Accessed: October 25, 2018.

[95] International technology roadmap for semiconductors. `"http://www.itrs.net/"`. Accessed: October 25, 2018.

[96] ITRS. Fo4 writeup: International technology roadmap for semiconductors 2003 edition. technical report. `"https://sourceforge.net/projects/deepscaletool/"`, February 2002.

[97] M. Cai et al. 7nm mobile soc and 5g platform technology and design co-development for ppa and manufacturability. In *2019 Symposium on VLSI Technology*, pages T104–T105, 2019.

[98] g3data, a tool for extracting data from scanned graphs. `"https://github.com/pn2200/g3data"`. Accessed: October 25, 2017.

[99] S. Sarangi and B. Baas. Deepscaletool. `"https://sourceforge.net/projects/deepscaletool/"`. Accessed: October 25, 2021.

[100] Satyabrata Sarangi and Bevan Baas. Energy-efficient canonical huffman decoders on many-core processor arrays and fpgas, 2022. In Review.

[101] Satyabrata Sarangi and Bevan Baas. Energy-efficient dynamic canonical huffman encoders on many-core processor arrays, 2022. In Preparation.

[102] Satyabrata Sarangi and Bevan Baas. Hardware architectures for gzip compression, 2022. In Preparation.