A Parallel Huffman Coder on the CUDA Architecture

Habibelahi Rahmani[#1], Cihan Topal[*2], Cuneyt Akinlar[#3]

[#]*Department of Computer Engineering*
[*]*Department of Electrical & Electronics Engineering*
Anadolu University, Turkey

[1]cenghabib@gmail.com
[2]cihant@anadolu.edu.tr
[3]cakinlar@anadolu.edu.tr

***Abstract*— 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.**

***Index Terms*— Huffman coding, variable length coding, CUDA, GPGPU, parallel computing, JPEG.**

I. 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, researchers 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 investigated 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 [7-12].

With the introduction of the GPGPU processing and the NVIDIA Computed Unified Device Architecture (CUDA) architecture [13, 14], which has a Single Instruction Multiple Data (SIMD) computation model, many researchers have concentrated on implementing parallel data intensive applications on the GPU. In [15], 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 32-bits, which severely limits the algorithm's usage for data having high entropies. In [16], 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.

978-1-4799-6139-9/14/$31.00 ©2014 IEEE

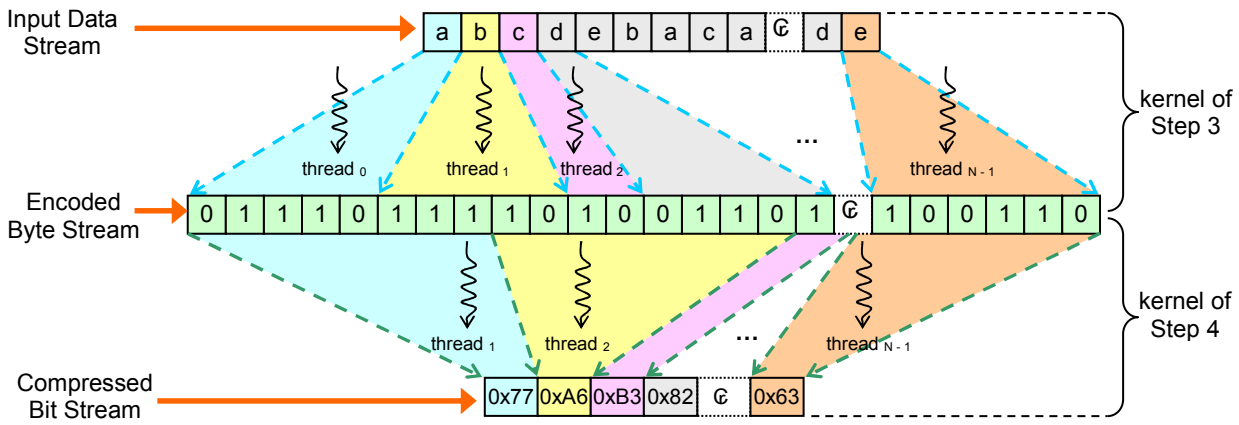

Fig. 1 Illustration of the proposed algorithm for 3rd and 4th steps. Each box represents 1 byte (8 bit) data.

II. Proposed Method

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 [15]. 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.

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)

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 lists the steps of our algorithm, and Fig. 1 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 Fig. 1 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.

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 [17].

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 Fig. 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 neighbouring 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 [15] 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 Fig. 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 Fig. 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.

III. 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.

In the first experiment, we evaluate the speedup of the proposed algorithm as the data size increases. Fig. 2 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 startup, 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 startup dominates the total time, so the speedup is not as big as expected.

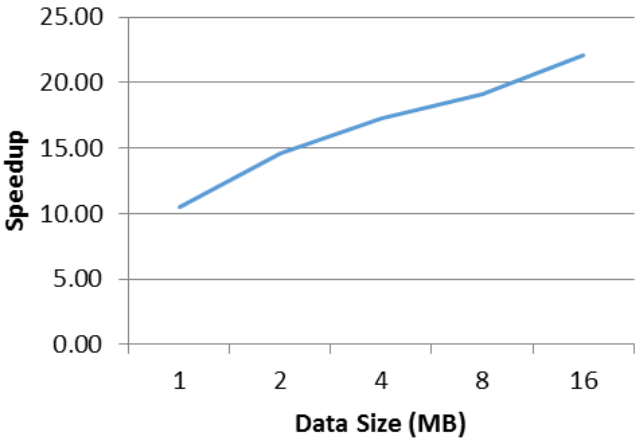

Fig. 2 Speedup achieved on GTX 480 compared to the serial implementation executed on a Core 2 Quad CPU running at 2.4 GHz as the data size increases. The entropy of the data is fixed at 5-bits/symbol.

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 large, 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.

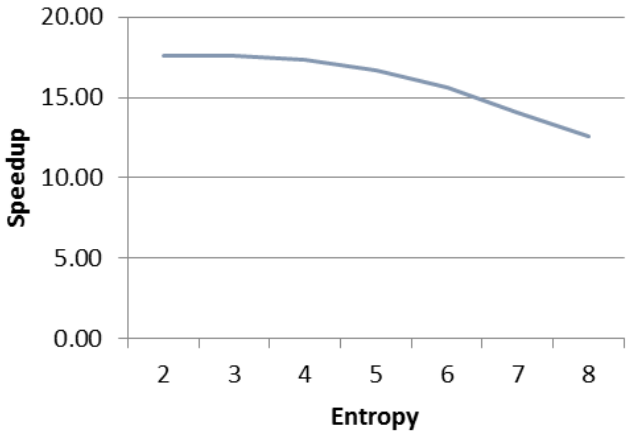

Fig. 3 Speedup achieved on GTX 480 compared to the serial implementation executed on a Core 2 Quad CPU running at 2.4 GHz as the entropy of the data increases. The data size is fixed at 8 MB.

Fig. 3 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.

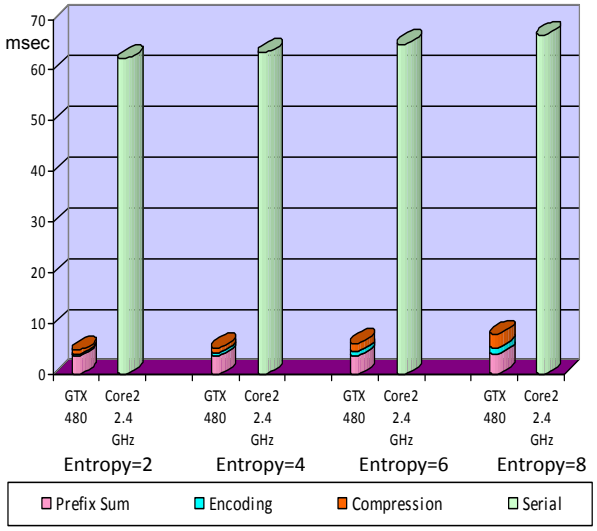

Fig. 4 Dissection of the running time of the parallel algorithm as the entropy of the input data increases from 2 to 8.

Fig. 4 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.

IV. Conclusion

In this paper, we present a parallel implementation of the famous Huffman coder on NVIDIA's CUDA architecture. After the codeword tree generation is performed serially, the rest of the encoding proceeds in parallel with each CUDA thread encoding a single input symbol. Experimental results show that we can achieve up to 22x speedup compared to the serial CPU implementation. It is important to stress that the proposed algorithm works for any input data and does not have any constraints on the codeword length or data entropy. Our future work would be to increase the achievable speedup by concentrating on Prefix Sum and Compression steps of the algorithm. Finally, we plan to perform the Huffman Tree Generation in parallel to get a complete parallel Huffman coder on the CUDA architecture.

References

[1] D. Huffman, "A method for the construction of minimum redundancy codes," in Proc. of the IRE, vol. 40, no. 9, pp.1098-1101, 1952.

[2] M. Adler, "Deflate algorithm," http://www.gzip.org.

[3] ISO/IEC JTC 1/SC 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.T. Biskup and W. Plandowski, "Guaranteed Synchronization of Huffman Codes with Known Position of Decoder," In Proc. Data Compression Conference, pp. 33-42, 2009.

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

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

[8] J.S. Patrick, J.L. Sanders, L.S. DeBrunner and V.E. DeBrunner, "JPEG compression/decopmression via parallel processing," in Proc. Signals, Systems and Computers, pp. 596-600, 1996.

[9] Y. Nishikawa, S. Kawahito and T. Inoue, "A Parallel Image Compression System for High-Speed Cameras," in Proc. International Workshop on Imaging Systems and Techniques, pp. 53-57, 2005.

[10] S.T. Klein and Y. Wiseman, "Parallel huffman decoding with applications to jpeg files," the Computer Journal, vol. 46, pp.487–497, 2003.

[11] S. Wahl, H.A. Tantawy, Z. Wang, P. Wener and S. Simon, "Exploitation of Context Classification for Parallel Pixel Coding in Jpeg-LS," in Proc. IEEE International Conference on Image Processing, pp. 2001-2004, 2011.

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

[13] NVIDIA Corporation Technical Staff, "NVIDIA CUDA programming guide 5.0," Technical report, http://docs.nvidia.com/cuda/cuda-c-programming-guide.

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

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

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

[17] M. Harris, S. Sengupta, J.D. Owens, "Parallel Prefix Sum (Scan) with CUDA," GPU Gems 3, 2007. S. M. Metev and V. P. Veiko, *Laser Assisted Microtechnology*, 2nd ed., R. M. Osgood, Jr., Ed. Berlin, Germany: Springer-Verlag, 1998.