Canonical Huffman Decoder on Fine-grain Many-core Processor Arrays

Satyabrata Sarangi
University of California Davis
Davis, USA
ssarangi@ucdavis.edu

Bevan Baas
University of California Davis
Davis, USA
bbaas@ucdavis.edu

ABSTRACT

Canonical Huffman codecs have been used in a wide variety of platforms ranging from mobile devices to data centers which all demand high energy efficiency and high throughput. This work presents bit-parallel canonical Huffman decoder implementations on a fine-grain many-core array built using simple RISC-style programmable processors. We develop multiple energy-efficient and area-efficient decoder implementations and the results are compared with an Intel i7-4850HQ and a massively parallel GT 750M GPU executing the corpus benchmarks: Calgary, Canterbury, Artificial, and Large. The many-core implementations achieve a scaled throughput per chip area that is 324× and 2.7× greater on average than the i7 and GT 750M respectively. In addition, the many-core implementations yield a scaled energy efficiency (bytes decoded per energy) that is 24.1× and 4.6× greater than the i7 and GT 750M respectively.

CCS CONCEPTS

• **Mathematics of computing** → Coding theory; • **Computer systems organization** → *Multiple instruction, multiple data.*

KEYWORDS

Canonical Huffman decoder, many-core processors

ACM Reference Format:
Satyabrata Sarangi and Bevan Baas. 2021. Canonical Huffman Decoder on Fine-grain Many-core Processor Arrays. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431424

1 INTRODUCTION

Data compression plays a critical role in reducing the cost of storage and communication as systems using data compression require fewer computing resources to store, transmit, and process data. Huffman coding [11] is a popular lossless data compression technique and is used in applications such as JPEG, DEFLATE, and others. Canonical Huffman coding is a type of Huffman coding and is the primary topic of this paper. Canonical Huffman coding has the advantage of less memory usage and a reduction in overall computation by eliminating the requirement of storing the complete Huffman tree, and thereby removing the need to traverse the whole tree from the root to leaf nodes during the decoding process [13].

Over the years there has been extensive work on Huffman codec implementations over different computing platforms. Notably, some of the earlier work was focused on optimizing the VLSI architectures for Huffman codecs [14, 17]. Aspar et al. proposed a real-time decoder implementation on FPGAs using an optimized lookup table (LUT) [4], which was well suited to achieve high frame rates in JPEG and MPEG implementations. Angulo et al. [3] implemented Huffman compression and decompression on GPUs by altering the original Huffman encoding scheme. Patel et al. [16] developed a block-based format for Huffman coding using Burrows-Wheeler and move-to-front transformation; however, this approach didn't help GPUs well in terms of performance. Ozsoy et al. [15] presented an implementation of the Lempel-Ziv-Storer-Szymanski (LZSS) loss less data compression algorithm by using NVIDIA GPUs and CUDA. Funasaka et al. [9] demonstrated how decompression could be performed very efficiently on GPUs and presented an adaptive loss-less compression method. However, this approach does not closely follow Huffman's method [18].

Parallelization in Huffman decoding is challenging due to the fact the location of a decoded symbol depends on the locations of all its predecessors, thus its serial dependency [12]. Therefore, most of the work mentioned earlier alters Huffman's original method by splitting up the input data into independent chunks which can be compressed and decompressed separately. This method results in poor compression efficiency and is unsuitable for several file formats [18].

Weißenberger and Schmidt [18] proposed a massively parallel Huffman decoder on GPUs based on the self-synchronizing property of the Huffman code. The use of this property enables parallel implementations using a large number of computing units at the cost of added complexity due to computation of synchronizing points across data chunks and its effect on overall area and energy efficiency, which is discussed later in Section 4.

The major contributions of our work are as follows:

- We propose LUT based bit-parallel and scalable architectures for canonical Huffman decoders on many-core processor arrays preserving Huffman's original method.
- The proposed area and energy efficient mapping of the architectures is evaluated on a programmable many-core processor array, KiloCore (KC) over standard benchmarks and metrics such as throughput per area and bytes decoded per µJoule of energy are compared to results with optimized software implementations on an Intel i7 and a massively parallel GPU [18].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7999-1/21/01.
https://doi.org/10.1145/3394885.3431424

- We also discuss the trade-offs for serial, LUT-based bit-parallel, and self-synchronization based parallel decoding approaches.

The remainder of the paper is organized as follows. Section 2 describes the canonical Huffman decoding process. Section 3 describes the implementations of the proposed mapping of canonical Huffman decoder kernels on many-core arrays, and Section 4 discusses the results and presents comparisons with the optimized Intel i7 and GPU software implementations. Section 5 concludes the paper.

2 CANONICAL HUFFMAN DECODER ALGORITHM AND EXAMPLE

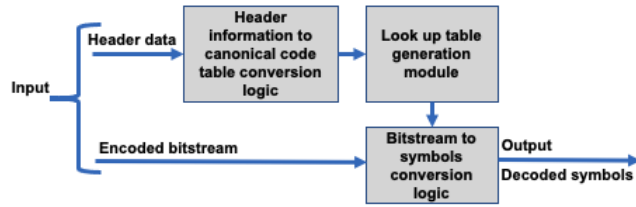

Figure 1: Block diagram of a canonical Huffman decoder.

Figure 1 shows a block diagram of the canonical Huffman decoding process. The first step requires parsing of the header information containing code lengths and symbols that are sorted lexicographically for different code lengths and subsequently, generating a look-up table (LUT) consisting of symbol and code length pairs. The next step does the actual decoding of the encoded bitstream by extracting the symbols from the LUT contents using codewords as table address indices.

Figure 2 details the canonical Huffman encoding process for a given encoded bitstream and header. Firstly, the number of symbols for each code length data from the header is used to generate the starting codeword for each code length following the pseudo code as shown in Figure 2(a). The next step is to read the starting symbol for each code length and assign the corresponding starting codeword to them. The codeword for other symbols with same code length get assigned by incrementing the codeword of the predecessor symbol by one.

For example, symbol 'C' gets the starting codeword which is '110' and symbol 'D' gets assigned with '111'. For a maximum code length of three as shown in the example, the depth of the LUT is eight and therefore, the LUT shown in Figure 2(c) has eight words and each of them stores both symbols and code lengths. The contents of the LUT are populated based on the prefix of the codeword for each symbol, which means the LUT's address index '000'to '011'points to the data related to symbol 'A' as it has the one-bit codeword '0'. Similarly, LUT's address indices '100' and '101' refer to the data related to symbol 'B' as the codeword for 'B' is '10'.

The next step is to read 3 bits (same as MAX_BITS for this example) at a time from the encoded bitstream, index it to the LUT address, and extract the corresponding symbol as the output. If a symbol has a code length of less than MAX_BITS, the difference in corresponding code length and MAX_BITS number of bits from the current chunk will be appended back in the beginning to the original bitstream buffer and the decoding process follows as mentioned above to result in 'CDBAA' as the decoded symbols for this example.

3 IMPLEMENTATIONS

3.1 Many-Core Processor Array Hardware

To demonstrate our many-core architectures, we utilize the KiloCore 32 nm chip [5, 7] to execute three canonical Huffman decoder designs. Each of the 1000 programmable processors in the array occupies 0.055 mm^2 and contains 575,000 transistors. Each processor has an instruction memory of 128x40-bits and a data memory of 256x16-bits [6]. The datapath inside each processor is pipelined into seven stages and contains a MAC with a 16x16-bit multiplier and a 40-bit accumulator. The array also contains a number of 64 KB SRAM modules.

3.2 Many-Core Software Implementations

The mapping of the canonical Huffman decoder on the many-core processor array primarily comprises the following tasks: parsing header information and creating symbol-code length pairs, LUT generation using a SRAM module with every word containing symbol-code length data indexed by the corresponding codeword, and bitstream decoding by accessing the LUT.

Firstly, the number of symbols per code length information in the header are passed to a single or multiple processors in a lane (for loop unrolled implementation) and symbols that are sorted lexicographically in the header based on the code length are passed to the corresponding processors in the next lane of processors as shown in Figure 3. Secondly, symbols and code lengths are grouped together, passed downstream, and get stored in the LUT with the corresponding shifted codewords acting as the address indices for the LUT. Once the starting codeword for a code length is computed, the rest of the codewords can be computed in parallel based on the number of symbols available for the specific code length. Finally, the encoded bitstream is streamed through a processor, two bytes of bitstream data gets buffered to generate the index address. The hashed symbol stored in the word outputs the array as the decoded symbol and corresponding code length data is extracted, which is used to generate the next chunk of data for indexing.

3.3 Baseline Implementation

In the baseline implementation shown in Figure 3 one processor is responsible to generate the starting codeword for each code length following the pseudo code shown in Figure 2(a). The starting codeword computation for each code length depends on the number of symbols encoded with the previous code length. Next, the processor downstreams the starting codeword data to the processor in south.

The first processor in that lane keeps the symbols with code length one and two, and passes the rest of the symbols to the corresponding processors in east. Each processor dedicated for a code length then maps the symbols with the corresponding codewords by assigning the first symbol with the starting codeword and subsequent symbols with a codeword by incrementing the codeword of the predecessor symbol.

The next task for a set of processors is to combine code length and symbol set as a word to be written into the LUT. Moreover, address generator processors generate the corresponding address

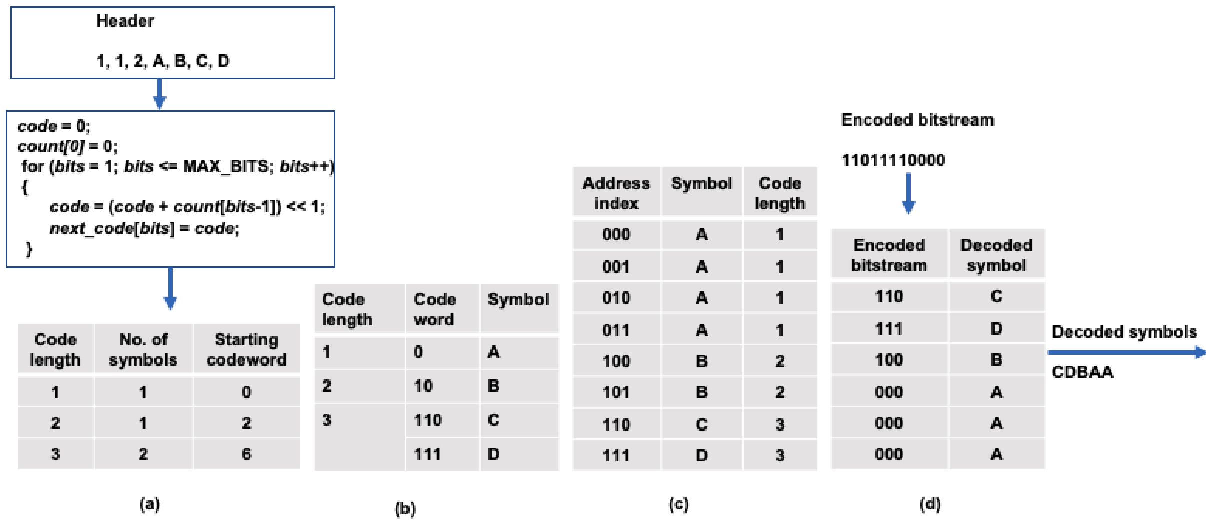

Figure 2: Example of a canonical Huffman decoding process (a) starting codeword for each code length computation (b) mapping of symbols to codewords (c) LUT containing symbols and code lengths (d) parsing encoded bitstream chunks and decoding of symbols.

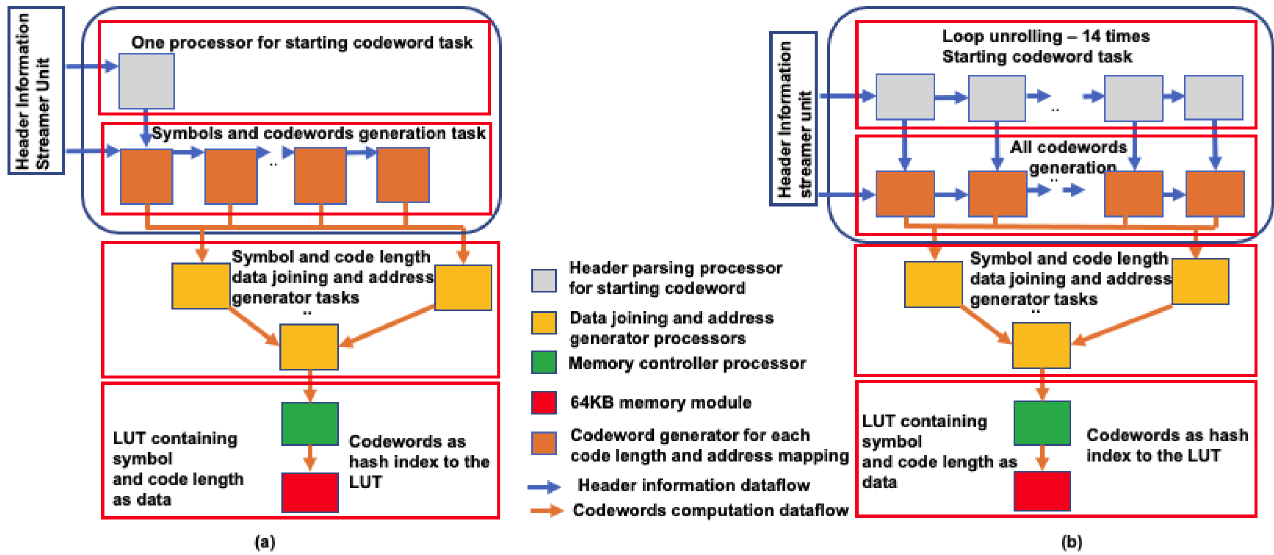

Figure 3: Mapping of header to lookup table conversion logic for: (a) baseline implementation and (b) optimized kernel implementation. This architecture involves starting codeword generation for each code length, subsequently assigns codewords to all symbols, and loads the symbol and corresponding code length data into a LUT.

for memory write operation using codewords as direct addresses for the memory module. A single SRAM module is used to implement the required LUT. The address generator processors set the start and end address based on the maximum code length available in the bitstream.

In the encoded bitstream task shown in Figure 4 the encoded bitstream data is buffered. Two bytes of data are buffered and sent to the processor to the south. The task of the corresponding processors to the south is to generate the read address and index the two bytes chunk to the memory module through the memory controller processor. Each memory access results in a single decoded symbol, but the corresponding code length of the symbol decides the next chunk of data to be indexed, either by shifting the considered chunk of data or obtaining the next chunk from the buffer processor. This task is done by the next chunk of bitstream determination task processors.

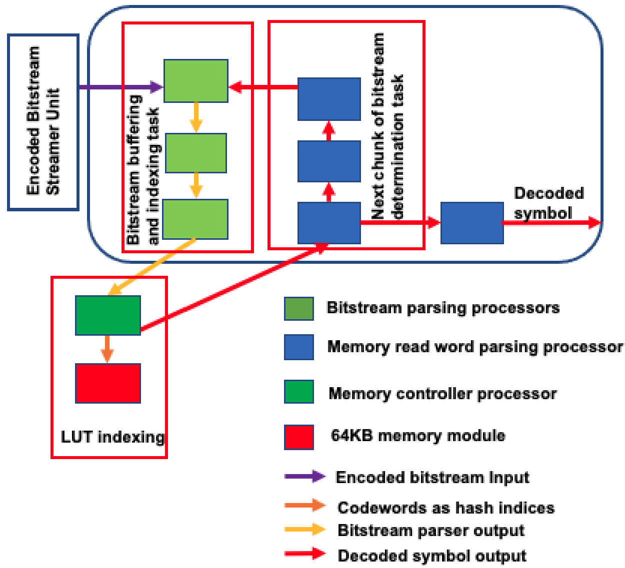

Figure 4: Mapping of the encoded bitstream parsing and LUT based decoding process. This architecture uses the bitstream chunk as hash index, access the LUT, and read the corresponding symbol and code length. The corresponding symbol is decoded as the output and code length is used to decide the next chunk of the encoded bitstream.

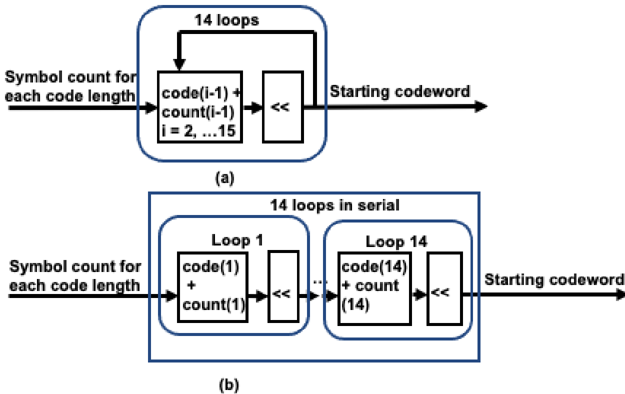

Figure 5: (a) No loop unrolling (b) loop unrolling 14 times to speed up the starting codeword task execution.

The proposed decoder architecture does not follow reading one bit at a time from the encoded bitstream and Huffman tree traversal based decoding method, which is relatively slow. Instead, a block of the encoded bits whose size is the same as the maximum code length for the given encoded bitstream is accessed and the look up table decodes a symbol in every iteration as explained earlier. Therefore, the decoder architecture is named the bit-parallel and LUT-based decoding method.

3.4 Optimized Kernel Implementation

One of the ideas to speed up the starting codeword computation task is to exploit loop unrolling and assigning the task across set of processors instead of one processor. The same has been implemented in two designs by adopting loop unrolling as shown in Figure 4 seven and fourteen times respectively for area and speed up trade-offs. The mapping of the symbol-code length data joining, LUT generation, and bitstream decoding tasks follow the same architecture as of the baseline implementation.

In terms of scalability, lane of processors dedicated for starting codeword and symbols-codewords mapping task can be replicated to cater to varying header and code length data across compression standards. Overall, the many-core processor array implementations presented in this paper support symbols with a maximum code length of 15, which is suitable for various compression standards.

3.5 Application Task Example Mapping

Figure 6 shows the mapping of the optimized kernel implementation onto the physical array using 52 processors and a memory module. The resulting task graph for the kernel is given as an input to a place-and-route tool designed for the many-core processor array, which shows mapping of the tasks discussed earlier to a set of I/O handlers, processors, and memory.

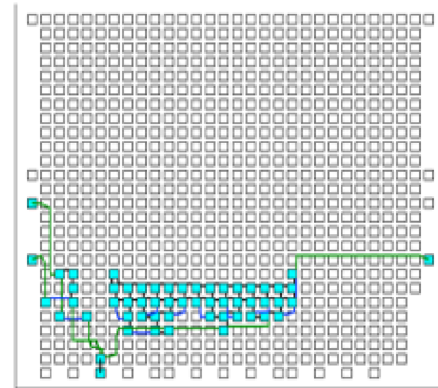

Figure 6: Application mapping of the optimized decoder kernel showing processors required for all tasks, a single 64 KB memory (on the bottom of the array), and I/O handlers.

4 RESULTS AND DISCUSSIONS

4.1 Platforms and Benchmarks

We consider the following computing platforms to evaluate the canonical Huffman decoder implementations in terms of area and energy efficiency.

- Intel i7-4850HQ containing four cores and eight threads with a base clock frequency of 2.3 GHz [1].
- GeForce GT 750M with 384 shader cores running at a base clock of 941 MHz [2].
- KiloCore featuring a 1000-processor array with each processor running at 1.2 GHz at 0.9 V [7].

All implementations have been tested and functionally verified before conducting experiments for various performance metrics. The following benchmarks for lossless text compression or decompression are used to validate the functionality of all designs and compare the area and energy efficiency metrics across all three computing platforms: Artificial corpus, Calgary corpus, Canterbury corpus, and Large corpus. Moreover, we added a large text file of size 1 GB to the Large corpus data sets to evaluate the implementations across a wide variety of file sizes and types of Huffman tree.

4.2 Experimental Setup

We simulate KiloCore implementations using a cycle-accurate simulator which is modeled based on the fabricated chip data [7]. The default simulated processor supply voltage is set to 0.9 V. The following key parameters have been analyzed for various implementations; throughput, area, power, and workload per unit energy. The optimized throughput and energy data are presented after opting supply voltage scaling while preserving the functionality across benchmarks. The Intel i7 and GPU simulation results are reported after speeding up the execution time by triggering the −O3 flag for both gcc and nvcc compilers respectively.

We have referred to the source code for canonical Huffman decoder implementation [8] for Intel i7-4850HQ implementation. The source code has been modified to do bit-parallel decoding instead of reading one bit at a time from the compressed bitstream to make the execution faster. For power calculations for the Intel i7-4850HQ chip, TDP/2 = 23.5 W is used since the actual power dissipation is not known. The die size of the Intel i7-4850HQ is reported as 260 mm^2 [1].

To implement the parallel decoder [18], we have considered a CUDA-enabled GT 750M GPU with compute capability 3.0, which is compatible to run the given implementation. The GT 750M GPU has a TDP ratings of 50 W and a TDP/2 = 25 W is used for power calculations since the actual power dissipation is unknown. The die size of the GT 750M GPU is reported as 118 mm^2 [2].

4.3 Throughput, Area, and Energy Efficiency Analysis

The optimized implementation on the many-core processor array exploiting loop unrolling by 14 for the starting codeword computation task results in the overall highest throughput per area and the baseline implementation gives the highest number of compressed bytes decoded per µJ of energy. KiloCore implementations yield a reduction in power dissipation of 72× and 37× when compared to Intel i7 and GPU implementations respectively. The GPU implementation is based on a massively parallel decoding algorithm and exploits all of its massive die area to yield an average throughput improvement of 20× and 54× when compared to KiloCore and Intel i7 implementations respectively. Unfortunately, the improvement in throughput results for the GPU implementation utilizing self-synchronizing property of the Huffman code comes at the cost of both power and chip area where the proposed many-core processor array implementations have an advantage—the area-efficient KiloCore designs give an area improvement of 181× and 51× over Intel i7 and GPU respectively.

Table 1 reports the implementation results in terms of throughput per area and bytes decoded/µJ for all the benchmarks and processors. It also shows the improvement in terms of both area and energy efficiency with a metric throughput per area × bytes compressed per energy where KiloCore achieves the best among all. For each metric the geometric mean of the results for all the files inside a benchmark are reported. The Intel i7-4850HQ (22 nm) and GeForce GT 750M (28 nm) results are scaled to 32 nm for a fair comparison with KiloCore. The many-core implementations achieve a scaled throughput per chip area that is **324**× and **2.7**× greater on average than the i7 and GT 750M respectively. In addition, the many-core implementations yield a scaled energy efficiency (bytes decoded per energy) that is **24.1**× and **4.6**× greater than the i7 and GT 750M respectively. Figure 7 shows a scatter plot with raw throughput per area vs. bytes compressed per energy where for every benchmark KiloCore outperforms both in terms of area and energy efficiency.

4.4 Bit-serial vs. Bit-parallel vs. Self-synchronization based Parallel Decoding

The tree parsing based bit-serial decoding approach which is used in Huffman decoder doesn't apply in case of canonical Huffman decoder as the header devoids of tree information. LUT based bit-parallel approach provides faster decoding compared to bit-serial based approach. We have considered a single LUT based decoding approach which is a good fit for a single SRAM module. However, multi-LUT based approach can be used for space constrained designs at the cost of slower decoding for symbols with code lengths higher than the code length meant for primary table.

Self-synchronization based parallel decoder [18] offers a great improvement in throughput utilizing all the computing resources available in a GPU. However, this approach adds more challenge due to added complexity of finding synchronization point for each encoded sequence, proper selection of an optimum thread block and subsequence size per available GPU resources. Therefore, an optimized bit-parallel approach using a small number of energy-efficient and high performance processors is more suitable in terms of both area and energy efficiency while preserving Huffman's original method. Moreover, the GPU implementation [18] also decodes only one symbol per memory access like the bit-parallel approach presented in this work with each symbol of size one byte.

5 CONCLUSION

In conclusion, we discuss various canonical Huffman decoding methods and present bit-parallel architectures based on the original Huffman's method. Many-core processors array shows a promising results in terms of both area and energy efficiency across standard lossless compression benchmarks when compared to general purpose CPU and GPU. The proposed mappings on the many-core processor array can be easily scaled to support variety of compression standards with varying code lengths support. The GPU implementation of a massively parallel decoder based on self-synchronizing sequences of Huffman code results in a better throughput at the cost of power and area. However, self-synchronizing property for all possible Huffman codes may not be viable. In general, LUT based bit-parallel implementation on many-core processor array offers

Table 1: Throughput per area (Area efficiency) and Bytes decoded per energy (Energy efficiency) data for Intel i7, GPU, and Many-Core (KiloCore) implementations. The i7 and GPU values are scaled to 32 nm (technology for KiloCore) to account for differences in technology generations using data by Holt [10].

Benchmark	Platform	Throughput/Area Mbps/mm^2	Bytes compressed/Energy Bytes/µJ	Throughput/Area * Bytes compressed/Energy (Normalized to i7-4850HQ) Mbps/mm^2 * Bytes/µJ
Artificial Corpus	i7-4850HQ	0.4	0.9	1.0
	GT 750M	79.1	6.6	1450
	Many-Core	**172.0**	**23.7**	**11,320**
Calgary Corpus	i7-4850HQ	1.0	1.4	1.0
	GT 750M	190.4	3.3	449
	Many-Core	**448.2**	**14.5**	**4642**
Cantrbry Corpus	i7-4850HQ	0.8	1.1	1.0
	GT 750M	64.9	5.1	376
	Many-Core	**229.4**	**22.0**	**5735**
Large	i7-4850HQ	0.3	0.3	1.0
	GT 750M	14.2	2.1	331
	Many-Core	**39.5**	**13.0**	**5706**

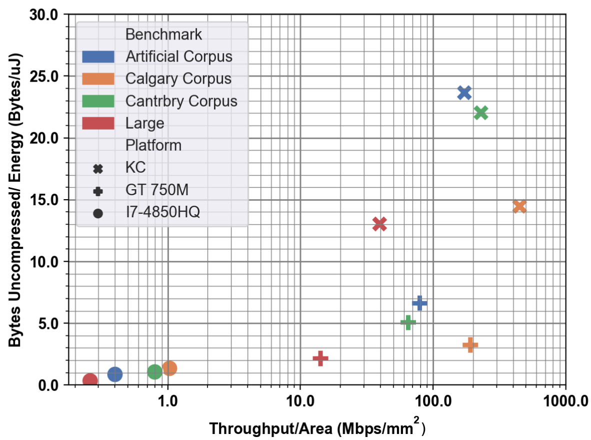

Figure 7: Comparison of scaled throughput per area (Mbps/mm^2) vs. scaled bytes compressed per energy (Bytes/µJ) for the many-core array (KiloCore), GeForce GT 750M GPU, and Intel i7 implementations.

better area and energy efficiency results while supporting all sorts of Huffman codes and yields no degradation in compression ratio.

REFERENCES

[1] 2020. *Intel Core i7-4850HQ.* Retrieved July 25, 2020 from https://www.notebookcheck.net/Intel-Core-i7-4850HQ-Notebook-Processor.90976.0.html

[2] 2020. *NVIDIA GeForce GT 750M.* Retrieved July 25, 2020 from https://www.techpowerup.com/gpu-specs/geforce-gt-750m.c2224

[3] C. A. Angulo, C. D. Hernández, G. Rincón, C. A. Boada, J. Castillo, and C. A. Fajardo. 2015. Accelerating huffman decoding of seismic data on GPUs. In *2015 20th Symposium on Signal Processing, Images and Computer Vision (STSIVA)*. 1–6.

[4] Z. Aspar, Z. Mohd Yusof, and I. Suleiman. 2000. Parallel Huffman decoder with an optimized look up table option on FPGA. In *2000 TENCON Proceedings. Intelligent Systems and Technologies for the New Millennium (Cat. No.00CH37119)*, Vol. 1. 73–76 vol.1.

[5] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, Bin Liu, A. Tran, E. Adeagbo, and B. Baas. 2016. KiloCore: A 32 nm 1000-processor array. In *2016 IEEE Hot Chips 28 Symposium (HCS)*. 1–23. https://doi.org/10.1109/HOTCHIPS.2016.7936218

[6] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. 2017. KiloCore: A Fine-Grained 1,000-Processor Array for Task-Parallel Applications. *IEEE Micro* 37, 2 (2017), 63–69.

[7] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. 2017. KiloCore: A 32-nm 1000-Processor Computational Array. *IEEE Journal of Solid-State Circuits* 52, 4 (2017), 891–902.

[8] Michael Dipperstein. 2018. *Huffman Code Discussion and Implementation.* Retrieved December 27, 2018 from https://michaeldipperstein.github.io/huffman.html

[9] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. 2017. Adaptive lossless data compression method optimized for GPU decompression. *Concurrency and Computation: Practice and Experience* 29, 24 (2017), e4283.

[10] W. M. Holt. 2016. 1.1 Moore's law: A path going forward. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. 8–13.

[11] D. A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[12] Y. Kim, I. Hong, and H. Yoo. 2015. 18.3 A 0.5V 54W ultra-low-power recognition processor with 93.5% accuracy geometric vocabulary tree and 47.5% database compression. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*. 1–3.

[13] J. Matai, J. Kim, and R. Kastner. 2014. Energy efficient canonical huffman encoding. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. 202–209.

[14] A. Mukherjee, N. Ranganathan, and M. Bassiouni. 1991. Efficient VLSI designs for data transformation of tree-based codes. *IEEE Transactions on Circuits and Systems* 38, 3 (1991), 306–314.

[15] A. Ozsoy and M. Swany. 2011. CULZSS: LZSS Lossless Data Compression on CUDA. In *2011 IEEE International Conference on Cluster Computing*. 403–411.

[16] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens. 2012. Parallel lossless data compression on the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–9.

[17] Seong Hwan Cho, T. Xanthopoulos, and A. P. Chandrakasan. 1999. A low power variable length decoder for MPEG-2 based on nonuniform fine-grain table partitioning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7, 2 (1999), 249–257.

[18] André Weißenberger and Bertil Schmidt. 2018. Massively Parallel Huffman Decoding on GPUs. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/3225058.3225076