



**WEST UNIVERSITY OF TIMIȘOARA
FACULTY OF MATHEMATICS AND COMPUTER
SCIENCE
BACHELOR STUDY PROGRAM: Computer Science**

BACHELOR THESIS

SUPERVISOR:
Conf. univ. dr. Pungilă Ciprian

GRADUATE:
Brega Stanislav

**TIMIȘOARA
2024**

WEST UNIVERSITY OF TIMIȘOARA
FACULTY OF MATHEMATICS AND COMPUTER
SCIENCE
BACHELOR STUDY PROGRAM: Computer Science

Optimization of Bzip2 file compression with OpenCL

SUPERVISOR:
Conf. univ. dr. Pungilă Ciprian

GRADUATE:
Brega Stanislav

TIMIȘOARA
2024

Abstract

This thesis presents an in-depth study and implementation of Bzip2 compression optimization using OpenCL. It aims to enhance the efficiency of handling large volumes of digital data without compromising compression effectiveness. The focus is primarily on overcoming the inherent speed limitations of the traditional Bzip2 algorithm by leveraging the parallel processing power of GPUs.

Initially, the thesis outlines the fundamental concepts and existing solutions in data compression, highlighting the trade-offs between compression rate and speed and setting the stage for the need for optimization. The implementation details of two distinct versions of the Bzip2 algorithm are then discussed: a standard single-core CPU version and an optimized GPU-accelerated version using OpenCL. This work demonstrates the adaptability of OpenCL to handle data-intensive tasks by allowing parallel block compression, which significantly speeds up processing times for large files.

The results of this research are quantified through rigorous testing on various file sizes, showcasing how the GPU-accelerated version dramatically improves processing times and maintains comparable compression rates to the original Bzip2 implementation. The tests indicate that while the GPU version introduces a slight increase in file size due to reduced block size for feasible memory management, it offers substantial time savings for large files.

Challenges encountered during this project included mastering the complexity of the compression algorithms and the steep learning curve associated with OpenCL programming. A notable limitation was the necessity to adjust the block size in the GPU implementation due to memory constraints.

Future work could explore optimizing the single-core CPU version of Bzip2 to narrow the performance gap with its original counterpart and expand the multithreading approach to include other stages of the compression process, such as Huffman coding.

This study confirms that significant performance enhancements can be achieved through parallel processing, suggesting that further exploration into GPU acceleration could provide more efficient solutions for data compression technologies.

Contents

1	Introduction	6
1.1	Motivation	6
1.2	Overview of Popular Compression Algorithms	6
1.2.1	Deflate	6
1.2.2	LZMA	6
1.3	Efficiency Trade-offs	7
1.4	Existing Solutions and Limitations	7
1.5	Objectives	7
2	Application Model Overview	8
2.1	External Perspective	8
2.1.1	Use Case Diagram	8
2.1.1.1	Compress File Use Case	8
2.1.1.2	Decompress File Use Case	9
2.1.1.3	Check File Integrity Use Case	10
2.1.2	Sequence Diagrams	11
2.1.2.1	Run-length encoding	11
2.1.2.2	Burrows-Wheeler Transform	11
2.1.2.3	Move-to-Front Transform	12
2.1.2.4	Huffman Coding	13
2.1.3	Application Usage	13
2.2	Internal Perspective	14
3	Implementation Details	16
3.1	Utility Classes	16
3.1.1	Config Class	16
3.1.2	CRC32 Class	17
3.2	BitStream Classes	18
3.2.1	BitOutputStream Class	18
3.2.2	BitInputStream Class	19
3.3	Burrows-Wheeler Transform	19
3.3.1	Suffix Arrays in BWT	19
3.3.2	Implementation Choices	20
3.4	Move-To-Front	20
3.4.1	Integration of MTF and RLE	20
3.4.2	MoveToFront Class	20
3.4.3	MTFAndRLE2StageEncoder Class	21
3.4.4	Details of the <code>encode()</code> Function	21

3.5	Huffman Encoding and Decoding	22
3.5.1	HuffmanAllocator Class	22
3.5.2	HuffmanStageEncoder Class	23
3.5.3	HuffmanStageDecoder Class	24
3.6	Block Compression and Decompression	25
3.6.1	BlockCompressor Class	25
3.6.2	BlockDecompressor Class	26
3.7	Input / Output Streams	28
3.7.1	OutputStream Class	28
3.7.2	InputStream Class	29
3.8	Parallelization and Usage of OpenCL	30
3.8.1	Implementation using OpenCL	30
4	Results	31
4.1	Compression Rates	31
4.2	Time Enhancement	32
5	Conclusion	34
5.1	Achievements	34
5.2	Challenges Encountered	34
5.3	Limitations	34
5.4	Future Work	34

Chapter 1

Introduction

1.1 Motivation

In today's world of rapidly advancing technology, we deal with an overwhelming amount of digital information. Managing and storing all the files, documents, and media we create and share every day efficiently has become a significant challenge. That is where file compression comes in — a clever way to shrink files so they take up less space and can be stored or sent more efficiently.

However, not all compression methods are designed equally, and data types and needs differ. Some prioritize reducing file size as much as possible, while others aim for a quicker process. This understanding leads us to the motivation behind this thesis—the challenge of making one particular algorithm, Bzip2, faster without sacrificing its impressive compression abilities.

1.2 Overview of Popular Compression Algorithms

1.2.1 Deflate

Among the many compression algorithms in use today, the Deflate algorithm, commonly associated with the .zip format, has become a standard for general-purpose compression. Deflate combines the LZ77 algorithm for string matching and Huffman coding for entropy encoding. In simpler terms, it identifies repeated patterns in the data and replaces them with shorter representations, reducing redundancy. Huffman coding is then applied to efficiently represent the most common elements in compressed data.

Deflate is widely used due to its balance between compression efficiency and speed. It provides a good compromise that suits a variety of applications, making it a popular choice for general file compression.

1.2.2 LZMA

Another notable algorithm in this category is the Lempel-Ziv-Markov chain. LZMA, used in formats like 7z, combines LZ77 and Markov chain algorithms. LZ77 identifies repeated substrings in the data and replaces them with references, similar to Deflate.

The Markov chain component models the probability of the following symbols based on the preceding symbols, enhancing the algorithm’s adaptability to various data types.

LZMA achieves high compression ratios by efficiently encoding short- and long-range patterns in the data, making it a modern alternative to Deflate with improved compression capabilities.

1.3 Efficiency Trade-offs

While Deflate and LZMA adjust compression efficiency and speed, other algorithms, such as Bzip2, prioritize higher compression ratios at the expense of speed. This trade-off is crucial when selecting compression algorithms, especially in scenarios where quick access to data is critical.

This thesis focuses on addressing the speed limitations of the Bzip2 compression algorithm. To achieve this, we will utilize parallel computing and, more specifically, GPU programming through OpenCL.

Unlike traditional CPUs (processors), modern GPUs (video cards) are equipped with a high number of cores (1000+), which is why they excel at handling parallel mathematical operations. OpenCL is a platform-independent framework that allows us to write executable code that can run on GPUs. In our context, OpenCL becomes a powerful tool, providing the means to explore and unlock the potential for optimizing Bzip2 through parallelization on these high-performance GPUs.

1.4 Existing Solutions and Limitations

While other implementations of Bzip2 already exist using parallel computing, a notable limitation lies in the platform specificity of some approaches. Projects utilizing CUDA, for example, offer impressive speed enhancements but are restricted to NVIDIA GPUs. CUDA is a parallel computing platform and programming model developed by NVIDIA, which allows the usage of GPUs for general-purpose processing. The critical difference between CUDA and OpenCL lies in their vendor support. CUDA is specific to NVIDIA, limiting its portability across different GPU architectures, while OpenCL is an open standard supported by multiple vendors, making it more versatile.

This limitation motivates the exploration of OpenCL, a more universal alternative that extends optimization opportunities to a broader range of hardware architectures.

1.5 Objectives

This thesis aims to explore the optimization of Bzip2 file compression through the utilization of OpenCL, emphasizing improving speed without compromising compression ratios. Ultimately, we will also compare our GPU-accelerated solution with the original version of the algorithm that runs on single-core CPUs.

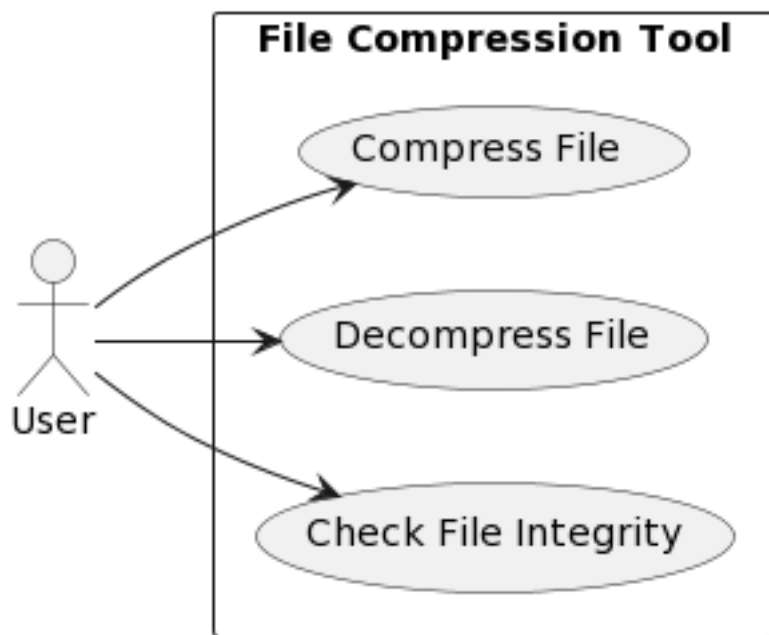
Chapter 2

Application Model Overview

The application will be a command-line tool designed to provide users with an interface for compressing and decompressing files using the optimized Bzip2 algorithm. The user will interact with the tool by executing commands in the command prompt or terminal. The primary functionalities will include compressing a specified file to reduce its size and decompressing a compressed file to restore it to its original state.

2.1 External Perspective

2.1.1 Use Case Diagram



2.1.1.1 Compress File Use Case

Use Case Name: Compress File

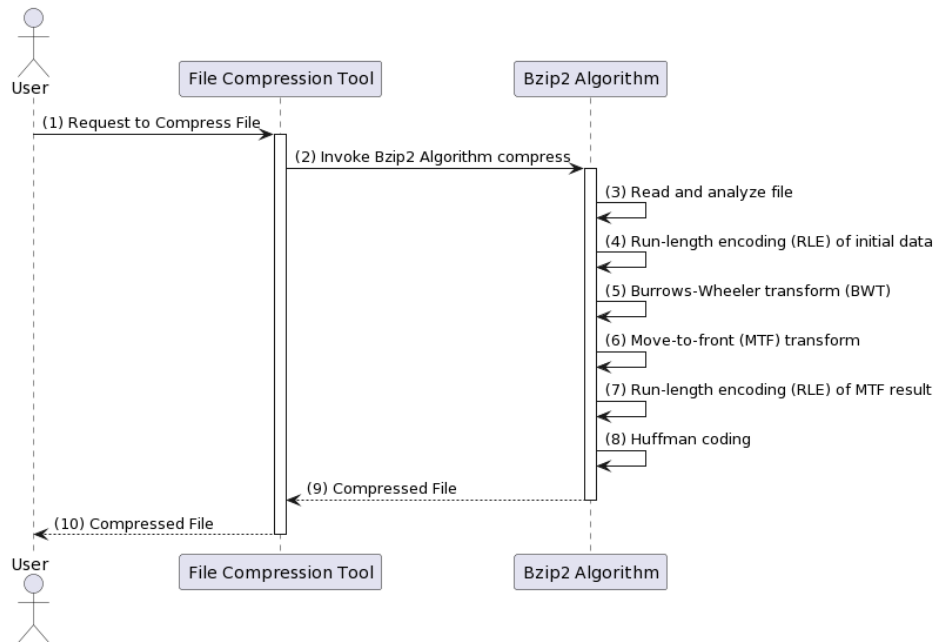
Actor: User

Description: This use case allows the user to compress a specified file, reducing its size for more efficient storage and transmission.

Pre-conditions: The file to be compressed must exist and be accessible. The user

should have the necessary permissions to read the file.

Post-conditions: The specified file is successfully compressed, generating a compressed version while preserving the original file for reference.



The uncompressed input data goes through a series of stages when compressing data. In BZip2, there are five stages[3]: Firstly, it employs Run-Length Encoding to identify and represent repeated sequences in the initial data efficiently. Subsequently, the algorithm applies the Burrows-Wheeler Transform[4] or block sorting, a technique for rearranging data to enhance compressibility. Following this, the Move-to-Front transform is employed, which involves reordering symbols based on their frequency of occurrence. Another round of Run-Length Encoding is then performed on the MTF-transformed data. Finally, the algorithm employs Huffman coding, a widely used entropy encoding method that assigns variable-length codes to different symbols, optimizing the overall compression.

2.1.1.2 Decompress File Use Case

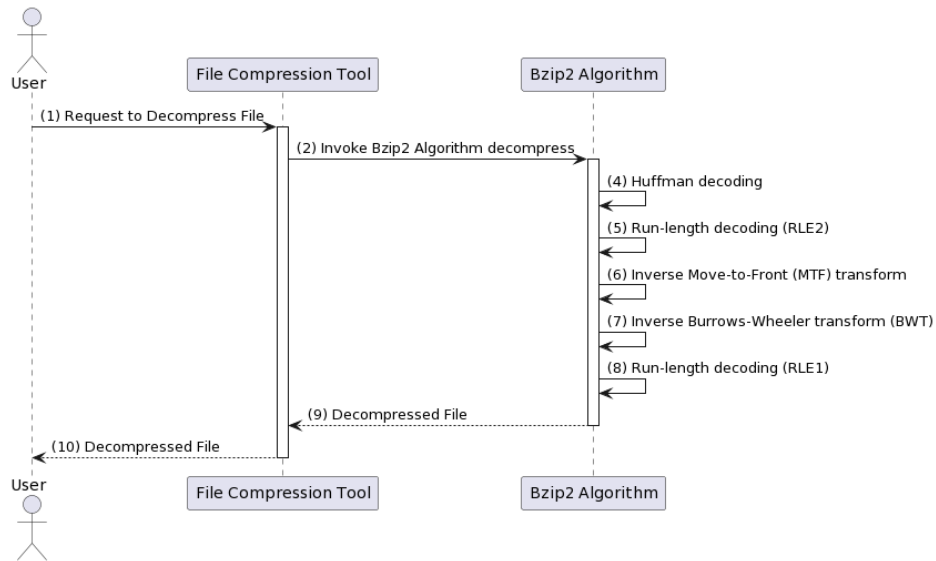
Use Case Name: Decompress File

Actor: User

Description: This use case lets the user decompress a compressed file, restoring it to its original state.

Pre-conditions: A compressed file must exist, and the user should have the necessary permissions to read and decompress the file.

Post-conditions: The specified compressed file is successfully decompressed, generating an uncompressed version while retaining the original compressed file for reference.



The same stages are applied for decompression in the algorithm but in reverse. Huffman and Run-Length coding also have a decoding mechanism, and the Move-to-Front and Burrows-Wheeler transformations have an inverse.

2.1.1.3 Check File Integrity Use Case

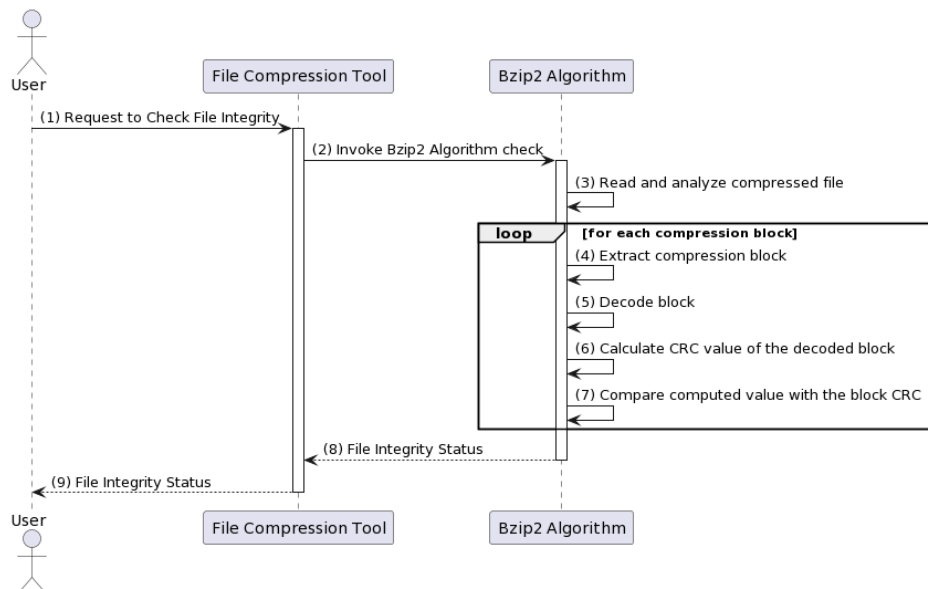
Use Case Name: Check File Integrity

Actor: User

Description: This use case allows the user to verify a file's integrity, ensuring it has not been corrupted during compression.

Pre-conditions: The file to be checked must exist, and the user should have the necessary permissions to read the file.

Post-conditions: The tool provides feedback on the file's integrity, indicating whether it is intact or if any corruption is detected. The original file remains unchanged.



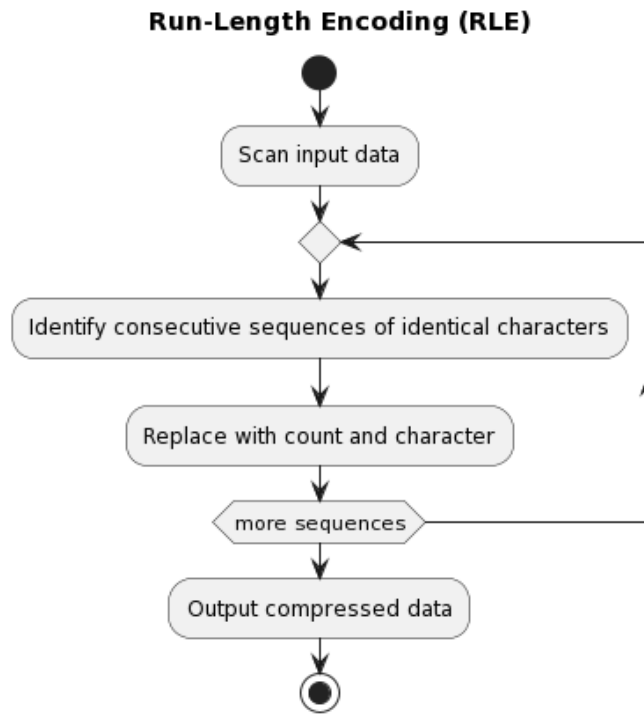
The compression process involves dividing the provided file into several blocks. During this operation, the algorithm compresses the data and keeps track of a CRC

(Cyclic Redundancy Check) value associated with the original content. Each block can be decoded when verifying integrity, and the calculated CRC values can be compared to the stored ones. If these values match, the compressed file retains its original data integrity, providing a reliable method for error detection and ensuring the accuracy of the decompressed data.

2.1.2 Sequence Diagrams

2.1.2.1 Run-length encoding

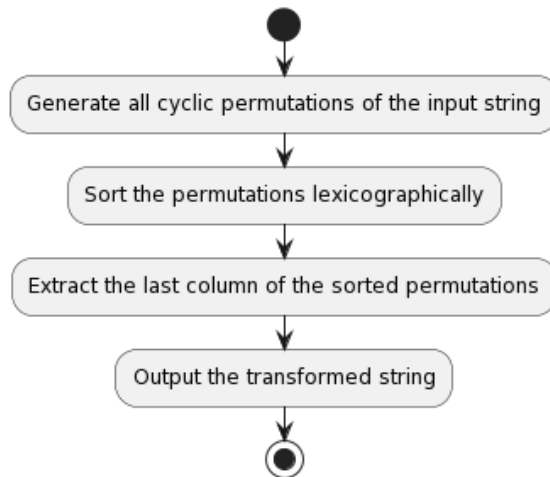
Run-Length Encoding (RLE) is a straightforward compression method that reduces data size by substituting repetitive character sequences with a count and the character itself. This technique is especially effective for data featuring long stretches of identical characters. RLE operates by examining the input data to identify consecutive identical characters. For each identified sequence, the repetition is substituted with a pair that includes the count of the occurrences followed by the character. This compression approach effectively shortens repetitive sequences in the original data into more compact representations.



2.1.2.2 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is a reversible method that reorders a string's characters to better suit compression techniques such as Move-to-Front encoding and Run-Length Encoding. This process includes sorting all cyclic permutations of the input string and collecting the last column from these sorted permutations. This reordering typically creates sequences of repeating characters, which subsequent encoding methods can compress effectively. The process of computing all permutations and sorting them is pretty time-consuming (with a naive approach $O(n^2 \log n)$ time); we will present an optimized process in the next chapter.

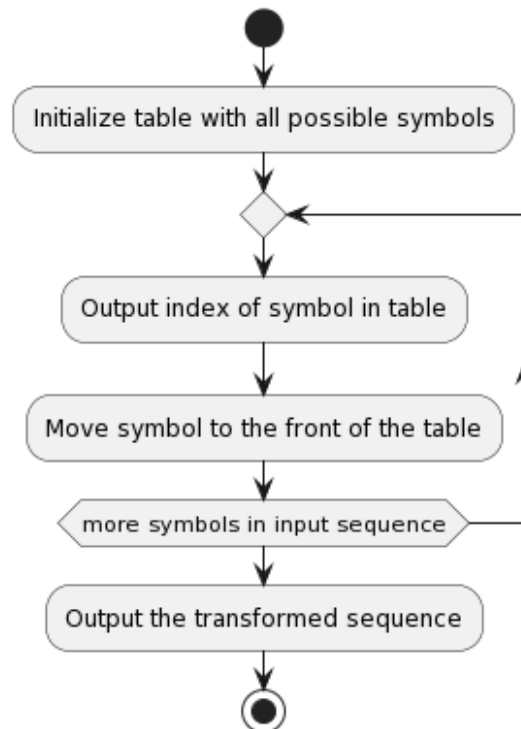
Burrows-Wheeler Transform (BWT)



2.1.2.3 Move-to-Front Transform

The Move-to-Front (MTF) Transform is an encoding method that manipulates a sequence of symbols according to their placement in a dynamic table. The MTF technique involves initializing a table with all possible symbols. As each symbol in the input sequence is processed, its position in the table is recorded, and the symbol is then moved to the front. This method takes advantage of the local repetition of symbols in the input sequence (achieved through the previous BWT step), often leading to more compact representations of repetitive symbols.

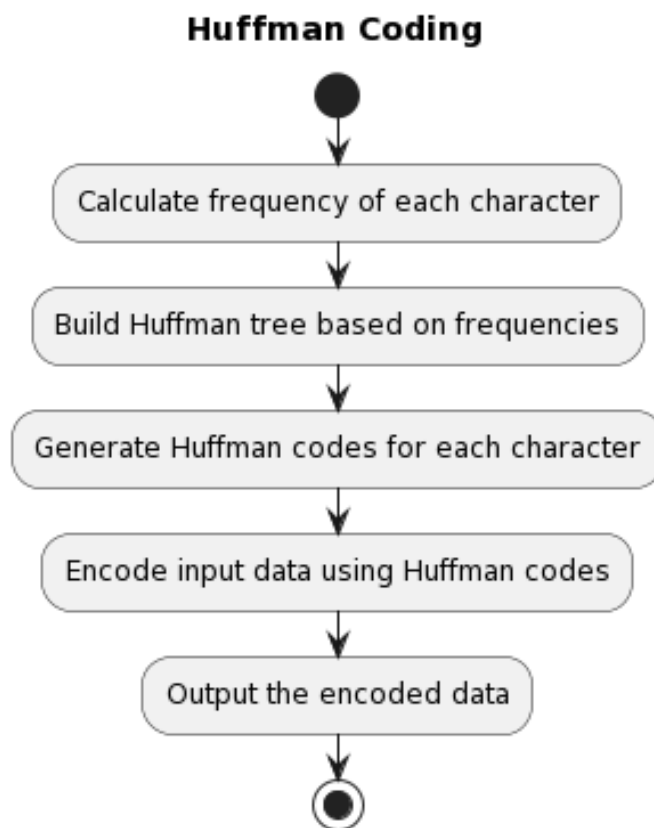
Move-to-Front Transform (MTF)



During the MTF, the algorithm can efficiently perform the next RLE since characters are processed one at a time.

2.1.2.4 Huffman Coding

Huffman Coding, a versatile technique for lossless data compression, allocates variable-length codes to characters based on their frequency. This adaptability ensures that common characters receive shorter codes while rarer characters get longer ones. The process involves building a binary tree, the Huffman tree, constructed from the frequency of characters in the input data. Characters are encoded by navigating the Huffman tree from the root to the leaves, with the path taken determining the code assigned to each character.



In practice, the frequency is calculated on the previous steps to save time, and most importantly, Bzip2 builds and uses multiple Huffman trees (2 to 6). For given input data, the algorithm chooses the most optimal tree, which will result in the highest compression rate, and the tree, which will generate the lowest Huffman code length.

2.1.3 Application Usage

The File Compression Tool can be executed as an executable in the console, providing various functionalities based on user input. Here are the possible command-line options:

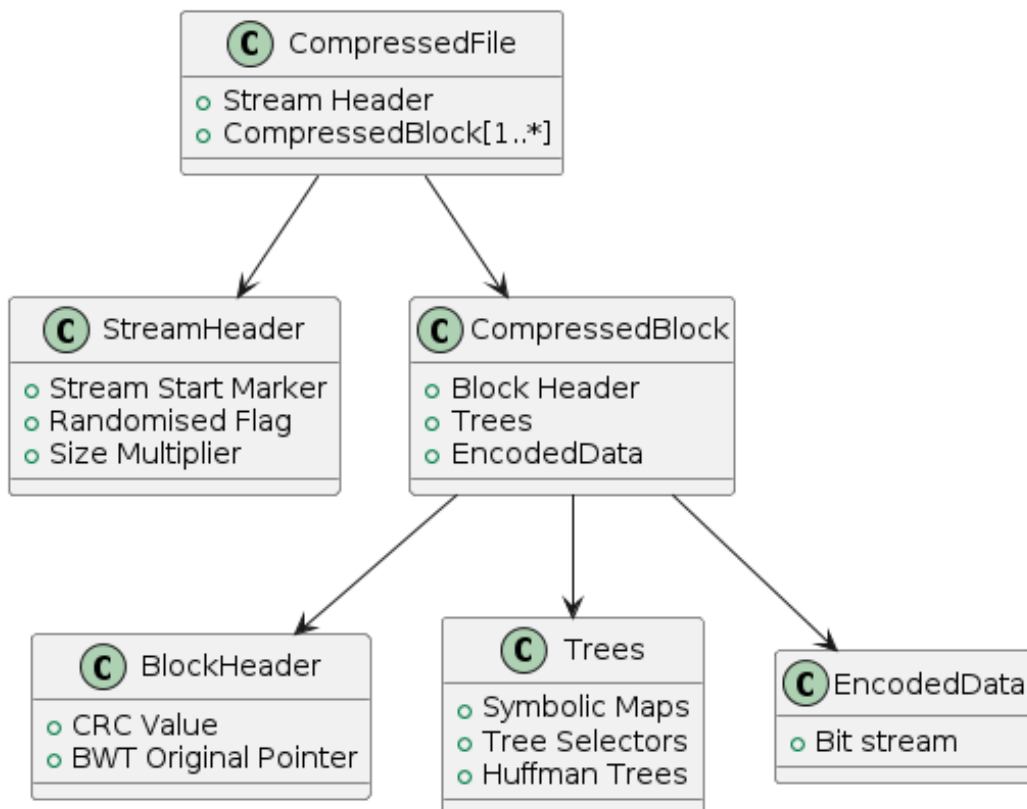
Usage: `executable [options] [file_path]`

Options:

<code>--dec</code>	<code>-d</code>	Decompress file.
<code>--check</code>	<code>-c</code>	Check the integrity of a compressed file.
<code>--keep</code>	<code>-k</code>	Keep the original (de)compressed file.
<code>--size</code>	<code>-s <1-9></code>	Set the block size multiplier for compression.
<code>--parallel</code>	<code>-p <1+></code>	Specify the number of parallel threads for GPU.
<code>--help</code>	<code>-h</code>	Display usage information.

When the tool is invoked with a file path without any flags, it initiates the compression use case, reducing the file size and creating a new file with the same name as the given file but with the ".bzip2" extension. If the "--dec" or "-d" flag is used, the tool employs the decompression algorithm, restoring the original content and removing the ".bzip2" extension. The "--check" or "-c" flag triggers an integrity check of the compressed file, ensuring reliability by verifying CRC values for each block without decompressing the actual file. The "--keep" or "-k" option allows users to retain the original file after (de)compression since the original algorithm also removes the original file[1] after successful execution. Additional options such as "--size" or "-s" allow users to specify the block size for compression, and "--parallel" or "-p" sets the number of parallel processing threads, optimizing performance on suitable hardware. The "--help" or "-h" option quickly references all these functionalities and usage information.

2.2 Internal Perspective



The structure of a bzip2 compressed file is designed to support efficient compression and straightforward decompression processes. The file consists of a single stream containing multiple compressed data blocks. Detailed below is the layout of the compressed file and the components within:

- **File Header:** At the beginning of each compressed stream, there is a header that includes:
 - **Stream Start Marker:** STREAM_START_MARKER - specific bit pattern indicating the start of a compressed stream.
 - **Randomised Flag:** A flag indicating whether a randomized approach was used before the Burrows-Wheeler Transform (this option is no longer supported in the latest bzip2 versions, so we will not bring it up later and assume it's always set to false).
 - **Size Multiplier:** Specifies the block size used in the compression process, ranging from 1 to 9. This multiplier determines the actual block size by multiplying it by a base block size (100000).[1]
- **Compressed Blocks:** Each block within the stream contains:
 - **Block Header:** This segment includes either a BLOCK_HEADER_MARKER or STREAM_END_MARKER - 2 fixed size values to know whether the decompression should continue or end. It also encapsulates the block's CRC value, essential for verifying data integrity, and the original pointer from the Burrows-Wheeler Transform (BWT), facilitating data reconstruction.
 - **Trees Section:**
 - * **Symbolic Maps (SymMap):** These maps detail the symbols used in the Move-to-Front (MTF) transformation.
 - * **Selectors:** Choose the appropriate Huffman trees for decoding.
 - * **Trees:** Contain the Huffman trees necessary for the entropy encoding stage.
 - **Encoded Data:** This segment includes the actual compressed data which is interpreted as a stream of bits for the Huffman processing.

Chapter 3

Implementation Details

This chapter will detail a possible working algorithm implementation based on the original manual[1], some additional details from the informal specification[3], and already publicly available implementations[2][5].

3.1 Utility Classes

The Bzip2 compression algorithm utilizes various utility classes to maintain configuration constants and perform critical calculations like cyclic redundancy checks (CRC). This section details the ‘Config’ and ‘CRC32’ classes that define operational parameters and ensure data integrity.

3.1.1 Config Class

The `Config` class contains constants that define key parameters used throughout the compression and decompression processes:

- **BLOCK_HEADER_MARKER_1 and BLOCK_HEADER_MARKER_2:** Markers used to identify the beginning of a block in the compressed stream.
- **ALPHABET_SIZE:** Represents the size of the alphabet used in the Huffman coding, typically 256 for byte-sized data.
- **BLOCKSIZE_DEFAULT:** The default block size for Bzip2 compression is the basis for calculating the maximum block size.
- **MAX_BLOCK_SIZE:** Maximum size of a block, calculated as nine times the default block size, allowing for varying compression settings.
- **BWT_BUCKET_A_SIZE and BWT_BUCKET_B_SIZE:** Sizes of the buckets used in the Burrows-Wheeler Transform for sorting and managing data.
- **HUFFMAN_GROUP_RUN_LENGTH:** The length of successive symbols encoded using the same tree.
- **HUFFMAN_MAXIMUM_ALPHABET_SIZE:** The maximum number of symbols in the Huffman alphabet, including special run-length symbols.

- **HUFFMAN_MAXIMUM_CODE_LENGTH** Maximum number of bits which a symbol can be translated to a Huffman code.
- **HUFFMAN_MINIMUM_TABLES** and **HUFFMAN_MAXIMUM_TABLES**: The minimum and maximum number of Huffman tables used in multi-table encoding schemes (2 to 6).
- **HUFFMAN_MAXIMUM_SELECTORS**: The maximum number of selector entries, which help in choosing the appropriate Huffman table during decoding.
- **HUFFMAN_SYMBOL_RUNA** and **HUFFMAN_SYMBOL_RUNB**: Special symbols used in 2nd RLE for encoding the running count in binary.
- **STREAM_END_MARKER_1** and **STREAM_END_MARKER_2**: Markers indicating the end of a compressed stream.
- **STREAM_START_MARKER_1** and **STREAM_START_MARKER_2**: Markers denoting the start of a compressed stream - necessary for initializing decoding processes.

3.1.2 CRC32 Class

C CRC32	
□	<u>Crc32Lookup : std::array<uint32_t, 256></u>
□	crc : int
●	getCRC() : int
●	updateCRC(value : int)
●	updateCRC(value : int, count : int)
●	reset()


The **CRC32** class calculates the cyclic redundancy check (CRC) values to verify data integrity. It uses a lookup table based on a polynomial to speed up the computation. Key functionalities include:

- **Crc32Lookup**: A predefined array that provides quick access to CRC values for byte-sized data segments.
- **getCRC()**: Returns the current CRC value, which is used to check the integrity of the data.
- **updateCRC(int value)**: Updates the CRC value based on the input byte.
- **updateCRC(int value, int count)**: Allows updating the CRC value repeatedly for a given count, useful in scenarios where the same data value occurs consecutively.
- **reset()**: Resets the CRC value to the initial state, preparing it for a new data set.

3.2 BitStream Classes

In the Bzip2 compression algorithm, precise bit-level manipulation is essential because the algorithm needs to efficiently pack and unpack data using variable-length codes, such as Huffman codes. These codes often do not align with byte boundaries, making bit-level operations crucial for higher compression ratios by minimizing the amount of unused data.

3.2.1 BitOutputStream Class

 BitOutputStream	
□	bitBuffer : std::bitset<BLOCK_MAX_BIT_SIZE>
□	bitCount : size_t
●	BitOutputStream()
●	getLeftBuffer() : std::vector<bool>
●	writeFileBytes(out : std::ostream, leftBuffer : std::vector<bool>)
●	writeBoolean(value : bool)
●	writeUnary(value : int)
●	writeBits(count : int, value : int)
●	writeInteger(value : int)
●	padding()

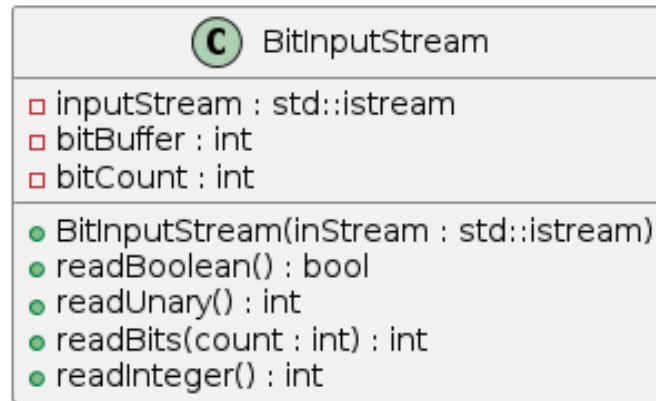
The `BitOutputStream` class manages the writing of bits to an output stream, essential for compressing data. It includes:

- **bitBuffer** (`std::bitset`): Stores bits until they are ready to be written out, ensuring that data is packed as compactly as possible.
- **bitCount** (`size_t`): Tracks the number of bits currently held in `bitBuffer`.

Because the system write operations can be slow, we use a buffer where the compressed data is stored for a block before it's written in the file. The functions used are:

- **writeBoolean(bool value)**: Writes a single bit into the buffer.
- **writeUnary(int value)**: Writes a series of '1' bits followed by a '0', effectively encoding unary numbers used in various compression schemes.
- **writeBits(int count, int value)**: Encodes a specific number of bits from an integer.
- **writeInteger(int value)**: Writes a complete integer to the stream in two segments, ensuring that larger data values are correctly encoded.
- **padding()**: Adds padding bits to align the output to byte boundaries, crucial for maintaining standard output formats (this is done only at the end of the stream).

3.2.2 BitInputStream Class



The `BitInputStream` class facilitates the reading of bits from an input stream, which is crucial for the decompression process. Key components and their functionalities include:

- **inputStream (std::istream&):** A reference to the file stream from where the data is read.
- **bitBuffer (int):** Temporarily holds bits read from the input stream until they are processed.
- **bitCount (int):** Counts the number of bits currently stored in `bitBuffer`.

The functionality provided by the class includes:

- **readBoolean():** Reads a single bit from the stream.
- **readUnary():** Decodes sequences of bits until a zero is encountered, helpful in reading unary numbers typical in run-length encoding.
- **readBits(int count):** Retrieves a specified number of bits, supporting decoding various multi-bit encoded elements.
- **readInteger():** Reads a 32-bit integer by reading two sequences of 16 bits, commonly used to reconstruct numeric values from the compressed data.

3.3 Burrows-Wheeler Transform

In the last chapter, we discussed the problem that the transformation can be the most time-consuming part of the compression sequence when implemented naively. Below, we will discuss how it can be implemented using an advanced data structure.

3.3.1 Suffix Arrays in BWT

Suffix arrays provide a sorted array of all suffixes of a string, which is essential for BWT since the transform involves sorting cyclic shifts of the input data. The resulting order of the suffixes directly contributes to generating the BWT output, where the last characters of each sorted cyclic shift are concatenated to form the transformed data.

3.3.2 Implementation Choices

Despite the existence of various $O(n)$ algorithms for constructing suffix arrays, these often involve compromises that are only ideal for some scenarios. For instance, some require the addition of a unique terminator symbol to the input data; they may utilize additional memory, which can be impractical for large datasets, or they can hide more prominent constants behind the linear complexity.

Given these considerations, we can opt for the DivSufSort[6] algorithm for this implementation. While DivSufSort operates with a time complexity of $O(n \log n)$, it balances efficiency and resource utilization well. It requires only two relatively small buffers (one size 256 and another size 65536), making it a more memory-efficient choice compared to other algorithms that offer linear time complexity but at the cost of higher memory usage.

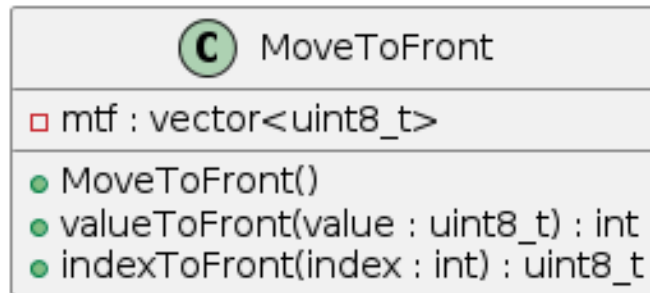
The detailed mechanics of building suffix arrays and their theoretical underpinnings are beyond the scope of this research.

3.4 Move-To-Front

3.4.1 Integration of MTF and RLE

The MTF stage is done by evaluating the bytes one by one. Thus, applying the RLE with no additional implementation effort is possible. The integration of RLE with MTF is particularly effective because the MTF transformation tends to produce many consecutive zeros when a frequently used symbol is encountered multiple times. Directly encoding these runs of zeros with RLE immediately after MTF can drastically reduce the size of the output, enhancing the efficiency of the subsequent Huffman coding stage.

3.4.2 MoveToFront Class




The **MoveToFront** class is designed to maintain and manipulate a list where frequently accessed elements are moved to the front, reducing their average access time.

- **mtf (std::vector<uint8_t>):** This vector holds the list of byte values in their current order, initialized to the natural order from 0 to 255.
- **valueToFront(uint8_t value):** Identifies a value's position in the list, moves it to the front and returns its prior position. This operation is critical for encoding symbols based on their frequency of use.

- **indexToFront(int index):** Moves the element at the specified index to the front of the list, facilitating dynamic reordering as data is processed.

3.4.3 MTFAndRLE2StageEncoder Class

 MTFAndRLE2StageEncoder
<ul style="list-style-type: none"> □ bwtBlock : vector<int>& □ bwtLength : int □ bwtValuesInUse : array<bool, 256>& □ mtfBlock : vector<uint16_t> □ mtfSymbolFrequencies : vector<int> □ alphabetSize : int □ mtfLength : int
<ul style="list-style-type: none"> ● MTFAndRLE2StageEncoder(bwtBlock : vector<int>&, bwtLength : int, bwtValuesInUse : array<bool, 256>&) ● encode() : void ● getMtfBlock() : vector<uint16_t>& ● getMtfLength() : int ● getMtfAlphabetSize() : int ● getMtfSymbolFrequencies() : vector<int>&

The `MTFAndRLE2StageEncoder` class combines MTF transformation with a secondary run-length encoding to prepare data for efficient Huffman coding.

- **bwtBlock (std::vector<int>&):** A reference to the block of data transformed by BWT.
- **bwtLength (int):** The length of the BWT block.
- **bwtValuesInUse (std::array<bool, 256>&):** Indicates which byte values are present in the BWT block.
- **mtfBlock (std::vector<uint16_t>):** Holds the MTF transformed data with additional run-length encoded values.
- **mtfSymbolFrequencies (std::vector<int>):** Tracks frequency of each symbol post-MTF and RLE, essential for optimal Huffman coding.
- **encode():** Performs the MTF transformation followed by run-length encoding. This method identifies runs of zeros (indicating repeated access of the first symbol in the MTF list) and encodes them using two special symbols, enhancing compression efficiency.

3.4.4 Details of the encode() Function

The `encode()` function within the `MTFAndRLE2StageEncoder` class performs both the MTF transformation and RLE. This function:


- Converts each byte from the BWT block into its MTF position.
- Checks for runs of zeros, which indicate repeated usage of the most recently accessed symbol.

- Encodes the length of these runs using a combination of two special symbols, `HUFFMAN_SYMBOL_RUNA` and `HUFFMAN_SYMBOL_RUNB`. These symbols help in encoding runs of zeros (consecutive accesses to the first item in the MTF list) as follows: `HUFFMAN_SYMBOL_RUNA` and `HUFFMAN_SYMBOL_RUNB` are alternately used to represent binary digits in a bijective base-2 number system, allowing for an efficient representation of run lengths. The length of a run is encoded by converting it into a binary number and then writing the binary digits using ‘RUNA’ for zeros and ‘RUNB’ for ones.
- Non-zero MTF positions are encoded directly, and the frequencies of each symbol, including the special RLE symbols, are recorded for use in Huffman coding.

3.5 Huffman Encoding and Decoding

The Bzip2 compression algorithm uses Huffman coding as a crucial part of its data compression strategy, necessitating efficient handling of Huffman trees. This involves classes designed for allocating Huffman code lengths and the actual encoding and decoding stages, implemented through the ‘`HuffmanAllocator`’, ‘`HuffmanStageEncoder`’, and ‘`HuffmanStageDecoder`’ classes.

3.5.1 HuffmanAllocator Class


 HuffmanAllocator
<ul style="list-style-type: none"> • <code>first(array : std::vector<int>, i : int, nodesToMove : int) : int</code> • <code>setExtendedParentPointers(array : std::vector<int>) : void</code> • <code>findNodesToRelocate(array : std::vector<int>, maxLength : int) : int</code> • <code>allocateNodeLengths(array : std::vector<int>) : void</code> • <code>allocateNodeLengthsWithRelocation(array : std::vector<int>, nodesToMove : int, insertDepth : int) : void</code> • <code>allocateHuffmanCodeLengths(array : std::vector<int>, maxLength : int) : void</code>

The `HuffmanAllocator` class is instrumental in calculating Huffman code lengths, which are essential for achieving optimal compression in Huffman coding. The class is designed with static methods to handle various aspects of Huffman tree manipulation and node length assignments.

- **`first(const std::vector<int>& array, int i, int nodesToMove)`:** This method performs a binary search to find the first node that can be relocated without violating the Huffman tree properties. It uses the ‘`nodesToMove`’ parameter to determine the minimum node index that can be considered for movement. The method helps balance the Huffman tree by ensuring nodes are placed in positions that help maintain a minimal maximum tree depth.
- **`setExtendedParentPointers(std::vector<int>& array)`:** Adjusts the parent pointers for each node in the Huffman tree. This method updates each node to point to its parent and combines the weights of child nodes, effectively building the tree from the leaves up to the root. This setup is essential for subsequent operations that depend on the tree structure, such as determining the maximum depth or optimal node placements.

- **findNodesToRelocate(const std::vector<int>& array, int maximumLength):** Identifies the deepest node that can be moved upwards in the tree without exceeding the specified maximum length. This method is used for optimizing the tree depth, ensuring that no Huffman code exceeds the maximum allowed length, thereby avoiding an increase in the average encoded symbol size.
- **allocateNodeLengths(std::vector<int>& array):** Assigns the optimal code lengths to each node in the Huffman tree based on their frequencies and positions in the tree. This method ensures that more frequent symbols have shorter codes, a fundamental property of efficient Huffman coding.
- **allocateNodeLengthsWithRelocation(std::vector<int>& array, int nodesToMove, int insertDepth):** Similar to **allocateNodeLengths**, additionally considering relocating specific nodes to better positions in the tree. This is used when standard node placement does not yield an optimal tree, typically when the tree needs more profound restructuring to fit within maximum code length constraints.
- **allocateHuffmanCodeLengths(std::vector<int>& array, int maximumLength):** Starts the entire process of setting up parent pointers, finding nodes to relocate, and allocating node lengths. It is the high-level entry point for transforming a frequency table into a set of Huffman code lengths.

3.5.2 HuffmanStageEncoder Class


 HuffmanStageEncoder
<div> <div> <div>□</div> <div>bitOutputStream : BitOutputStream&</div> </div> <div> <div>□</div> <div>mtfBlock : std::vector<uint16_t>&</div> </div> <div> <div>□</div> <div>mtfLength : int</div> </div> <div> <div>□</div> <div>mtfAlphabetSize : int</div> </div> <div> <div>□</div> <div>mtfSymbolFrequencies : std::vector<int>&</div> </div> <div> <div>□</div> <div>huffmanCodeLengths : std::vector<std::vector<int>></div> </div> <div> <div>□</div> <div>huffmanMergedCodeSymbols : std::vector<std::vector<int>></div> </div> <div> <div>□</div> <div>selectors : std::vector<uint8_t></div> </div> </div>
<div> <div>●</div> <div>HuffmanStageEncoder(&outputStream, &mtfBlock, mtfLength, mtfAlphabetSize, &mtfSymbolFrequencies)</div> </div> <div> <div>●</div> <div>encode() : void</div> </div> <div> <div>●</div> <div>generateHuffmanOptimisationSeeds() : void</div> </div> <div> <div>●</div> <div>optimiseSelectorsAndHuffmanTables(storeSelectors : bool) : void</div> </div> <div> <div>●</div> <div>assignHuffmanCodeSymbols() : void</div> </div> <div> <div>●</div> <div>writeSelectorsAndHuffmanTables() : void</div> </div> <div> <div>●</div> <div>writeBlockData() : void</div> </div>

The **HuffmanStageEncoder** class is designed to handle data encoding using Huffman codes efficiently. This class manages the complex interactions between different stages of Huffman encoding, from generating initial code length guesses to writing the final Huffman-encoded data.

- **Constructor(...):** Initializes the encoder with the necessary streams and data structures for MTF (Move-To-Front) processed data. It also calculates the number of Huffman tables based on the length of the MTF sequence, which helps adapt the Huffman encoding strategy to the data size.
- **encode():** Coordinates the entire process of encoding a data block using Huffman codes. This includes optimizing Huffman tables, assigning Huffman codes to symbols, and writing encoded data and necessary metadata.

- **generateHuffmanOptimisationSeeds():** Generates initial guesses for Huffman code lengths based on symbol frequencies. This method divides the symbol frequencies among different Huffman tables to balance the load and minimize the overall bit cost of encoding the data.
- **optimiseSelectorsAndHuffmanTables(bool storeSelectors):** Iteratively refines Huffman tables and selector choices to improve compression efficiency. Selectors determine which Huffman table is used for a group of symbols, thus affecting the compression ratio.
- **assignHuffmanCodeSymbols():** After optimizing the tables, this method assigns actual Huffman codes to the symbols based on the final table configurations. It ensures that symbols are encoded with the minimum possible lengths determined by their frequencies.
- **writeSelectorsAndHuffmanTables():** Writes the selectors and their corresponding Huffman tables to the output stream.
- **writeBlockData():** Outputs the actual Huffman-encoded data to the stream.

3.5.3 HuffmanStageDecoder Class

 HuffmanStageDecoder
<ul style="list-style-type: none"> □ bitInputStream : BitInputStream& □ selectors : std::vector<uint8_t> □ minimumLengths : std::vector<int> □ codeBases : std::vector<std::vector<int>>> □ codeLimits : std::vector<std::vector<int>>> □ codeSymbols : std::vector<std::vector<int>>> □ currentTable : int □ groupIndex : int □ groupPosition : int
<ul style="list-style-type: none"> ● HuffmanStageDecoder(&inputStream, alphabetSize, tableCodeLengths, selectors) ● nextSymbol() : int ● createHuffmanDecodingTables(alphabetSize, tableCodeLengths) : void

The **HuffmanStageDecoder** class is designed to decode data encoded using Huffman codes. It relies on a set of Huffman decoding tables that it constructs based on the encoded data's metadata, ensuring accurate reconstruction of the original data.


- **Constructor(...):** Initializes the decoder with the bit input stream and the selectors that determine which Huffman table to use for decoding each group of symbols. It also constructs the Huffman decoding tables based on the provided code lengths for each symbol in the Huffman tables.
- **nextSymbol():** Using the current Huffman table, this method fetches the next symbol from the input stream. It increments the position within the current group and switches to the following table when the current group ends, as the selectors dictate. The method reads bits from the stream and compares them against the limits of the current Huffman table to decode the symbol.
- **createHuffmanDecodingTables(alphabetSize, &tableCodeLengths):** Sets up the Huffman decoding tables. This method processes the lengths of Huffman

codes to create bases and limits for each table, facilitating quick symbol resolution during decoding.

- **Handling of Huffman Groups and Selectors:** The decoding process is structured around Huffman groups, with selectors indicating which Huffman table to use for each group. This allows the decoder to adaptively switch between different sets of Huffman codes, optimizing decoding based on the varying symbol frequency distributions within other parts of the data.

3.6 Block Compression and Decompression

3.6.1 BlockCompressor Class


 BlockCompressor
<ul style="list-style-type: none"> □ bitOutputStream : BitOutputStream □ crc : CRC32 □ blockLength : int □ blockLengthLimit : int □ blockValuesPresent : std::array<bool, 256> □ block : std::vector<uint8_t> □ bwtBlock : std::vector<int> □ rleCurrentValue : int □ rleLength : int
<ul style="list-style-type: none"> ● BlockCompressor(blockSize : int) ● isEmpty() : bool ● getCRC() : int ● getBitOutputStream() : BitOutputStream ● write(value : int) : bool ● write(data : std::vector<char>, offset : int, length : int) : int ● close() : void ● reset() : void

Bzip2 is a block-oriented algorithm, meaning the input stream gets divided into fixed-size blocks before the encoding steps are applied. Because the first RLE step can change the input size (as repeated characters are replaced by a single one followed by a counter), the block is processed only when the RLE output reaches the fixed length.

- **Constructor (int blockSize):** Sets up the compressor with a specified block size. This size dictates how much data is handled in one compression cycle, with an added allowance for BWT wraparound. The block length limit is set slightly lower than the block size to leave room for control structures in the compressed format.
- **isEmpty():** Checks whether the current compression block is empty, indicating that no data has been written or the last written data has been completely processed and reset.
- **getCRC():** Retrieves the current cyclic redundancy check (CRC) value, which is used to verify the integrity of the data once decompressed.

- **write(int value):** Writes a single byte into the compression block, managing it through an RLE process to minimize redundancy. If the current run length reaches its maximum or a different byte is encountered, the method triggers ‘written’ to finalize the current run and start a new one.
- **write(const std::vector<char> &data, int offset, int length):** Processes multiple bytes from an input vector, appending them to the current block while respecting the block size limits. It returns the number of bytes successfully written.
- **close():** Finalizes the compression of the current block. It ensures any remaining RLE data is written, performs the BWT to reorder data for better compressibility, applies MTF and RLE encoding, and passes the data through Huffman encoding. It writes necessary metadata to the bit output stream, like block headers and symbol tables.
- **reset():** Resets the compressor’s state to prepare for a new data block. This includes resetting the CRC, clearing the data blocks, and resetting all tracking variables used in RLE and other transformations.
- **writeSymbolMap():** Generates and writes a symbol presence map to the output stream, which the decoder uses to reconstruct the complete set of used byte values.
- **writeRun(int value, int runLength):** Encodes a sequence of identical bytes using RLE, adjusting the CRC accordingly. This method optimizes the data size by collapsing repeated values into shorter representations.

3.6.2 BlockDecompressor Class

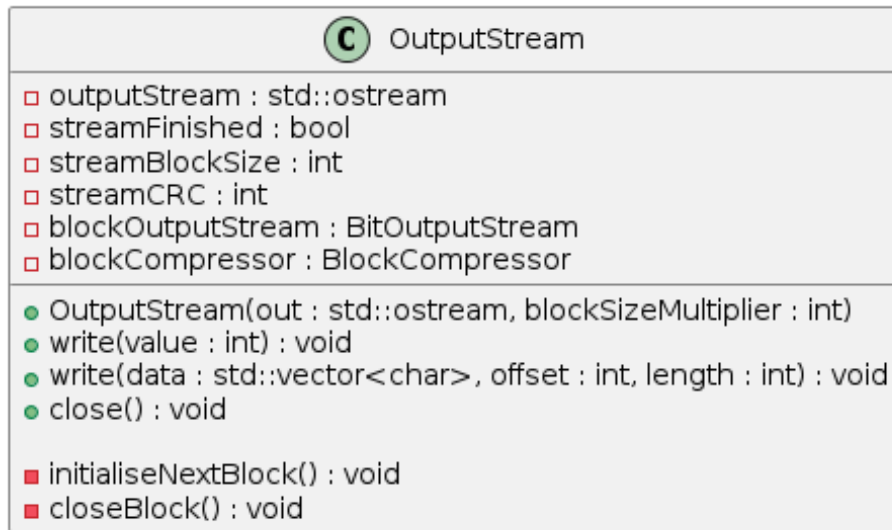
 BlockDecompressor
<ul style="list-style-type: none"> □ bitInputStream : BitInputStream □ crc : CRC32 □ blockCRC : int □ blockRandomised : bool □ huffmanSymbolMap : std::vector<uint8_t> □ bwtByteCounts : std::vector<int> □ bwtBlock : std::vector<uint8_t> □ bwtMergedPointers : std::vector<int> □ bwtCurrentMergedPointer : int □ bwtBlockLength : int □ bwtBytesDecoded : int □ rleLastDecodedByte : int □ rleAccumulator : int □ rleRepeat : int
<ul style="list-style-type: none"> ● BlockDecompressor(inputStream : BitInputStream, blockSize : int) ● read() : int ● read(destination : std::vector<uint8_t>, offset : int, length : int) : int ● checkCRC() : int ● initialiseInverseBWT(bwtStartPointer : int) : void ● decodeNextBWTByte() : int

The `BlockDecompressor` handles data reconstruction from its compressed form through several inverse stages of the compression process.

- **Constructor (`BitInputStream& inputStream, int blockSize`):** Initializes the decompressor with the input stream containing the compressed data and the size of blocks used during compression. It begins to set up by reading the block's header, including the CRC checksum and the BWT start pointer, and then preparing the Huffman decoding stage.
- **`read()`:** Retrieves the next decompressed byte from the block. This method handles the inverse of the Run-Length Encoding (RLE) applied during compression. It adjusts internal counters and states based on the decompressed data to maintain the correct output sequence. It also manages the cyclic redundancy check (CRC) for output data to ensure integrity.
- **`read(std::vector<uint8>& destination, int offset, int length)`:** Decodes a specified number of bytes into a given buffer. This function calls `read()` repeatedly to fill the buffer with decompressed data, handling EOF conditions gracefully.
- **`checkCRC()`:** Compares the calculated CRC of the decompressed data against the stored CRC value to verify the integrity of the decompressed block.
- **`readHuffmanTables()`:** Processes the Huffman table data from the input stream. This method reconstructs the Huffman tables used during compression, allowing the Huffman decoder to interpret the compressed data correctly.
- **`decodeHuffmanData(HuffmanStageDecoder& huffmanDecoder)`:** Decodes the Huffman encoded data using the prepared Huffman tables. This process reconstructs the original Move-to-Front (MTF) sequence and applies inverse transformations to the original Burrows-Wheeler Transform (BWT) sequence.
- **`initialiseInverseBWT(int bwtStartPointer)`:** Initializes the inverse BWT process using the start pointer read from the block header. This method sets up the structures necessary for the inverse transformation of the BWT, preparing for the final reconstruction of the original data.
- **`decodeNextBWTByte()`:** Retrieves the next byte from the inverse BWT sequence.

3.7 Input / Output Streams

3.7.1 OutputStream Class

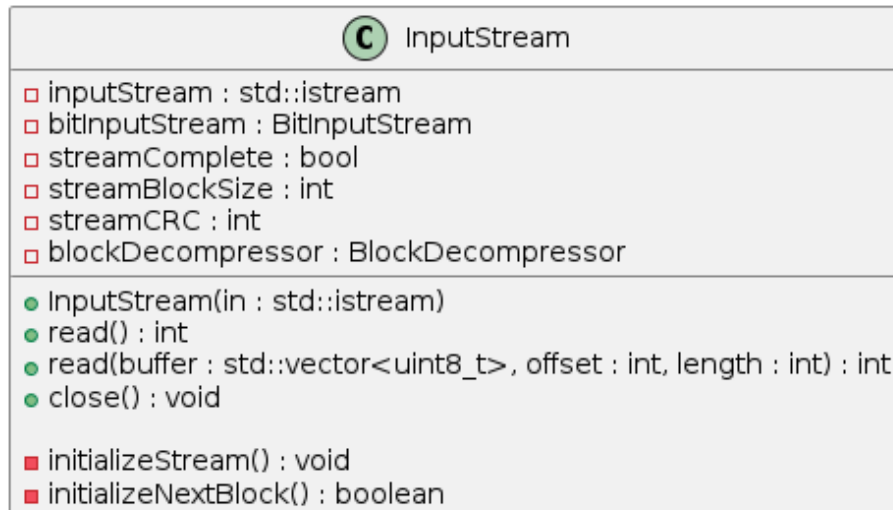


The `OutputStream` class manages the high-level operations of outputting compressed data. It integrates directly with ‘`std::ostream`’ to allow writing to various output streams, such as files.

- **Constructor (`std::ostream& out, int blockSizeMultiplier`):** Initializes the output stream with reference to a ‘`std::ostream`’ object for data output and a multiplier that determines the size of blocks to compress. This setup includes writing the stream start marker and setting the initial block size based on the multiplier, affecting the granularity and compression efficiency.
- **`write(int value)`:** Writes a single byte to the compressed stream. It passes this byte to the ‘`BlockCompressor`.’ If the compressor cannot accept more data (indicating the current block is full), it finalizes the current block, initializes a new block, and continues writing.
- **`write(const std::vector<char>& data, int offset, int length)`:** Writes a sequence of bytes from a buffer to the compressed stream. It handles block transitions seamlessly, ensuring that each block is filled and processed correctly before starting a new one.
- **`close()`:** Finalizes the compression process. This method ensures that any remaining data in the current block is compressed and written out, followed by writing the stream end markers. It then flushes the underlying bit stream to the ‘`std::ostream`,’ ensuring all data is written to the output medium.
- **`initialiseNextBlock()`:** Prepares a new block for compression. It creates a new instance of ‘`BlockCompressor`,’ linked to the current bit output stream, ready to receive data for compression.
- **`closeBlock()`:** Closes the current compression block. It instructs the ‘`BlockCompressor`’ to finalize its data, writes the compressed block to the output

stream, and updates the overall stream CRC based on the block's CRC. This CRC update is crucial for ensuring data integrity across multiple blocks.

3.7.2 InputStream Class



The `InputStream` class is designed to facilitate the reading and decompression of data that has been compressed using the Bzip2 algorithm. It interacts directly with an input stream and manages the decompression process block by block.

- **Constructor (std::istream& in):** Initializes the 'InputStream' with a 'std::istream' reference, setting up the 'BitInputStream' to handle the bit-level input from the stream. This setup prepares the class to start reading compressed data immediately.
- **read():** Attempts to read the next byte from the decompressed data. If no 'BlockDecompressor' is active, it initializes the stream and prepares the first block for decompression. If the current block ends, it tries to initialize the next block.
- **read(std::vector<uint8_t>& buffer, int offset, int length):** Reads a sequence of decompressed bytes directly into a buffer. This method utilizes 'read()' internally to fill the buffer, handle block transitions, and ensure that the requested number of bytes is read until the end of the compressed data is reached.
- **close():** Marks the stream as complete and cleans up resources. This method is crucial for ensuring that no further reading occurs once the end of the stream is reached, and it properly releases any resources tied to the decompression process.
- **initializeStream():** Reads the initial setup from the compressed stream, including the stream start marker and block size. This method validates the compressed data format, setting up the decompression process according to the specifications found in the header.
- **initializeNextBlock():** Prepares for decompression of the next block if available. It checks for block end markers or the stream end marker. If a new block

is found, it initializes a new ‘BlockDecompressor’ with the correct settings. The overall stream CRC is verified to ensure data integrity if the stream end is detected.

3.8 Parallelization and Usage of OpenCL

Parallelization in compression algorithms is often challenging due to the sequential nature of most data dependencies within the algorithms. Initially, the idea was to parallelize individual steps of the Bzip2 compression process, such as RLE, BWT, MTF, and Huffman coding. However, these steps depend on the sequential processing of input data, making effective parallelization difficult without significantly reengineering the algorithmic approach.

The breakthrough came with the realization that instead of trying to parallelize the internal steps of the compression algorithm, it would be more feasible to parallelize the processing at the block level. Given that each block of data is known and constant in size, each block could be prepared independently for BWT, MTF, and Huffman encoding. This approach allows multiple blocks to be processed in parallel, significantly improving the compression process’s throughput, especially for large data sets.

3.8.1 Implementation using OpenCL

OpenCL provides a framework to execute tasks across various computing devices like CPUs, GPUs, and other processors. This project uses OpenCL to manage and accelerate the computation of multiple compression blocks on GPUs. To facilitate the use of OpenCL, an API wrapper[7] is used that handles the creation of necessary computational resources such as device selection, memory management, and kernel execution. The block-level parallelization is managed as follows:

- **Device Initialization:** The most suitable device (GPU with the highest computing capability available on the system) is selected to perform the computations.
- **Memory Management:** Buffers for input blocks, output data, and intermediate results are allocated in the device memory.
- **Kernel Execution:** Kernel, a function executed on the OpenCL device, is written to perform BWT, MTF, and Huffman encoding on data blocks. The kernel functions are designed to handle one block independently, allowing for the simultaneous processing of multiple blocks.

The modified `BlockCompressor` and `OutputStream` classes were adapted to manage the parallel execution. The `OutputStream` class, in particular, is responsible for creating the memory structures that are passed to the kernel, writing the data to the kernel before running the encoding process, reading, and ensuring that the compressed output of multiple blocks is correctly sequenced. The `BlockCompressor` class is no longer responsible for starting the encoding sequence since this is done in the kernel.

Chapter 4

Results

This chapter evaluates the compression algorithms implemented in two distinct versions of the software: the initial version running on a single-core CPU and the GPU-accelerated version utilizing OpenCL, where the multithreaded approach is executed. The performance of these versions is compared in terms of compression rates and execution times against the metrics obtained from the original compression library using random sample data files of different big sizes.

4.1 Compression Rates

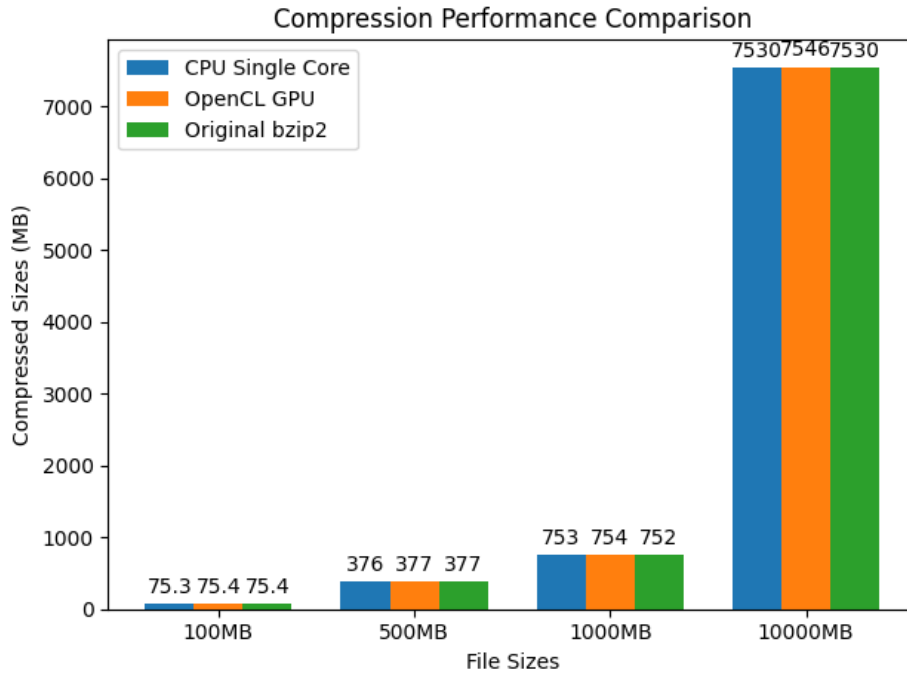


Figure 4.1: Comparison of compression performance between CPU, GPU, and original bzip2 versions

In the first analysis section, we investigate the compression rates achieved by both the single-core CPU and OpenCL GPU-accelerated versions of the software, compared

to the original bzip2 library. The graph in Figure 4.1 illustrates the compression performance across different file sizes.

Our analysis reveals that both the single-core CPU and GPU-accelerated implementations closely match the compression effectiveness of the original library, confirming the robustness of the adapted algorithms. However, the GPU version does show a slight increase in the resulting file sizes compared to the original bzip2 and CPU versions. This discrepancy is primarily due to a reduction in the default block size from 100000 to 10000 bytes. This adjustment was necessary to manage stack memory usage and thread spawning on the GPU, as larger block sizes were found to be impractical due to excessive memory usage.

4.2 Time Enhancement

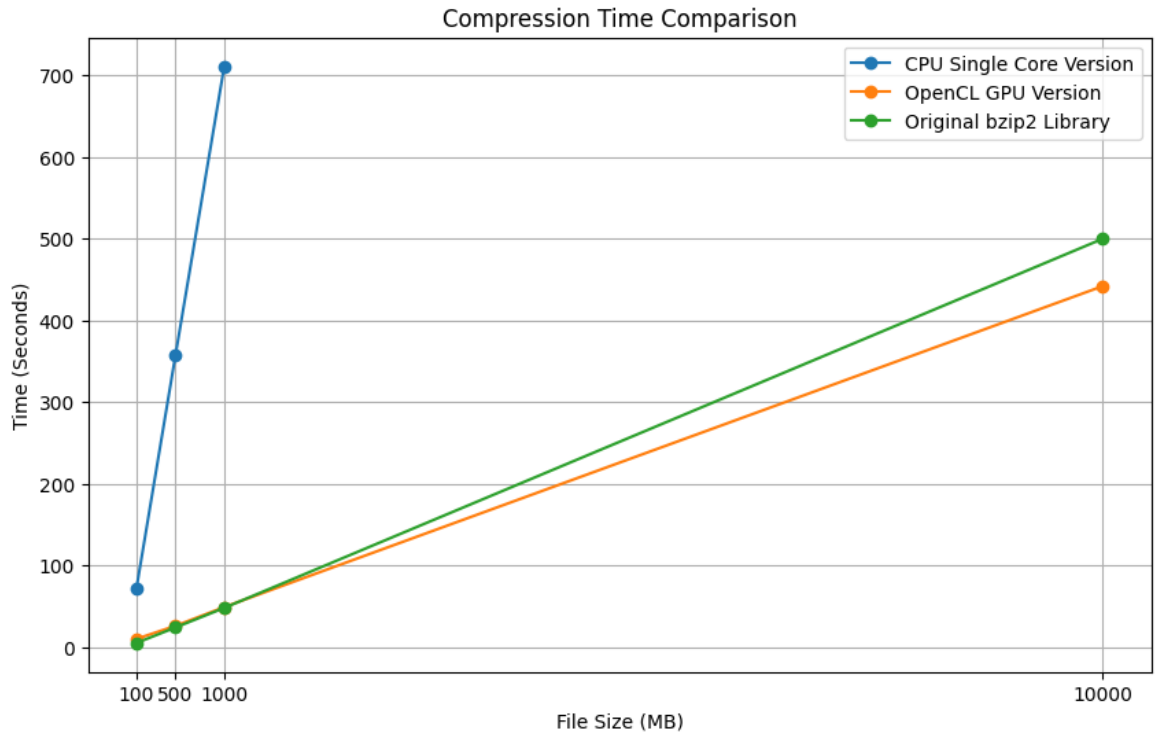


Figure 4.2: Time taken to compress files of different sizes by CPU, GPU, and original bzip2 versions

In this subsection, we analyze the time enhancement provided by the GPU-accelerated version of the software compared to both the single-core CPU version and the original bzip2 library. The line graph in Figure 4.2 demonstrates the time taken to compress files of varying sizes.

The measurements were conducted on a system running Windows 10, equipped with a Ryzen 7 7700X processor and a Radeon RX 6750XT GPU, which supports up to 2560 cores and delivers 12.861 TFLOPS. It is important to note that the CPU version, while significantly slower than the others, was not intended to compete directly with the highly optimized original bzip2 library. The primary goal was to assess potential speed improvements using GPU acceleration.

For the GPU-accelerated process, the parallel block count was set to 1024, a standard support level for modern GPUs, optimizing the use of available cores. The results indicate that the time growth rate is approximately 16 times slower than the CPU version. For larger data volumes, the GPU-accelerated version competes with and can surpass the original library's performance. However, for smaller files, the overhead associated with kernel initialization and data transfer to and from the GPU results in slower performance than the original bzip2.

Chapter 5

Conclusion

This project successfully implemented a GPU-accelerated version of a compression algorithm alongside a single-core CPU version, which was compared with the original bzip2 library. The results demonstrate that while the GPU-accelerated version can save significant time for larger files, the CPU version remains substantially slower.

5.1 Achievements

The primary achievement of this research was the development and benchmarking of two versions of a compression algorithm, highlighting the potential of GPU acceleration in data compression tasks. The ability to parallelize the compression process on a GPU significantly enhanced performance, especially for large data files.

5.2 Challenges Encountered

Several significant challenges were encountered during the project:

- **Compression Algorithm Complexity:** Understanding and implementing the intricate bit manipulation required by the compression algorithms was notably challenging.
- **OpenCL Implementation:** Learning and applying OpenCL was demanding, mainly due to the older libraries and the necessity to develop C kernel code, which required a deep understanding of both the hardware and software aspects of GPU programming.

5.3 Limitations

The project has a minor limitation: the GPU version required a reduced default block size to manage memory constraints effectively, potentially affecting performance for even larger files.

5.4 Future Work

For future improvements, several aspects call for further investigation:

- **CPU Version Performance:** Understanding why the CPU version underperforms compared to the original bzip2 library is crucial. This could involve deeper profiling to identify bottlenecks.
- **Multithreaded Approaches:** There is room to expand the multithreading approach, especially in further parallelizing the Huffman coding step. Since blocks are further divided into segments of 50 symbols, this represents a potential area for performance enhancement.
- **Other Encoding Steps:** Further research could also explore the parallelization of other encoding steps, potentially unlocking additional performance gains.

This research has laid a solid foundation for enhancing GPU acceleration data compression techniques. It also opens up several avenues for future exploration that could further bridge the gap between computational efficiency and compression effectiveness.

Bibliography

- [1] Bzip2 Manual, Julian Seward
- [2] Bzip2 Library, Julian Seward
- [3] Bzip2 Format Specification, Joe Tsai
- [4] A Block-sorting Lossless Data Compression Algorithm, M. Burrows and D.J. Wheeler, May 10, 1994
- [5] bzip2 Java implementation, Mateusz Bartosiewicz
- [6] libdivsufsort, Yuta Mori
- [7] OpenCL-Wrapper, Dr. Moritz Lehmann