A PARALLEL HUFFMAN CODER ON THE CUDA ARCHITECTURE

Habibelahi RAHMANİ

Master of Science Thesis

Computer Engineering Thesis

July - 2014

ANADOLU ÜNİVERSİTESİ

JÜRİ VE ENSTİTÜ ONAYI

Habibelahi Rahmani'nin "**A Parallel Huffman Coder On The CUDA Architecture"** başlıklı **Bilgisayar Mühendisliği** Anabilim Dalındaki, Yüksek Lisans Tezi 23.07.2014 tarihinde, aşağıdaki jüri tarafından Anadolu Üniversitesi Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin ilgili maddeleri uyarınca değerlendirilerek kabul edilmiştir.

	Adı-Soyadı	İmza
Üye (Tez Danışmanı)	**: Doç. Dr. CÜNEYT AKINLAR**	**……**
Üye	**: Yrd. Doç. Dr. ALPER KÜRŞAT UYSAL**	**…….**
Üye	**: Yrd. Doç. Dr. MEHMET KOÇ**	**…….**

Anadolu Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun …… tarih ve …… sayılı kararıyla onaylanmıştır.

Enstitü Müdürü

ABSTRACT

Master of Science Thesis

A PARALLEL HUFFMAN CODER ON THE CUDA ARCHITECTURE

Habibelahi RAHMANİ

Anadolu University
Graduate School of Sciences
Computer Engineering Program

Supervisor: Assoc. Prof. Dr. Cüneyt AKINLAR

2014, 39 pages

We present a parallel implementation of the widely-used entropy encoding algorithm, the Huffman coder, on the NVIDIA CUDA architecture. After constructing the Huffman codeword tree serially, we proceed in parallel by generating a byte stream where each byte represents a single bit of the compressed output stream. The final step is then to combine each consecutive 8 bytes into a single byte in parallel to generate the final compressed output bit stream. Experimental results show that we can achieve up to 22x speedups compared to the serial CPU implementation without any constraint on the maximum codeword length or data entropy.

Keywords: Huffman coding, variable length coding, CUDA, GPGPU, parallel computing, JPEG

ÖZET

Yüksek Lisans Tezi

CUDA MİMARİSİ ÜZERİNDE PARALEL HUFFMAN KODLAYICI

Habibelahi RAHMANİ

Anadolu Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Doç. Dr. Cüneyt AKINLAR

2014, 39 sayfa

Çalışmamızda, geniş kullanıma sahip olan entropi kodlama algoritması Huffman kodlayıcının, NVIDIA CUDA mimarisi üzerinde, paralel uygulaması sunulmuştur. Huffman kod sözcük ağacının seri olarak oluşturulmasının ardından, paralel olarak her baytın sıkıştırılmış akım çıktısı olan tek bir biti temsil ettiği bir bayt akımı oluşturularak ilerlenmiştir. Son adımda, art arda gelen her 8 bayt paralel olarak tek bayt içerisinde birleştirilerek, son sıkıştırılmış bit akım çıktısı oluşturulmuştur. Deneysel sonuçlar, kod sözcüklerin uzunluğunda veya veri entropisinde her hangi bir kısıt olmadan, seri CPU uygulamaya göre 22 kat hız kazanıldığını göstermiştir.

Anahtar Kelimeler: Huffman kodlama, değişken uzunluklu kodlama, CUDA, GPGPU, paralel hesaplama, JPEG

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Assoc. Prof.Dr. Cüneyt AKINLAR for his patience, advice, criticism and encouragements throughout this research.

I would like to thank Res. Asst. Cihan TOPAL for his guidance, patience, advice, criticism and encouragements throughout this research.

I would like to thank Res. Asst. Tuba GÖKHAN for her guidance, support and help for preparing this research's report.

Finally, I wish to thank my family and my friends for their support and patience through- out my research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

CUDA **:** Compute Unified Device Architecture

GPU **:** Graphics Processing Unit

CPU **:** Central Process Unit

SIMD **:** Single Instructıon Multiple Data

GPGPU **:** General Purpose Graphics Processing Unit

1. INTRODUCTION

Huffman coding is an entropy encoding algorithm that produces uniquely decodable codewords in variable length to minimize the average codeword length [1]. It is widely used by many communication protocols, compression algorithms, and image and video formats [2] [3]. It is most suitable to compress large volumes of data having a small number of different symbols.

A typical serial implementation of a Huffman coder consists of two separate steps. In the first step, the input stream is processed to compute the frequency of each input symbol, and then a binary codeword tree is generated using the symbol frequencies. In the resulting tree, each symbol has a variable length encoding with the most frequently occurring symbol having the shortest codeword, and the least frequently occurring symbol having the longest codeword. Having computed the codeword for each symbol, the second step of the algorithm proceeds by simply appending the codeword for each symbol of the input stream one after the other to obtain the final encoded stream, which is a slow operation.

Although both steps of the algorithm can be made parallel, researches have concentrated on the second step, i.e., the encoding, since it consumes a lot more time than the first step. At the same time, parallelizing the second step is more challenging due to the fact that the codewords for each symbol has variable length and it is not clear where the codeword for an arbitrary symbol of the input should be written in the final output stream. This problem is trivial in the case of a serial implementation where the codewords are easily appended one after the other to the output stream. Dividing the data into chunks and encoding them separately is also not a feasible solution since it requires bitwise arrangements on the encoded data chunks to obtain a single encoded stream at the end of the operation.

Despite the difficulties in implementing the Huffman coder in parallel, many researchers have looked at the problem from different aspects. In [4], the authors look at parallel codeword generation using “n” CREW processors, but they do not talk about parallel encoding. In [5], the authors present a parallel decoding method that limits error propagation in the event of a bit error in the

encoded stream. In [6], the authors present different hardware architectures for dynamic Huffman coding. Making the Huffman coding faster in the context of JPEG and MPEG encoding is also a widely researched topic [7] [8].

With the introduction of the GPGPU processing and the NVIDIA Computed Unified Device Architecture (CUDA) architecture [9] [10] , which has a Single Instruction Multiple Data (SIMD) computation model, many researchers have concentrated on implementing parallel data intensive applications on the GPU. In [11], authors present a data parallel algorithm for variable length encoding and achieve high speedups by making limitations on the codeword length. Specifically, the author assumes that the total codeword length for four consecutive symbols of the input stream cannot exceed 64-bits, which severely limits the algorithm's usage for data having high entropies. In [12], the authors present a modified Huffman coder that composes the data into independently compressible and decompressible blocks for concurrent compression and decompression, and achieve up to 3x speedup.

In this study, we present a parallel Huffman encoding algorithm which works without any constraints on the maximum codeword length and entropy. We tested the proposed method with a large set of test data with different size and distributions, and we obtained promising results in terms of speedup.

1.1 Huffman Algorithm

Towards the end of 1940, at the beginning of the Information Theory, the idea to develop new effective coding techniques had new begun. Investigators were searching about the ideas of entropy, information content, and redundancy. One of the notion went into, that if the occurrences of symbols in a message were known, there would be a way to encode symbols, so the size of message become small. This worth considering work was being done before invent of modern digital computer. Although, now it appearing normal that information theory goes together at the same level with computer science, but just after World War II, there were no modern computers for all practical tasks. So encoding symbols using base 2 arithmetic algorithm development idea was really major breakthrough [13].

According to data compression format, there are two types of data compression, lossless (entropy encoding) and lossy. If losing some data does not matter the lossy data compression is used, i.e. (video, voice, image), otherwise lossless data compression algorithms are used, i.e. (text, source codes.)

Huffman coding is an entropy encoding algorithm that produces uniquely decodable codewords in variable length to minimize the average codeword length [1] . It is widely used by many communication protocols, compression algorithms, and image and video formats [2] [3]. It is most suitable to compress large volumes of data having a small number of different symbols. This algorithm developed by David A. Huffman while he was a Ph.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" [14].

A typical serial implementation of a Huffman coder consists of two separate steps. In the first step, the input stream is processed to compute the frequency of each input symbol, and then a binary codeword tree is generated using the symbol frequencies. In the resulting tree, each symbol has a variable length encoding with the most frequently occurring symbol having the shortest codeword, and the least frequently occurring symbol having the longest codeword. Having computed the codeword for each symbol, the second step of the algorithm proceeds by simply appending the codeword for each symbol of the input stream one after the other to obtain the final encoded stream [13] [15].

1. Step: consist of the following steps (Expressed in the following example)

Example:

Input symbol stream: **‘**ABABCDDEFGAFDCAABBCCDDEEFFGAAAFFFFF**’**

Count the frequency: Count the appearance of the symbols in stream

Table 1.1-1: Symbols and Counts

8	**4**	**4**	**5**	**5**	**9**	**2**
A	B	C	D	E	F	G

Building tree: The procedure for constructing the tree is simple. The different symbols are considered as stream of leaf nodes that are connected by a binary tree. Every node owns a number, which basically shows the count of symbol in the string. The tree can be constructed with the following steps:

1) Locate two nodes which less occurred in the string.
2) For created two nodes a parent node is created. The sum of the two child's weight is assigned to parent node of them.
3) The two child nodes which were used are removed from the list. The new created parent node is added to the free nodes list.
4) The paths to the children is 0, 1 arbitrarily when decoding.
5) The upper steps are continuing until one free node is left in the list. This last free node is the root of the tree.

The above steps can be used to the symbols used in the given example. Illustrated in Figure 1.1-1 to Figure 1.1-7.

As mentioned before these seven nodes are going to be the leaves of the decoding tree.

At first we pack the two nodes with lowest weights: In our list B and G have the lowest weights which are 4 and 2. One parent node is created for these two which is assigned a weight of 6. The used Nodes B and G removed from the list. Figure 1.1-1.

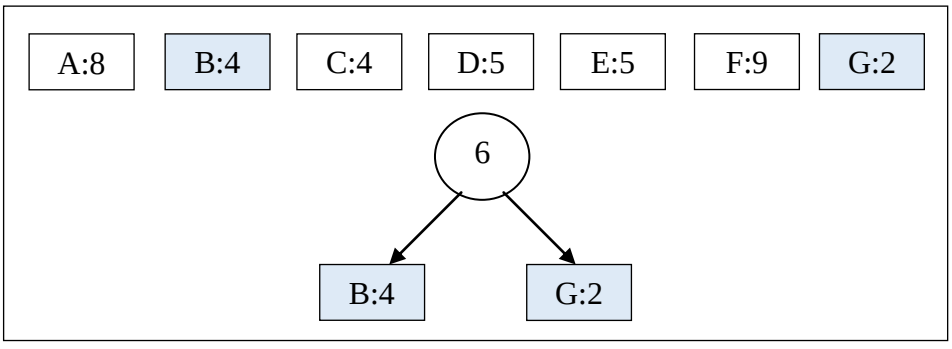

Figure 1.1-1: Tree Generation Step 1

In the second step also we pack the two nodes with lowest weights from the new list which are the D and C nodes. Then a new parent node is created for D

and C. For new created node weight is 9. D and C are deleted from the node list. Our tree and list structure is shown in Figure 1.1-2.

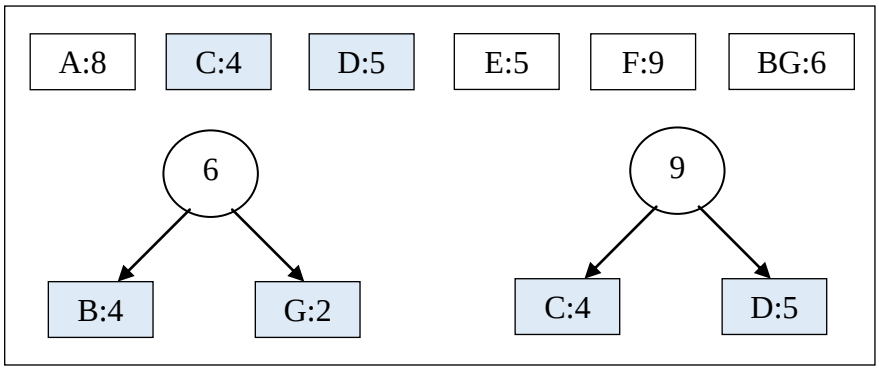

Figure 1.1-2: Tree Generation Step 2

On the next pass, the two nodes with the lowest weights are the parent node for the B/G pair and E node. These are tied together with a new parent node, which is assigned a weight of 11, and the children are removed from the free list. This process is so until all free nodes finished. The process is illustrated in Figure 1.1-3, Figure 1.1-4, Figure 1.1-5, Figure 1.1-6 and in Figure 1.1-7.

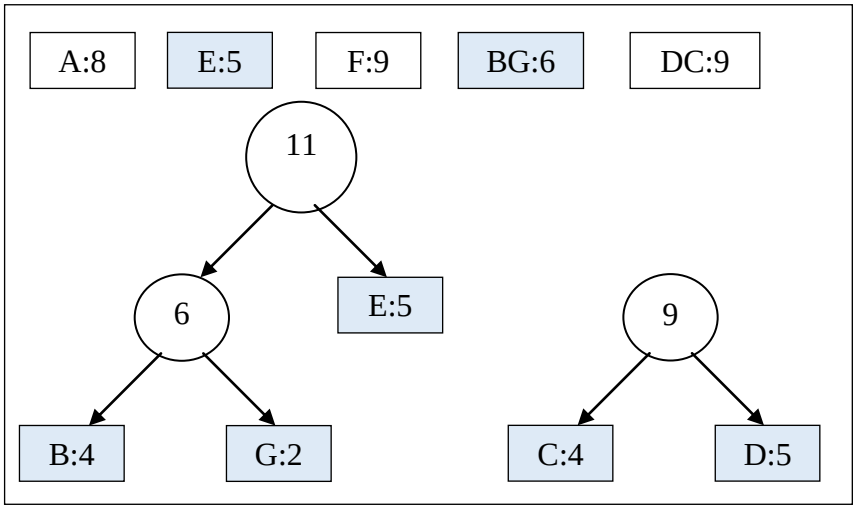

Figure 1.1-3: Tree Generation Step 3

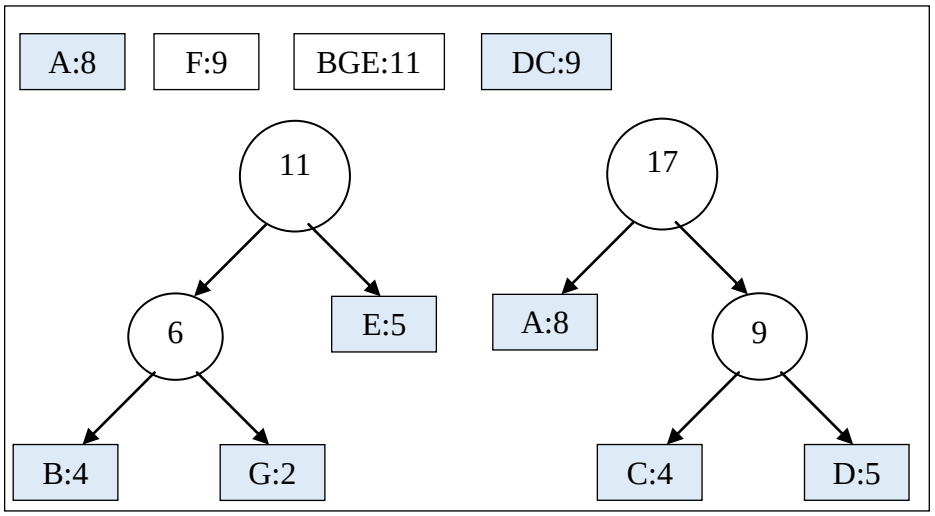

Figure 1.1-4: Tree Generation Step 4

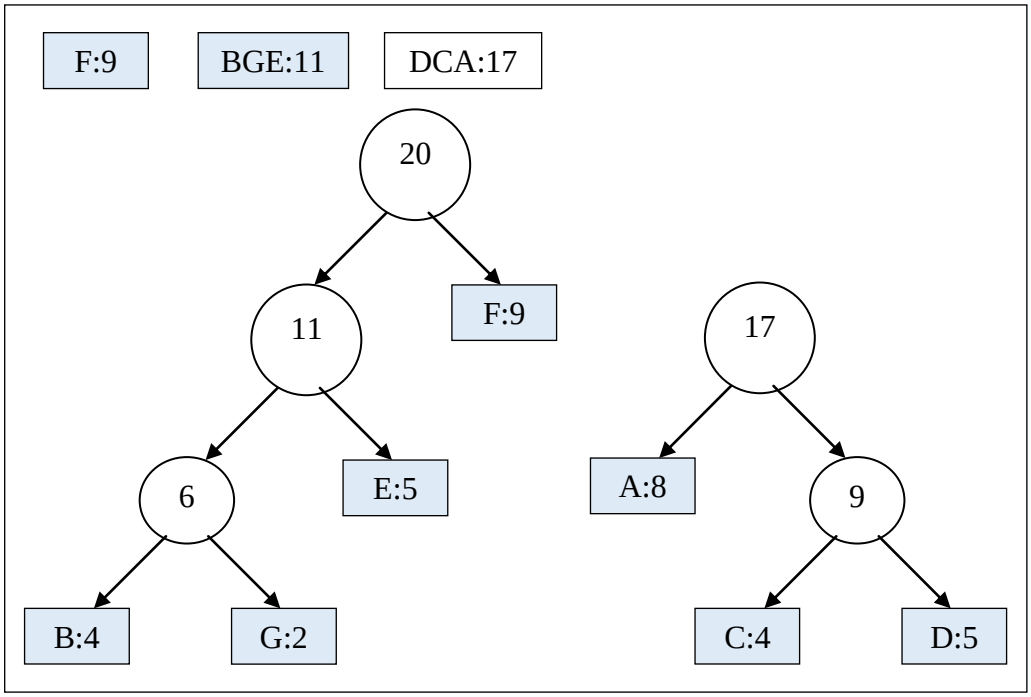

Figure 1.1-5: Tree Generation Step 5

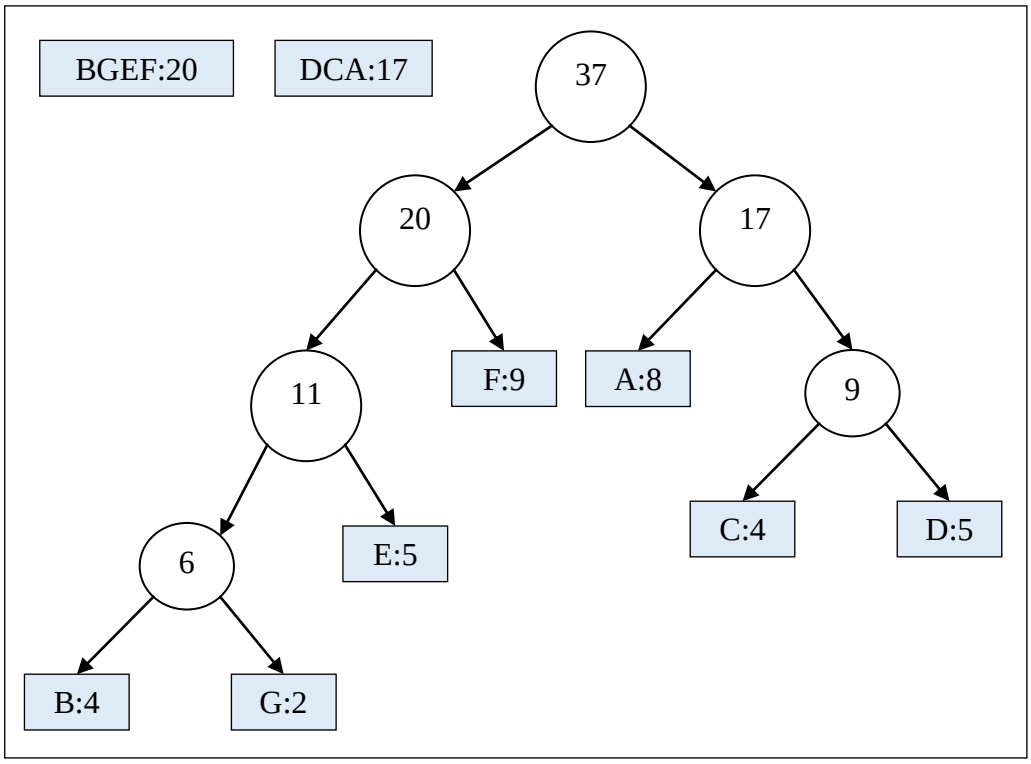

Figure 1.1-6: Tree Generation Step 6

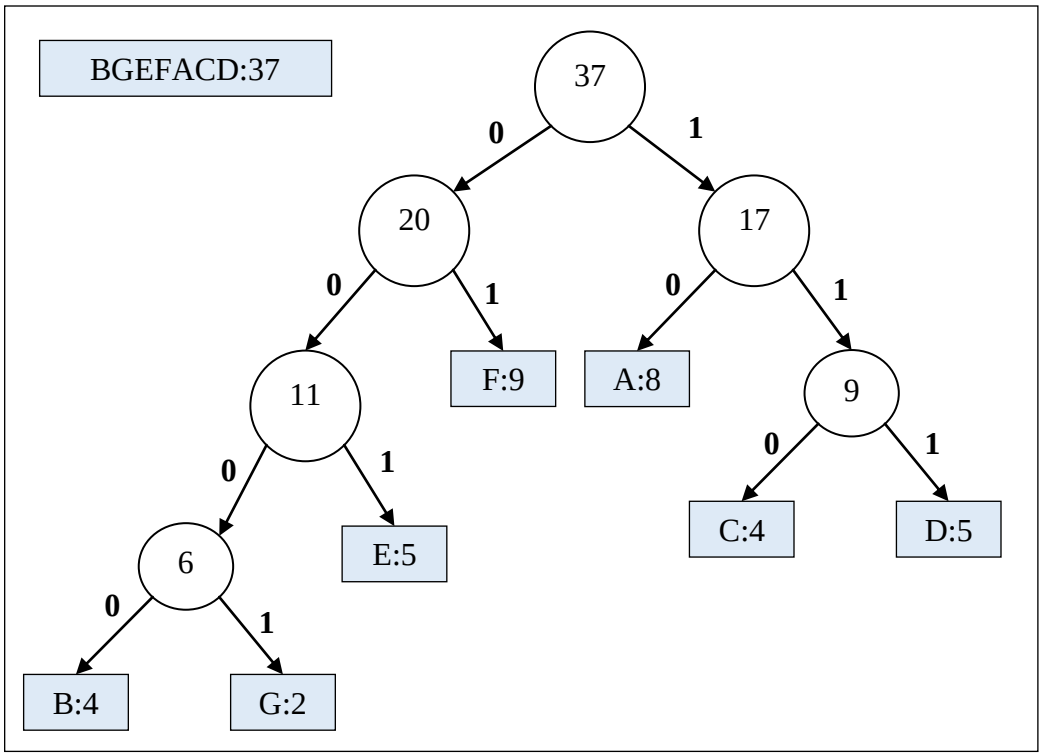

Figure 1.1-7: Tree Generation Step 7

Prefix Code Extraction: To extract the symbols, we have to walk on Huffman tree from leaf to root, collecting bits as we put for every parent node. The new created bits codes are get in reverse order, accordingly we have to reverse them again.. These steps will give codes for each symbol which are shown in the following table.

Table 1.1-2: Generated Codes

A	10
B	0000
C	111
D	110
E	001
F	01
G	0001

As we can see, the new generated codes are prefix codes (Since no code is a prefix to another code); Huffman codes can be easily decoded when we face them in a stream. The symbol with big weight, A and F, has been assigned the smaller codes, and the symbol with the small weights, B and G, has been assigned the big codes [13].

2. Step: In this step we are building a bit stream by picking the corresponding prefix code and writing in a sequence. So, working on our previous example will get result like this.

A Encoded String: ‘10’

AB Encoded String: ‘100000’

And so on. So, our final result is:

‘10000010000011111011000101000110011101111010000000001111111101100010010101000110101001010101’

Finally our work can be summarized in the Table 1.1-3.

Table 1.1-3: Generated Codewords Summary

Symbol	Count	Original Symbol Size	Total Size In the String per symbol	Huffman symbol Size	Total Size of Huffman Bits per symbol
A	8	8	64	2	16
B	4	8	32	4	16
C	4	8	32	3	12
D	5	8	40	3	15
E	5	8	40	3	15
F	9	8	72	2	18
G	2	8	16	4	8

Size of original string: 296 bits

Size of encoded string: 100 bits

1.2 GPU Programming

GPU computing, a graphics processing unit (GPU) scientific, engineering, and enterprise are used together with a CPU to accelerate applications. In 2007, its leadership, NVIDIA's GPUs, now worldwide government laboratories, universities, institutions, small and medium enterprises are strengthening energy-efficient data centers [16].

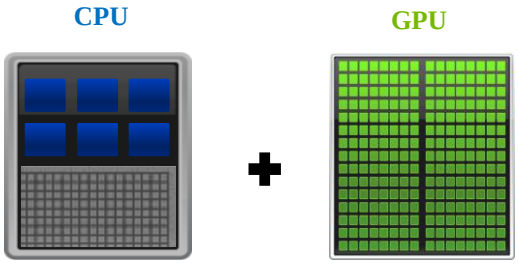

Figure 1.2-1: CPU and GPU Core Architecture [16]

CPU is optimized for sequential batch process consists of a few seeds, GPU is consists of lots of work that is designed to simultaneously execute

thousands of smaller, more efficient core. GPUs has thousands core for handle efficiently parallel workloads. GPU computation, the code continues to work on the rest of the CPU, even though the account-intensive parts of the application by installing GPU delivers outstanding application performance. From the perspective of users, applications run significantly faster. [16]

1.3 CUDA

The first GPUs that support only specific fixed-function pipelines were designed as graphics accelerators. The programmability of hardware increased in the late 1990s, resulting in NVIDIA's first GPU in 1999. In short period when NVIDIA coined the term GPU, artist and game developers didn't have the priority in doing ground-breaking work regarding the technology. Researchers hinted of excellent floating point performance. The General Purpose GPU (GPGPU) movement had reduced [17].

Although GPGPU was hard by the time, even for graphic programs for example OpenGL. Developers had to do scientific calculation onto problems that could be showed as triangles and polygons. A group of Stanford University researches work to reimagine the GPU as a "streaming processor" that resulted memorized the latest graphics APIs [17].

During 2003, researches in head Ian Buck unveiled Brook made the way the first widely adopted programming model to extend C with data-parallel constructs. Make use of concepts such as streams, kernels and reduction operators, the Brook compiler and runtime system exposed the GPU as a general-purpose processor in a high-level language. The highlighted point was Brook programs were not only easier to write than hand-tuned GPU code, they were seven times faster than similar existing code [17].

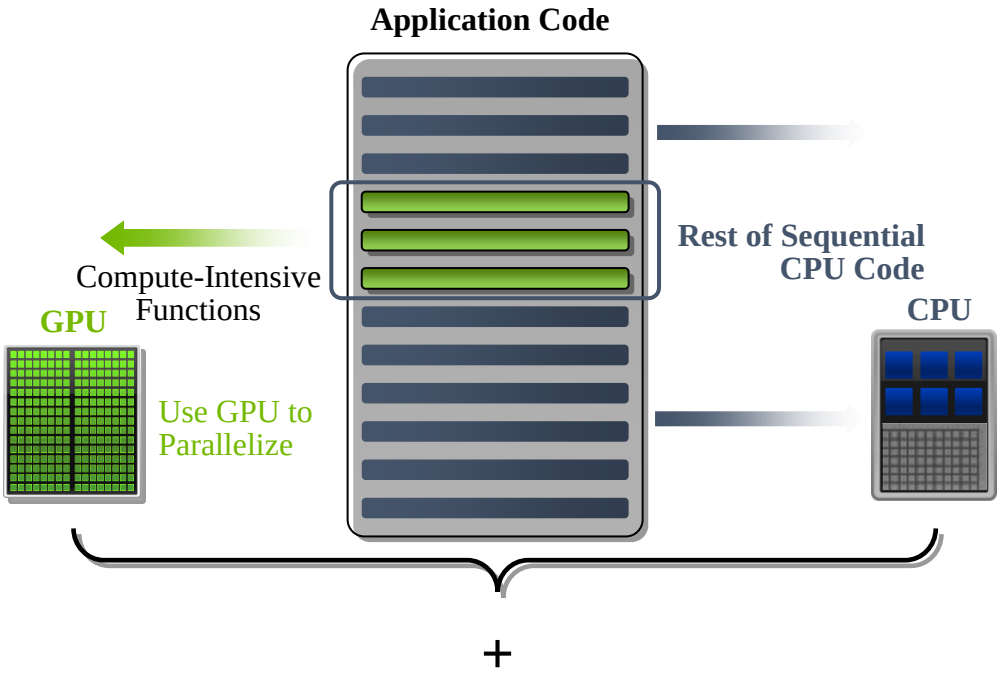

Figure 1.3-1: CUDA Program Execution Architecture [16]

NVDIA was aware of that blazingly fast hardware had to be coupled with intuitive software and hardware tools, and asked Ian Buck to join the company and start evolving a solution to seamlessly run C on the GPU. Using the hardware and software together, the world's first solution for general-computing on GPUs explained CUDA by NVDIA in 2006 [17].

1.3.1 CUDA Architecture

By using CUDA architecture shows that performance off application is much better than CPU based application. Actually this difference basically in GPU architecture, improvement for operations such as graphic operation and those intense highly rated parallel needed operations are required [9] [18] [19].

Figure 1.3.1-1: The GPU Devotes More Transistors to Data Processing [9]

The purpose of GPU being image processing, 3d compiling and signal processing applications, GPU architecture improved based on data processing transistors heavily existed.

1.3.2 Programming Model

CUDA, comes along with high level language C and a platform which give Access to software development. Languages helped by CUDA platform shown in Figure 1.3.2-1.

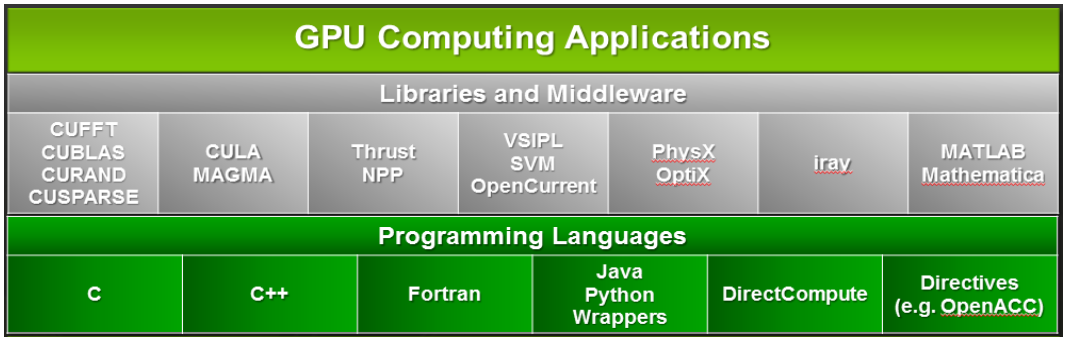

Figure 1.3.2-1: GPU Computing Applications [9]

In terms of designed CUDA architecture allows scalable programming. Developers don't have to deal with GPU kernel. It provides the same transaction running into thousands of channels. The least amount of channel is executed again in the very small amount of CPU cores.

1.3.2.1 Kernels

CUDA C expands standard C language and allows user to identify kernel called C functions. Kernel functions different from normal C function when executed N times Works N units Works in parallel on separate channels [9] [19].

Kernel functions are described in terms __global __. Number of channels able to run Kernel is indicated by <<< ... >>> expression. Each of channel running Kernel is given private key value (ID). This value is accessed via "threadIDx" variable.

In the representation of Figure 1.3.2.1-1, saxpy kernel was called with a <<<B,T>>> configuration type. In here, number of blocks B, and T in each block refers to the number of channels.

```
void saxpy(unsigned int n, float a,
        float *x,float *y)
{
        for(int i=0;i<n;i++)
        {
                y[i]=a*x[i]+y[i];
        }
}


void serial_sample()
{
        //call serial saxpy
function
        saxpy(n,0.2,x,y);
}
```

(a)

```
__global__ void saxpy(unsigned int n, float a,float
*x,float *y)
{
        int i=blockIdx.x*blockDim.x+threadIdx.x;
        if(i<n)
        {
                y[i] = a*x[i] + y[i];
        }
}

void parallel_sample()
{
        //Launch parallel saxpy kernel
        //using n/256 blocks of 256
        //threads each
        saxpy<<<ceil(n/256,256)>>>(n,0.2,x,y);
}
```

(b)

Figure 1.3.2.1-1: A Sample CUDA Application in C Language

1.3.2.2 Channel Hierarchy

Variable threadIDx, which allows us to reach the ID value of the channel is an element included of 3 elements. With this structure, vector, matrix, or 3D data sets allows easy adjustment of on operations. [19]

The relationship between the Channel index values and the channel ID values, will varies according to the size of the data block. For one-dimensional data block channel index and ID values are equal to each other.

Two-dimensional, A data block size for D_x and D_y, index for the channel in x and y coordinates is calculated as value of: $K_{in}=x+y\ D_y$.

$$K_{in}=x+y\ D_y$$

x: A Channel consisting of channels in the matrix column number
y: A Channel consisting of channels in the matrix row number
D_x: The number of columns of channel matrix
D_y: The number of columns of row matrix

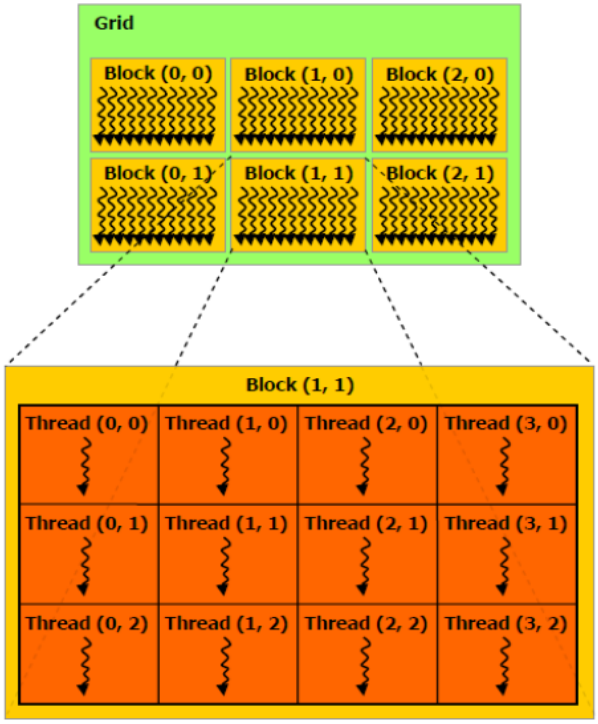

Figure 1.3.2.2-1: Grid of Thread Blocks [9]

The number of channels that can be opened in a block varies according to hardware. In existing products, this number can be up to 1024. To fully cover processed data could be utilized by evenly shaped blocks. Thus the total number of channels, in a block the number of channels is obtained by multiplying the total number of blocks [9].

Blocks can be either one-dimensional or two-dimensional clusters. This structure is called the grid. In Figure 1.3.2.2-1, a channel structure is shown which consists of 6 blocks. Table 1.6-1 contains info about properties of blocks, grid, and channel according to compute capacity.

Any block in the grid, can be expressed by a one-dimensional or two-dimensional index. This index value is called as blockIdx. Mentioned block size can be obtained out of block “blockDim” element.

Channels within the block can operate interactively. In addition, the common shared memory structure having high access speed is available. This structure regulates access, and mechanisms to ensure synchronization of channels within the block are also available.

1.4 Memory Model

CUDA memory type has access to many channels. Figure 1.4-1 shows channels and the type of memory they can access each channel has a special memory access field. Each of the block has a common memory area which has access to all channels. The life of these memory types, is limited by the life of the block. All channels are entitled to the same global memory access [19].

A part from this, there are two different type of memory. They are, respectively, constant and texture memory.

Unchangeable memory area varies according to used graphic card calculated capacity value (compute capability). Table 1.6-1. Unchangeable memory total area is shown according to calculation capacity.

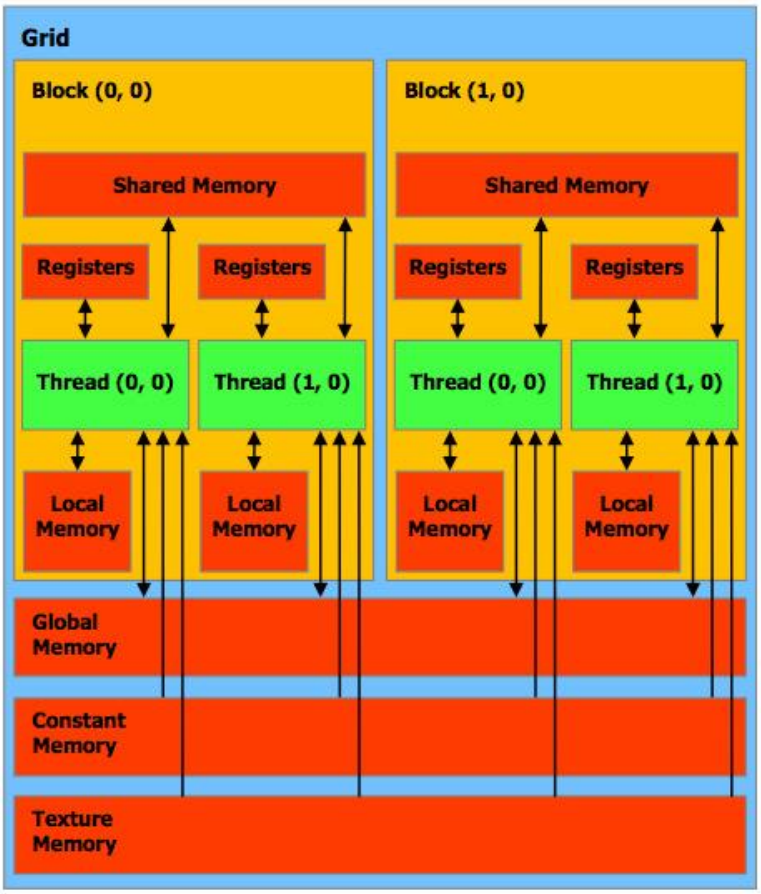

Figure 1.4-1: Memory Access Hierarchy [9]

Unchangeable memory area in available NVIDIA products is 64 KB and below of it. It is therefore difficult to use effectively.

Texture memory is a technology that supports higher performance value according to global memory help by GPU. Applications for the interface to allow read-only transactions. In use basically in connection with a data source. Then the functions and data provided by CUDA is managed. Global unchangeable, and texture memory spaces are designed to be accessible by all kernels.

1.5 Running Model

Some basic information about CUDA model are listed below [19].

1) In CUDA study model kernels are run on the grid
    a. At the same time only one kernel is run.
2) A channel block of a multiprocessor is run.
    a. The channels between processors is not shared to the same block.
3) A multiprocessor can run more than one blog asynchronously.
    a. Here, runnable channel number is limited by source of processor.
    b. Common (shared) memory space is shared between the blocks.
    c. Local memory (register) is shared between channels.

1.6 Compute Capacity

Computing capacity is expressed in numbers large version and a small version. Larger version numbers are the same products have the same core architecture. Small version number will vary according to updates made on the architecture. For example, Fermi [20] architecture is expressed by computing capacity 2.x of architecture product. Fermi architecture products prior to the calculation of the capacity of the CUDA architecture is expressed by 1.x [19].

According to the capabilities and features of multi-processor computing capacity is different. In Table 1.6-1; blocks, grids and channels, calculation capacity of the memory space rates upon the properties are located.

Table 1.6-1: Technical Specifications per Compute Capability [9]

Technical specifications	Compute capability (version)							
	1.0	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.1**	**5.0**
Maximum dimensionality of grid of thread blocks	2				3			
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8					16		32
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB
Number of shared memory banks	16				32			
Amount of local memory per thread	16 KB				512 KB			
Maximum width for a 1D surface reference bound to a CUDA array	Not supported				65536			
Maximum width and number of layers for a 1D layered surface reference					65536 × 2048			
Maximum width and height for a 2D surface reference bound to a CUDA array					65536 × 32768			
Maximum width, height, and number of layers for a 2D layered surface reference					65536 × 32768 × 2048			
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array					65536 × 32768 × 2048			
Maximum number of instructions per kernel	2 million				512 million			

1.7 Software Stack

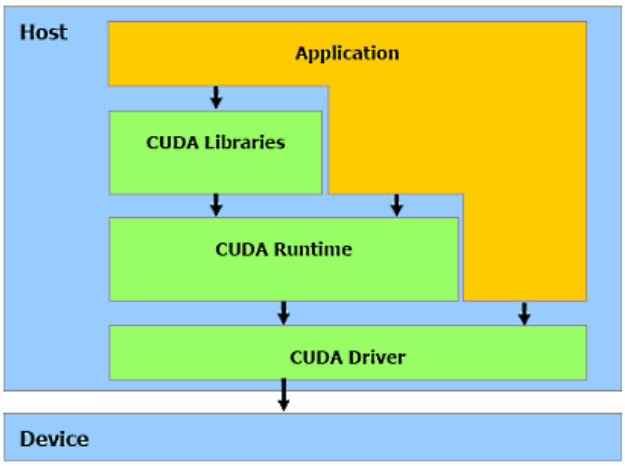

Figure 1.7-1: Software Stack [20]

CUDA software stack is composed of a device driver, an application programming interface, Runtime software and two number for general usage of high-level math library. Figure 1.7-1 shows the CUDA software stack and the elements.

2. PROBLEM DEFINITON AND RELATED WORK

Although both steps of the algorithm can be made parallel, researches have concentrated on the second step, i.e., the encoding, since it consumes a lot more time than the first step. At the same time, parallelizing the second step is more challenging due to the fact that the codewords for each symbol has variable length and it is not clear where the codeword for an arbitrary symbol of the input should be written in the final output stream. This problem is trivial in the case of a serial implementation where the codewords are easily appended one after the other to the output stream. Dividing the data into chunks and encoding them separately is also not a feasible solution since it requires bitwise arrangements on the encoded data chunks to obtain a single encoded stream at the end of the operation.

In [12] (“Accelerating Lossless Data Compression with GPUs”), the authors present a modified Huffman coder that composes the data into independently compressible and decompressible blocks for concurrent compression and decompression, and achieve up to 3x speedup.

In [11] Ana Balevic (“Parallel Variable-Length Encoding on GPGPUs”) worked on Variable-Length Encoding using CUDA. In The presented algorithm, each thread processing some data symbols. Although, the mentioned algorithm reach some speedups but also it has some constraints on data symbols code word lengths. If total length of consecutive 4 data symbols code words lengths exceeds 32 the speedup slows down dramatically, if this length exceeds 64 the algorithm well not work and fails.

3. PROPOSED METHOD

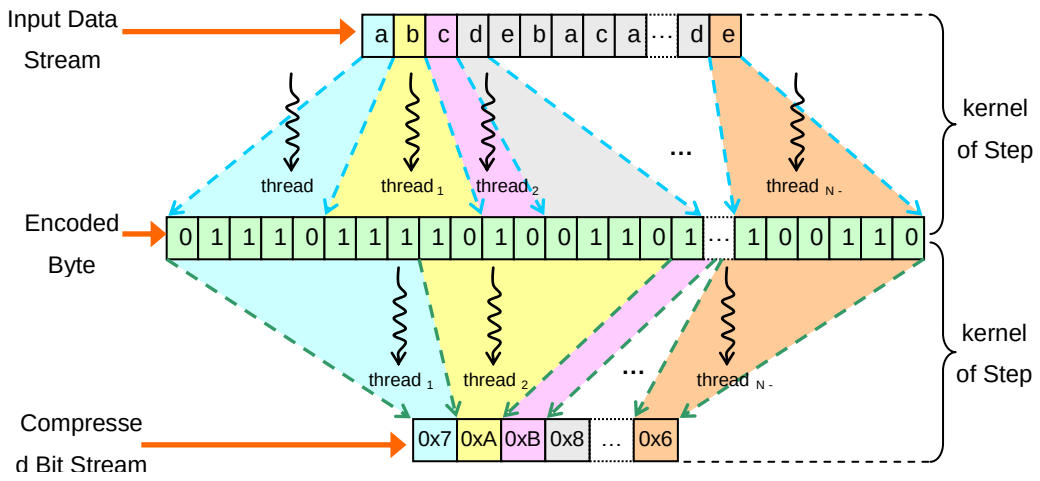

Figure 3-1: Illustration of the proposed algorithm for 3rd and 4th steps. Each box represents 1 byte (8 bit) data.

The Huffman coder consists of two major steps. In the first step, the input data is processed to generate a codeword tree. In the second step, the codeword for each symbol is appended one after the other to create the final compressed bit stream. This second step of Huffman algorithm poses the real challenge due to the variable-length nature of the symbol codewords. During a serial implementation, we start with an empty encoded stream. We can then take the next symbol from the input stream and append its codeword to the end of the encoded stream, and do this until all symbols in the input stream are exhausted. During a parallel implementation where separate CUDA threads are utilized to encode each symbol of the input stream, it is not clear where the thread should write the corresponding codeword in the final encoded stream. We can compute the bit-offsets for each input symbol and have different threads write the corresponding codewords to the appropriate positions in the encoded bit stream as done in [11]. But then, all threads have to be synchronized properly as many of them would need to access the same memory location. This not only creates a huge synchronization problem, but also requires concurrent writes to the same memory slots.

To avoid these two problems, we have followed a different path during encoded stream generation: Our main idea is to have each thread write its

symbol's codeword as a byte stream where each byte represents a single bit of the codeword. For example, if the codeword for the symbol is 5- bits longs, then the thread generates 5 bytes with each byte representing a single bit value of the codeword, which is either 0 or 1. Notice that since each thread is writing its codeword to a separate memory slot, neither synchronization among the threads nor concurrent writes to the same location is a problem anymore.

On the downside, more memory needs to be used to hold the encoded byte stream. The only problem that needs to be solved here is where in memory a thread will write its codeword during encoded byte stream generation. To solve this problem, the byte offset for each symbol's codeword in the encoded byte stream needs to be computed, which can easily be done by using a parallel prefix sum algorithm.

After the encoded byte stream is generated in parallel, a final step is now necessary to combine 8 consecutive bytes into a single byte to generate the final encoded bit stream. Notice that during this step, CUDA threads again work independently without stepping onto each other's feet.

Algorithm 1. Parallel Huffman Coding Steps

1. Huffman Tree Generation . . . . . (Serial in Host)

2. Prefix Sum Computation . . . .(Parallel in GPU)

3. Encoded Byte Stream Generation . . (Parallel in GPU)

4. Compressed Bit Stream Generation .(Parallel in GPU)

Figure 3-2: Algorithm 1. 2. 3. 4. Steps

Algorithm 1 lists the steps of our algorithm, and Figure 3-2 illustrates the general idea. After the Huffman Tree Generation is done serially on the CPU, the rest of the computation, that is the encoding, is performed in parallel on the GPU. Encoding consists of three separate and consecutive steps.

The first of these is the Parallel Prefix Sum to compute the codeword offsets for each input symbol in the intermediate encoded byte stream. In Figure 3-3 for example, the byte offset of the first input symbol 'a' is zero; the byte offset of the second input symbol 'b' is five; the byte offset for the third input symbol

'c' is ten; and the byte offset of the fourth input symbol 'd' is twelve. The offsets for the rest of the input symbols can easily be computed.

```
k ← tid
for  threads k = 1 to N  in parallel
      bitpos[1..N ] ← pref ixsum(cwlen[1..N ])
EndFor

for  threads k = 1 to N  in parallel
      symbol ← data[k]
      cw, cwlen ← cwtable[symbol]
      bitStartPos← bitpos[k]
      for bits j=0 to cwlen
            ByteStream[bitStartPos+ j]← cw [j]
      EndFor
EndFor

for  threads k = 1 to N  in parallel
      for bits i=0,j=128 to 8
            mask=0x01
            mask ← ByteStream[k*8 + i] && mask
            temp ← temp | mask * j
            j ← j>>1
      EndFor
      CompressedByteStream[k]←temp
EndFor
```

Figure 3-3: Proposed Algorithm Pseudo Code

Prefix Sum can easily be done serially in *O(n)* steps, but modelling it as an efficient parallel algorithm is a tough problem. There are many different Parallel Prefix Sum implementations on the CUDA architecture each having different advantages and disadvantages with different constraints due to the hardware restrictions. In our study, we employed a slightly modified version of the Parallel Prefix Sum algorithm presented in [21].

Having computed the codeword offsets for each input symbol, we now proceed to the third step of our algorithm; that of generating an intermediate encoded byte stream. This is illustrated in Figure 3-1. As seen, a separate CUDA thread is launched to handle one symbol of the input stream, and that thread simply writes the symbol's codeword to its corresponding memory slots in the encoded byte stream. For example, $thread_0$ writes 01110 to the first five bytes of the encoded byte stream, $thread_1$ writes 11110 to bytes five through ten, and

$thread_2$ writes 10 to bytes eleven and twelve. The rest of the threads work similarly.

Notice that there is no need for inter-thread synchronization during the encoded byte stream generation. Since each thread performs writes to non-overlapping memory slots, each can proceed independently and perform its operation without the need for any synchronization or coordination with neighboring threads. That is the main idea with generating an intermediate encoded byte stream. Since the codewords are of variable-length, generating the final compressed bit stream directly as done in [11] would have created a huge thread synchronization problem since many threads would have to perform concurrent writes to the same memory location in the final compressed bit stream.

During encoded byte stream generation, threads make use of the CUDA global memory (GM) rather than the shared memory (SM) for the following reasons. First, we do not perform any computational operation on the data. In other words, each byte that we reach from GM is used only once. Therefore, pulling all data to SM and pushing them back becomes unnecessary. The other reason is the automatic caching property of the recent CUDA GPUs, which makes it unnecessary to explicitly pull the data to SM for fast access as was done in previous CUDA GPUs.

The last step of the algorithm is the compressed bit-stream generation from the encoded byte stream (refer to Figure 3-1). This is a massively parallel step. Each thread reads 8 consecutive bytes from encoded byte stream and generates a single byte of the compressed bit stream. For example, thread0 in Figure 3-1 takes the first eight bytes with values 01110111, and compresses them into a single byte having the value 0x77. This is now the first byte of the final compressed bit stream. The other threads work similarly to output the final bit stream. Notice again that each thread works independently requiring no thread synchronization whatsoever. Further notice that each thread accesses different memory locations; that is, there is no concurrent read or write operations to the same memory slot during this step.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed parallel Huffman coder and compare its performance to the serial implementation executed on a CPU. To execute our GPU-based parallel Huffman code, we employ an NVIDIA GTX 480 GPU card; and to execute the serial Huffman code, we employ an Intel Core 2 Quad CPU running at 2.40 GHz.

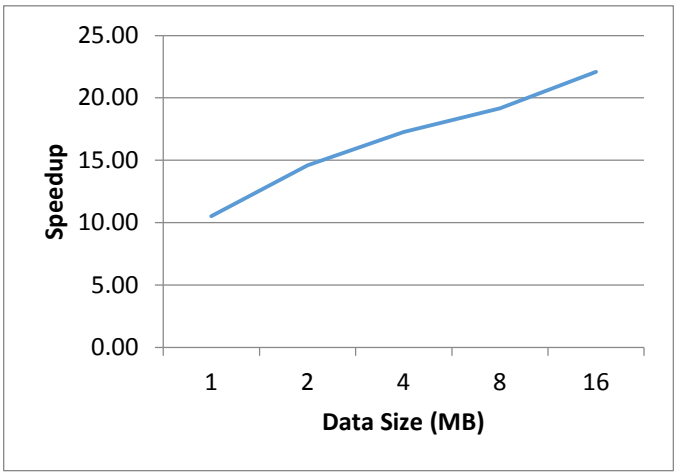

Figure 4-1: Speedup achieved on GTX 480 compared to the serial implementation executed on a Core 2 Quad CPU running at 2.4 GHz as the data size increases. The entropy of the data is fixed at 5-bits/symbol.

In the first experiment, we evaluate the speedup of the proposed algorithm as the data size increases. Figure 4-1 shows the achieved speedup as the data size increases from 1 MB to 16 MB. The entropy of the input data, i.e., the average codeword length used to encode a symbol is fixed at 5-bits/symbol so that we can directly see the effects of the data size on the performance. As expected, the speedup increases as the data size increases, and at 16 MB, we achieve about 22x speedup. The reason for better speedups for big data sizes is due to the fact that after initial start-up, the pipeline of the GPU gets full for large volumes of data and processing dominates the total time. Whereas for small data sizes, the initial start-up dominates the total time, so the speedup is not as big as expected.

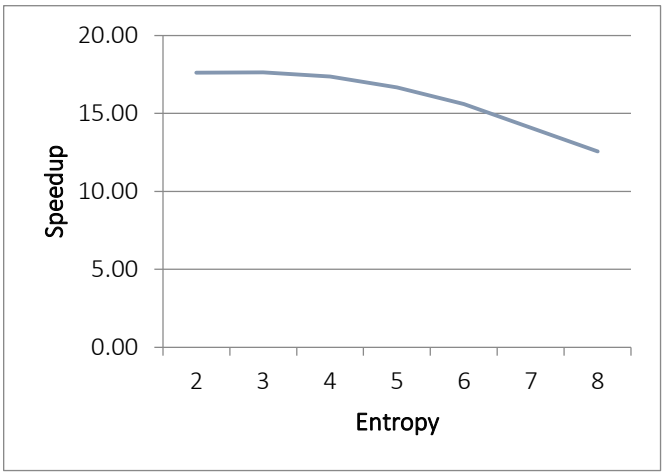

Figure 4-2: Speedup achieved on GTX 480 compared to the serial implementation executed on a Core 2 Quad CPU running at 2.4 GHz as the entropy of the data increases. The data size is fixed at 8 MB.

In the second experiment, we fixed the input data size at 8 MB, and changed the entropy of the data to see the effects of the entropy on the achievable speedup. For data having high entropy, the average codeword length of the symbols will be small, which would also mean that the data is not amenable for compression. Conversely, if the average codeword length of the symbols is big, i.e., the codeword lengths deviate too much from the average, then the data is very amenable for compression. Figure 4-2 sketches the speedup values of the proposed parallel algorithm over the serial one as the entropy changes from two to eight. As seen from the figure, if the entropy of the input data is low, then we can achieve about 17x speedups; whereas, when the entropy of the input data is high, the speedup drops down to about 12x. This is again expected since with low entropy, the resulting compressed bit stream will be of smaller size, which means that the algorithm has to deal will less amounts of data. Conversely, with high entropy, the resulting compressed bit stream will be of larger size, which means that the algorithm has to deal with bigger volumes of data. Since shuffling data in the GPU is a slow operation, the speedup drops with higher entropies.

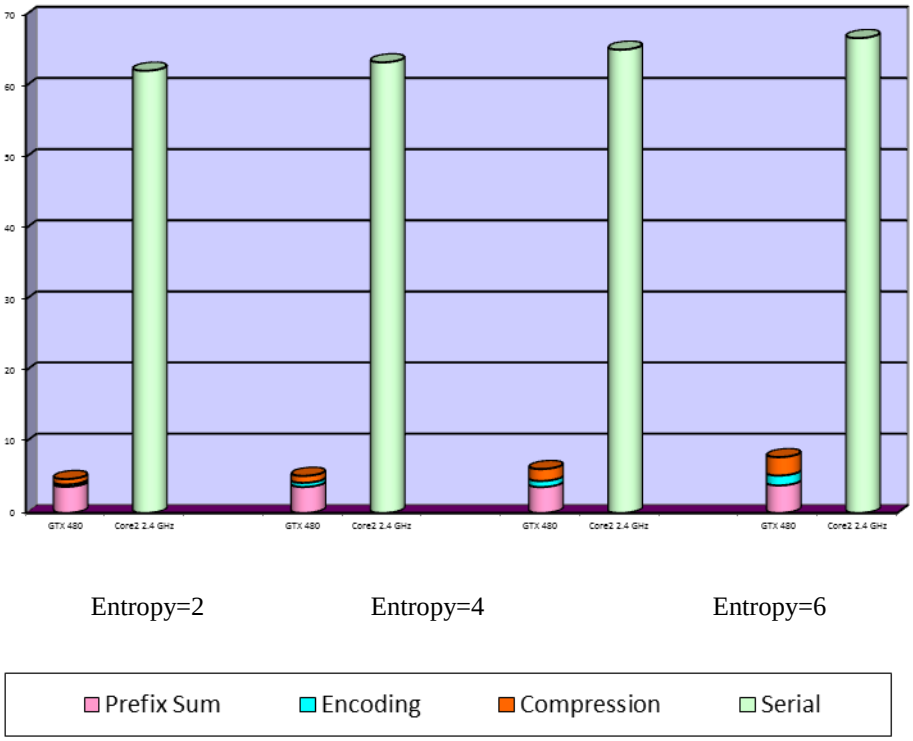

Figure 4-3: Dissection of the running time of the parallel algorithm as the entropy of the input data increases from 2 to 8.

Figure 4-3 shows the dissection of the total running time of the parallel algorithm for data having different entropies, and compares it to the serial Huffman coders. The running time of the parallel code is divided into three parts: (1) Prefix sum to compute the offset of each codeword in the byte stream, (2) Encoding to create the byte stream, (3) Compression to actually compress the byte stream into bit stream output by the algorithm. As can be seen from the figure, Prefix Sum is the major contributor to the running time of the parallel algorithm. To be specific, for data having entropy 8, about 50% of the time is spent on Prefix Sum, about 17% on Encoding and the remaining 33% on Compression. For data having lower entropies, the contribution of Prefix Sum to the total running time increases more. This tells us that to reduce the total running time of the proposed algorithm, Prefix Sum must be made faster followed by Compression. It appears that Encoding has the least contribution to the total running time and is already fast enough.

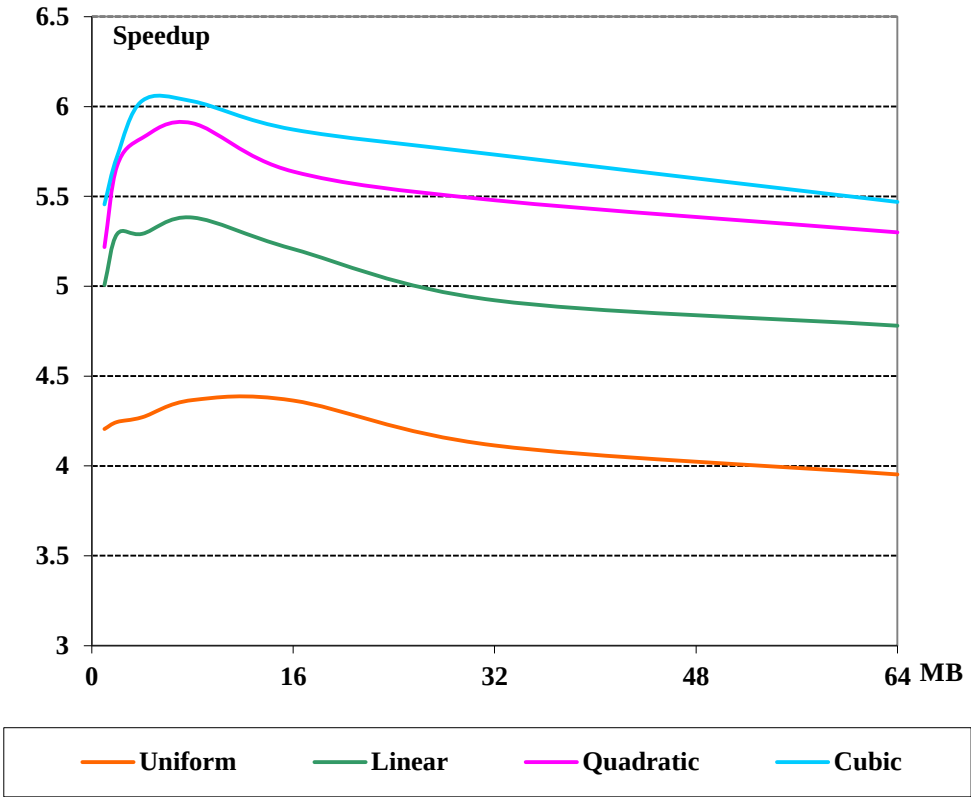

Figure 4-4: Execution time comparison of serial vs. parallel algorithm for different size of data

In Figure 4-4 sketch of the speedup values of the proposed parallel algorithm over the serial one is presented for different types of distributions and data sizes. It is clearly seen that the speedup value increases as the entropy of the data decreases and the deviation of codewords increases. This is an unexpected result for us due to the following assumption. Before the experiments, we expect that the bigger entropies result in codewords with similar lengths and this situation equalizes the overhead per thread. It is a fact that, CUDA architecture performs better if a task can be divided to the threads in an even manner. However, we see that this assumption was wrong.

REFERENCES

[1] D. Huffman, "A method for the construction of minimum redundancy codes," *Proceding of the I.R.E. ,* vol. 40, no. 9, pp. 1098-1101, 1952.

[2] M. Adler, "Deflate algorithm," [Online]. Available: http://www.gzip.org. [Accessed 11 7 2014].

[3] I. J. 1. 29, *ISO/IEC JTC 1/SC 29/WG 1 – Coding of Still Pictures (SC 29/WG 1 Structure),* 1992.

[4] P. Berman, M. Karpinski and Y. Nekrich, "Approximating Huffman Codes in Parallel," *Journal of Discrete Algorithms,* vol. 5, no. 3, pp. 479-490, 2007.

[5] M. Biskup and W. Plandowski, "Guaranteed Synchronization of Huffman Codes with Known Position of Decoder," *Data Compression Conference,* pp. 33-49, 2009.

[6] L. Liu, J. Wang, R. Wang and J. Lee, "Design and hardware architectures for dynamic Huffman coding," *Computers and Digital Techniques,* pp. 411-418, 1995.

[7] P. Howard and J.S. Vitter, "Parallel Lossless Image Compression Using Huffman and Arithmetic Coding," *Data Compression Conference,* pp. 299-308, 1992.

[8] S. Klein and Y. Wiseman, "Parallel Huffman Decoding with Applications to JPEG Files," *The Computer Journal,* vol. 46, no. 5, 2003.

[9] N. C. T. Staff, "NVIDIA CUDA programming guide 5.5," [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide. [Accessed 11 7 2014].

[10] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39-55, 2008.

[11] A. Balevic, "Parallel Variable-Length Encoding on GPGPUs," *Parallel Processing Workshops*, pp. 26-35, 2010.

[12] R. Cloud, M. Curry, H. Ward, A. Skjellum and P. Bangalore, "Accelerating Lossless Data Compression with GPUs," *Journal of Undergraduate Research*, vol. 3, pp. 26-29, 2009.

[13] M. Nelson and J.-L. Gailly, The Data Compression Book, IDG Books Worldwide, Inc., 1995.

[14] Suman, "Enhancement in File Compression Using Huffman Approach," *International Journal of Innovations in Engineering and Technology*, vol. 2, no. 2, pp. 117-123, 2014.

[15] S. Korkmaz, *Türkçe Metinlerin Statik Huffman Algoritması Kullanılarak Sıkıştırılmasında Sıkıştırma Oranı Optimizasyonu,* Konya: Selçuk Üniversitesi, 2003.

[16] N. C. T. Staff, "What is Accelarated Computing?," [Online]. Available: http://www.nvidia.com/object/what-is-gpu-computing.html#sthash.aelvwh72.dpuf. [Accessed 11 7 2014].

[17] N. C. T. Staff, "History of GPU Computing," [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html#sthash.TCH5KEhi.dpuf. [Accessed 11 7 2014].

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, W. Sheaffer and Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed*

Computing, pp. 1370-1380, 2008.

[19] E. Yıldız, *NVIDIA CUDA ile Yüksek Performanslı Görüntü İşleme,* İstanbul: İstanbul Üniversitesi, 2011.

[20] P. N. Glaskowsky, "NVIDIA's Fermi: The First Complete GPU Computing Architecture," [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf. [Accessed 11 7 2014].

[21] M. Harris, S. Sengupta and J. Owens, GPU Gems 3: Parallel Prefix Sum (Scan) with CUDA, Germany:Springer-Verlag: Laser Assisted Microtechnology, 2nd ed., 2007.