A

Dissertation

On

Analysis of Parallel Lempel-Ziv Compression Using CUDA

Submitted in partial Fulfillment of the requirement

For the award of the Degree of

Master of Technology

In

Computer Science & Engineering

Submitted By

Milind Mathur

University Roll No. 2K11/CSE/06

Under the esteemed guidance of

Mr. Vinod Kumar

Associate Prof, Computer Engineering Department, DTU, Delhi

DELHI TECHNOLOGICAL UNIVERSITY

2011-2013

DELHI TECHNOLOGICAL UNIVERSITY

DELHI - 110042

CERTIFICATE

This is to certify that the dissertation titled "**Analysis of Parallel Lempel-Ziv Compression Using CUDA**" is a bonafide record of work done at **Delhi Technological University** by **Milind Mathur, Roll No. 2K11/CSE/06** for partial fulfilment of the requirements for degree of Master of Technology in Computer Science & Engineering. This project was carried out under my supervision and has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma to the best of our knowledge and belief.

Date: ____________

(Mr. Vinod Kumar)

Associate Professor & Project Guide

Department of Computer Engineering

Delhi Technological University

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to all the people who have supported and encouraged me during the course of this project without which, this work could not have been accomplished.

First of all, I am very grateful to my project supervisor Mr. Vinod Kumar, Associate Professor, for providing the opportunity of carrying out this project under his guidance. I am deeply indebted to him for the support, advice and encouragement he provided without which the project could not have been a success. I am grateful for the knowledgeable insights of our Head of the Department, Dr. Daya Gupta in contributing to the success of this project.

I am also thankful to my Mom and Dad for being there for me at all times. I would also like to thank my mentor and elder brother Rachit for inspiring me to go through with the project. I am grateful for the support of my closest friends Puneet, Sourabh and Shalini for helping me think in the right direction. Last but not the least; I am grateful to Delhi Technological University for providing the right resources and environment for this work to be carried out.

Milind Mathur
University Roll no: 2K11/CSE/06
M.Tech (Computer Science & Engineering)
Department of Computer Engineering
Delhi Technological University
Delhi – 110042

ABSTRACT

Data compression is a topic that has been researched upon for years and we have standard formats like zip, rar, gzip, bz2 in generic data; jpeg, gif in images; . In this age where we have lots of data with internet being ubiquitous, there is a strong need for fast and efficient data compression algorithm. Lempel-Ziv family of compression algorithms form the basis for a lot of commonly used formats. Some modified form of LZ77 algorithm is still used widely as a lossless run length encoding algorithm.

Recently Graphics Processing Units (GPUs) are making headway into the scientific computing world. They are enticing to many because of the sheer promise of the hardware performance and energy efficiency. More often than not these graphic cards with immense processing power are just sitting idle as we do our tasks and are not gaming. GPUs were mainly used for graphic rendering but now they are being used for computing and follow massively parallel architecture. In this dissertation, we talk about hashing algorithm used in LZSS compression. We compare the use of DJB hash and Murmur Hash in LZSS compression. We compare it to the more superior LZ4 algorithm. We also look at massively parallel, CUDA enabled version of these algorithms and the speedup we can achieve with those at our disposal.

We conclude that for very small file (of order of KBs) we should use the LZ4 algorithm. If we don't have a CUDA capable device LZ4 is our best bet. But CUDA enabled versions of these algorithms outperform all the other algorithms easily and a speedup up to 10x is possible with GPU only of 500 series and even better with the newer GPUs.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Most modern systems are equipped with what many call a graphics card. The power of a Graphics Processing Unit (GPU) is not really unleashed until most are playing demanding graphic games. Technologies like the NVIDIA OPTIMUS [16] also put the graphics card to sleep when most of us are not playing games as it is not required then.

With the advent of Common Unified Device Architecture (CUDA) by NVIDIA, it has attracted the attention of many computational scientists. More and more people are looking to exploit the latent power residing under their computer's hood. NVIDIA's CUDA is leading the way into general purpose GPU computing. But its adoption is hampered by the effort required for rewriting optimized code for CUDA. Plus the tools that are needed like debuggers, memory leak checkers are now maturing to a level that they can be used for enterprise software development.

1.1 Problem Description

Data Compression algorithms are ubiquitous. They operate on firmware, BIOS, chips and mainframe computer systems. In today's world where everybody has an internet footprint and performs activities on the internet, data explosion is taking place. The web is a great example of it. Imagine the data that most search engines have to crawl and store. Imagine your email service like Gmail. How many emails have you received till date? And what is total length of data in your single account? Multiply that by the number of email accounts present on Gmail. That would probably be a very large number of Terabytes. A lot of storage space can be saved by applying a data compression algorithm.

We cannot use any data compression algorithm. We want the algorithm to be efficient and fast. Efficient in the sense that it must have a good compression ratio and be fast as the influx of data on servers is very large.

1.2 Related Topics

The Lempel Ziv family of algorithms has been around since 1970s have been the basis of a lot of modern algorithms. These are used the most common formats of data compression we use today for all sorts of files (data, image, video etc.). The concept of massively parallel processing via the GPU and the SIMD like architecture of the NVIDIA GPUs, they are perfect tools to get the job of data compression done and freeing up valuable clocks on the CPU for other processing.

1.3 Proposed Work

Lempel Ziv family of algorithms is the building block for many modern algorithms out there today. They have all the desired features we asked for. In this dissertation we first discuss the LZ77, LZSS algorithms and see its performance by using murmur hash in it. We will then have a look at more modern LZ4 algorithm that is again a LZ77 derivate. We then analyze the performance improvement by using the massively parallel (CUDA enabled) version of these algorithms: CuLZSS and CuLZ4.

2 RUN LENGTH ENCODING

Run length encoding is representing data in terms of runs of data. A run of data is simply a sequence of contiguous bytes. For instance given the string:

```
aaaabbbbccdeeeee
```

Can be encoded most simply in the format:

```
<literal><literal_length>
```

And the resulting encoded string we would have is:

```
a4b4c2d1e5
```

The above means that ‘a’ is the first literal that is repeated 4 times. Followed by ‘b’ which is also repeated 4 times, ‘c’ is repeated 2 times and so on. It is interesting to note that the single occurrence of ‘d’ which would occupy 1 byte uncompressed is occupying 2 bytes in this arrangement.

3 DICTIONARY ENCODING

This type of encoding is a type of substitution encoding and is also a form of lossless compression. Some set of strings are stored in a special data structure or 'dictionary'. Occurrences of those are replaced by an index into the dictionary.

There can be two types of dictionary encoding.

a) Static :

In this kind of dictionary encoding, the dictionary remains static and does not evolve or change during the entire process of encoding. This kind of encoding can be found where we want to represent a single kind of document and can have a fixed dictionary containing the words that are to be represented in it.

b) Dynamic:

In this kind of dictionary encoding, the dictionary is dynamic and is ever changing with new entries being created or deleted as more and more input text is passed to it. This is the 'Huffman' style coding where the tree is constructed as per the input text.

4 LEMPEL ZIV ALGORITHMS

Lempel Ziv algorithms are lossless data compression algorithms that find wide use in today's computing needs. In this family of algorithms LZ77 and LZ88 were published in papers by Abraham Lempel and Jacob Ziv in 1977 and 1978. Since then a lot of work has been done on them to improve on them and create algorithms like LZSS, LZW and LZMA etc. These form the core of modern compressors and decompressors.

These algorithms are dictionary based encoding algorithms. They rely highly on string factorization [5]. In other words strings are represented as compact form of other strings at some other points.

4.1 LZ77

Proposed in 1977 by Abraham Lempel and Jacob Ziv, this is a widely popular compression algorithm. It follows the 'sliding window' algorithm where a window slides over the input from the start to the end [21].

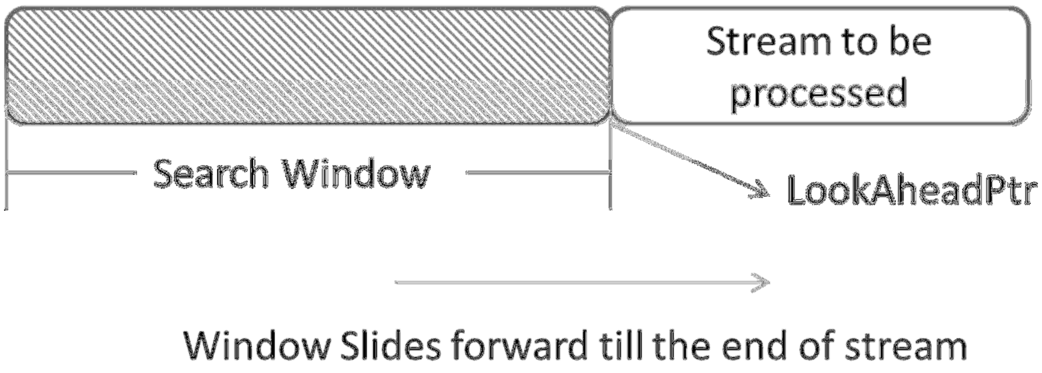


Figure 1: Sliding Window Encoding

The first few characters are pushed into what is called a search window. This search window has a size limited by the size of offset field. This search window expands to the maximum size and then slides along the input text. The look-ahead pointer marks the end of search window and the window slides on till the look-ahead pointer reaches the end of input stream. All bytes are to be represented as a triple:

```
<Offset><match_length><next_character>
```

The whole concept is finding a match of the string starting at look-ahead pointer in the previously visited / encoded stream of characters [8]. The number of matched characters forms the match length and the distance from the look-ahead pointer to the start of match in search window is called the offset. The offset is thus measured from the end of search window to the start of match length. Once a match is found, the search window moves past the entire match length. The figure below shows how the movement of the window takes place.

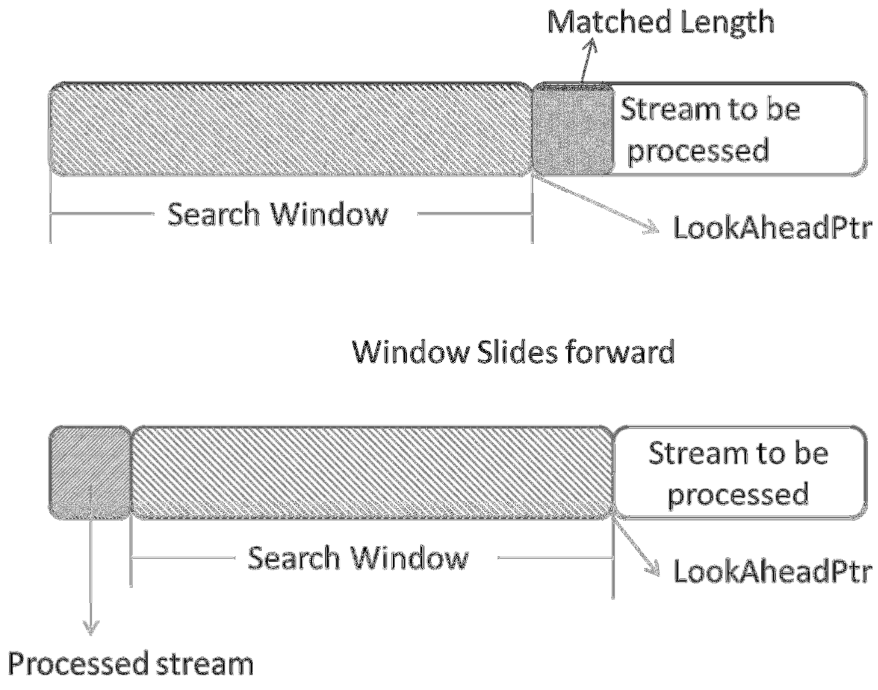


Figure 2: Movement of window

In LZ77 all entries are represented in the form of a triplet. Each triplet has 3 fields: the length, the distance and the next symbol that follows the match. The numbers shown in the figure are in bit width.

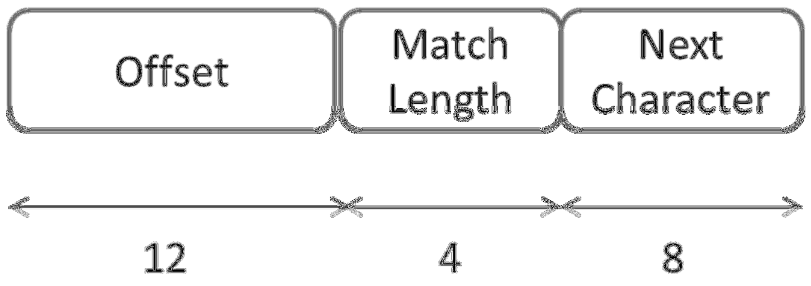


Figure 3: LZ77 Format Specification

It is clear from the above that the max values that we can have are:

Field	Max Value
Offset / Search Window Size	12 bits = 4 KB
Match Length	4 bits = 15 characters
Next Character	8 bits = 1B

Table 1: LZ77 Fields

The above translated in simple English is that the maximum amount of data an entry can hold is of length 15 which may start at an offset that may be as far back as 4 KB from the start of end of the search window.

The interesting thing to note here is that all non-matched characters will have offset and length zero and next character representing the desired character. This is in effect wasting a lot of space. The single character occupying 1B is occupying 3B. This is exactly the problem addressed in LZSS.

4.2 LZ78

Proposed in 1977 by Abraham Lempel and Jacob Ziv, it uses a more dynamic dictionary. It starts with a dictionary of all characters used in the input text. The indexes are numbered such as to leave space for addition of entries at a later stage of the process. As unseen patterns of text are visited they are added to the dictionary and given an index for use at a later stage. Next time the same portion of text is seen, only its index is output. There is also a threshold on how many entries the dictionary can hold at time. If that limit is not set, the size of dictionary may explode very fast.

4.3 LZW

Lempel-Ziv-Welch algorithm is an improved version of LZ78 published by Lempel, Ziv and Terry Welch [14]. It believes in growing the input symbols in dictionary. If a match of a string S is found then the entry for that is written to output and the entry is removed and the new S followed by the next symbol in input. It is most suited for image files and is a standard for GIF files and is implemented in UNIX's compress utility.

4.4 LZMA

Lempel Ziv Markov Chain Algorithm is again a lossless data compression algorithm. Unlike the others we discussed this is an entropy encoding algorithm [19]. It uses a scheme similar to LZ77, but uses a variable dictionary size. The LZ77 like output is encoded by a range encoder and uses sophisticated models and makes probabilistic prediction of each bit. The .7z format is based upon the LZMA algorithm.

5 RELATED ALGORITHMS

5.1 DEFLATE / ZIP

It is perhaps the most widely used algorithm. It is used in ZIP file formats, PNG image files and GZIP data compression. PKWARE owns the patent to the algorithm. However there are patent free implementations of the same. It is basically encoding with LZ77 followed by Huffman Coding. It has 2 main motives: Duplicate string reduction as we have seen with LZ77 and bit reduction with Huffman codes [2].

5.2 TAR

TAR is a file format used widely in the UNIX world. Most of the algorithms discussed above address the problem of compressing a single file. But what if we have a complete folder or a group of files? That's where TAR comes into scene. TAR is a store format that has specification for entries of multiple files and their metadata (relative paths). It provides no compression.

5.3 GZIP

It is often used in conjunction with tar and thus the famous format tar.gz is used. It is a compression format which uses LZSS followed by Huffman encoding. The interesting thing to note here is that the Huffman tree is not included with the compressed file as it can be reconstructed by the decoder as it has been specified to be right heavy [3]. If there is only a single child to any node it is the right child. It is very useful and a lot of UNIX packages are compressed using this method. UNIX utilities built around this algorithm are gzip and gunzip that is self-explanatory.

6 LZSS

Lempel Ziv Storer Szymanski is a derivative of LZ77 published in 1982 by James Storer and Thomas Szymanski. It addresses a major problem of LZ77 that it forced everything to be encoded as triplets [4]. This meant that a mismatch (a string not occurring in search window), will occupy 3 bytes instead of 1. So they came up with a new format. They started outputting an encoded/decoded bit before each entry. If we have the encoded bit, then what follows is a pair of offset and match length. If we have the decoded bit, it means the next byte is the un-encoded character as is. They also eliminate need for the third entry of the next character from the triple. Thus the entry now is only a pair and the start bit, which indicates the type of entry that follows.

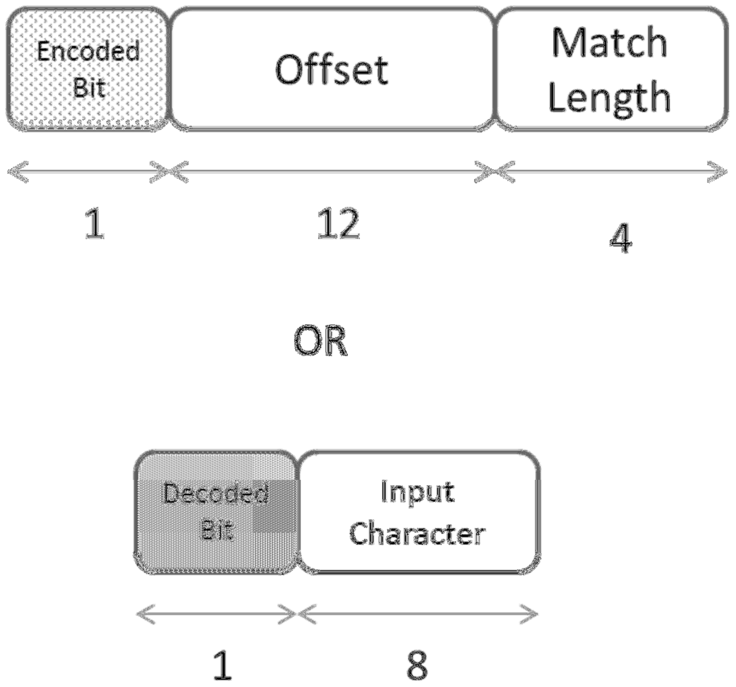


Figure 4: LZSS Format Specification

The above figure shows the two types of entries possible. The field widths mentioned are in bits. The following table analyzes the limits of having such an entry.

Field	Max Value
Offset / Search Window Size	12 bits = 4 KB
Match Length	4 bits = 18 characters

Table 2: Limits in LZSS

The table is similar to LZ77. But the most important thing to note is representing Match length of 18 in 4 bits. The whole concept is that since we have an encoded/decoded bit, we can represent match length as 3 + the value in match. Since if we are outputting this record we cannot have a match less than 3 (min match length). LZSS is widely popular and forms the basis of many algorithms.

The pseudo code of the algorithm follows:

```
LZSS(input,len)
    rindex := 0
    hash first 3 elements
    write 3 un-encoded elements to output
    lookAheadPtr := 3
    while rindex < len
        look for a match of sequence starting at lookAheadPtr in
        the hash
        look at match lengths for sequences for the next 8
        characters and make the best choice
        if found in hash
            output encoded bit and output offset and match
            length
        else
            output the un-encoded character
```

```
// slide window
for i:=0 to match_length
      if search window is full
            remove the entry from hash for lookAheadPtr –
            Max_Offset from the window
      add entry for sequence starting at lookAheadPtr + i
      in hash
lookAheadPtr += match_length
rindex += match_length
```

It is clear from the above algorithm that the most challenging tasks for us in the algorithm are:

1. Finding pattern match starting at look-ahead pointer in search window
2. Hashing the sequences

There is a lot of work done in how to improve speeds for pattern matching in LZ algorithms [7]. We can use any of those techniques like Hashing, Knuth-Morris-Pratt [11] Matching, Binary Search Trees, Tries and Suffix Trees etc. The standard implementation in GZIP and ZLIB use the standard DJB2 hash algorithm.

6.2 DJB2 Hash Algorithm

It was proposed by Daniel Julius Bernstein a professor in University of Illinois. It is still a very widely used algorithm mainly because of its high speed, simplicity and good enough distribution. The DJB2 hash algorithm's XOR version is described below:

```
DJB2_XOR_32(key,len,seed)
```

```
hash := seed
for i:=1 to len
      hash = (hash * MUL) ^ key[i]
return hash
```

Good and known value for MUL is 33 and seed is 5381. These values are also called magic values [6].

7 LZSS-MURMUR

What we propose here is to use the more modern, fast and well distributed Murmur hash to be used. It was proposed by Austin Appleby in 2008 [13]. Major advantage of murmur hash is its distribution. For even small deviation of input, we get hashes that are way apart. This proves very useful to us as this means there are lesser collisions and thus it improves our searching for pattern time.

7.1 Murmur Hash Algorithm

It has gone through a few versions and the latest one as of 2013, Murmur3 is described below:

```
Murmur3_32(key, len, seed)
// Note: In this version, all integer arithmetic is performed with
// unsigned 32 bit integers.
// In the case of overflow, the result is constrained by the
// application of modulo arithmetic.

    c1 := 0xcc9e2d51
    c2 := 0x1b873593
    r1 := 15
    r2 := 13
    m  := 5
    n  := 0xe6546b64

    hash := seed

    for each fourByteChunk of key
        k := fourByteChunk

        k :=  k * c1
        k := (k << r1) OR (k >> (32-r1))
        k := k * c2

        hash := hash XOR k
        hash := (hash << r2) OR (hash >> (32-r2))
        hash := hash * m + n

    with any remainingBytesInKey
        remainingBytes := SwapEndianOrderOf(remainingBytesInKey)
// Note: Endian swapping is only necessary on big-endian machines.
// The purpose is to place the meaningful digits towards the
// low end of the value, so that these digits have the greatest
// potential to affect the low range digits in the subsequent
// multiplication.  Consider that locating the meaningful digits in
// the high range would produce a greater effect upon the high digits
// of the multiplication, and notably, that such high digits are
// likely to be discarded by modulo arithmetic under overflow.
// We don't want that.

        remainingBytes := remainingBytes * c1
        remainingBytes :=
               (remainingBytes << r1) OR (remainingBytes >> (32 - r1))
        remainingBytes := remainingBytes * c2

        hash := hash XOR remainingBytes
    end for

    hash := hash XOR len

    hash := hash XOR (hash >> 16)
    hash := hash * 0x85ebca6b
    hash := hash XOR (hash >> 13)
    hash := hash * 0xc2b2ae35
    hash := hash XOR (hash >> 16)
    return hash
```

8 LZ4

The modern LZ4 algorithm is a newer algorithm proposed in 2012 that completely rewrites the Lempel Ziv format. It has gained a lot of popularity as of late and is used as back end Apache Hadoop, Rare logic: Real time data analysis, Apache Lucene search engine, GRUB boot loader, Enlightenment Desktop Environment, ZFS file system and FreeBSD etc.

It is based on LZ77 and follows the sliding window algorithm. The figure below describes the LZ4 format specification.

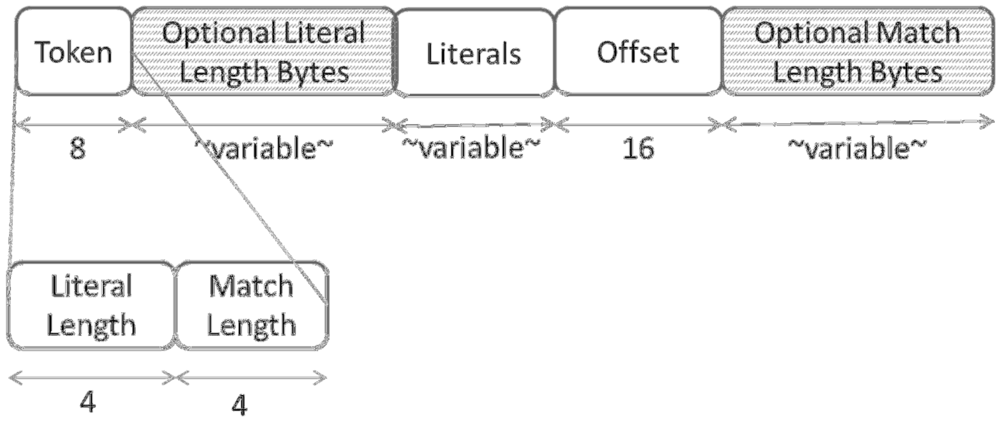


Figure 5: LZ4 Format Specification

The format has the following fields:

1. Token: width 8 bits

   It is split into two 4 bit fields. The higher 4 bits represent the literal length. Lower 4 bits represent the match length.

2. Optional Literal Length Bytes: width variable

   If literal length nibble in token is 15 or 0xF (all 1s) then one byte entry of this will necessarily exist. If any of them is all 1s or 0xFF or 255 then another byte is used.

Total literal length is calculated as sum of all these entries till one entry where there are not all 1s. Thus size of this field is variable.

3. Literals: width defined by literal length

   These include the characters that don't have a match in the search window. The only thing to note is that these literals are contiguous.

4. Offset: width 2 bytes or 16 bits

   This determines how far back the offset can be from the look-ahead pointer. Since this a 2 byte field so the max window size is 4KB.

5. Optional Match Length Bytes: width variable

   If match length nibble in token is 15 or 0xF (all 1s) then one byte entry of this will necessarily exist. If any of them is all 1s or 0xFF or 255 then another byte is used. Total literal length is calculated as sum of all these entries till one entry where there are not all 1s. Thus size of this field is also variable.

So let's look at the max entries table:

Field	Max Value
Offset / Window Size	16 bits = 4 KB
Match Length	4 bits + 8 * x bits = variable
Literal Length	4 bits + 8 * x bits = variable

Table 3: Limits in LZ4

The number of literals is defined by the literal length entry present before it. So, there can be three types of LZ4 entries.

a) Those that have only match length and no literals. Thus size of this entry is 1B token + 2B offset + variable match length.

b) Those that have only literal lengths. The size of this entry is 1B token + variable Literal length + 2B offset (all 0s)

c) Those that have both a literal length and match length. It means first expand the literals to output and then use the match length then. The size of this entry is largest: 1B token + variable Literal length + 2B offset + variable match length.

Finally we can now describe the LZ4 algorithm,

```
LZ4(input,len)
    rindex := 0
    hash first 13 elements
    write type b) entry
    lookAheadPtr := 13
    while rindex < len
        look for a match of sequence starting at lookAheadPtr in
        the hash
        if a match of at least 4 found in hash
            output relvant type entry a) or c)
        else
            add to current entries literal list
        // slide window
        for i:=0 to match_length
            if search window is full
                remove the entry from hash for lookAheadPtr –
                Max_Offset from the window
            add entry for sequence starting at lookAheadPtr + i
            in hash
        lookAheadPtr += match_length
        rindex += match_length
```

9 CUDA - COMPUTE UNIFIED DEVICE ARCHITECTURE

9.1 Overview

CUDA (aka Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce [1]. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest Nvidia GPUs become accessible for computation like CPUs. Unlike CPUs, however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general-purpose (i.e., not exclusively graphics) problems on GPUs is known as GPGPU.

CUDA has several advantages over traditional general-purpose computation on GPUs (GPGPU) using graphics APIs [9]:

- Scattered reads – code can read from arbitrary addresses in memory
- Shared memory – CUDA exposes a fast shared memory region (up to 48KB per Multi-Processor) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Faster downloads and readbacks to and from the GPU
- Full support for integer and bitwise operations, including integer texture lookups

Limitations of CUDA are:

- Copying between host and device memory may incur a performance hit due to system bus bandwidth and latency (this can be partly alleviated with asynchronous memory transfers, handled by the GPU's DMA engine)

- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not affect performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent tasks.

9.2 CUDA processing flow

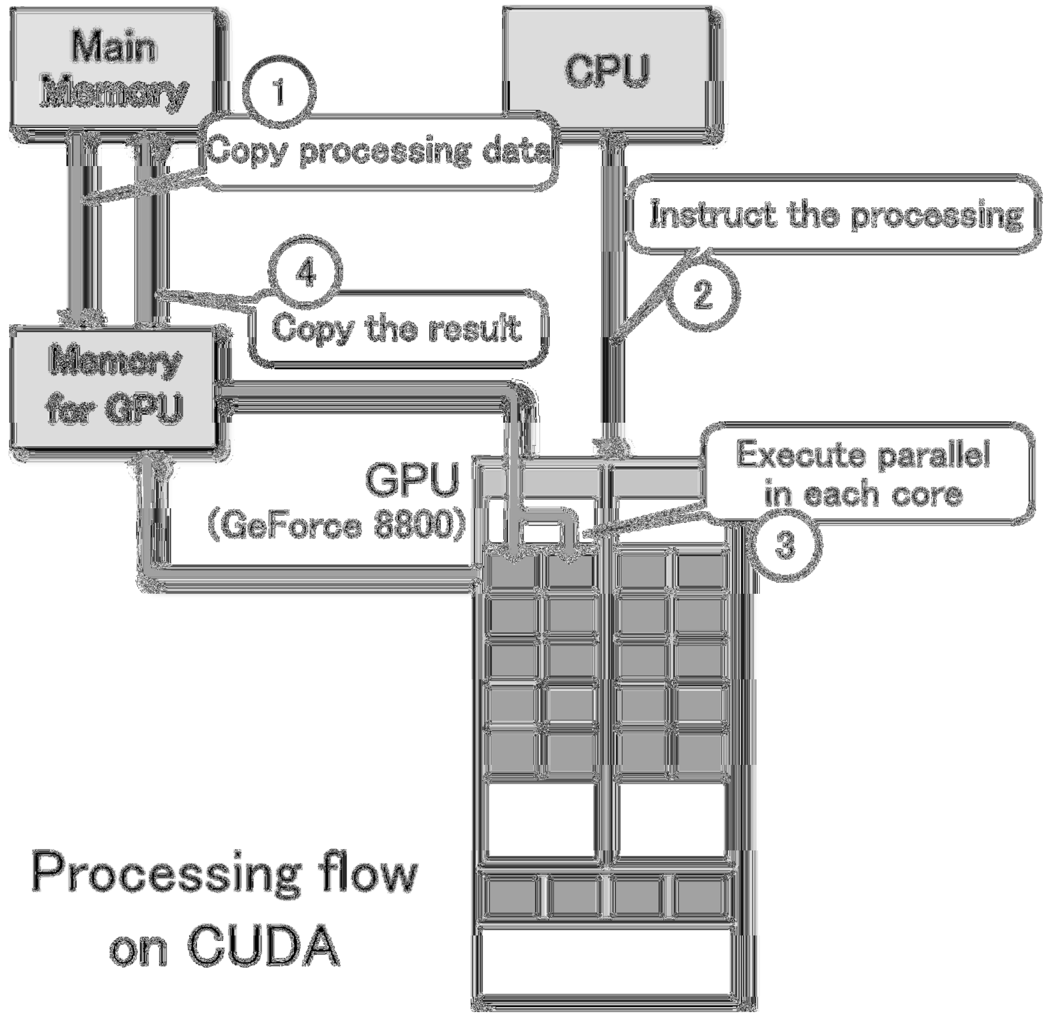


Figure 6: CUDA processing flow [10]

The diagram above shows the flow of a typical CUDA program. The flow is four fold:

1. Copy the data to be processed to GPU DRAM.

2. Instruct the processing and launch a CUDA kernel to do something with the data you copied earlier.
3. Executes the CUDA kernel in parallel using the GPU's SIMD model. Multiple kernels can also be scheduled in parallel to increase the utilization on the GPU.
4. Copy the result back to the system RAM.

9.3 Architecture

We know that in CUDA the threads run in parallel on each core. This means that all branching synchronization instructions are very crucial. We know that there are CUDA Blocks, each of which contains a group of CUDA threads. Certain numbers of CUDA threads are also clubbed to form what is called a Warp [20]. There can be no diversion in a Warp. A Warp goes for data parallelism. If there is branching or threads in a Warp are to diverge then the total Warp is serialized. This is the biggest challenge in CUDA [17]. __syncthreads() function is used to synchronize the Warp at a point and do the remaining tasks in parallel(SIMD) from that point on.

Also in CUDA we have few different types of memory and their access times are thus ordered as per their location. There are temp caches and registers of each CUDA thread which are accessed the fastest. Next we have the shared memory [18]. Each kernel can define a certain amount of shared memory which then can be accessed very quickly. Then data from the DRAM on GPU is slower than the two listed above. Lastly, if you need data from the CPU RAM, it is the slowest as it has to be copied to GPU DRAM and then used from the DRAM.

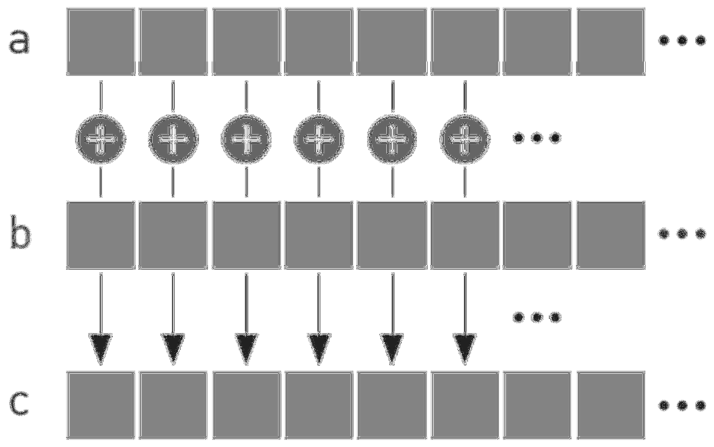


Figure 7: Vector Addition CUDA

The standard vector addition is the simplest way to explain SIMD CUDA architecture. Here

C[i] = A[i] + B[i]

Therefore there is no issue in synchronizing the Warp for such a kernel and a lot of speedup can be achieved with this kind of tasks. But, unfortunately, most of our daily tasks involve a higher level of divergence and CUDA cannot be used in a straight forward manner.

10 CULZSS AND CULZ4

These are the CUDA enabled versions of serial algorithms. CuLZSS is the massively parallel LZSS Murmur while CuLZ4 is the massively parallel LZ4. What having the GPU at our disposal does is that we can use memory hungrily in the DRAM of the GPU without worrying about running out of memory in RAM or affecting the performance of other running programs. What we want to do here is to divide the file into chunks and process them together in parallel [12]. This is similar to the behavior of a download manager. It would cause a slight loss in the compression ratio with the loss of 4 KB window that could have been continuous, but now would belong to a different thread in a different thread. Thus, the chance of a match in the starting portion of a file is reduced. Such losses can be neglected when looking at large files and compressed file sizes.

10.1 Algorithm

The following describes in very high level pseudo code, the working of the algorithm for CuLZSS / CuLZ4.

```
CuLZ(input,length)
    Divide the input into chunks
    For each chunk that will be processed in parallel
        Copy chunk to GPU
        Hash triples via massively parallel CUDA kernel
        Search for maximum match in a new CUDA kernel for each thread
        based on thread ID
        Store entries in GPU

        If chunk done

              Move output entries to RAM

              Write the entries to file

        End if

End for
```

11 EXPERIMENTAL SETUP

Here we look to compare using the DJB2 hash and the Murmur hash in LZSS. Both of these are just serial implementations of LZSS with one using DJB2 hash and other Murmur hash. We will use the following standard files to clock our performance.

File Type	**Size (MB)**
BibTex	0.009773
Text (Homer's Illiad)	0.804038
Image (Bitmap)	4.1
C Source Code	21.9
Dictionary	27.6
Linux Kernel Tarball	178.7

Table 4: Sample File Sizes

11.1 Performance metrics

We will measure the performance on the basis of following criteria:

11.1.1 Time taken

This represents the time taken to compress the chosen sample files using a particular algorithm.

11.1.2 Throughput

This represents the rate at which the compression took place. It is the total time it took to read the file to memory and back to disk as compressed file. It will be measured in Kilo Bits per Second (Kbps).

11.1.3 Compression Ratio

It gives a good idea of the amount of compression we have achieved.

Compression Ratio = Size Original / Size Compressed

11.1.4 Speedup

This represents the relative improvement we see in the throughput of both algorithms.

Speedup = Time taken in LZSS DJB2 /

Time taken in target algorithm

11.1.5 Collisions in Hash

This represents the number of collisions in the respective hashes, taking place for each file. They will give us a clue to which hash function is better distributed.

11.2 Hardware Used

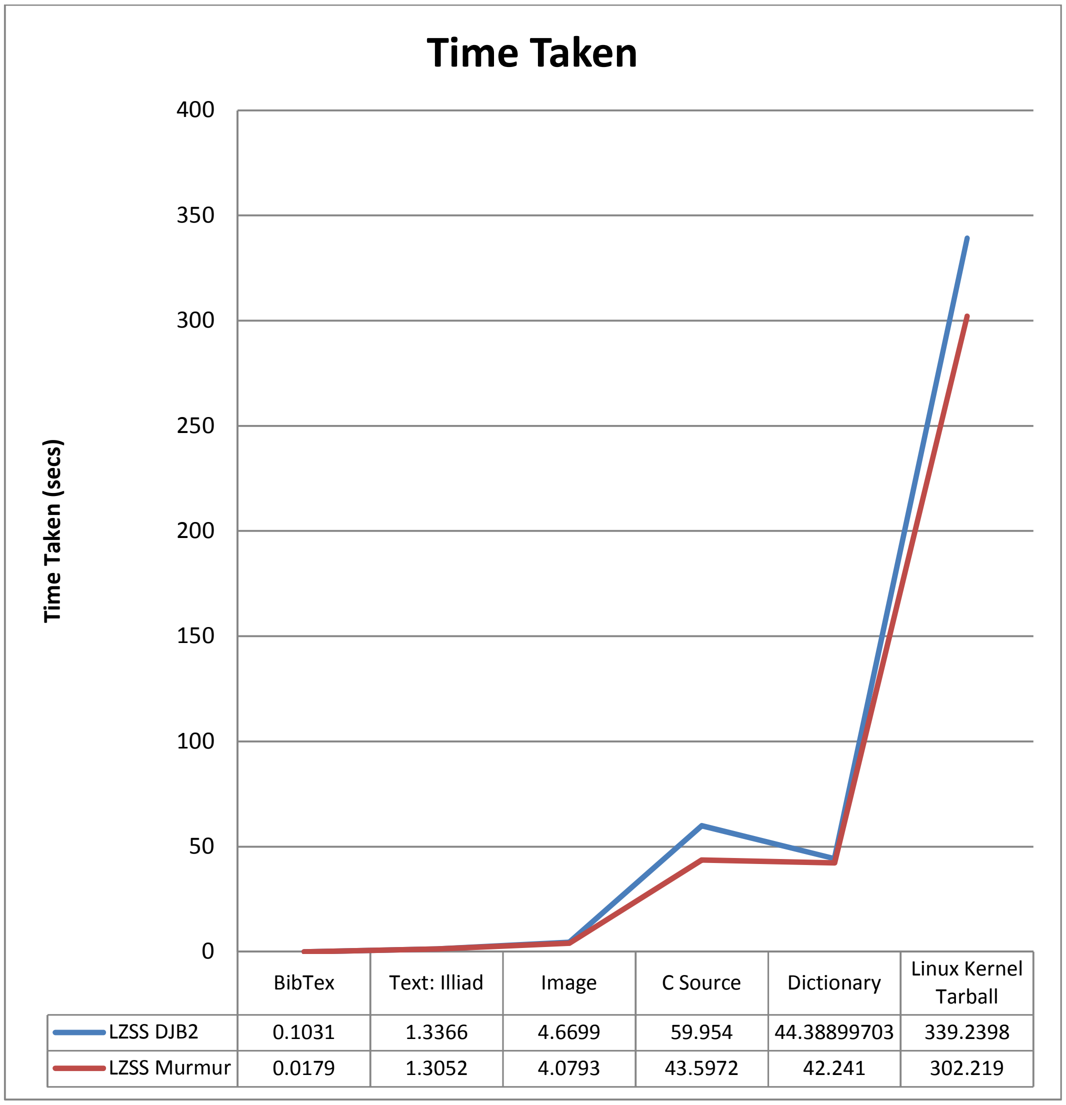
The machine from which all the result data is collected is equipped with 8GB DDR3 RAM, which is not used much as most algorithms here use a small memory footprint. It has the Intel I7 2630-QM processor with a maximum clock speed of 2.90 GHz which has 4 cores and can process 8 threads at a time. The GPU used to test our algorithms is a NVIDIA

525M GPU with 1GB DRAM and 96 cores. None of the devices used are overclocked. The CUDA toolkit version 5 is used with Arch Linux kernel 3.10. The proprietary version of the NVIDIA driver is used in conjunction with the toolkit.

12 RESULTS

12.1 Serial LZSS v/s Serial LZSS-Murmur

12.1.1 Time Taken

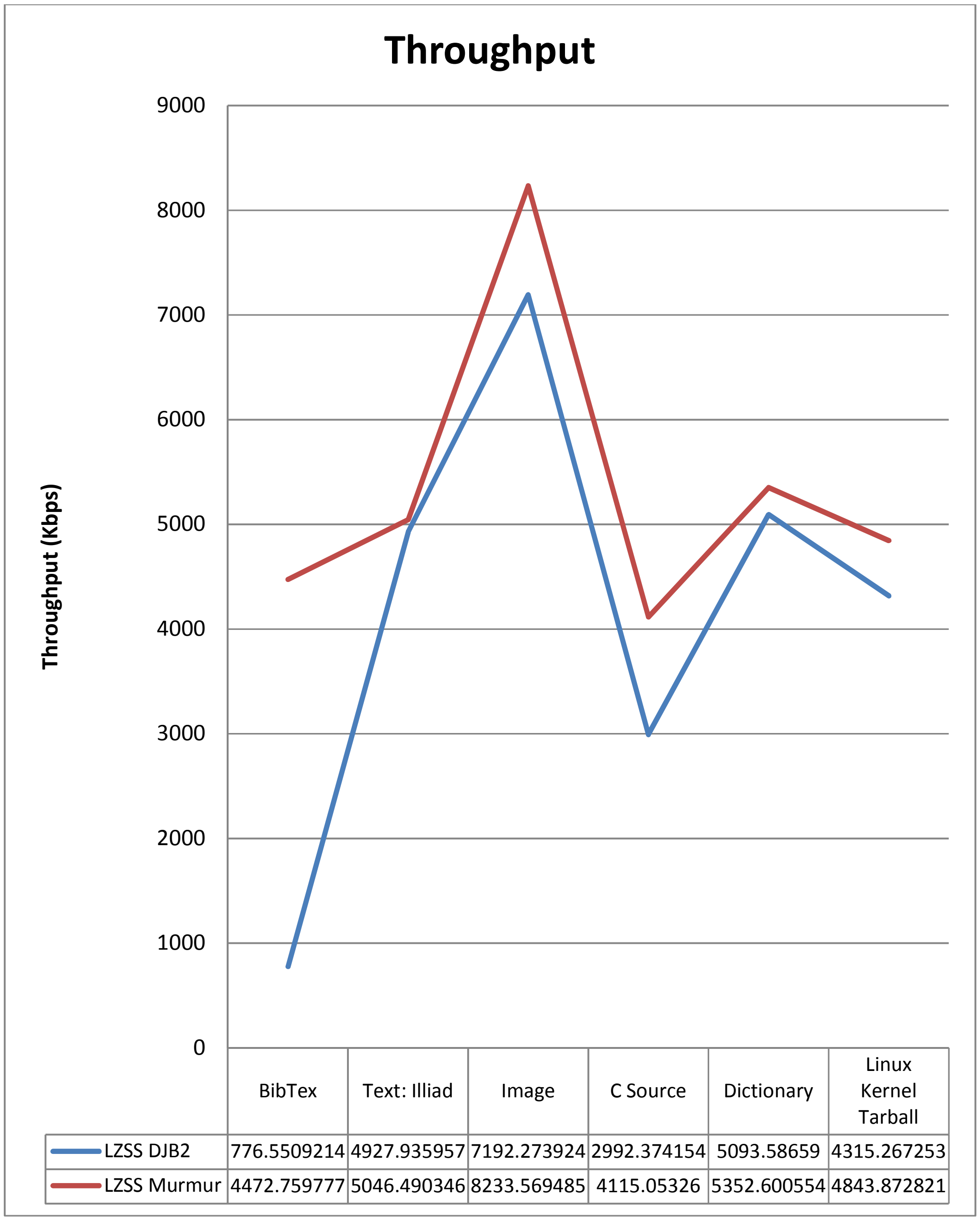


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZSS DJB2	0.1031	1.3366	4.6699	59.954	44.38899703	339.2398
LZSS Murmur	0.0179	1.3052	4.0793	43.5972	42.241	302.219

Figure 8: Time Taken: LZSS DJB v/s LZSS Murmur

The graph represents the time taken by each file to be compressed. So we want the line to be as low as possible. The graph above clearly shows that performance of Murmur hash is better than DJB2 hash.

12.1.2 Throughput

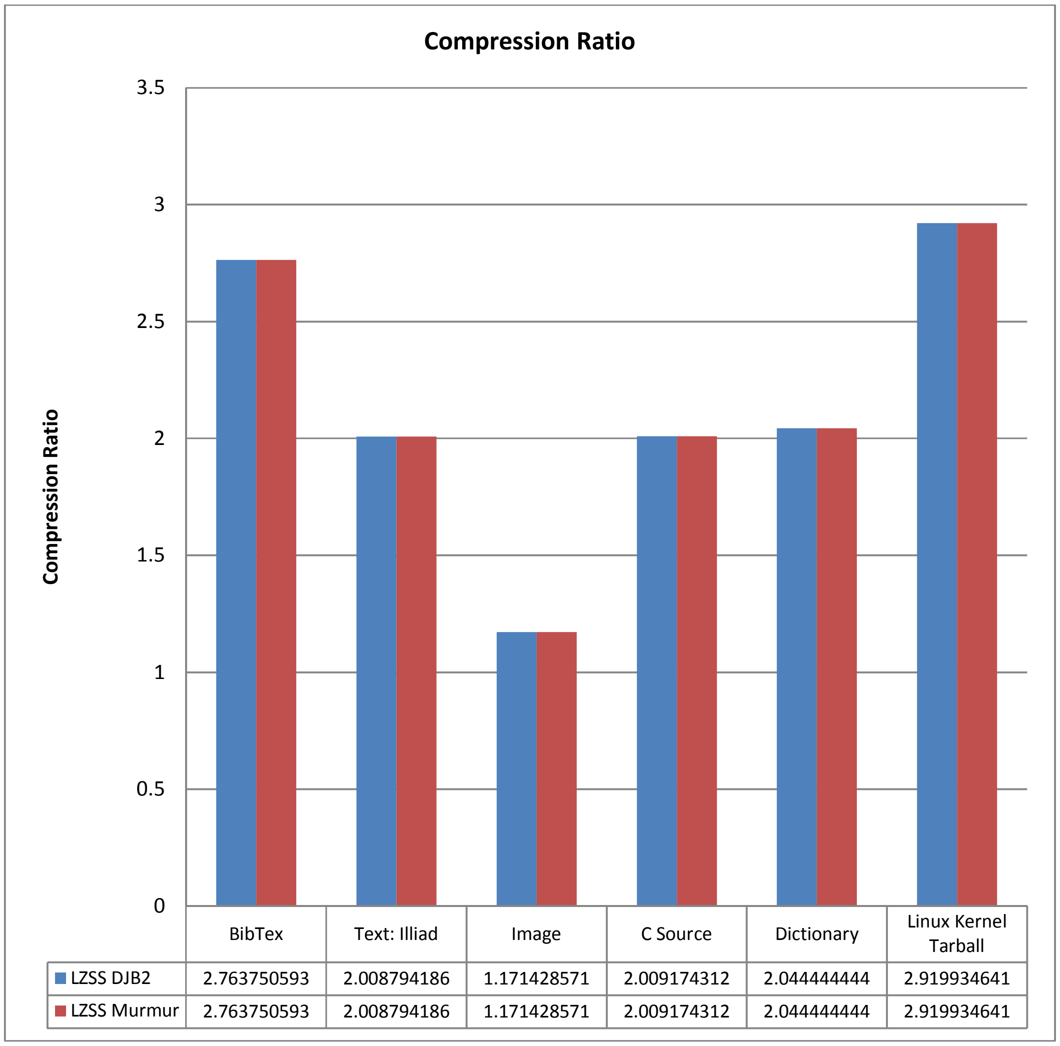


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZSS DJB2	776.5509214	4927.935957	7192.273924	2992.374154	5093.58659	4315.267253
LZSS Murmur	4472.759777	5046.490346	8233.569485	4115.05326	5352.600554	4843.872821

Figure 9: Throughput: LZSS DJB2 v/s LZSS Murmur

We expect that since Murmur hash is performing faster than DJB hash, the throughput of Murmur hash algorithm will be better which is illustrated by the above graph.

12.1.3 Compression Ratio

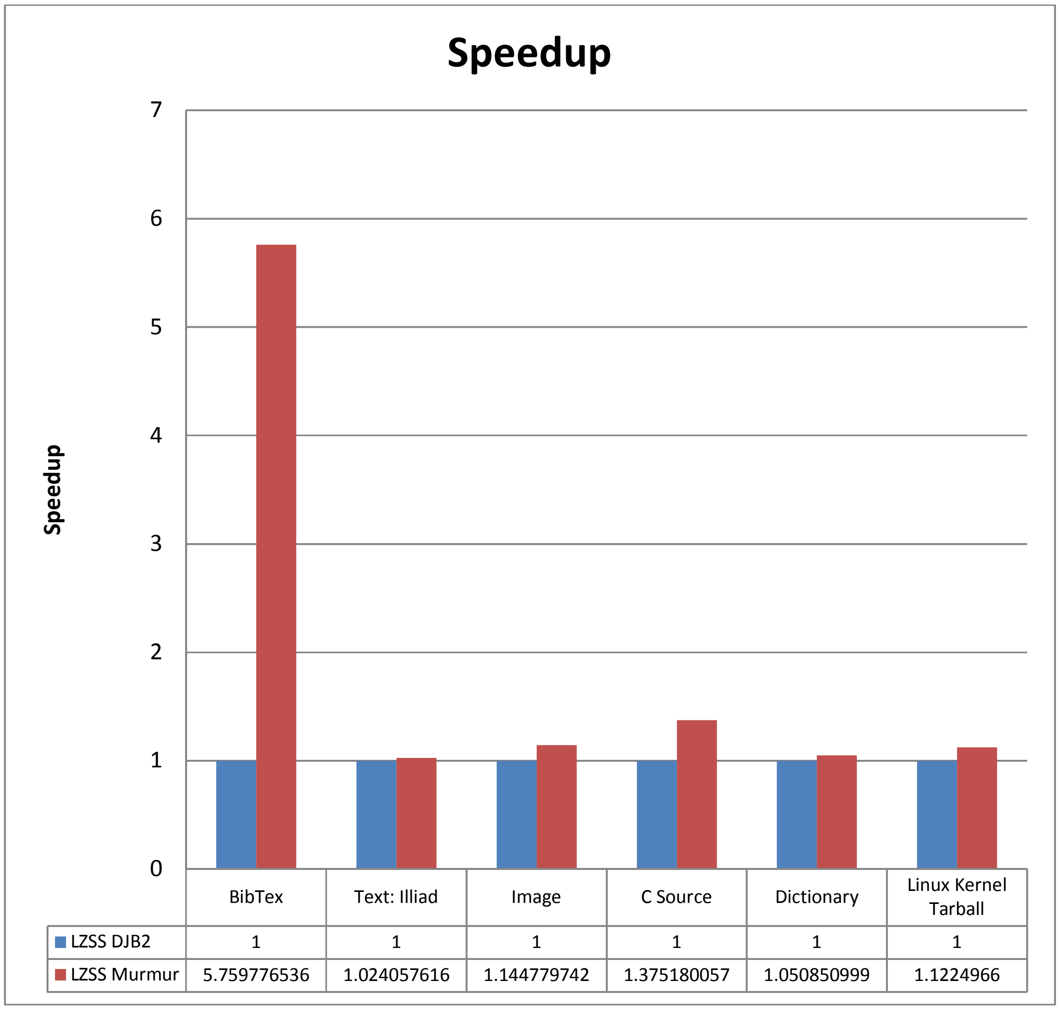


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS DJB2	2.763750593	2.008794186	1.171428571	2.009174312	2.044444444	2.919934641
🟥 LZSS Murmur	2.763750593	2.008794186	1.171428571	2.009174312	2.044444444	2.919934641

Figure 10: Compression Ratio: LZSS DJB2 v/s LZSS Murmur

The graph above illustrates that compression ratio is not affected by change in hash algorithm. If we look at the algorithms described earlier, we also know for a fact that compression ratio should remain the same.

12.1.4 Speedup

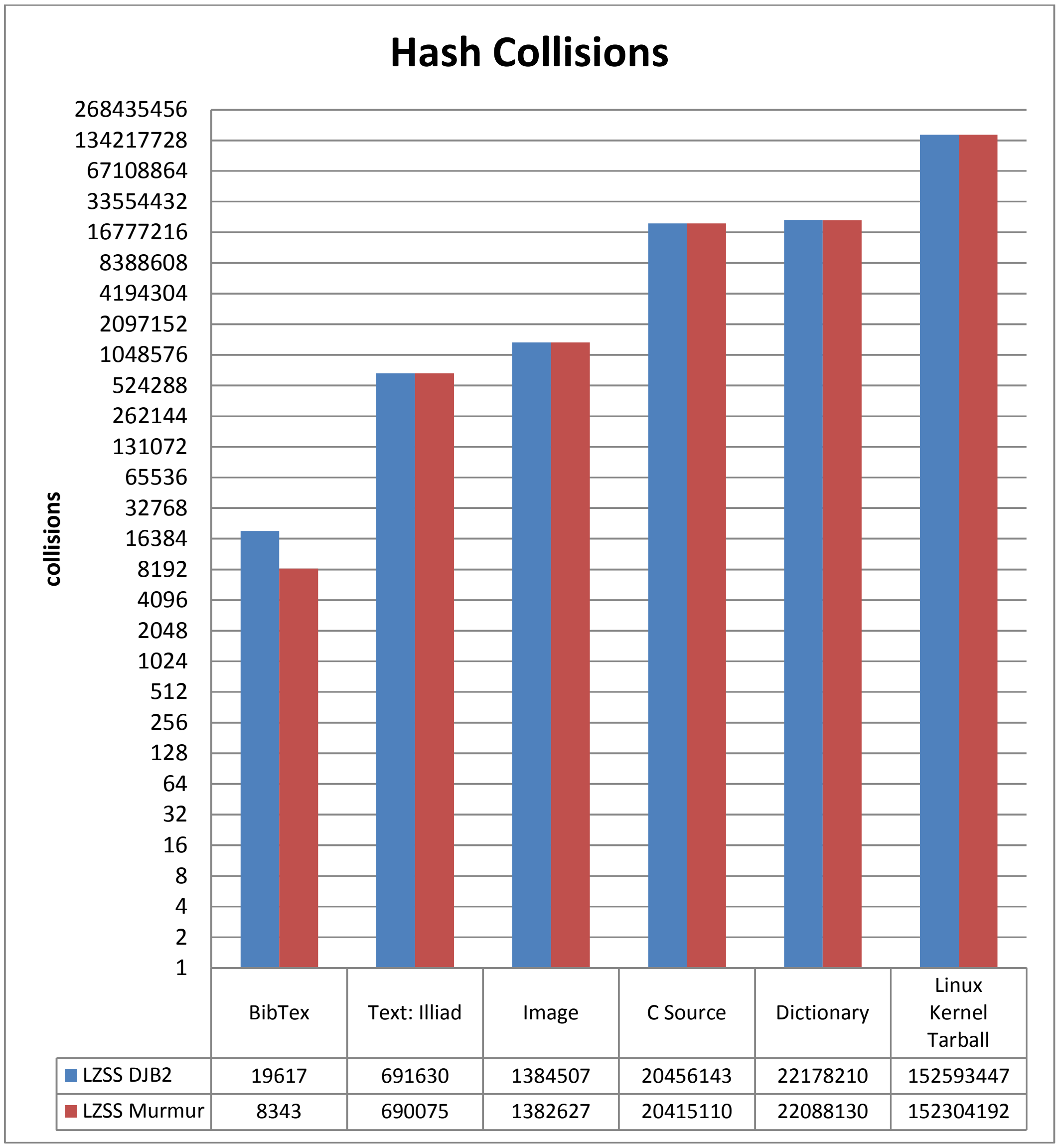


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS DJB2	1	1	1	1	1	1
🟥 LZSS Murmur	5.759776536	1.024057616	1.144779742	1.375180057	1.050850999	1.1224966

Figure 11: Speedup LZSS DJB2 v/s LZSS Murmur

This graph shows that for very small files murmur hash is showing better distribution than DJB2 hash. Murmur hashing algorithm is faster than DJB2 hash and thus there is a 1.02x to 1.3x speedup achieved by using the Murmur hash algorithm. Overall, simply by using murmur hash algorithm we can achieve a little speedup over the standard algorithm.

12.1.5 Collisions in hash



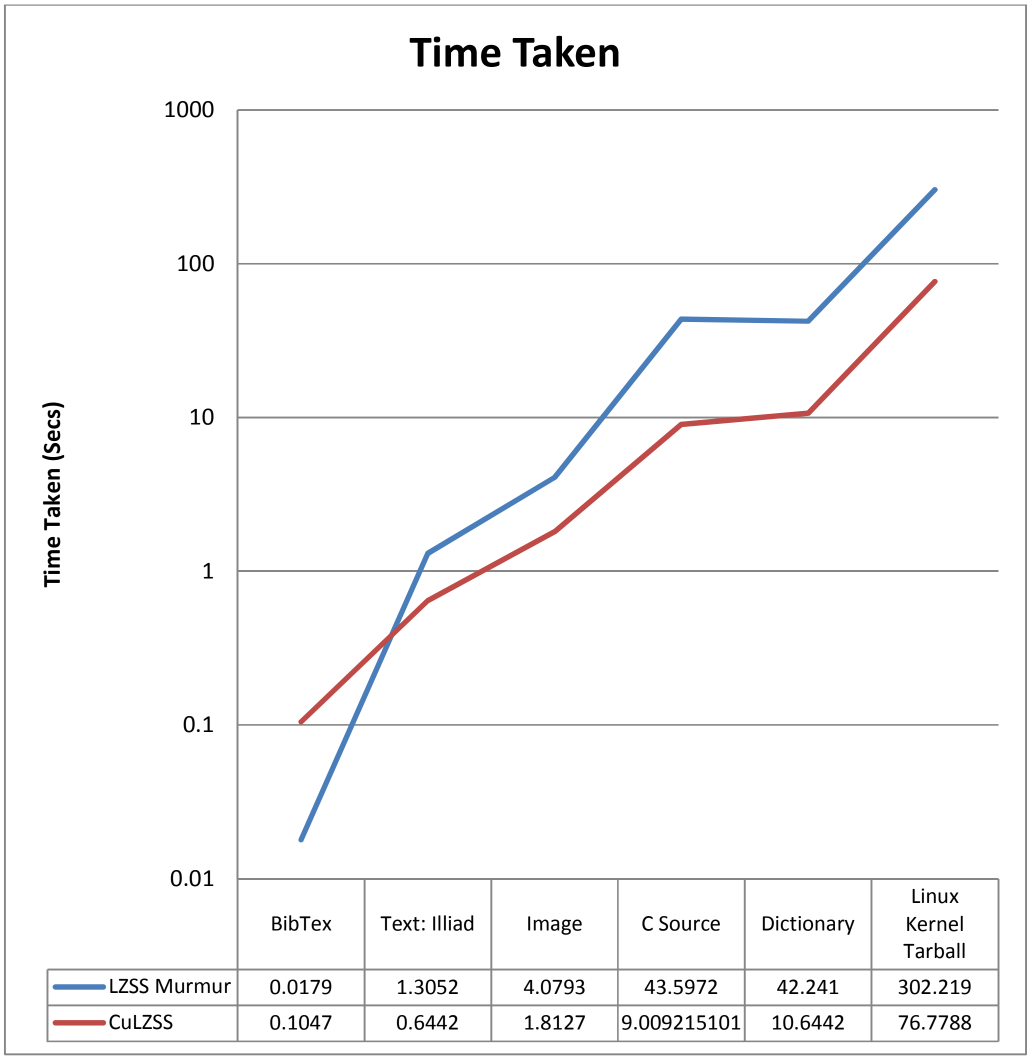
	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS DJB2	19617	691630	1384507	20456143	22178210	152593447
🟥 LZSS Murmur	8343	690075	1382627	20415110	22088130	152304192

Figure 12: Collisions in hash: LZSS DJB2 v/s LZSS Murmur

The above graph illustrates the reason for speedup in BibTex file. The reason is that Murmur hash is just a little bit more distributed and hence lesser collisions lead to faster throughput for BibTex. But for all the rest we can say that, collisions are roughly the same using any hash algorithm (only murmur hash performing a little better).

12.2 LZSS Murmur v/s Parallel CuLZSS

12.2.1 Time Taken

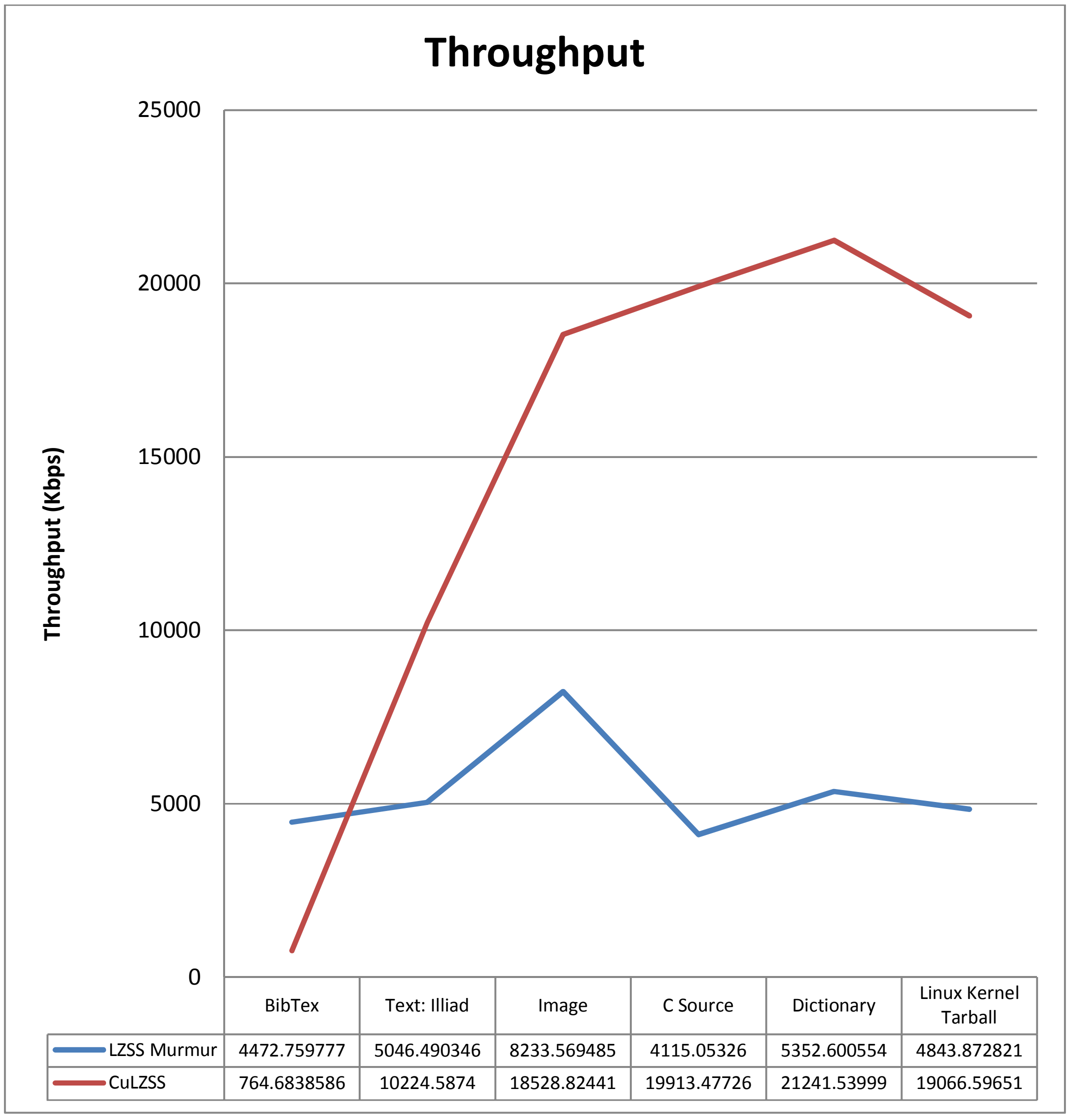


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZSS Murmur	0.0179	1.3052	4.0793	43.5972	42.241	302.219
CuLZSS	0.1047	0.6442	1.8127	9.009215101	10.6442	76.7788

Figure 13: Time taken LZSS Murmur v/s CuLZSS

The plot above shows that for very small files time CuLZSS is costlier. As file size is growing the parallelism for CuLZSS comes into effect. We can see CuLZSS performing a lot better for significant file sizes.

12.2.2 Throughput

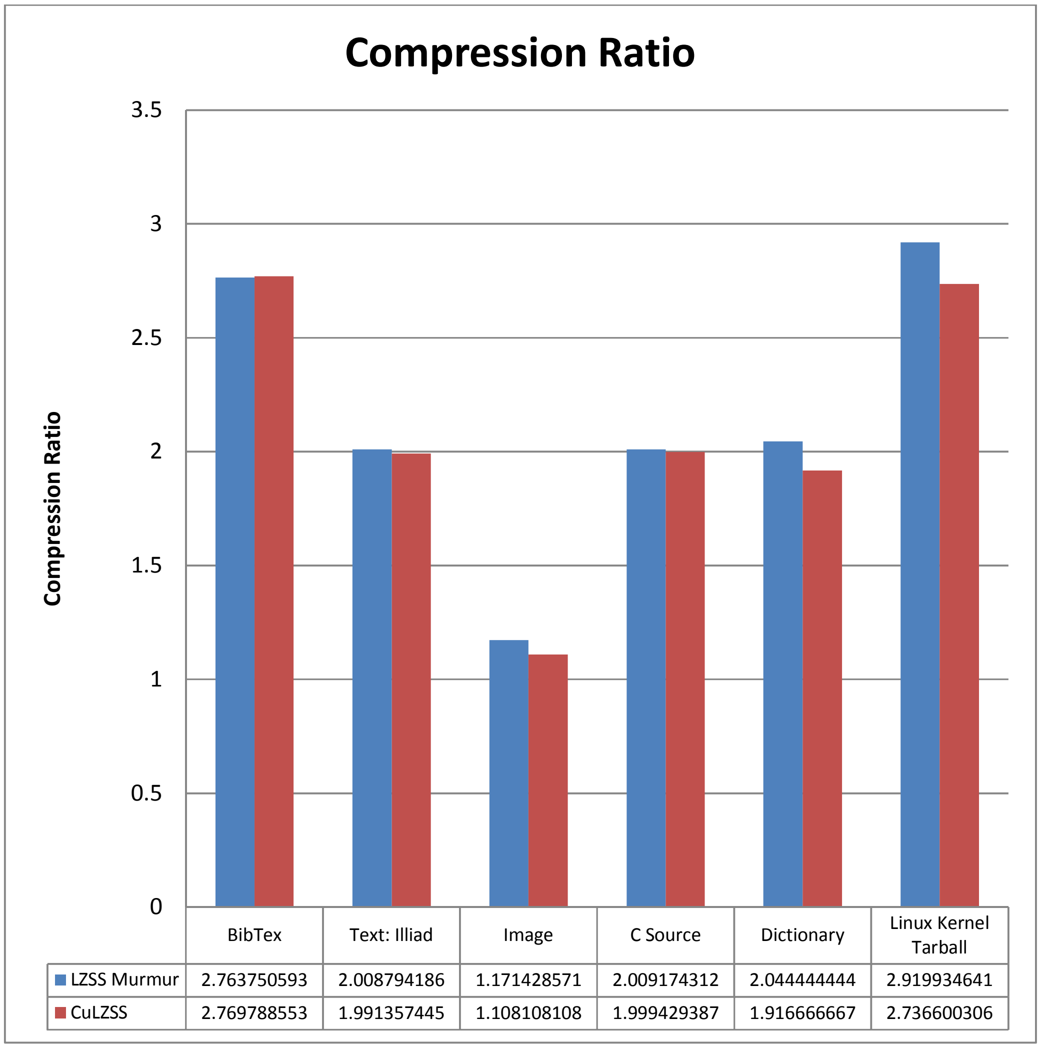


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZSS Murmur	4472.759777	5046.490346	8233.569485	4115.05326	5352.600554	4843.872821
CuLZSS	764.6838586	10224.5874	18528.82441	19913.47726	21241.53999	19066.59651

Figure 14: Throughput LZSS Murmur v/s CuLZSS

As in the previous graph, here also for smaller files we get lower throughput, but for significant size files we get a better throughput.

12.2.3 Compression Ratio

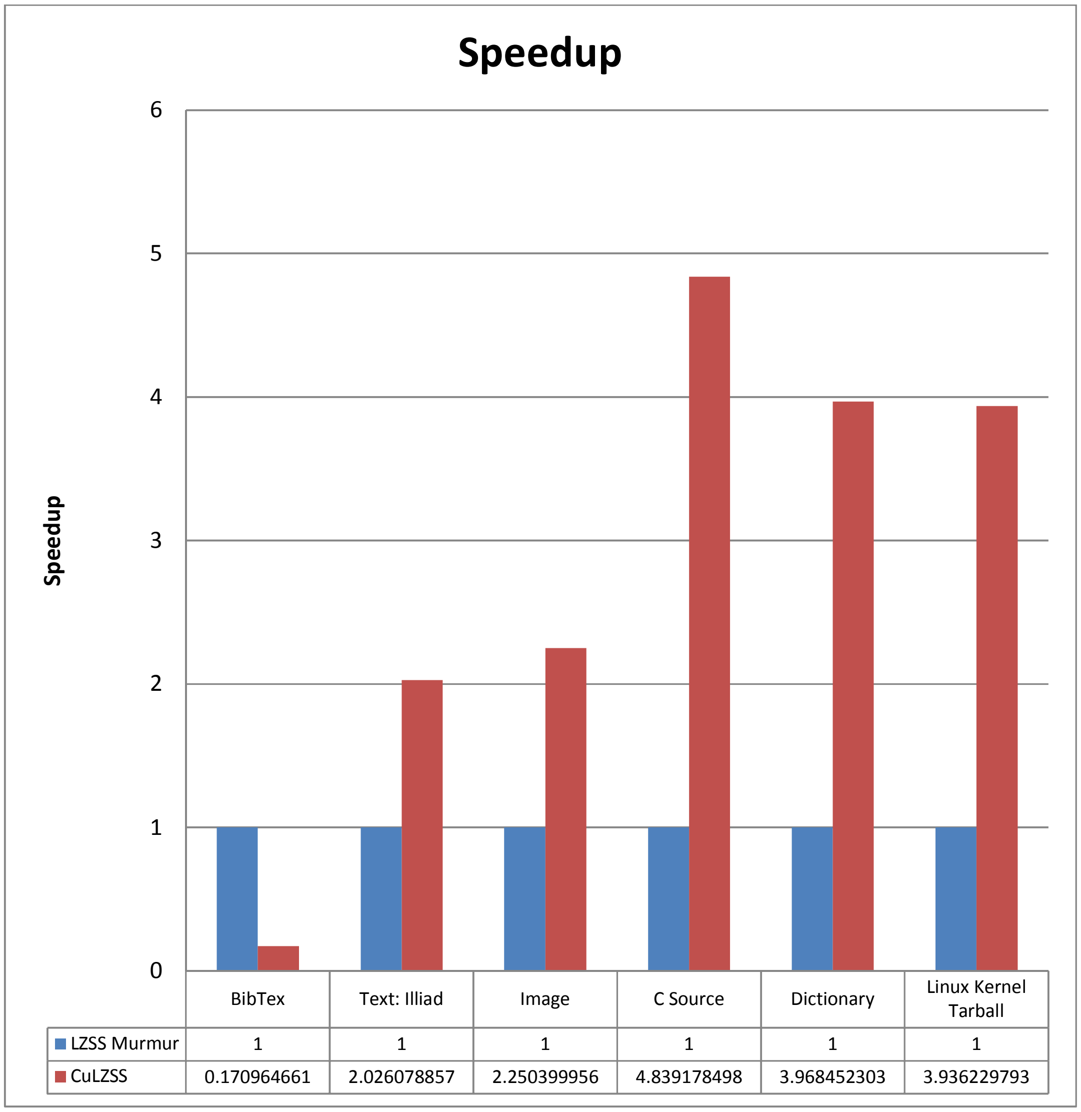


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS Murmur	2.763750593	2.008794186	1.171428571	2.009174312	2.044444444	2.919934641
🟥 CuLZSS	2.769788553	1.991357445	1.108108108	1.999429387	1.916666667	2.736600306

Figure 15: Compression Ratio LZSS Murmur v/s CuLZSS

The compression ratio is roughly the same but tilting towards LZSS Murmur as the parallel one cuts into the window for parallel threads.

12.2.4 Speedup



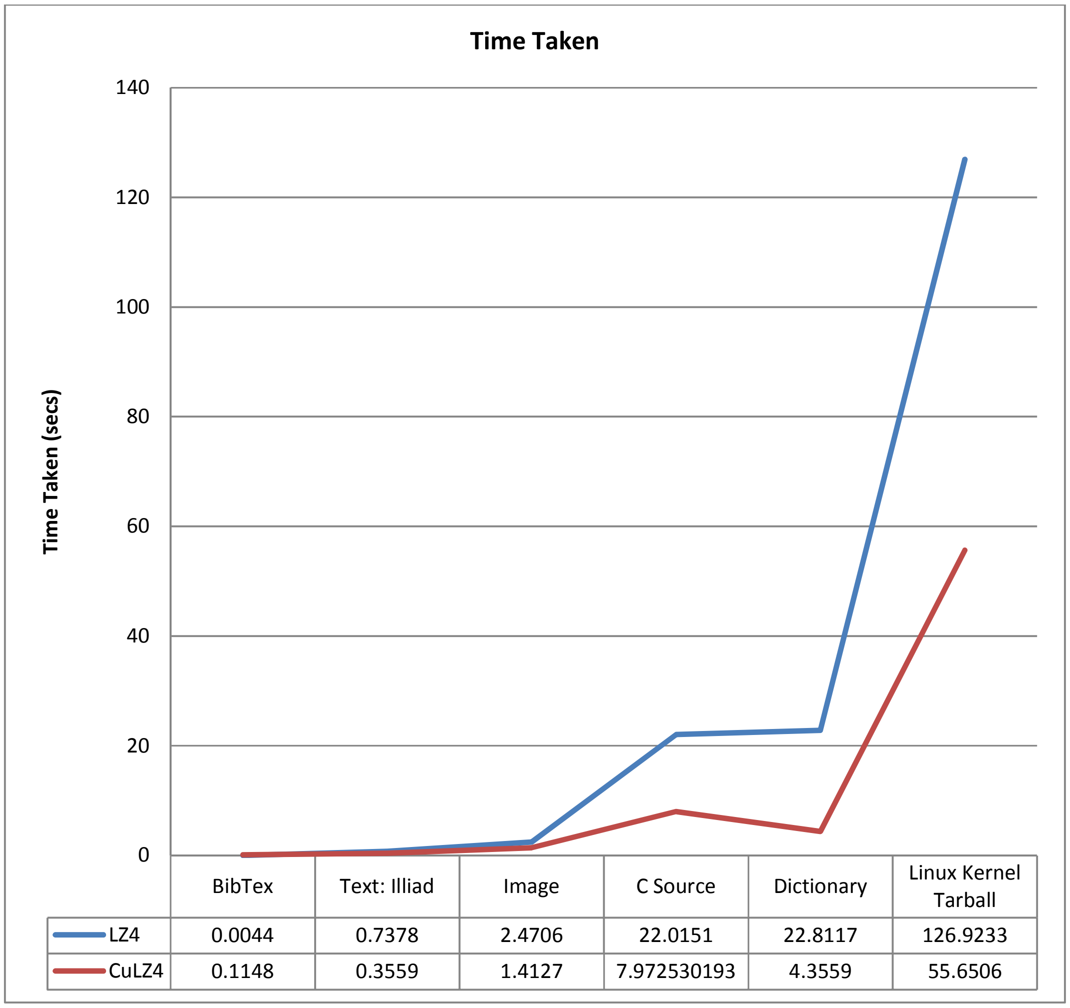
	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS Murmur	1	1	1	1	1	1
🟥 CuLZSS	0.170964661	2.026078857	2.250399956	4.839178498	3.968452303	3.936229793

Figure 16: Speedup LZSS Murmur v/s CuLZSS

Again for very small files, the CUDA version will not perform better but as the file size becomes significant, the CUDA version does go 2x to 4x speedup on the serial version.

12.3 LZ4 v/s Parallel CuLZ4

12.3.1 Time Taken

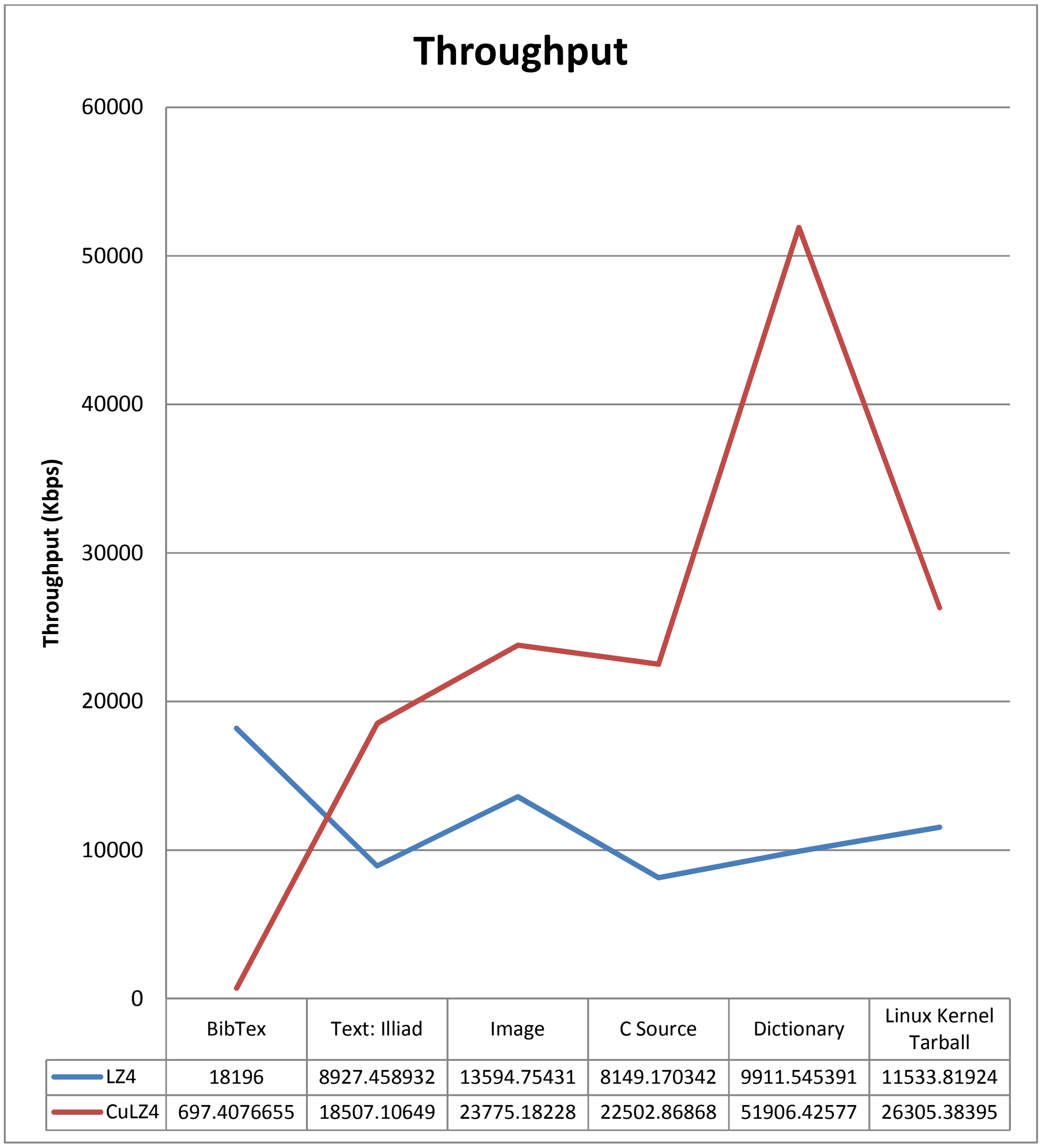


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZ4	0.0044	0.7378	2.4706	22.0151	22.8117	126.9233
CuLZ4	0.1148	0.3559	1.4127	7.972530193	4.3559	55.6506

Figure 17: Time taken LZ4 v/s CuLZ4

We see that again for smaller files like BibTex, LZ4 is lightning fast but as file size goes up we get better performance with CUDA.

12.3.2 Throughput

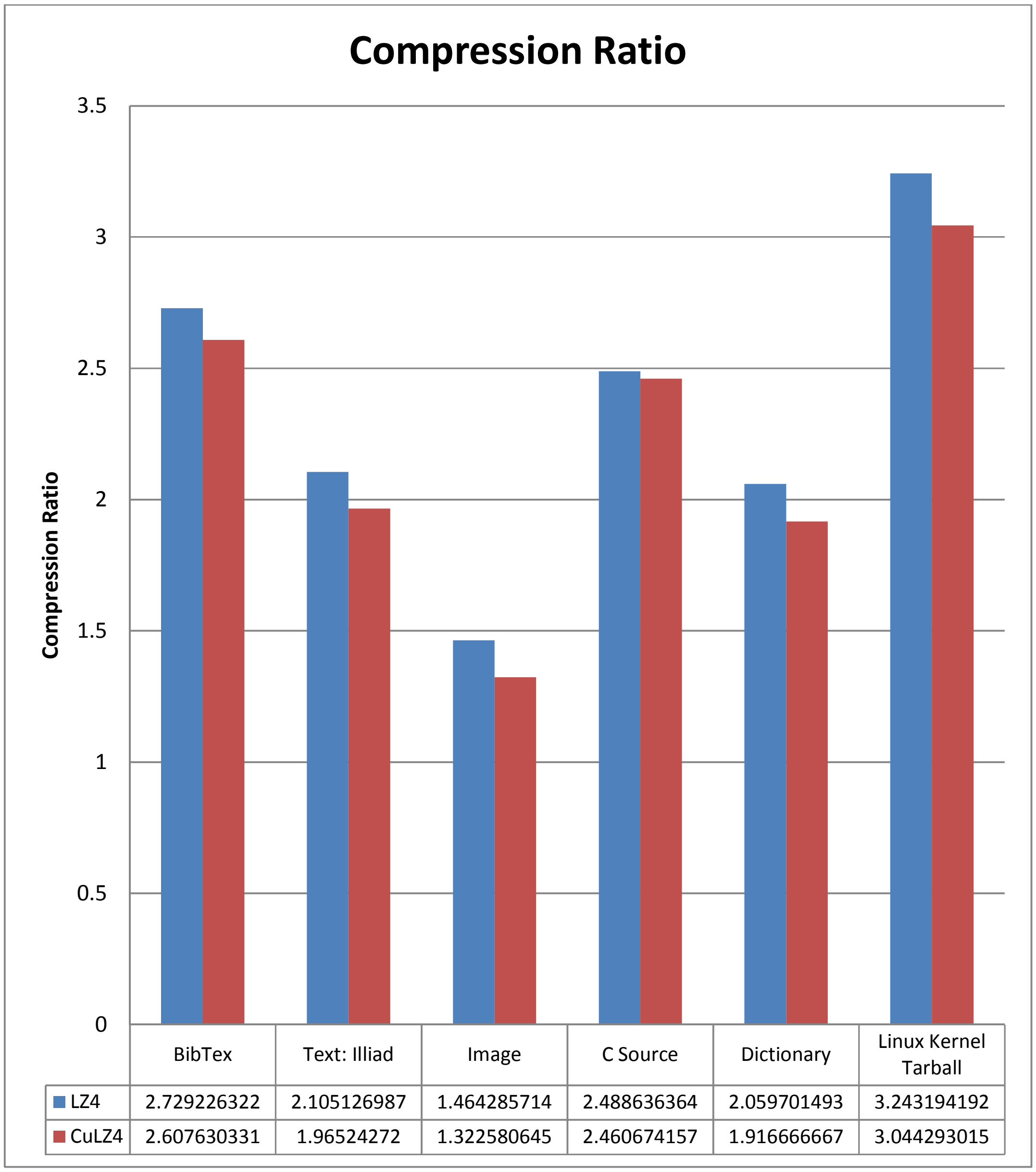


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZ4	18196	8927.458932	13594.75431	8149.170342	9911.545391	11533.81924
CuLZ4	697.4076655	18507.10649	23775.18228	22502.86868	51906.42577	26305.38395

Figure 18: Throughput LZ4 v/s CuLZ4

Same reason as the previous graph, we have better throughput for CuLZ4.

12.3.3 Compression Ratio

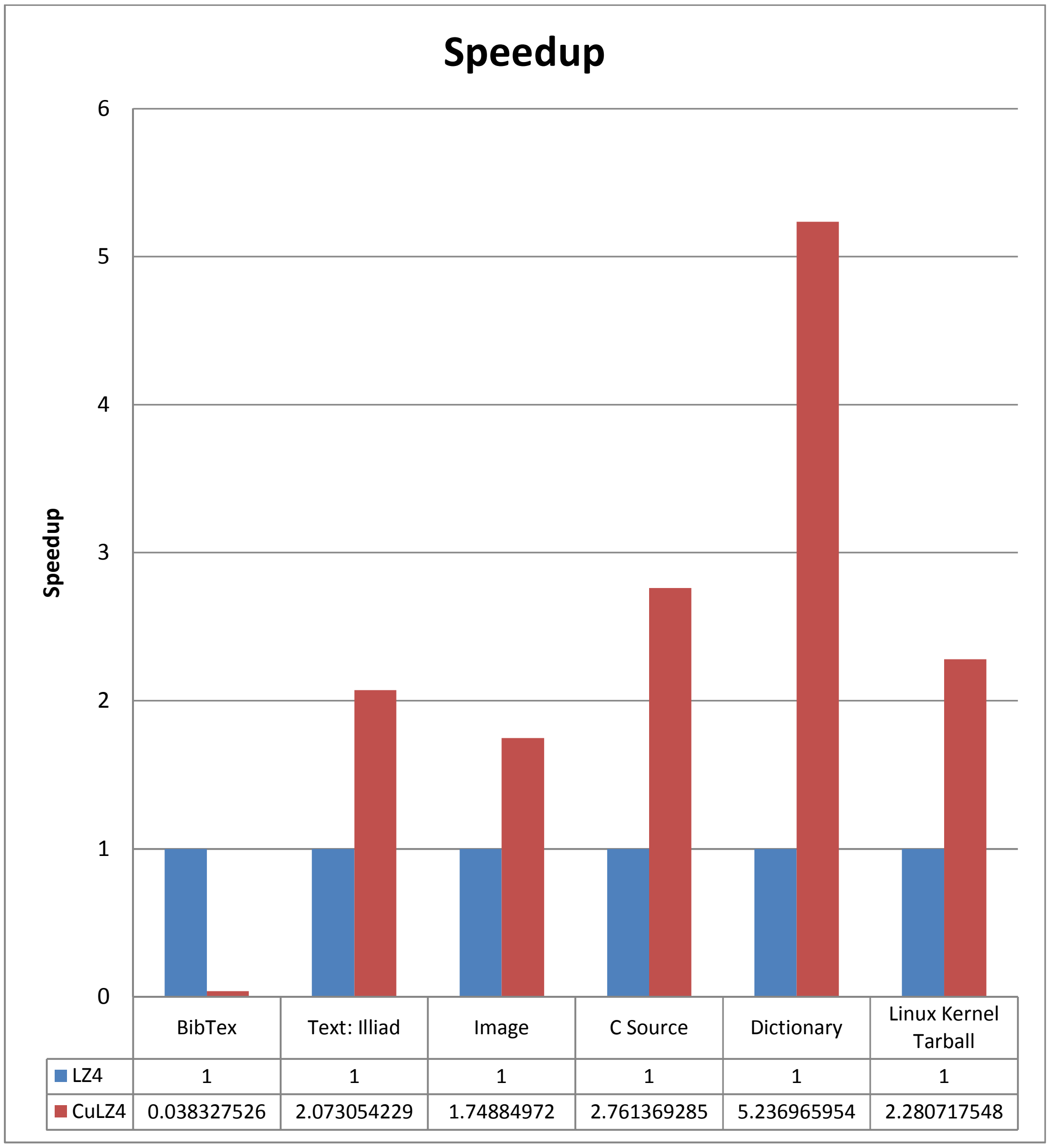


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZ4	2.729226322	2.105126987	1.464285714	2.488636364	2.059701493	3.243194192
🟥 CuLZ4	2.607630331	1.96524272	1.322580645	2.460674157	1.916666667	3.044293015

Figure 19: Compression Ratio LZ4 v/s CuLZ4

Because of losing on the 4 kb window, we get roughly similar, but lower compression ratio with CuLZ4.

12.3.4 Speedup



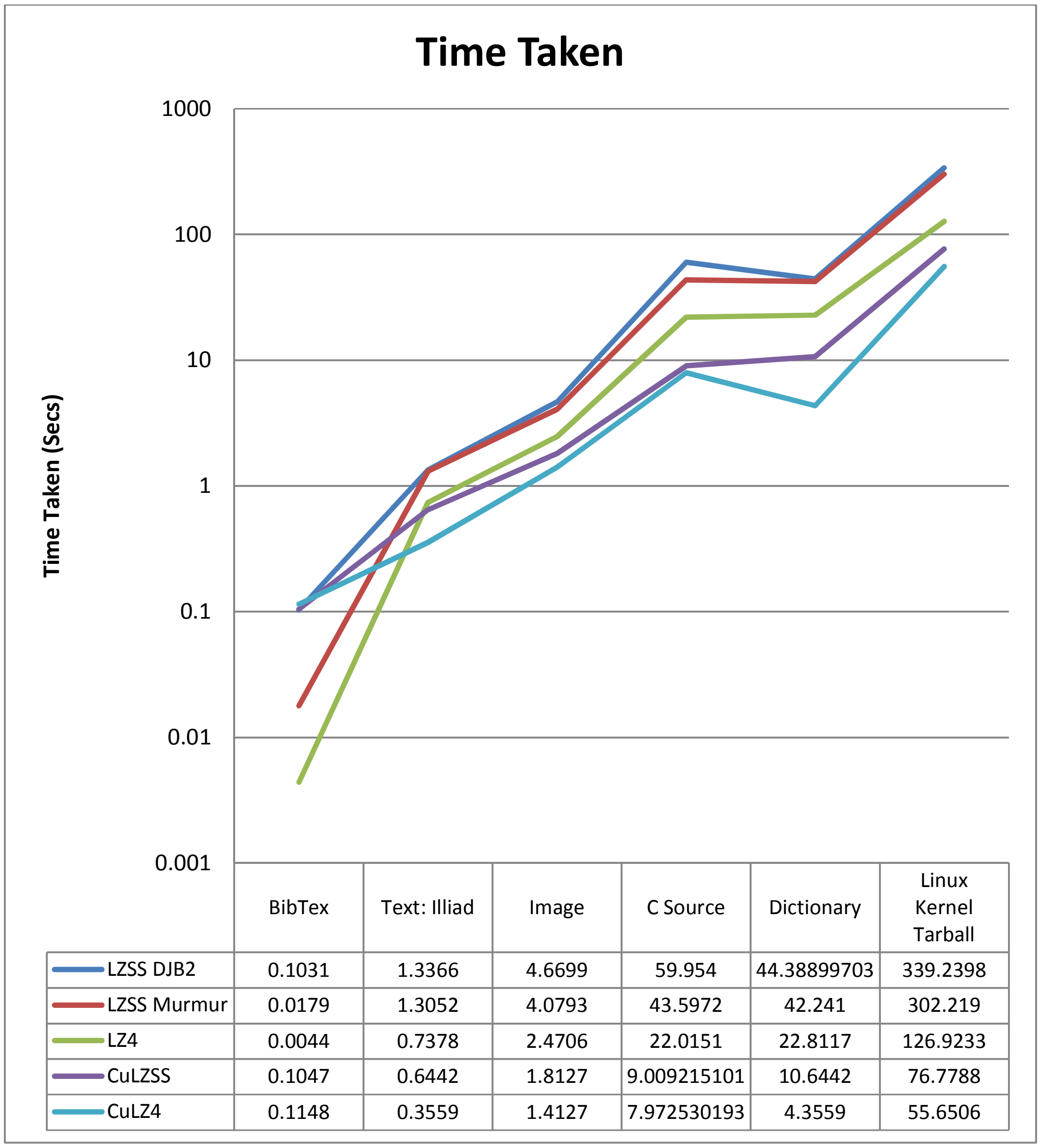
	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZ4	1	1	1	1	1	1
🟥 CuLZ4	0.038327526	2.073054229	1.74884972	2.761369285	5.236965954	2.280717548

Figure 20: Speedup LZ4 v/s CuLZ4

We get a 2x to 5x speedup for significant size files in CuLZ4.

12.4 Overall

12.4.1 Time Taken

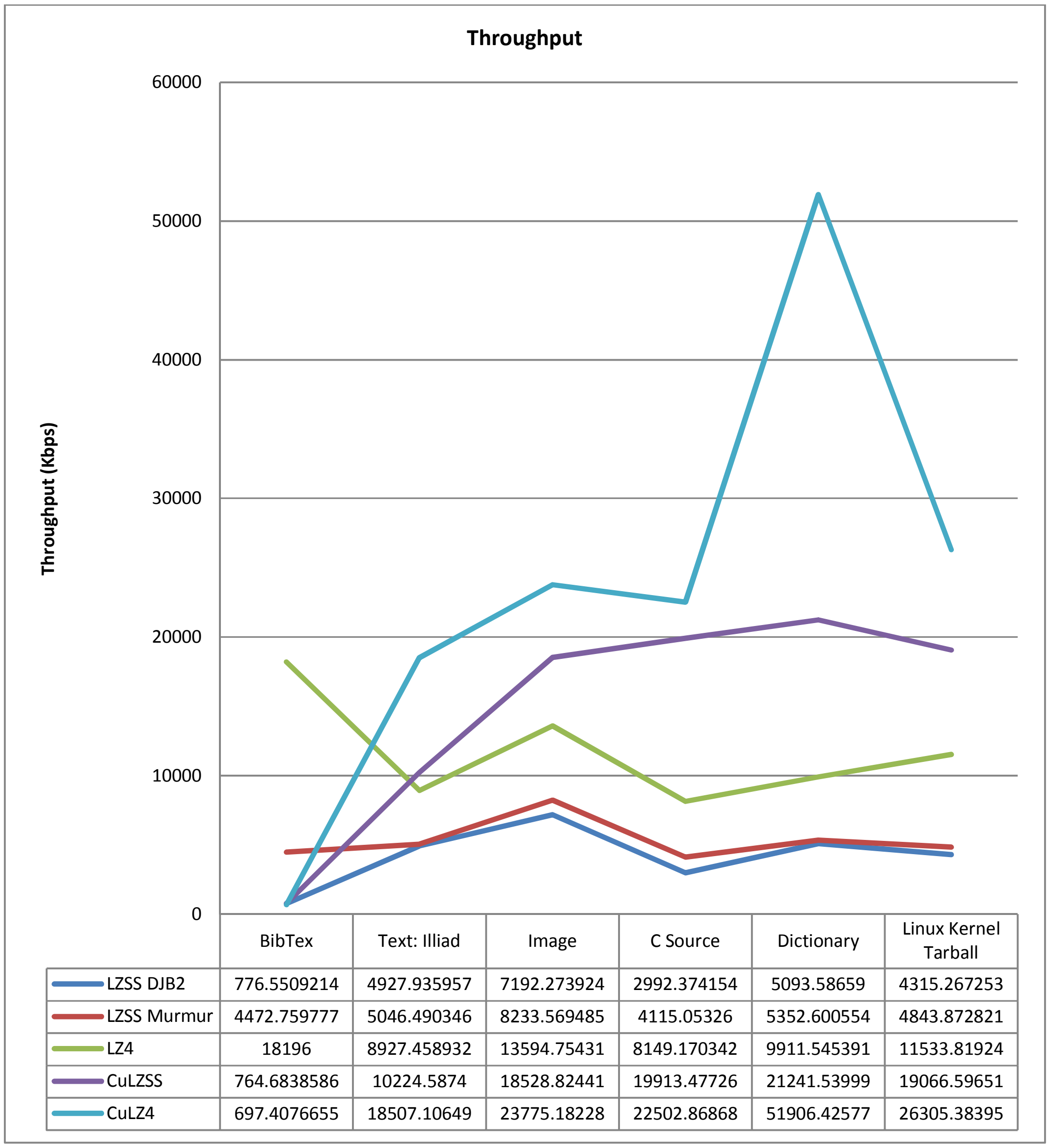


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZSS DJB2	0.1031	1.3366	4.6699	59.954	44.38899703	339.2398
LZSS Murmur	0.0179	1.3052	4.0793	43.5972	42.241	302.219
LZ4	0.0044	0.7378	2.4706	22.0151	22.8117	126.9233
CuLZSS	0.1047	0.6442	1.8127	9.009215101	10.6442	76.7788
CuLZ4	0.1148	0.3559	1.4127	7.972530193	4.3559	55.6506

Figure 21: Time taken comparison

The above figure retraces the flow of this dissertation. The standard DJB2 takes the longest and CuLZ4 turns out to be best one out of the ones discussed.

12.4.2 Throughput

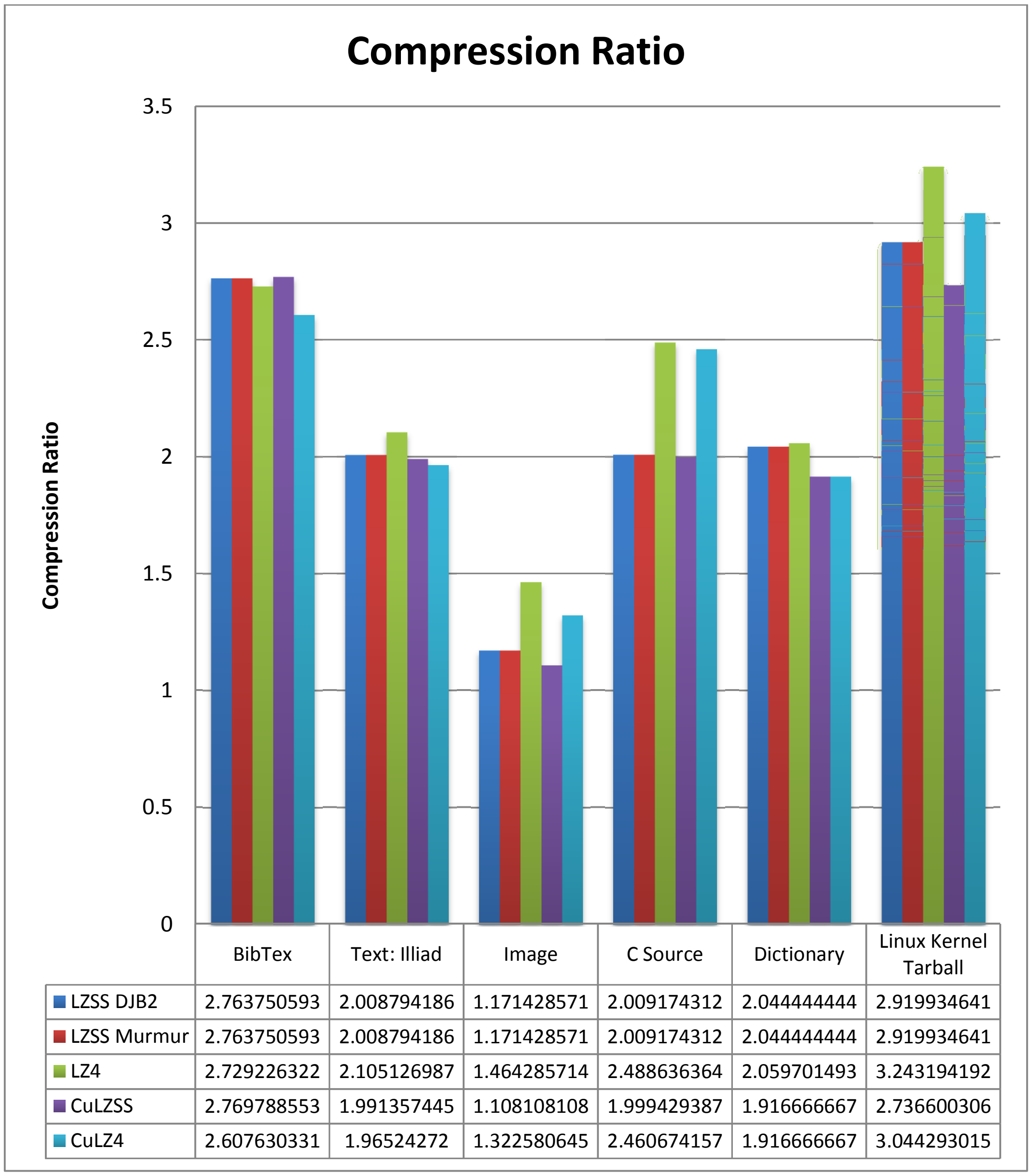


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
LZSS DJB2	776.5509214	4927.935957	7192.273924	2992.374154	5093.58659	4315.267253
LZSS Murmur	4472.759777	5046.490346	8233.569485	4115.05326	5352.600554	4843.872821
LZ4	18196	8927.458932	13594.75431	8149.170342	9911.545391	11533.81924
CuLZSS	764.6838586	10224.5874	18528.82441	19913.47726	21241.53999	19066.59651
CuLZ4	697.4076655	18507.10649	23775.18228	22502.86868	51906.42577	26305.38395

Figure 22: Throughput Comparison

This builds on the previous graph illustrating, for smaller file size we should use LZ4 and for significant sizes, we should go to the more powerful CuLZ4.

12.4.3 Compression Ratio

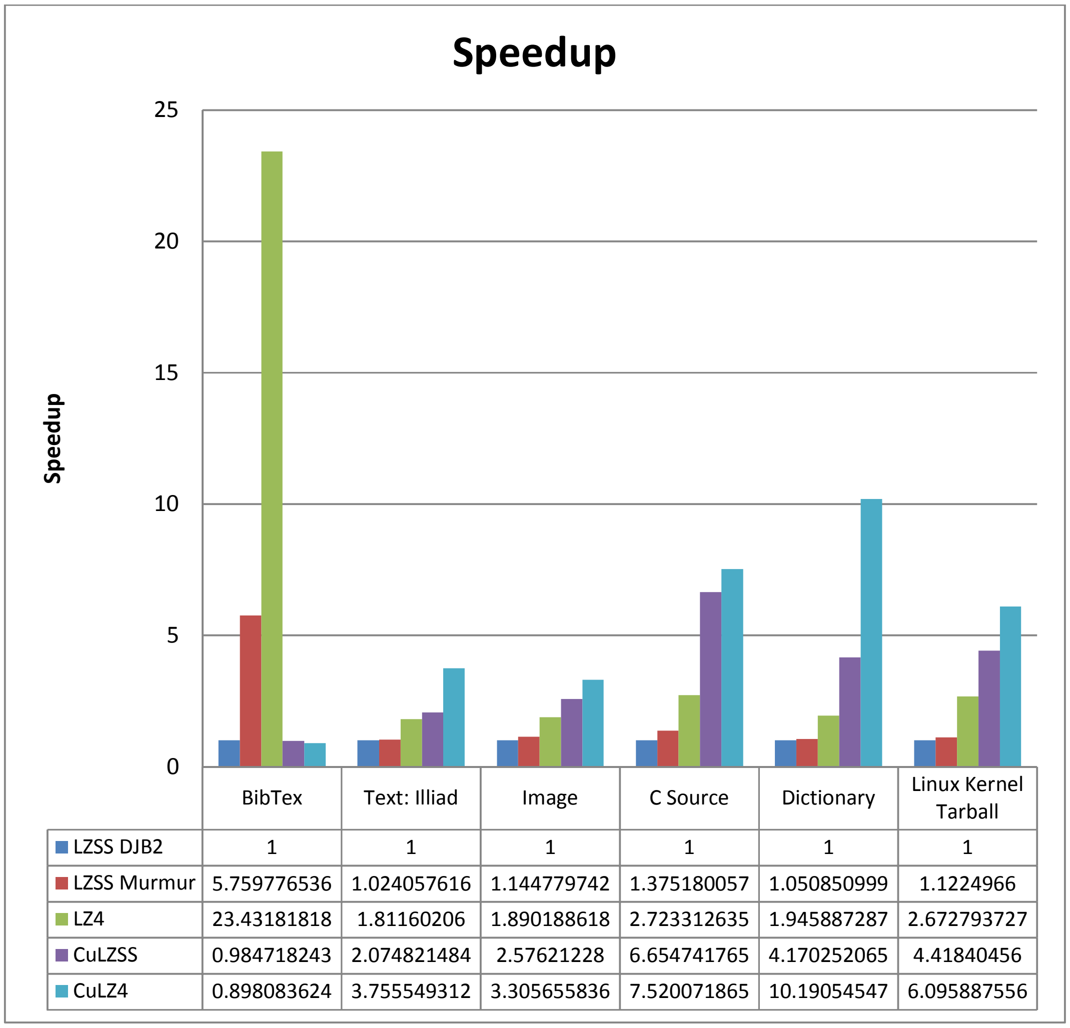


	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS DJB2	2.763750593	2.008794186	1.171428571	2.009174312	2.044444444	2.919934641
🟥 LZSS Murmur	2.763750593	2.008794186	1.171428571	2.009174312	2.044444444	2.919934641
🟩 LZ4	2.729226322	2.105126987	1.464285714	2.488636364	2.059701493	3.243194192
🟪 CuLZSS	2.769788553	1.991357445	1.108108108	1.999429387	1.916666667	2.736600306
🟦 CuLZ4	2.607630331	1.96524272	1.322580645	2.460674157	1.916666667	3.044293015

Figure 23: Compression Ratio Comparison

We see the compression ratio of LZ4 edging others. As file size is increasing the ratio is getting higher and higher as compared to others.

12.4.4 Speedup



	BibTex	Text: Illiad	Image	C Source	Dictionary	Linux Kernel Tarball
🟦 LZSS DJB2	1	1	1	1	1	1
🟥 LZSS Murmur	5.759776536	1.024057616	1.144779742	1.375180057	1.050850999	1.1224966
🟩 LZ4	23.43181818	1.81160206	1.890188618	2.723312635	1.945887287	2.672793727
🟪 CuLZSS	0.984718243	2.074821484	2.57621228	6.654741765	4.170252065	4.41840456
🟦 CuLZ4	0.898083624	3.755549312	3.305655836	7.520071865	10.19054547	6.095887556

Figure 24: Speedup Comparison

This paints a clear picture of the performance of the algorithms. For very small files we get a speedup of up to 23x using the serial LZ4. Using the CUDA enabled version for significant file sizes we get a good speedup up to 10x the standard algorithm used today.

13 CONCLUSION

From the above results we can conclude that the standard LZSS algorithm with DJB2 hash can be improved serially with Murmur hash without a change in format specification. Also we can afford to change the format specification LZ4 performs a whole lot better than the LZSS format. LZ4 also provides a better compression ratio.

The GPU can be utilized to really make compression fast. For files of size of a couple MBs and up, we should use the more powerful CuLZSS and CuLZ4. For smaller files (of a few hundred KBs in size) it takes the CUDA versions longer due to the time spent in transfer of data from RAM to DRAM on the graphics processor. Thus, for most systems that stores user data online, The CUDA enabled versions would be definitely a huge time saver. Also for the home users, they can get more out of their existing hardware with CUDA enabled compression.

Serially LZ4 is shown to be better algorithm both speed wise and compression ratio wise. If given a choice into parallel, the CUDA enabled version of LZ4 is really fast. In today's multitasking and demanding computing requirements, the CuLZ4 can be a boon. It can be coupled with some cloud storage software to increase storage (compression) and access times.

14 FUTURE WORK

We have seen how using the untapped potential of the dormant graphics card under the hood of our computers can be used to help speedup data compression. General purpose graphics processing unit computing is a highly researched topic today. We are still in its early days, with NVIDIA leading from the forefront. As more advanced and powerful graphics processing units come into existence like the NVIDIA Kepler which has dynamic parallelism and Hyper-Q and clock speeds reaching memory clocks of 6GHz and 1536 processing cores with a clock speed of 1058 MHz, paves the way for more complex algorithms that are more GPU friendly and compute intensive [21]. The only concern that remains is the heat that is generated from the GPU. However, the energy efficiency of GPU is generally very high as compared to the processing power to generated, but still its use in enterprise environments is limited by this factor. A deviation from the standard Lempel Ziv formats can come into place which may be more SIMD and CUDA favorable. Data will continue to explode as more people create more footprints the web and elsewhere. Ergo, data compression will always be of interest for many scientists. More generic implementations of day to day software can be created with OpenCL or such languages which are not limited to NVIDIA GPUs but can be used with just about any GPU. With CPU clock speeds peaking, the GPUs are the future to a faster computing environment and more and more algorithms need to be made CUDA friendly.

15 REFERENCES

[1] Anderson, Nate, Jens Mache, and William Watson. "Learning CUDA: lab exercises and experiences." In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 183-188. ACM, 2010.

[2] Deutsch, L. Peter. "DEFLATE compressed data format specification version 1.3." (1996).

[3] Deutsch, L. Peter. "GZIP file format specification version 4.3." (1996).

[4] Dipperstein, Michael. "LZSS (LZ77) discussion and implementation." (2008).

[5] Eastman, Willard L., Abraham Lempel, Jacob Ziv, and Martin Cohn. "Apparatus and method for compressing data signals and restoring the compressed data signals." U.S. Patent 4,464,650, issued August 7, 1984.

[6] Eitz, Mathias, and Gu Lixu. "Hierarchical spatial hashing for real-time collision detection." In *Shape Modeling and Applications, 2007. SMI'07. IEEE International Conference on*, pp. 61-70. IEEE, 2007.

[7] Farach, Martin, and Mikkel Thorup. "String Matching in Lempel—Ziv Compressed Strings." *Algorithmica* 20, no. 4 (1998): 388-404.

[8] Gasieniec, Leszek, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. "Efficient algorithms for Lempel-Ziv encoding." In *Algorithm Theory—SWAT'96*, pp. 392-403. Springer Berlin Heidelberg, 1996.

[9] Harris, Mark. "Optimizing parallel reduction in CUDA." *NVIDIA Developer Technology* 2 (2007).

[10] Kirk, David. "NVIDIA CUDA software and GPU parallel computing architecture." In*ISMM*, vol. 7, pp. 103-104. 2007.

[11] Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." *SIAM journal on computing* 6, no. 2 (1977): 323-350.

[12] Lerchundi Osa, Gorka. "Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU." (2011).

[13] Murmur Hash, last Modified: March 1,2011, https://sites.google.com/site/murmurhash.

[14] Nelson, Mark R. "LZW data compression." *Dr. Dobb's Journal* 14, no. 10 (1989): 29-36.

[15] Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron. "Scalable parallel programming with CUDA." *Queue* 6, no. 2 (2008): 40-53.

[16] Nvidia optimus technology. http://www.nvidia.com/object/optimus_technology.html

[17] Nvidia, C. U. D. A. "C programming best practices guide." *Cuda Toolkit* 2 (2009).

[18] Nvidia, C. U. D. A. "Programming guide." (2008).

[19] Pavlov, Igor. "LZMA Software Development Kit." (2009).

[20] Yang, H., M. Li, K. Koizumi, and H. Kudo. "Accelerating backprojections via CUDA architecture." In *9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, vol. 9, pp. 52-55. 2007.

[21] Ziv, Jacob, and Abraham Lempel. "A universal algorithm for sequential data compression." *Information Theory, IEEE Transactions on* 23, no. 3 (1977): 337-343.

16 APPENDIX

16.1 Hashing CUDA Kernel

```
/**
 * hash on device specific to thread
 */
__device__ unsigned int hash(int i,int index,byte *dreadBuf,int dreadBufSize)
{
	const uint c1 = 0xcc9e2d51;
	const uint c2 = 0x1b873593;
	const uint r1 = 15;
	const int len = 3;
	uint hash = 0;
	uint b1 = dreadBuf[index*oneMB + i];
	uint b2=0;
	if(i + 1 < dreadBufSize)
		b2 = dreadBuf[index*oneMB + (i + 1)];
	uint b3=0;
	if(i + 2 < dreadBufSize)
		b3 = dreadBuf[index*oneMB + (i + 2)];
	uint remainingBytes=(b1<<16) | (b2<<8) | b3;

	remainingBytes = remainingBytes * c1;
	remainingBytes = (remainingBytes << r1) | (remainingBytes >> (32 - r1));
	remainingBytes = remainingBytes * c2;

	hash = hash ^ remainingBytes;

	hash = hash ^ len;

	hash = hash ^ (hash >> 16);
	hash = hash * 0x85ebca6b;
	hash = hash ^ (hash >> 13);
	hash = hash * 0xc2b2ae35;
	hash = hash ^ (hash >> 16);
	return hash % HASH_SIZE;
}
/**
 * insert the sequence of 3 in the hash
 */
__device__ void insertHash(int i,int index,byte *dreadBuf,node *dhashTable,int dreadBufSize)
{
	uint key=hash(i,index,dreadBuf,dreadBufSize);
	int j;
	for(j=0;j<MAX_CHAIN_LENGTH;j++)
```

```
		{
			if(!dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
				(MAX_CHAIN_LENGTH*key) +j].valid)
			{
				 dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
				(MAX_CHAIN_LENGTH*key) +j].valid = true;
				 dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
				(MAX_CHAIN_LENGTH*key) +j].indexInReadBuf = i;
			}
		}
}

/**
 * remove the sequence of 3 from the hash
 */
__device__ void removeHash(int i,int index,byte *dreadBuf,node *dhashTable,int dreadBufSize)
{
		uint key=hash(i,index,dreadBuf,dreadBufSize);
		int j;
		for(j=0;j<MAX_CHAIN_LENGTH;j++)
		{
			if(dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
				(MAX_CHAIN_LENGTH*key) +j].valid &&
dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
								(MAX_CHAIN_LENGTH*key)
+j].indexInReadBuf==i)
			{
				 dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
				(MAX_CHAIN_LENGTH*key) +j].valid = false;
				 dhashTable[index*HASH_SIZE*MAX_CHAIN_LENGTH +
				(MAX_CHAIN_LENGTH*key) +j].indexInReadBuf = -1;
			}
		}
}
/**
 * the kernel init point
 */
__global__ void kernel(byte *dreadBuf,int *dreadBufSize,node *dhashTable,
			byte* dOutput,int *dOutputSize,byte *dliterals,entry *dtheEntry)
{
		int index = blockIdx.x * blockDim.x + threadIdx.x;
		if(index < WORK_SIZE)
		{
			int i,j,lookahead,start=0,sl;
			pii p;
			//printf("My WorkLoad: %d\n",dreadBufSize[index]);
			for(i=0;i<13 && i<dreadBufSize[index];i++)
			{
				dtheEntry[index].countLiterals++;
				dliterals[index*MAX_LITERALS +i] = dreadBuf[index*oneMB +i];
				//printf("Read: %c\n",dreadBuf[index*oneMB +i]);
```

```
                        insertHash(i,index,dreadBuf,dhashTable,dreadBufSize[index]);
                }
                //printf("index: %d %p %p\n",index,dtheEntry,dliterals[index]);
                writeEntry(index,dtheEntry,dliterals,dOutput,dOutputSize);
                lookahead=13;
                for(;lookahead + dtheEntry[index].countLiterals<dreadBufSize[index];)
                {
                        //printf("Read: %c\n",dreadBuf[lookahead +
dtheEntry[index].countLiterals]);
                        //printf("loop::%d",lookahead + dtheEntry[index].countLiterals);
                        p=getMaxMatchLength(lookahead +
dtheEntry[index].countLiterals,index,
                                        dreadBuf,dhashTable,dreadBufSize[index]);
                        if( p.first >=4 )
                        {
                                dtheEntry[index].matchLength = p.first;
                                dtheEntry[index].offset = lookahead - p.second;
                                //theEntry.offset = pii.second;
                                sl = dtheEntry[index].countLiterals +
dtheEntry[index].matchLength;
                                writeEntry(index,dtheEntry,dliterals,dOutput,dOutputSize);
                                //slideWindow(sl);
                                i += sl;
                                while( i - start >= MAX_OFFSET)
                                {

        removeHash(start,index,dreadBuf,dhashTable,dreadBufSize[index]);
                                        start++;

                                }
                                for(j=0;j<sl;j++)
                                {

        insertHash(lookahead,index,dreadBuf,dhashTable,dreadBufSize[index]);
                                        lookahead++;
                                }
                        }
                        else
                        {
                                //add to literal list
                                dliterals[index*MAX_LITERALS +
dtheEntry[index].countLiterals] = dreadBuf[index*oneMB + lookahead +
dtheEntry[index].countLiterals];
                                dtheEntry[index].countLiterals++;
                        }
                        if(dtheEntry[index].countLiterals == MAX_LITERALS)
                        {
                                sl=dtheEntry[index].countLiterals;
                                writeEntry(index,dtheEntry,dliterals,dOutput,dOutputSize);
                                //slideWindow(sl);
                                i += sl;
```

```
                              while( i - start >= MAX_OFFSET)
                              {

        removeHash(start,index,dreadBuf,dhashTable,dreadBufSize[index]);
                                  start++;
                              }
                              for(j=0;j<sl;j++)
                              {

        insertHash(lookahead,index,dreadBuf,dhashTable,dreadBufSize[index]);
                                  lookahead++;
                              }
                          }
                      }
                      if(dtheEntry[index].countLiterals)
                          writeEntry(index,dtheEntry,dliterals,dOutput,dOutputSize);
        }
}
```

16.2 DJB Hash Implementation

```
/**
 * Assumes the 3 indices starting at I are valid in
 * read buffer.
 * @i represents the index in readBuf to start hashing from
 */
int hash(int i)
{
        int key = 5381;    //Magic Number
        for (int j = 0; j < 4; j++)
        {
           key = (key *33) ^ readBuf[(j+i) % READ_BUFFER_SIZE];
           key %= HASH_SIZE;
        }
        return key;
}
```

16.3 Murmur Hash Implementation

```
/**
 * Assumes the 3 indices starting at I are valid in
 * read buffer.
 */
int hash(int i)
```

```
{
        // Note: In this version, all integer arithmetic is performed with unsigned 32 bit integers.
        //      In the case of overflow, the result is constrained by the application of modulo
arithmetic.

        const uint c1 = 0xcc9e2d51;
        const uint c2 = 0x1b873593;
        const uint r1 = 15;
        const int len = 3;
        uint hash = 0;
        uint b1 = readBuf[i];
        uint b2 = readBuf[(i + 1)% READ_BUFFER_SIZE];
        uint b3 = readBuf[(i + 2)% READ_BUFFER_SIZE];
        uint remainingBytes=(b1<<16) | (b2<<8) | b3;

        remainingBytes = remainingBytes * c1;
        remainingBytes = (remainingBytes << r1) | (remainingBytes >> (32 - r1));
        remainingBytes = remainingBytes * c2;

        hash = hash ^ remainingBytes;

        hash = hash ^ len;

        hash = hash ^ (hash >> 16);
        hash = hash * 0x85ebca6b;
        hash = hash ^ (hash >> 13);
        hash = hash * 0xc2b2ae35;
        hash = hash ^ (hash >> 16);
        return hash % HASH_SIZE;
}
```