Faster Image Compression (LZW algorithm) Technique Using GPU Parallel Processing

Ateeq-Ur-Rahman Soobhee 13301025

Kamrun Nahar Ruma 13101035

Md. Fakhrul Ahsan 14201050

F. M. Fahmid Hossain 13301018

Supervisor: Md. Ashraful Alam

Co-Supervisor: Md. Saiful Islam

Department of Computer Science and Engineering

BRAC University

Submission of this report was done according to the requirements of the Degree of Bachelors in Computer Science and Engineering in the Department of Engineering and Computer Science.

Submitted on:

26th DECEMBER, 2017

DECLARATION

We, hereby declare that this thesis report is derived from our own work along with endeavor and also it has not been submitted anywhere for any sort of recognition yet. Our dedication towards the successful completion of the thesis resulted in arranging all the required content materials for the thesis purpose. Other sources of information have been acknowledged whenever those were required to be mentioned or utilized and reference section at the end of the report holds all the sources of information that were cited throughout the report.

Signature of Supervisor:

____

Dr. Md. Ashraful Alam

Assistant Professor

Department of Computer Science

and Engineering

BRAC University

Signature of Co-Supervisor:

____

Md. Saiful Islam

Lecturer

Department of Computer Science

and Engineering

BRAC University

Signature of Authors:

1. Ateeq-Ur-Rahman Soobhee ________________________

2. F. M. Fahmid Hossain ________________________

3. Kamrun Nahar Ruma ________________________

4. Md. Fakhrul Ahsan ________________________

ACKNOWLEDGEMENT

To start off, we would like to convey our deepest sense of heartfelt gratitude towards the Almighty Allah. Secondly, we would love to share our sincerest gratitude to our advisor Dr. Md. Ashraful Alam and our co-advisor Md. Saiful Islam for their constant support, patience, motivation, and immense knowledge in our research. Their guidance and assistance helped us in all parts of the progress. Also, special thanks to Mr. Mahfuze Subhani for his immense guidance throughout the time.

Finally, we must express our cordial gratefulness to our beloved parents, brothers and sisters for ensuring an environment full of peace and harmony around us. And last but not the least; we are grateful to all of our friends including Mr. Emrul Kais who helped us sincerely to complete our thesis.

Table of Content

Declaration

Acknowledgement

Abstract

Keywords

Chapter 1: Introduction

1.1 Introduction..1

Chapter 2: LZW Algorithm and GPU Parallel Processing

2.1 LZW Overview..3

2.2 Algorithm..3

2.3 Encoding..3

2.4 Decoding..5

2.5 Background Information of GPU..6

2.6 GPU Architecture..7

2.6.1 Tesla Architecture..7

2.6.2 Fermi Architecture..8

2.6.3 Kepler Architecture..8

2.6.4 Maxwell Architecture..9

2.6.5 Pascal Architecture..9

2.7 CUDA..10

2.7.1 Units of CUDA..11

2.7.2 Memory Units in CUDA..14

2.7.3 CUDA C..15

2.8 Difference Between CPU and GPU Processing......................................16

Chapter 3: Proposed Method

3.1 Proposed Method......................................18

Chapter 4: Result and Discussion

4.1 Result and Discussion......................................21

Chapter 5: Conclusion......................................27

References......................................28

List of Figures

Figure 1: Encoding steps....4

Figure 2: Decoding steps....6

Figure 3: Streaming Multiprocessor (SM)....7

Figure 4: Processing Flow on CUDA....11

Figure 5: 1D Grid with 2D Blocks and Threads....11

Figure 6: 1D Block with 2D Threads (3x2)....12

Figure 7: Global Thread ID generation from 1D block....13

Figure 8: Memory model of CUDA....15

Figure 9: Typical architecture of a CPU....17

Figure 10: Typical architecture of a GPU....17

Figure 11: Input images....21

Figure 12: Lossless Picture Quality....23

Figure 13: Identifying Input and Output file....23

Figure 14(a): Comparison between original image size and compressed size....24

Figure 14(b): Comparison between original image size and compressed size....25

Figure 15: Comparison between execution time of CPU and GPU....26

List of Tables

Table 1: Difference between CPU and GPU..16

Table 2: Size comparison between original and compressed file......................................24

Table 3: Comparison between execution time of CPU and GPU.......................................25

ABSTRACT

Since the beginning till present, the technology demands to store as massive data as possible in as little space as possible. As web, mobile, desktop and all other applications use image for different purposes, image compression technique has become one of the most important applications in image analysis as well as in computer science. Though image compression is an old concept, yet it's considerably time consuming processes has opened a new field of research in image compression. In this paper, LZW (Lempel-Ziv-Welch) algorithm which is a lossless image compression algorithm with the implementation of parallel processing for faster computation has been proposed. As a consequence, the experimental result verifies much faster and satisfactory computation time in millisecond scale than the conventional technique along with keeping the decoded image in lossless format.

Keywords: *image compression, LZW, CUDA, GPU, parallel processing, DICOM*

Chapter 1: Introduction

1.1 Introduction

Image compression means compressing or reducing the image file in such a way that we don't have to loss any data to an objectionable level. We need image compression to lessen the cost of the storage and also for a fast transmission. More data can be stored in the memory space if we get our images compressed and transmission will be faster because of the reduced size of the image.

An image can be compressed in two ways; lossy and lossless. Lossy methods are usually used for natural images where imperceptible loss of precision is acceptable to achieve a significant amount of reduction in bit rate. The negligible differences that a lossy compression produces may also be called visually lossless. On the other hand, the lossless compression is preferred for archival purposes and often for medical imaging, technical drawings, clip art, or comics.

Our goal is to reduce a given file to a file with significantly less size with zero data loss and ensure faster encoding time. To ensure zero data loss we used LZW ((Lempel-Ziv & Welch)) algorithm. At first, this algorithm takes string of characters and converts into a string of using dictionary which is actually a code table that can map string into codes. By adding new substring the code table is made. Then, the input string is being converted into a string of characters. We have used a TIFF, GIF, DCM file as an input and after compressing and decompressing we get the output data same as input.

Tagged Image File format or TIFF is used to maintain image quality and file security.
The medical imaging system also known as DICOM (DCM file extension) image. The format ensures that all the data stays together, as well provides the ability to transfer said information between devices that support the DICOM format. The idea of big data is already been introduced and in the recent future, it is going to be used in almost every sectors possible. As a DCM file is saved on a disk or flash drive, so when big data will be used the companies or industries responsible for managing these files will have to spend a lot. [13]
GIF file format is commonly used for images on the web and in the software programs.
For building and manipulating images, GPU is used which will accelerate the computation. The applications which are traditionally handled or manipulated by the CPU can also be processed by the GPU.A parallel computing architecture called CUDA(Compute Unified Device Architecture), provided by NVIDIA, and is the computing engine for NVIDIA GPUs.

As both CPU and GPU can compute parallel we believe this will help us to compute in image processing. As the computation will be faster this will give a lesser encoding time for our work. That is why to make the encoding time lesser, we have used parallel processing in CUDA-enabled GPU.

Chapter 2: LZW Algorithm and GPU Parallel Processing

2.1 LZW Overview

Lemple-Ziv-Welch (LZW) is an errorless, lossless general purpose data compression algorithm, which is simple to execute as well as versatile at the same time. The LZW is mostly effective where repeated patterns are available in the data. Whenever a new pattern or substring is found, it is placed in the dictionary so that it will be easy to find next time. The advantage of this dictionary is that it will stop repetition of the same substring. Also because of the same reason, there will be such entries in the dictionary that might never be used. LZW addresses spatial redundancies in an image. By spatial we mean to say that the elements that are duplicated within a structure, such as pixels in a still image and bit patterns in a file as well. This algorithm can be implemented on few popular formats like GIF, TIFF, DCM etc. [12]

2.2 Algorithm

The algorithm generates a dictionary or code table, in each stage step of compression it takes the input bytes as a sequence. if the sequence is not in the dictionary, the sequence will be added. This way the codes 256 through 4095 will be created in the dictionary, as the algorithm process proceeds. LZW may have a dictionary-entry that will be never used.

2.3 Encoding

The LZW starts working with a dictionary of 256 characters where these are arranged in the case of 8 bits. Then it uses those characters as the standard character set. After that the algorithm consider 8 bits as a single character and read data all the 8 bits at a time and encode the data by replacing with the number it represents as the index in the dictionary. Whenever it finds a new string or substring, it adds it to the dictionary; and every time it finds an existing substring it just reads in a new character and concatenate this string with the current on to create a new substring. Later, if the LZW again finds the substring, it will be replaced with a single number. Normally, a maximum number is assigned to a dictionary so that, the process does not run away with its memory. It is mandatory for a code to have a bigger bit than a character although, many substrings are created frequently and being replaced by a single code, the ultimate goal of compression will be achieved.

The following pseudo code [own defined] which refers our encoding procedure:

Char*c*;

String*s*= empty string;

while (input)

{

c = read a character;

if (dictionary contains $s+c$)

{$s = s+c$;}

else

{

encode*s* to output;

add*s*+*c* to dictionary;

$s = c$;

}

}

encode s to output;

The encoding steps would follow the steps like showed in Figure 1:

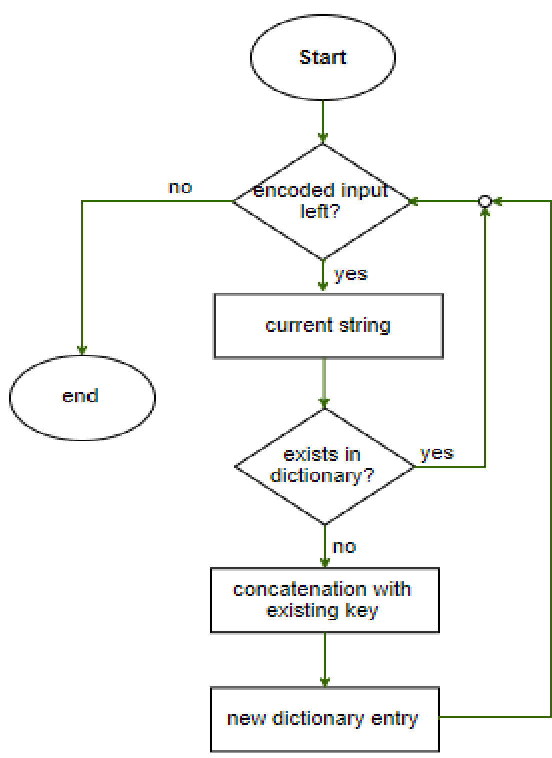


Figure1: Encoding steps

2.4 Decoding

The decompression process follows a straightforward direction for LZW. It can use the dictionary which was created while encoding so, no need of dictionary creation. In the decoding process the first step is that the LZW decoder takes a numeric number which indicate the value in the dictionary. The first character of this substring is concatenated to the current working string. This new concatenated string is added to the dictionary. After that, the decoded string becomes current working string and the process repeats.

The following pseudo code [own defined] which refers our decoding procedure:

```
string s;
charc;
while (input from code stream)
{
        c = read character from input;
        if (c exists in the dictionary)
                {
                        c will be output;
                        add s+ c[0] in dictionary;
                        s=c;
                }
        else
                {
                        s+s[0] will be output;
                        s=c;
                }
        s will be output;
}
```

The decoding steps would follow the steps like showed in Figure 2:

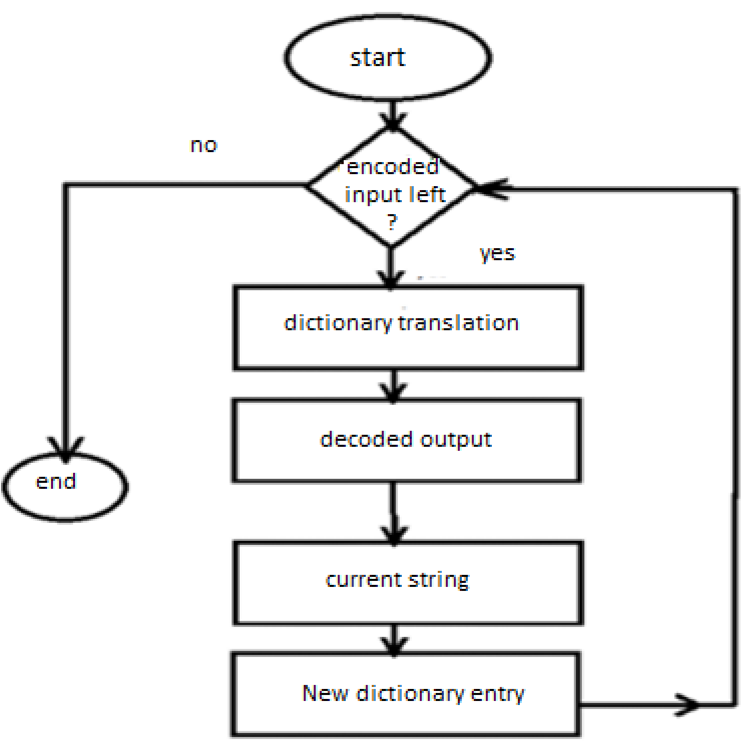


Figure 2: Decoding steps

2.5 Background Information of GPU

A GPU (Graphics Processing Unit) is a specific electronic device which is specially designed to manipulate and alter memory to increase the speed of creation of images in a frame buffer intended for output to a display device rapidly. Modern Graphics Processing Units are very efficient at manipulating computer graphics as well as image processing. The GPUs are more efficient than the general purpose CPUs because of their highly parallel structure. GPU complete this type of operations in such algorithms where the processing of large blocks of data can be done in parallel. The term GPU was popularized by NVIDIA in 1999, who marketed the “GeForce 256” as "the world's first GPU", or Graphics Processing Unit. The “GeForce 256” was introduced as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines which is capable of processing a minimum of 10 million polygons per second". GPU accelerated computing the portion of a program where a large block is used and a normal CPU takes much time to compute it but the remainder of the code still runs on the CPU. From a user’s perspective, applications simply run much faster in a GPU. A GPU’s computing is much faster than that of a CPU’s because GPU has a massively parallel architecture including more efficient and thousands of smaller

cores functioned for multiple tasks (multi-threading) simultaneously whereas CPU consists of few cores optimized for sequential serial processing.

2.6 GPU Architecture

There are few differences between GPU and CPU processor design. NVIDIA's GPU having variations among streaming multiprocessor (SMs) and each of these streaming processors comprise of numerous scalar processors also known as cores. NVIDIA introduced us with different types of GPU architecture such as Kepler, Fermi and Tesla etc. Figure 3 shows the architecture of streaming multiprocessor.

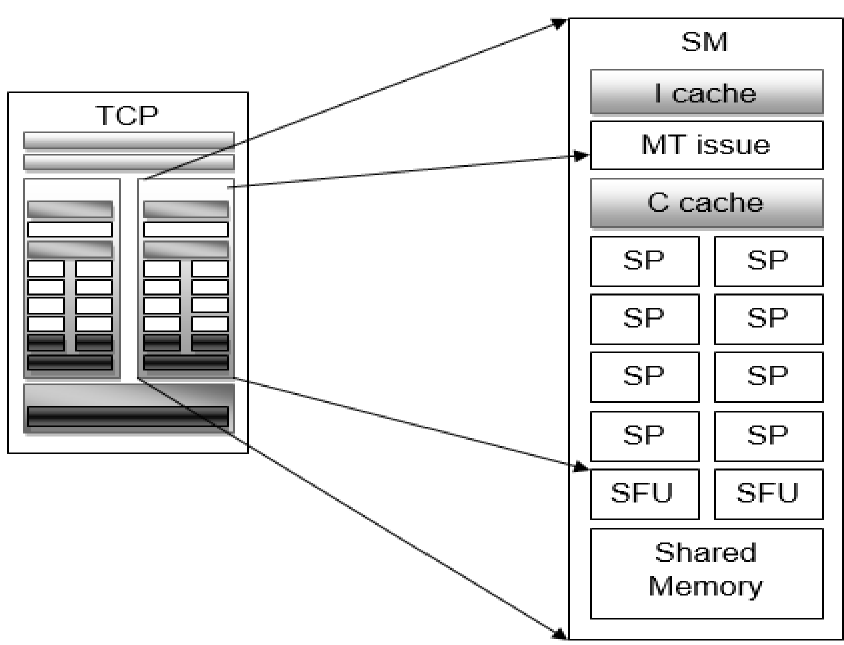


Figure 3: Streaming Multiprocessor (SM)

2.6.1 Tesla Architecture

Tesla is NVIDIA's first introduced micro architecture which a unified Shader model. In this architecture, the driver supports Direct3D Shader model 4.0 / OpenGL 2.1 (later drivers have OpenGL 3.3 support) architecture. The design is a major turnover for NVIDIA in GPU functionality and capability, because they were moving from the separate functional units like pixel shaders, vertex shaders etc. within previous GPUs to a homogeneous collection of universal floating point processors also called Stream processors. These floating point

processors can perform a more universal set of tasks. The Tesla was the first GPU which had unified shader with 128 processing elements which distributed in 8 shader cores. [5]

2.6.2 Fermi Architecture

After Tesla, NVIDIA introduced with its successor and named it Fermi. Fermi is another GPU multiprocessor architecture like Tesla. G80 was the result of NVIDIA's initial vision of what a unified graphics and parallel processor should look like. G200 was the upgraded version of G80. It extended the performance and functionality of G80. It was world's first computational GPU which was designed with an entirely new approach. When NVIDIA started the groundwork for both G80 and G200, they started gathering extensive feedback from the users and marked some area for further improvement such as ECC support, faster atomic operation, faster context switching, true cache hierarchy, more shared memory and to improve double precision performance.

The very first Fermi architectured GPU, which featured up to 512 CUDA cores, was created with 3.0 billion transistors. All these transistors were implemented in 16 SMs (Streaming Multiprocessors) of 32 cores each. For a 384-bit memory interface, the GPU has six partitions of 64 bit each which helps the GPU to support up to a total of 6GB of GDDR5 DRAM memory. [5]

2.6.3 Kepler Architecture

Kepler is the codename for a NVIDIA GPU micro architecture which is the successor of the Fermi micro architecture. It was first introduced in April, 2002. This engineering was mainly focused on energy efficiency or power effectiveness. Most GeForce 600 series to 700 series and some of GeForce 800 arrangements were manufactured in 28 nm depending on Keplermicro architecture [5]. When compared to older NVIDIA GPUs, Kepler helped newer versions to achieve a memory clockspeed of 6GHz. Kepler based NVIDIA members ensure some following standard features:

- Next generation Streaming Multiprocessor
- CUDA compute capability 3.0 to 3.5
- Polymorph-Engine 2.0
- PCI Express 3.0 interface
- Dynamic parallelism

- NVIDIA GPUDirect

2.6.4 Maxwell Architecture

Maxwell is an NVIDIA GPU micro architecture which is considered as the upgraded version of Kepler rmicro architecture. GeForce GTX 750 and GeForce GTX 750 Ti were the very first Maxwell based products launched by NVIDIA in the market. The models where the Maxwell architecture was introduced to were all manufactured in 28 nm. The devices which were designed on the basis of Maxwell architecture were specially functioned for power efficiency. The L2 cache used in Kepler was of 256 KiB which later on increased to 2 MiB on Maxwell. This massive increase in memory resulted the reduction for memory bandwidth. After this less bandwidth architecture was provided, the memory bus was reduced to 128 bit which used to be of 192 bit on Kepler and resulting more power saving. An improved Streaming Multiprocessor (SM) was provided using Maxwell architecture and influenced more power saving. The improved SM (Streaming Multiprocessor) on Maxwell architecture was renamed to SMM. The structure of the warp scheduler was taken from Kepler, with the texture units and FP64 CUDA cores still shared, but the layout of most execution units were divided so that each warp schedulers in an SMM controls one set of 32 FP32 CUDA cores, one set of 8 load/store units and one set of 8 special function units. This is in contrast to Kepler, where each SMX has four schedulers that schedule to a shared pool of execution units. [5]

2.6.5 Pascal Architecture

After Maxwell micro architecture NVIDIA developed a new micro architecture named Pascal. It considered as the Successor of Maxwell micro architecture. The first Pascal micro architecture was reveled in the market on April 2016 and the first chip was GP100. The architecture name was derived from a 17th century mathematician Blaise Pascal. The Streaming multiprocessors in different GPU micro architecture consist different number of CUDA cores. For example, Tesla had only 8, Fermi had 32, Kepler had 192 and Maxwell had 128 CUDA cores; where Pascal consists of 64 CUDA cores. The Pascal micro architecture based GP100 SM consists of two different processing blocks with having 32 single precision CUDA core each. It also has a wrap scheduler, an instruction buffer, 2 dispatch units and 2 texture mapping units. The Maxwell micro architecture supports CUDA compute capability 6.0. [5]

2.7 CUDA

CUDA was first introduced by NVIDIA in 2007. It is a general purpose computational programming which is specialized in parallel GPU architectures. CUDA which stands for Compute Unified Device Architecture is a NVIDIA GPU architecture which is in GPU (Graphics Processing Unit) card. CUDA uses extension of C++ also referred to CUDA C for programming purpose. CUDA has a huge advantage in computational power to the programmers and it is being liked by them since it provides a lot of freedom to work on a broader basis. Depending on the GPU model, CUDA also provides many co-operating cores. These cores can communicate and also exchange information among each other as a result; running multithreaded application there is no need for streaming computing in GPU.

As mentioned earlier, CUDA follows C programming language and able to works in such way that the thousands of threads used in CUDA can perform parallel. All of these threads can perform or execute a lot of functions or codes at the same time. These threads execute same codes on different data. It has introduced us with a whole new meaning for general purpose computing with GPUs. CUDA programs consists of one or two parts of a code that executes in the device (GPU) whereas, rest of the portion executes on the host (CPU). It is preferable to execute some parts of a code in GPU where a huge parallelism is needed otherwise in terms of little multithreading it is wiser to use CPU since copying data from host to device is time consuming. In large scale parallelism the GPU overcomes copy speed and provides performance boost.

Figure 4 describes how the whole process works step by step. At first we run main method form host (CPU). From host memory the processing data is copied to memory of GPU. At the same time CPU gives instruction of processing to the device (GPU).after that GPU cores or threads take part in executing the GPU-enabled CUDA code simultaneously. In the next step, the executed result is stored in the memory of GPU. At last, the device memory sends the result to CPU memory which is the result of the portion implemented in GPU.

Figure 4 clearly describes the above process:

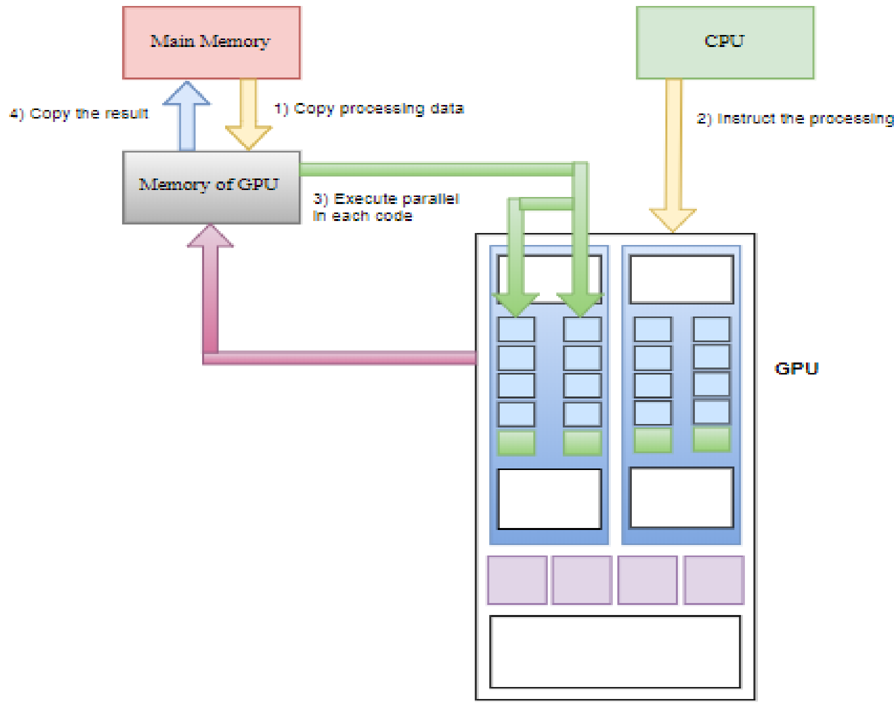


Figure 4: Processing Flow on CUDA

2.7.1 Units of CUDA

Grids

Grid is a group of all the threads which are running in the same kernel at the same time. GPUs cannot share grids. It is not possible to synchronize between the blocks within a grid. Grids can be executed only in the part of GPU where an entire grid is handled by a single GPU [5]. Figure 5 shows architecture of a grid:

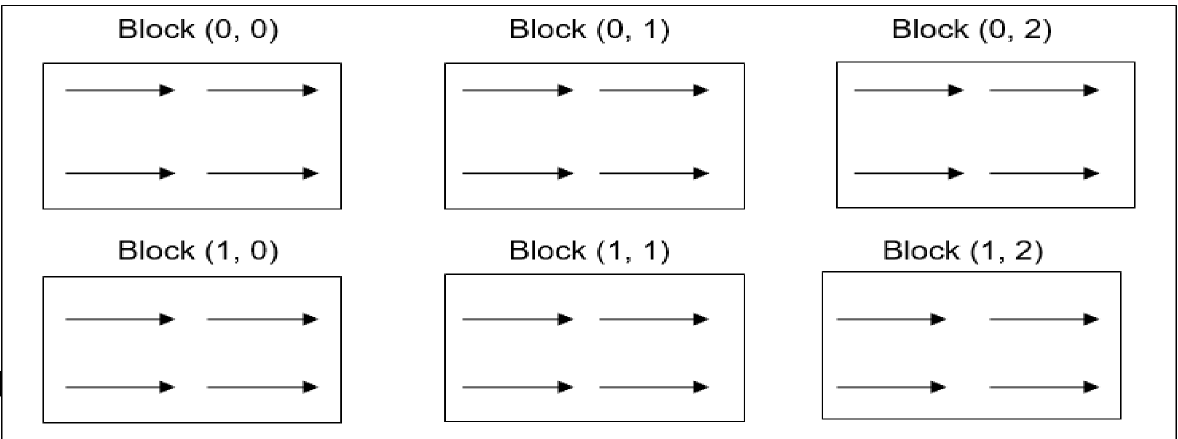


Figure 5: 1D Grid with 2D Blocks and Threads

Blocks

A lot of blocks are compressed in a single grid whereas each block is composed of multiple threads which could execute concurrently or serially and in no specific order but all the grids in a thread uses the same program in a block. A single block has unique ids and these ids can be interpreted as 1D, 2D or 3D according to the architecture. Multiprocessors (MPs) are the place where blocks are executed. A GPU chip is designed as a combination of multiprocessors (MPs) where each multiprocessor is responsible for executing one or more blocks in a grid. A block can never be shared or divided between multiple multiprocessors (MPs). [5] Figure 6 shows the architecture of a block in GPU:

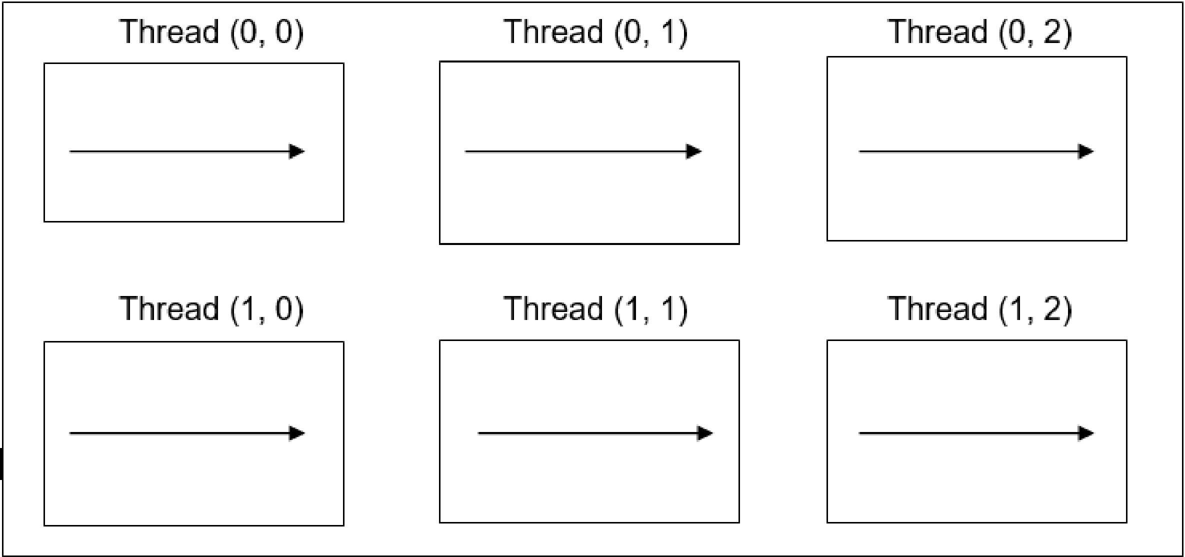


Figure 6: 1D Block with 2D Threads (3x2)

Threads

Each block consists of multiple threads. It is only the execution of a kernel with a provided index. Every thread work with a unique index to access elements in array such that all the threads in a block can process the entire data set combined. Threads can run with an individual core of microprocessor although they are not bound to only with multiple microprocessors. A thread has its own ID and it can be interpreted as 1D, 2D or 3D depending on block pattern. Threads are mainly executed in stream processors (SPs). Every Multi Processor is further separated into a number of SPs. With each stream processors the handling one or more threads in a block is possible. If a multiprocessing unit can run 768 threads, for example and a GPU device has 4 of them; then within a given time not more than 4*768 threads will run parallel.

CUDA built-in variables

● **blockIdx.x, blockIdx.y, blockIdx.z** are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.

● **threadIdx.x, threadIdx.y, threadIdx.z**are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.

● **blockDim.x, blockDim.y, blockDim.z** are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis)

Thread identification and manipulation

We must have a good idea about the proper manipulation of the blocks and threads in order to carry out intense calculations in the most efficient manner so that we can make the best use of the features of CUDA.

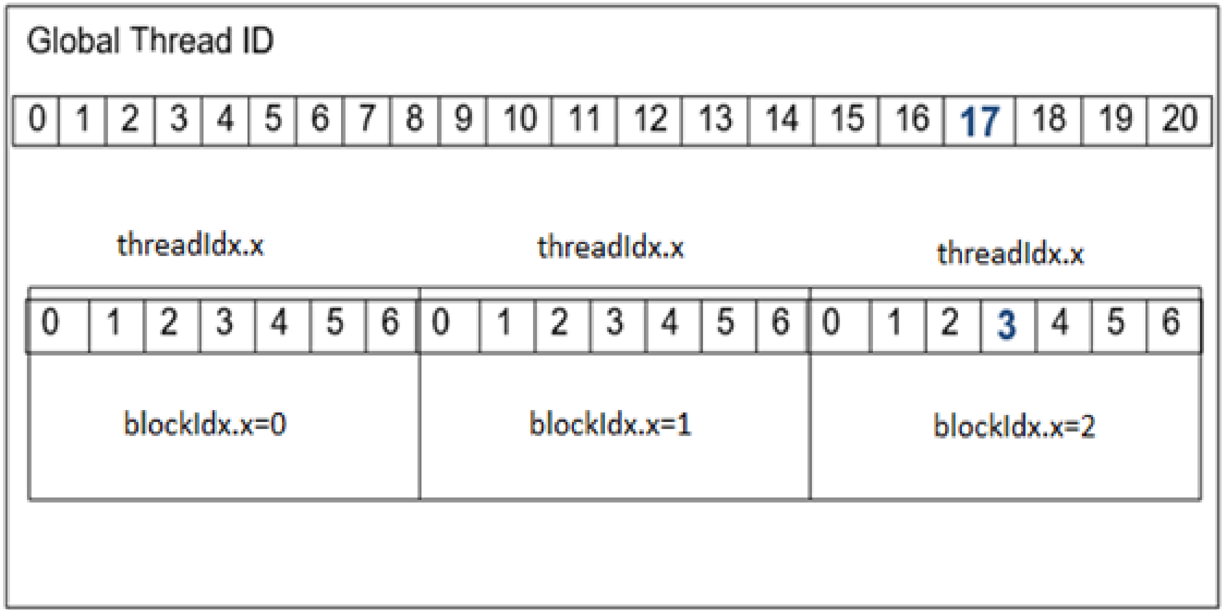


Figure 7: Global Thread ID generation from 1D block

Assume a hypothetical 1D grid and 1D block architecture: 3 blocks, each with 6 threads.

So for example, if we want to calculate the global thread ID of 26:

● gridDim.x = 3 x 1

● blockDim.x = 7 x 1

● Global Thread ID = (blockIdx.x * blockDim.x) + threadIdx.x

● = (2 x 7) + 3 = 17

2.7.2 Memory units in CUDA

Global memory: SDRAM chips are the source of this type of memory which is connected to the GPU chip. Any multiprocessor thus any thread within a multiprocessor can read or write from or to any location in the global memory. It also may call device memory.

Texture cache: This type of memory stays in the multiprocessors which can be filled with data from device memory so that it can act a cache. Threads running in the microprocessors are restricted to read-only access of this memory.

Constant cache: It is a read-only memory which lays in each MP.

Shared Memory: Like the other types of memory shared memory lays within a multiprocessor although it is smaller. It can be read/written by any thread in a block which is assigned to that MP.

Registers: each microprocessor has a number of registers that are shared between its SPs. Different types of memory which are available in CUDA like global memory, texture memory, and shared memory are shown in Figure 12. Global memory which is also referred to as device memory is visible to every thread within the same grid in computing with large size. Shared memory is only visible to threads in the same compute block which is very fast to access although it has a much smaller capacity than that of a global memory.

Figure 8 describes the memory model of CUDA architecture where a CUDA-enabled GPU consists of many grids which includes multiple blocks and each block contains number of threads.

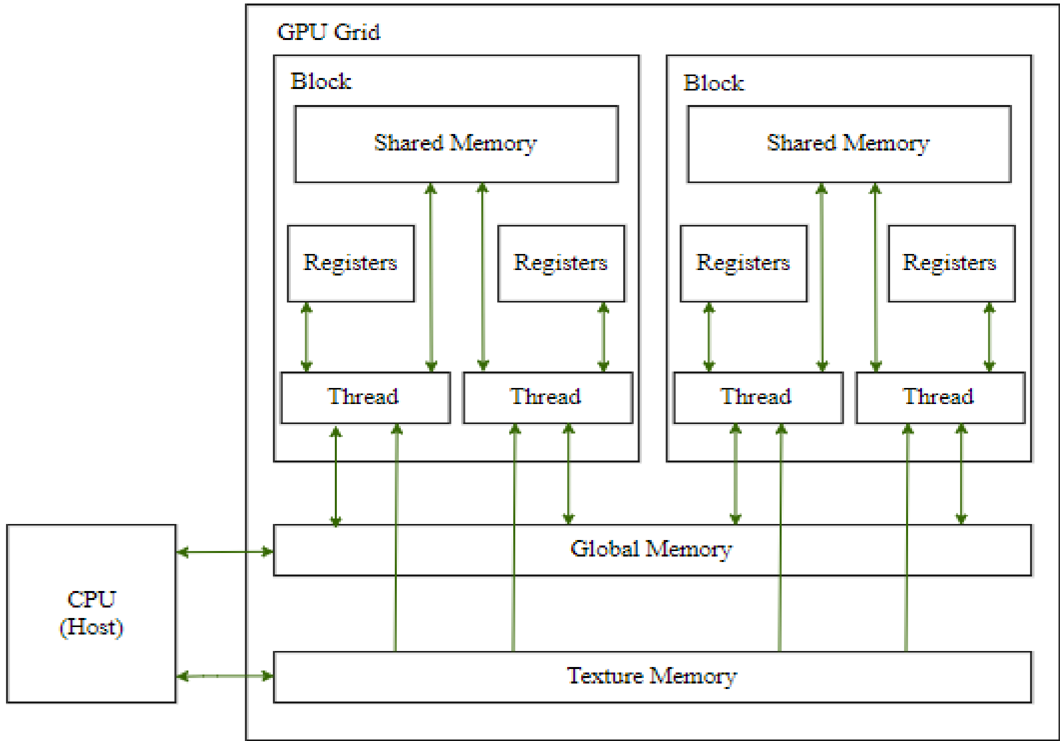


Figure 8: Memory model of CUDA

2.7.3 CUDA C:

Although the term CUDA C indicates the involvement of only C programming language, both the C and C++ languages can be used in CUDA C. Eventually, CUDA C is C/C++ with few extensions which allow programmers to execute a code faster in GPU by running multiple threads in parallel. NVIDIA developed CUDA (Compute Unified Device Architecture) in such a way so that, one can convert a C code in a GPU enable language.

CUDA C uses three types of functions; firstly, the host functions which can only be called and executed by CPU. These are the functions mostly similar to C. Secondly, the functions, the qualifier, _global_ which are only executed by the device and called by CPU itself. These types of functions are called kernel. The return type of kernel function is always void. The final type of function is device functions can be exampled by the qualifier, _device_ which can only be called and run by the device. This type of function can return any type of value.

The built in device variables blockDim, blockIdx and threadIdxused to identify and differentiate GPU threads that execute the kernel in parallel.

2.8 Difference between CPU and GPU Processing

The CPU (Central Processing Unit) is mostly considered as the brain of the PC. Now-a-days, that brain is being replaced by performance by another part of the PC - the GPU (Graphics Processing Unit). All PCs have chips that render the display images to monitors. All these chips are different from each other. Graphics controller provided by Intel provides basic graphics which means it can only display applications like Microsoft PowerPoint, low resolution video, basic games etc. Unlike a graphics controller the GPU is in a class by itself. The GPU by born is a powerful programmable and computational device. The architecture of a CPU is composed to just a few cores and a lot of primary memory that can handle quite a few multithreading software parallel. On the other hand, GPU is composed to hundreds of cores which allow it to execute thousands of threads at a time simultaneously. A GPU with 100+ cores can speed up some software by 1000 times alone although that software can have thousands of threads.

Table 1: Differences between CPU and GPU

CPU	GPU
1. CPU consists of few cores with good amount of cache memory	1.GPU consists of hundreds of cores
2.CPU can handle few software thread at one time	2.GPU can take care of thousands at a time
3.Can reduce latency more efficiently	3.Not as efficient as CPU in reducing latency
4.CPU is not much power and cost efficient	4.GPU is more power and cost efficient
5.Host code runs on CPU	5.CUDA runs on GPU

Figure 9 and 10 shows a typical architecture of a CPU and GPU accordingly:

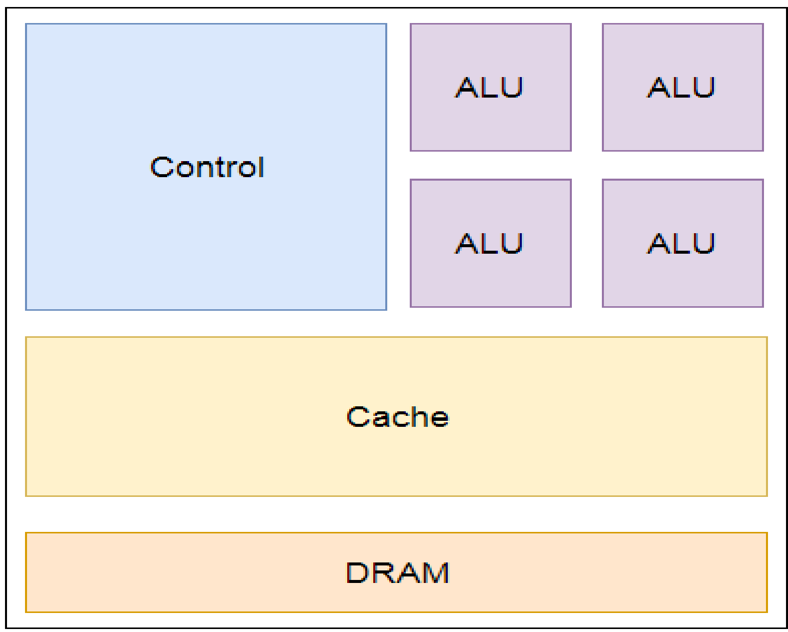


Figure 9: Typical architecture of a CPU

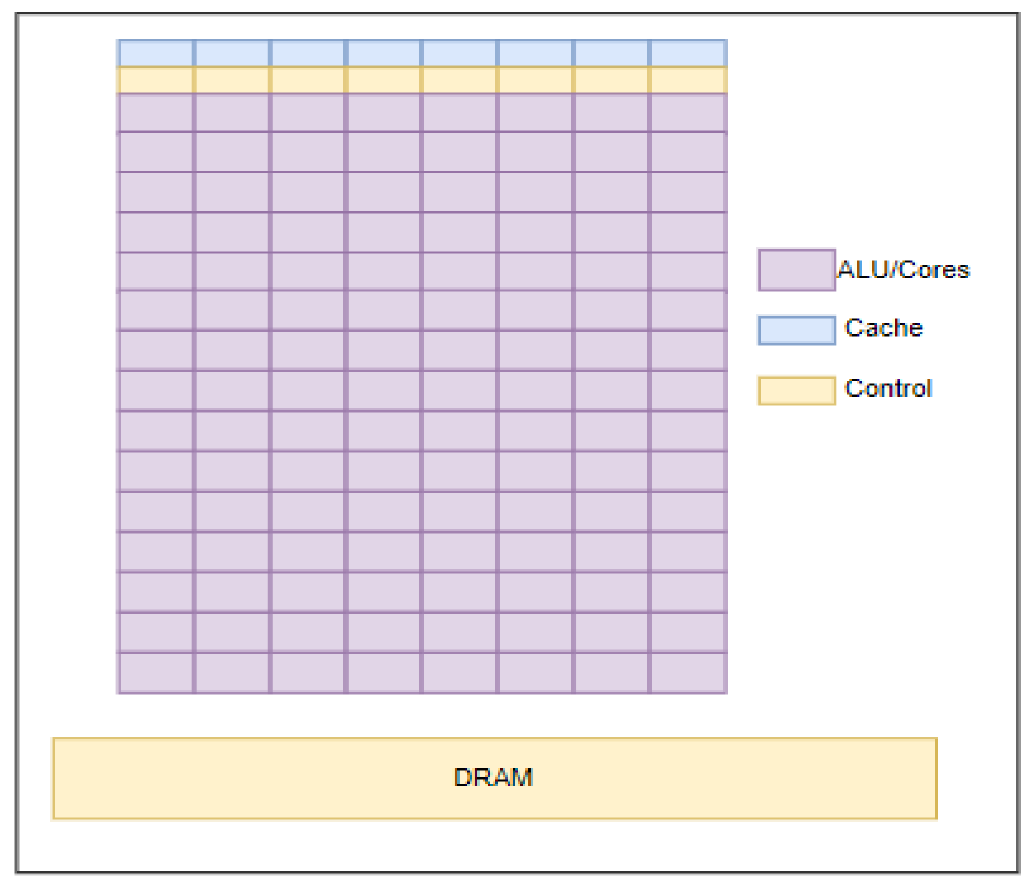


Figure 10: Typical architecture of a GPU

Chapter 3:Proposed Method

3.1 Proposed Method:

The methodology of our whole research work was mainly based upon making the encoding/decoding time more efficient with the help of CUDA technology introduced by NVIDIA but we also took space complexity as one of our concerns.

Image can be compressed in two ways. One of them is lossless and another one is lossy. LZW compression is a lossless compression method. It takes string of characters as input and convert the input into a string of codes. To do so, the process uses code table (or dictionary) that maps strings into codes. When a new sub string appears, the code table is updated. Since generating the code table is a repeated process, it is very hard to parallelize LZW compression conventionally using a parallel processing in CUDA-enabled GPU, we can make the encoding time of the algorithm faster the application or calculation which is supposed to be done by the CPU can be done fast if we can use GPU to compute simultaneously. As both CPU and GPU can calculate simultaneously, this combined calculation can help us to process image providing shorter encoding time. Our goal is to implement LZW compression algorithm using several acceleration techniques using CUDA, although it is a very hard task. Suppose that a GPU generates a compressed image generated by a computer graphics or image processing CUDA program and we want to archive it as a LZW-compressed DCM image in the SSD connected to the host PC. The image is transferred to the host PC, and compressed and written in the SSD using a CPU.

Before starting to code on CUDA programming language we had to study the architecture and operational features of the Graphics Card that we used to run our complete code on(i.e.: Collecting information about Grids, Blocks and Threads).

Moreover, we had to learn about the complete computation process of the LZW algorithm and its processing steps. To run the compression algorithm comprehensively at a faster rate, firstly we analysed the portions of our C code that had to be implemented with CUDA code. Because, we only can visually feel the difference in computational ability between GPU and CPU when the GPU code is set on the areas where there are large amount of calculations, needed to be processed (i.e.: loop)

After running both the codes for 5 times consecutively we then analyse the execution time for both the codes in a chart, which gives the comparison between the CUDA code and C code .It also gives us idea about the average time required to execute the whole system.

At first, we learned about the advantages of different compression algorithms such as DCT, JPEG, Huffman Coding, LZW and so on. But from these, we chose LZW because it is a lossless compression algorithm. And LZW algorithm always provides efficiency in terms of space complexity and time complexity both for the case of files that have redundancy in data. Since .dcm files have redundancy in colour code, we chose this to be as our dataset of proposed method. Also, .dcm file is widely used in medical imaging and any loss of data is not suitable in medical imaging, hence we chose LZW algorithm to be run on parallel computation process.

The proposed method was verified by bus through a successful completion of an experiment. The experiment was conducted on a PC, its photo is attached below.

The specification of the PC is given below:

● Microsoft Windows 8.1 Pro64 bit

● Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 3401 MHz, 4 Core(s), 8 Logical Processor(s)

● 16 GB RAM

NVIDIA GeForce GTX 660

● 2GB GDDR5 Memory

We used the NVIDIA GeForce GTX 660 as our GPU needed to operate parallel processing.

Advantages of NVIDIA GeForce GTX 660:

1) Performance is very pleasing when the fact of price is kept in mind

2) Common markets are introduced to kepler technologies by this GPU

Disadvantages of NVIDIA GeForce GTX 660:

1)AMD GPU of the same price range perceives much faster computational speed than this GPU

2)AMD GPUs of the same specifications are much more power efficient than this NVIDIA GPU.

GPU Engine Specs:

❑960 CUDA Cores

❑980 Base Clock (MHz)

❑1033 Boost Clock (MHz)

❑78.4Texture Fill Rate (billion/sec)

Memory Specs:

❑6.0 Gbps Memory Clock

❑2048 MB Standard Memory Config

❑GDDR5 Memory Interface

❑192-bit GDDR5 Memory Interface Width

❑144.2Memory Bandwidth (GB/sec) 30

Feature Support:

❑GPU Boost, PhysX, TXAA, NVIDIA G-SYNC-ready Important Technologies

❑ 3D Vision, CUDA, Adaptive VSync, FXAA, 3D Vision Surround, SLI Other Supported Technologies

❑4.3OpenGL

❑12 API Microsoft DirectX

❑PCI Express 3.0 Bus Support

❑Yes Certified for Windows 7

❑Yes 3D Vision Ready

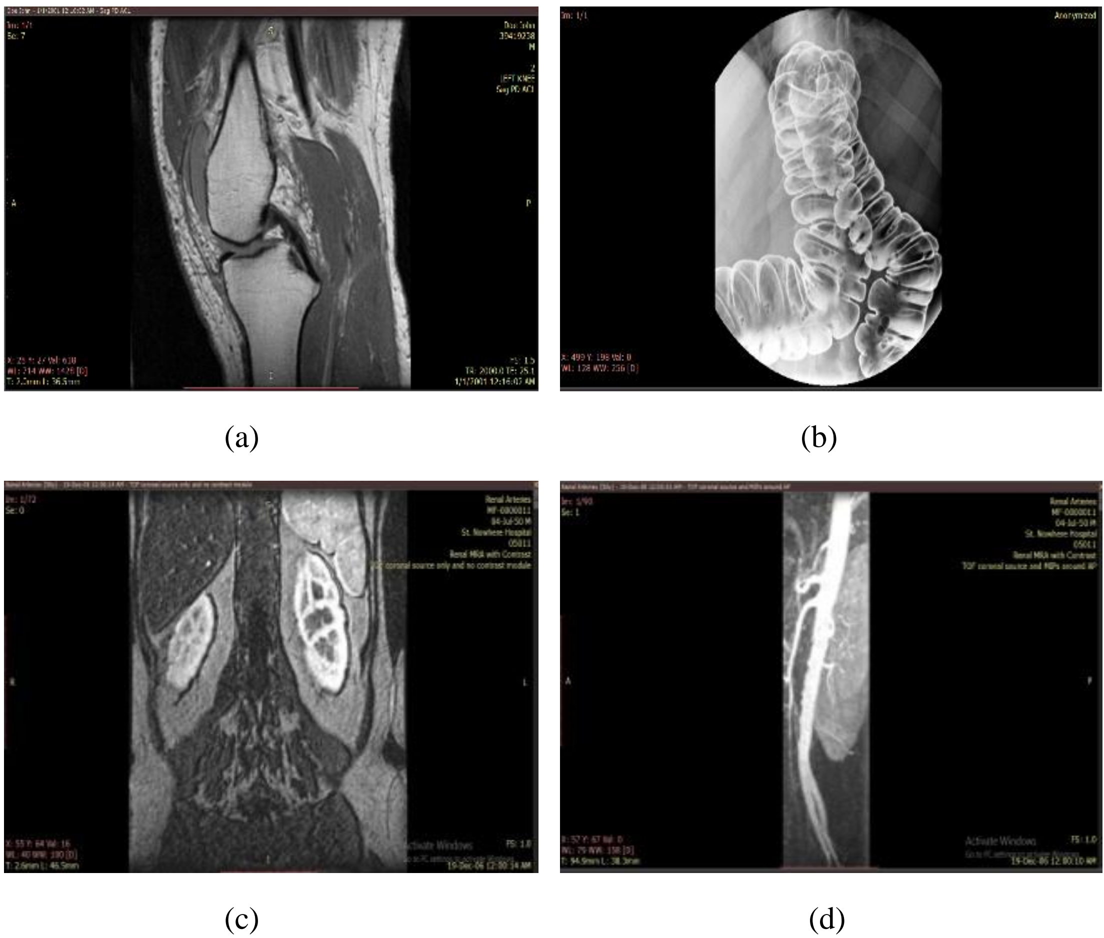
Chapter 4: Results

4.1 Result and Discussion:

Our proposed method is successfully verified because we have found significant amount of decrease in compression and decompression time of LZW(Lempel–Ziv-Welch) compression algorithm after the inclusion of CUDA programs in our implementation. The process was comprehensively run and programmed under Microsoft Visual Studio 2010 and CUDA 5.5 in the Operating System of Windows 8.1 .We took .dcm (DICOM) format files as our input of the compression algorithm. For comparison the implementation was done in two different ways, one with pure ANSII C code and another with the inclusion of CUDA programs in that C code. When the two different codes were fed with our given .dcm format files, we found significant amount of increase in efficiency time in terms of comparing the CUDA code with the C code.

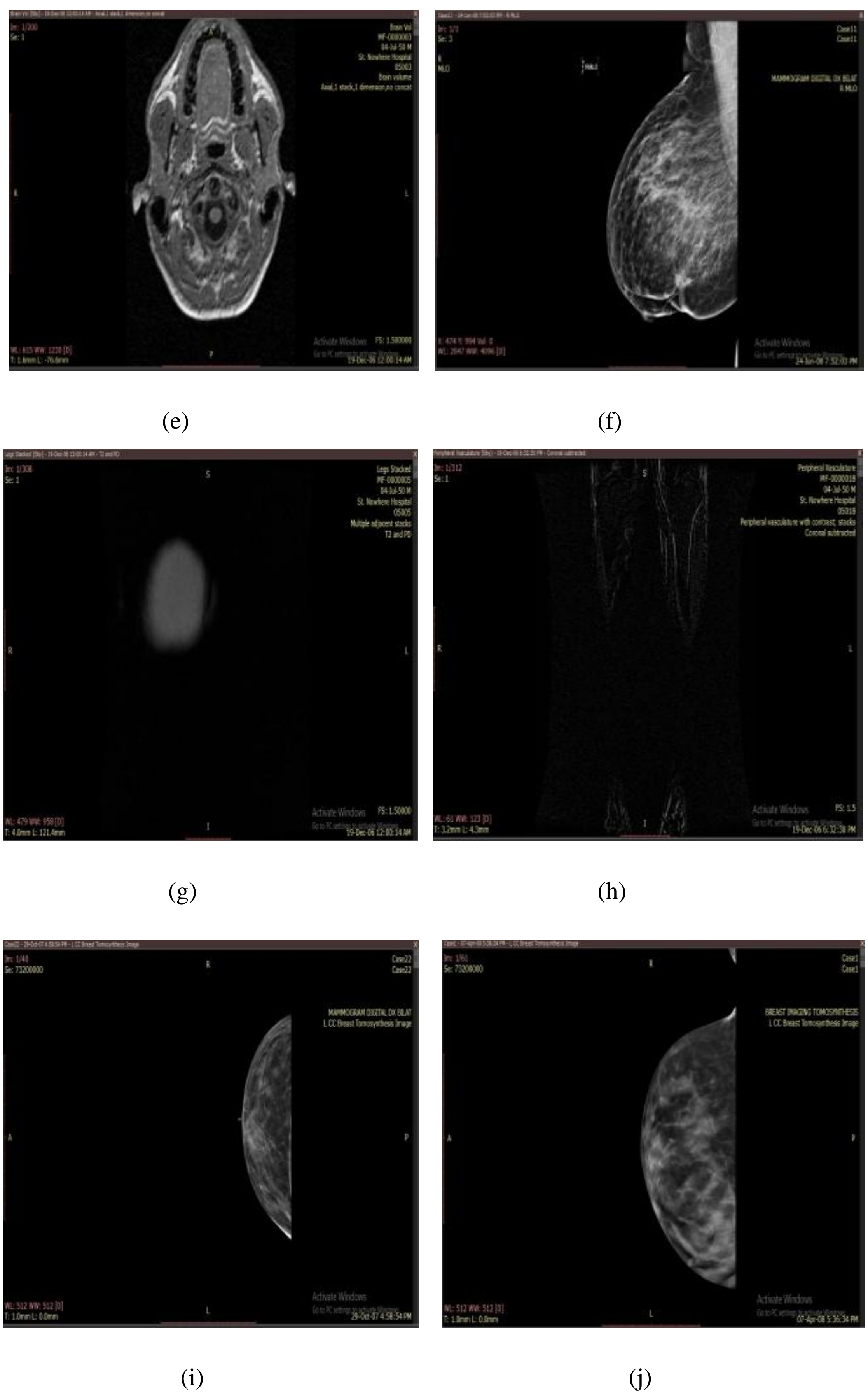
Results of existing LZW Algorithm fed with input:

Images that are used to create the charts in the results are given bellow.



(a) (b)

(c) (d)



(e) (f)

(g) (h)

(i) (j)

Figure 11: Input images

Figure 12 attached below is evidence that shows that our system is lossless and both of the images (input and output) have exactly the same resolution and file size.

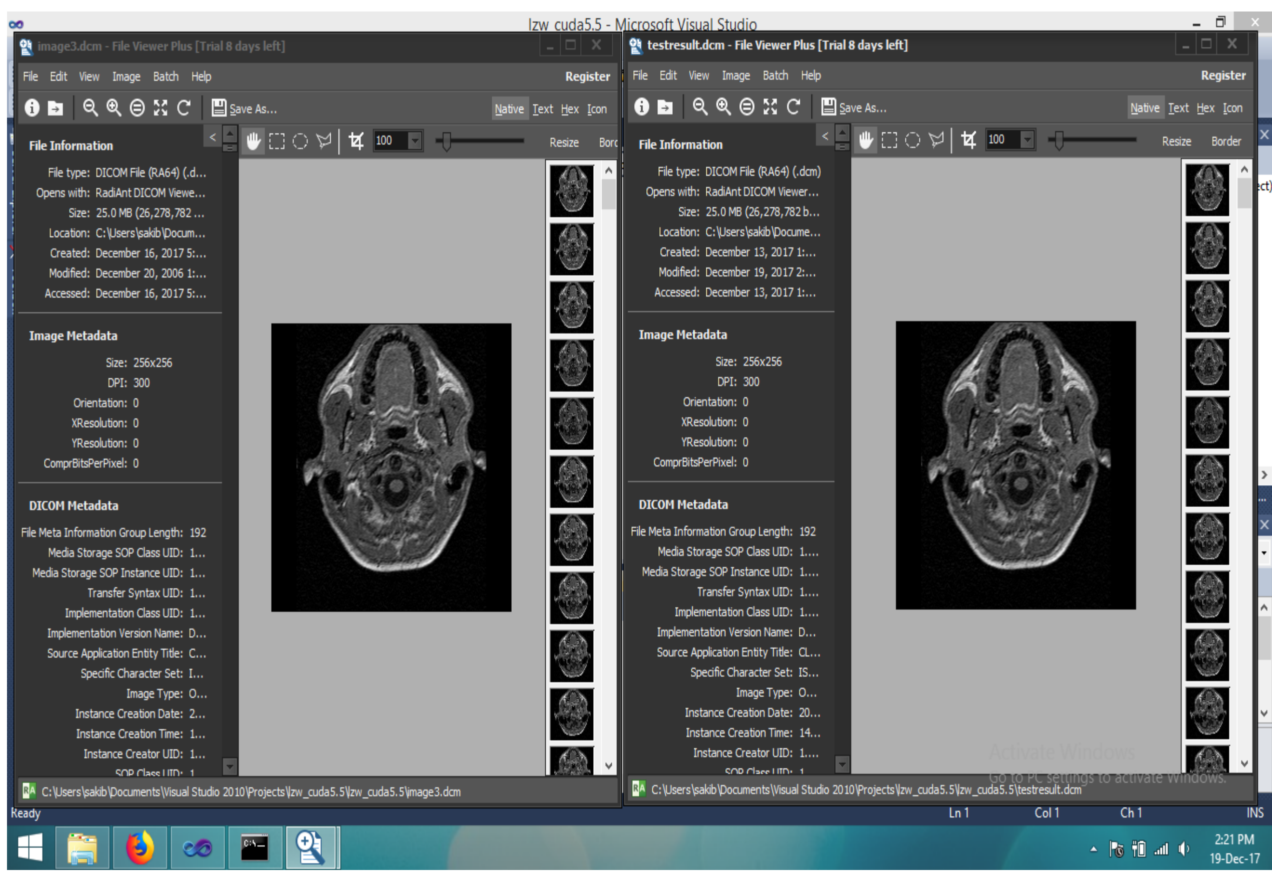


Figure 12: Lossless Picture Quality

In the Figure 13 below, image3.dcm is our input file for this particular example, test.dcm is the compressed file and testresult.dcm is the final output file.

image3.dcm	20-Dec-06 1:37 AM	DICOM File (RA64)	25,663 KB
kernel.cu	19-Dec-17 12:44 PM	CU File	11 KB
lzw_cuda5.vcxproj	14-Dec-17 12:59 AM	VCXPROJ File	9 KB
lzw_cuda5.vcxproj.user	06-Dec-17 4:55 AM	Visual Studio Proj...	1 KB
test.dcm	19-Dec-17 2:19 PM	DICOM File (RA64)	20,066 KB
testresult.dcm	19-Dec-17 2:19 PM	DICOM File (RA64)	25,663 KB
vc100.pdb	19-Dec-17 2:06 PM	Program Debug D...	428 KB

Figure 13: Identifying Input and Output file

Table 2: Size comparison between original and compressed file

	Original Size (kb)	Compressed Size (kb)	Size reduced (%)
Image 1	130	120	7.69
Image 2	257	197	23.35
Image 3	9234	6279	32.00
Image 4	11583	8977	22.5
Image 5	25663	20066	21.81
Image 6	26628	16410	38.37
Image 7	39682	36730	7.44
Image 8	160147	65416	59.15
Image 9	362819	100240	72.37
Image 10	553290	294604	44.75

The results of existing LZW Algorithm fed with a .dcm file as input is shown below in Figure 14(a) and chart 14(b).

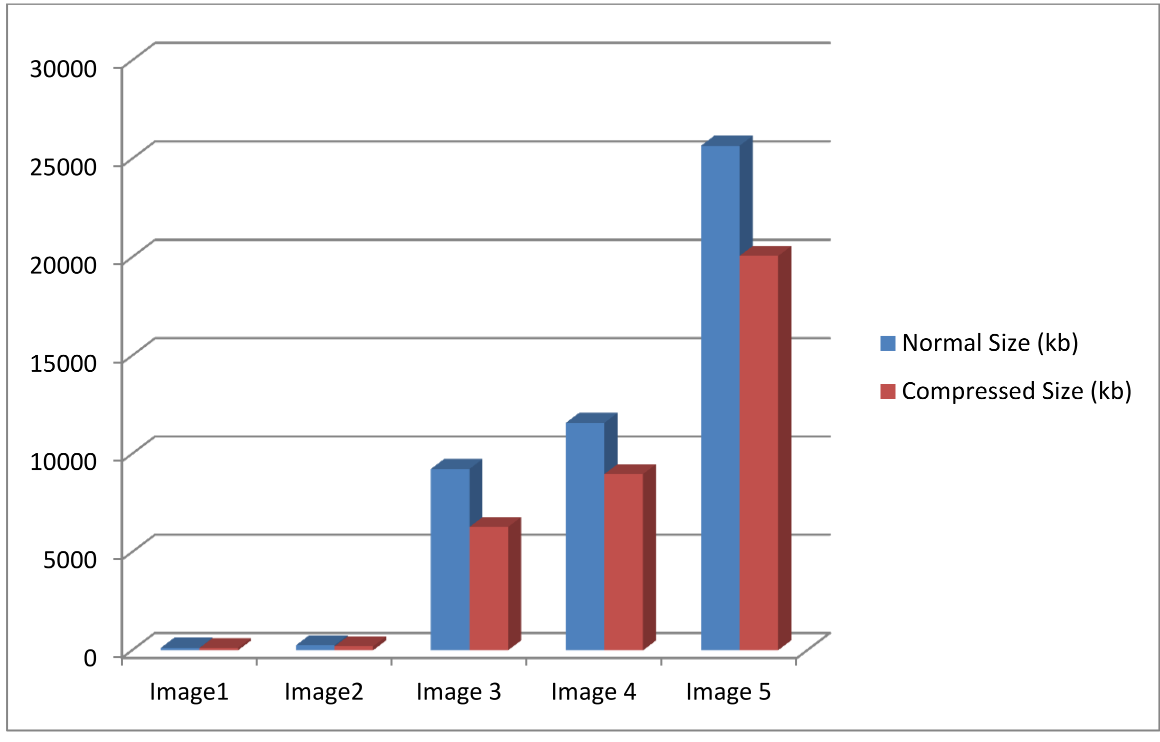


Figure 14(a): Comparison between original image size and compressed size

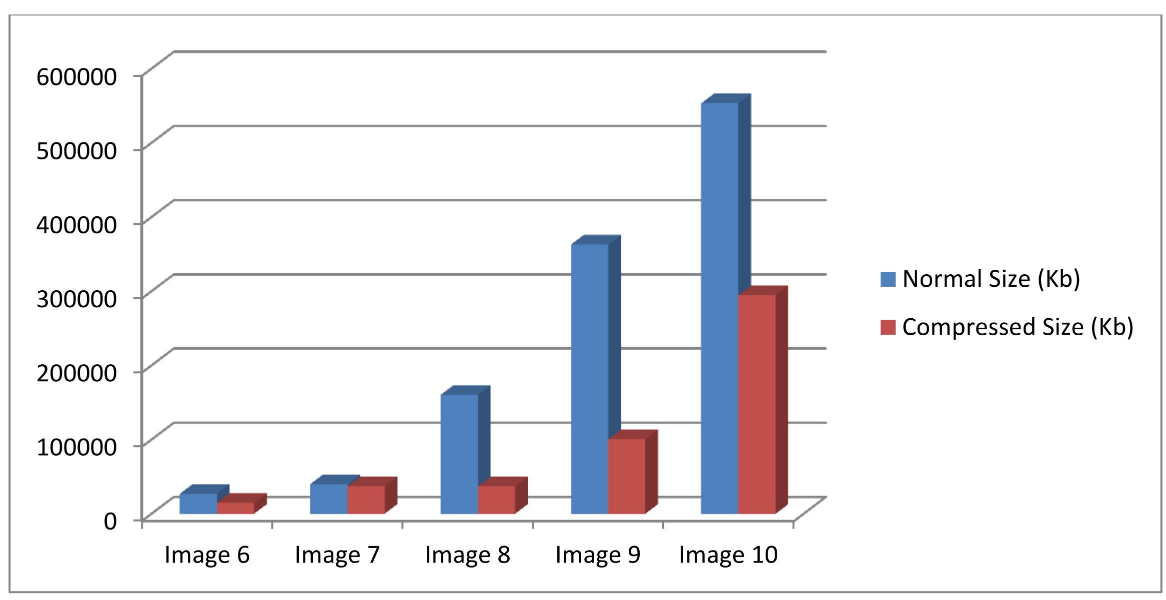


Figure 14(b): Comparison between original image size and compressed size

The experimental results using NVIDIA GeForce GTX 660 and Intel Core i7 show that we have obtained approximately 15% efficiency in encoding time depending on the image.

Results of LZW Algorithm CUDA version fed with input:

The results of LZW Algorithm CUDA version fed with a .dcm file as input is shown below.

Table 3: Comparison between execution time of CPU and GPU

	Image size (kb)	CPU time (ms)	GPU time (ms)	Time Difference(ms)
Image 1	130	3914.8	3485.397	429.403
Image 2	257	4198.38	3803.84	394.54
Image 3	9234	6619.866	5879.145	740.721
Image 4	11583	9086.88	8887.145	199.735
Image 5	25663	9116.15	9007.135	109.015
Image 6	26628	9507.68	9268.94	238.74
Image 7	39682	15361.84	14772.71	589.13
Image 8	160147	45201.4	33467.15	11734.25
Image 9	362819	67311.3	54969.35	12341.95
Image 10	553290	136407.3	120936.9	15470.4

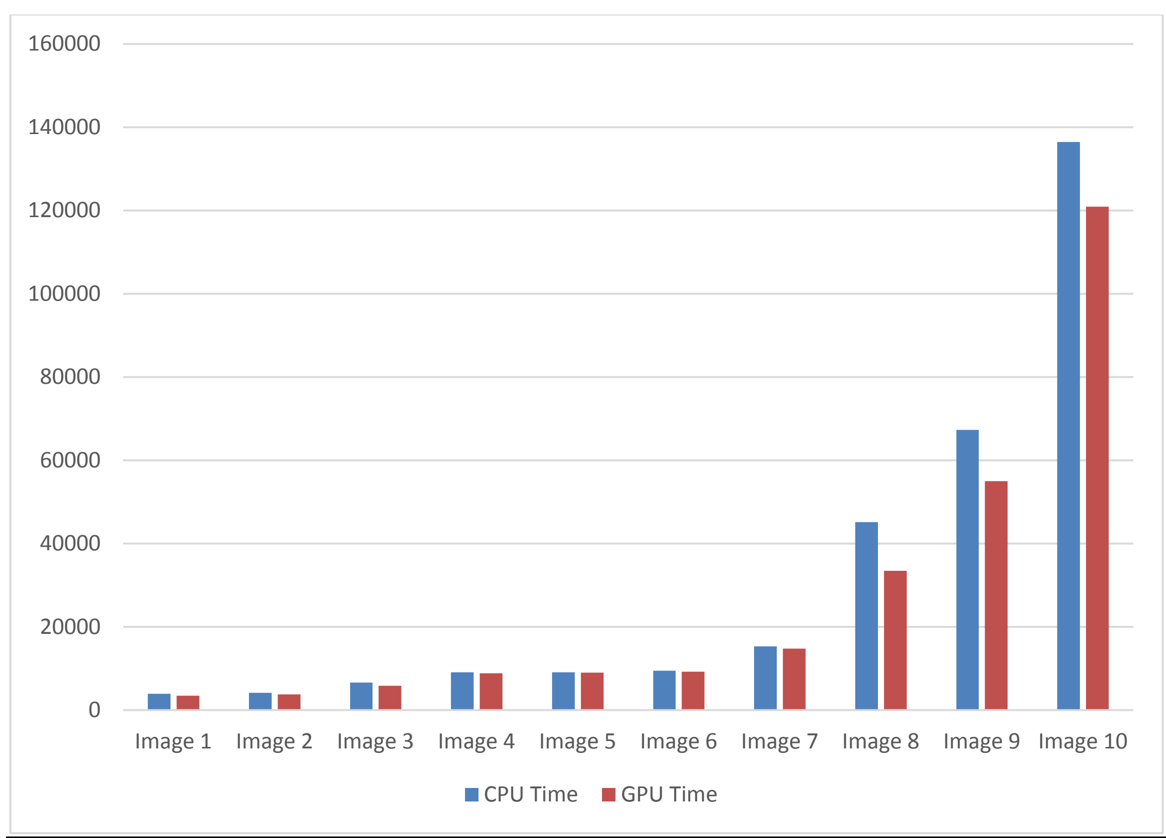


Figure 15: Comparison between execution time of CPU and GPU

Chapter 5: Conclusion

In this paper, we have proposed a faster computation method for LZW image compression algorithm using GPU parallel processing. At first, an image is taken as string of characters as input and converted into a string of codes while updating dictionary to encode with the help of GPU. In spite of having a satisfactory result, hurdle of some limitations have put this experiment away from an even greater result. Firstly, we could not have the opportunity to perform the experiment with higher specifications of GPU as the result will definitely be different with higher computation power of GPU. Most importantly, we could not have scope to implement dynamic parallelism with our GPU with computation capability of 3.0 as it needs GPU with at least computation capability of 3.5. Dynamic parallelism, the ability to parallel processing the nested loops of programs in GPU will bring out an improvement of this experiment. In conclusion, this paper evinces that a faster computation time can be achieved with the help of GPU parallel processing than the conventional technique. We intend to do further research and development in this particular algorithm with the concept of GPU parallel processing which will help us to get more efficient encoding time.

References:

[1] W. W. Hwu, GPU Computing Gems Emerald Edition. Morgan Kaufmann, 2011.

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in Proc. of International Conference on Networking and Computing, Dec. 2011, pp. 68-76.

[3] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0", Mar 2015.

[4] Y. Takeuchi, D. Takafuji, Y. Ito and K. Nakano, "ASCII art generation using the local exhaustive search on the GPU," in Proc. Of International Symposium on Computing and Networking, Dec. 2013, pp. 194-200.

[5] Islam, M., Subhani, M., Tareque, M. and Rafi, M. (2017). Real-time 3D integral imaging system using a faster elemental image generation method using GPU parallel processing. [online] Dspace.bracu.ac.bd. Available at:
http://dspace.bracu.ac.bd/xmlui/handle/10361/8199 [Accessed 19 Oct. 2017].

[6] Ieeexplore.ieee.org. (2017). Differential evolution algorithm on the GPU with C-CUDA - IEEE Conference Publication. [online] Available at:
http://ieeexplore.ieee.org/document/5586219/ [Accessed 10 Dec. 2017].

[7] Ieeexplore.ieee.org. (2017). Fractal Compression with GPU Support - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/document/7968599/ [Accessed 11 Dec. 2017].

[8] Ieeexplore.ieee.org. (2017). Different approaches for implementation of Fractal Image Compression on medical images - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/document/7955187/ [Accessed 11 Dec. 2017].

[9] Ieeexplore.ieee.org. (2017). Speeding up the runtime performance for lossless image coding on GPUs with CUDA - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/document/6572477/ [Accessed 13 Dec. 2017].

[10] Ieeexplore.ieee.org. (2017). A performance model of fast 2D-DCT parallel JPEG encoding using CUDA GPU and SMP-architecture - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/document/7040947/ [Accessed 20 Aug. 2017].

[11] Ieeexplore.ieee.org. (2017). File compression with LZO algorithm using NVIDIA CUDA architecture - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/abstract/document/6319497/ [Accessed 12 Nov. 2017].

[12] Ieeexplore.ieee.org. (2017). Dictionary design algorithms for vector map compression - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/document/1000014/ [Accessed 13 Nov. 2017].

[13] Ieeexplore.ieee.org. (2017). Fast LZW Compression Using a GPU - IEEE Conference Publication. [online] Available at: http://ieeexplore.ieee.org/document/7424730/ [Accessed 23 Dec. 2017].