# Parallel Huffman Coding

Francesco Bozzo - 229312
francesco.bozzo@studenti.unitn.it

Michele Yin - 229359
michele.yin@studenti.unitn.it

*Abstract*—**This report aims to explain the design and implementation of a parallel encoding and decoding tool for the Huffman algorithm. To scale horizontally with increasing hardware resources, the application uses both MPI for multiprocessing and OpenMP for multithreading. Our tool is able to process both single files and nested folders obtaining very competitive results with respect the other online-available benchmarks.**

## I. Introduction

The *Huffman algorithm* is designed to find a more convenient bit representation to store data through lossless compression. Instead of considering groups of eight bits to encode data, the Huffman algorithm uses variable-length sequences of bits defined as *alphabet*.

The objective of our work is to build a scalable Huffman encoder and decoder that are able to exploit the provided hardware resources. Moreover, the application should handle both single files and nested folders properly.

Even if nowadays the Huffman algorithm is mainly used for teaching purposes, its prefix mechanism is still part of many notable standards, such as Deflate (PKZIP's algorithm), JPEG, and MP3 compression algorithms. For this reason, there are no state-of-the-art online-available implementations to compare our tool with. In fact, differently from our work, many of the tools we found do not consider:

- low-level optimizations, such as to use in-memory buffers to speed up I/O timings [1]–[5];
- handling files of size bigger than some GiBs [1]–[3], [5]–[7];
- handling nested folders [1]–[8].

Unfortunately, due to lack of documentation, missing files, and some restrictions due to interactive prompting, we were able to execute properly only two other online-available implementations [3], [7].

## II. Serial Huffman Algorithm

### A. Priority Queue

The Huffman algorithm uses a minimum priority queue (MPQ) to build its alphabet efficiently. This specific data structure ensures logarithmic insertion and deletion times with respect to its size. We implemented the MPQ using a standard C array considered as a minimum heap, by ensuring that the min-heap property holds at every insertion and deletion:

$$A[i] \le A[l(i)], A[i] \le A[r(i)] \tag{1}$$

where $A[i]$, $A[l(i)]$, and $A[r(i)]$ are respectively a node, its left child, and its right child in a min-heap tree.
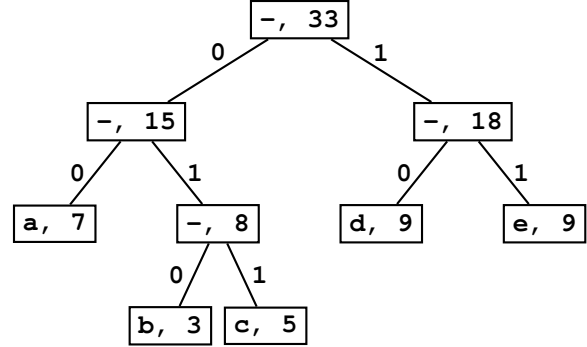


Fig. 1: An example of Huffman tree.

We also considered a parallel approach to implement MPQ [9], but it has been discarded due to the high complexity compared to the slim gains on a MPQ of at most 256 elements.

### B. Encoding

The Huffman encoding procedure makes use of a MPQ to build its alphabet efficiently. The idea is to build a tree similar to Figure 1 that defines all the variable length prefix sequences of bits: these sequences are represented by the path from the tree root to a leaf. Using a greedy approach, the Huffman encoding ensures that the less frequent a byte is in a file, the more likely is to have a longer Huffman representation, which means that his path from the root to its specific leaf is longer.

Once having computed the frequencies for each one of the 256 different bytes in a file, the Huffman algorithm populates a MPQ with Huffman tree nodes storing the byte value and its frequency. Successively, it removes the least two frequent bytes from the queue, creates a new node with children the two extracted nodes, assigns it to a dummy character and the sum of the frequencies of its two children, and inserts it into the MPQ. After $n - 1$ iterations, the last node in the MPQ is the actual root of the Huffman tree [10]. Algorithm 1 explains in the detail this procedure.

When the Huffman tree is built, it is possible to generate the Huffman alphabet by visiting the tree using a DFS algorithm; assigning 0 to each left-child traverse and 1 for the right one as presented in Algorithm 2. Finally, it is possible to compress the file by creating a stream of bits corresponding to its content using the Huffman alphabet.

---

**Algorithm 1:** Build the Huffman tree

---
**1** // Populate the min priority queue with characters and their frequencies
**2** **for** $i = 1$ **to** $n - 1$ **do**
**3**     Q.insert(f[i], Tree(f[i], c[i]))

**4** // Repeat until the queue has only a single element left
**5** **for** $i = 1$ **to** $n - 1$ **do**
**6**     // Get the two least frequent nodes
**7**     z1, z2 = Q.deleteMin(), Q.deleteMin()
**8**     // Create and insert inner tree node into the queue
**9**     z = Tree(z1.f + z2.f, null)
**10**     z.left, z.right = z1, z2
**11**     Q.insert(z.f, z)

**12** // The last element in the queue is the root of the Huffman tree
**13** **return** *Q.deleteMin()*

---

---

**Algorithm 2:** Encode(node)

---
**1** **while** *not eof()* **do**
**2**     bit = read()
**3**     **if** *bit == 0* **then**
**4**        Encode(node.left)
**5**     **else**
**6**        Encode(node.right)
**7**     **if** *node is leaf* **then**
**8**        **return** *node.value*

---

### C. Decoding

Finally, having the encoded file and the Huffman tree, it is straightforward to decompress the file to its original shape. The prefix alphabet ensures to have that it is possible to visit the tree using a DFS approach and get a unique decoded version of the file using an approach similar to Algorithm 2.

### D. Implementation

In order implement the Huffman encoding and decoding algorithm, several details have been taken into account.

- The byte frequencies computed during the encoding process are saved in the compressed file. In this way, the decoding procedure can easily rebuild the Huffman tree and therefore the alphabet.
- Both the encoding and decoding procedures make use of buffers to improve I/O performance: the streams of bytes are first written inside the buffer and then saved on the disk.
- The encoding procedure works with chunks of 4096 bytes. This ensures that the tool can handle very large files when dealing with bit buffers. The decoding procedure deals with chunks of maximum size of $4096 * 32$ bytes, since the maximum length of a Huffman tree traverse from root to leaf is of 256 bits, which is 32 times a byte.

As explained later, dealing with chunks makes handy the parallel implementation of the encoding and decoding algorithms.

- In the encoded file, the chunk sizes and offsets are saved at the end of the file: this is to avoid considering the last bits of the byte of a chunk, which may not be relevant because the encoded representation may not be a multiple of a byte.

## III. PARALLEL HUFFMAN IMPLEMENTATION

To parallelize the serial Huffman algorithm over threads and processes, we decided to follow this simple yet effective structure:

- Multiple processes should handle groups of file and folders separately.
- Multiple threads of the same process should work on different chunks of the same file in parallel.

Specifically, there are multiple reasons behind our architectural design for this project. Here we present a list of the main ones:

- In most operating systems a file is a resource that the OS gives to a single process to avoid race conditions.
- Most operating systems allow multiple processes to open the same file in reading mode, but only few allow to open the same file in writing mode on multiple processes. This is because it may lead to concurrency and data integrity issues. By ensuring to have only a single process that opens a specific file, we avoid all these issues.
- Because threads of the same process share the address space, we can avoid the expensive data transfer across processes. When data is read or written to file, there is no need to transfer data between threads.

In the next section, we start by explaining the basic multi-threading operations on a single file. Later on, multiprocessing with multiple files is covered.

### A. Multithreading

The parallel algorithm for a single file follows the exact same procedure as for the serial version, with just a few key differences to allow multithreading.

Suppose we are dealing with $m$ threads and a single input file. Here follows a common procedure used for both encoding and decoding, with very small differences in between them.

1) A single file is divided into chunks of a fixed size. When that is done, the single process forks and creates $m$ threads.
2) A single thread (not necessarily the main one) reads $m$ chunks in a shared memory buffer.
3) Each one of the $m$ threads works in parallel on the processing of its assigned chunk using a buffer. Although all the chunks are in shared memory buffer, each thread is assigned only to a single memory section: in this way there are no racing conditions when writing in the buffer. This is done by creating a `unsigned char buffer[m][buffer_size]`.
4) When all threads are done computing, either the encoding or decoding of their chunks, a single thread writes
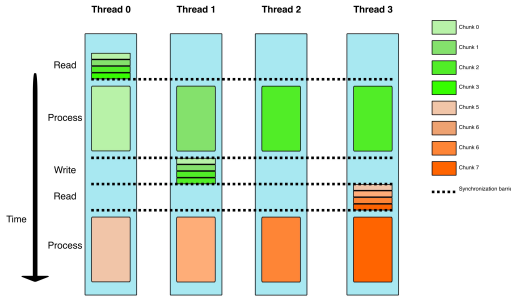
Fig. 2: Simple schema for processing a single file with multiple threads.

the processed chunks to the output file. Again, since all threads share a memory space, there is no need to perform data transfer.

Moreover, our implementation also parallelizes the counting of byte occurrences in the input file, since this is required in order to create the Huffman tree. This is done only during the encoding phase, as for the decoding the occurrences are already stored in the encoded data. The procedure is very similar to the one just described, with the key difference that there is no output file to write, but only byte occurrences stored in an in-memory array unique for each thread to avoid concurrency. When the counting is done, all threads join their counts into a single array. Figure 2 shows a schema of the parallel workflow for processing a single file.

### B. Multiprocessing

Additionally, we wanted our tool to be able to process multiple files and folders: for this use case, we exploit multiprocessing. To achieve this, we devised a simple algorithm to distribute files across multiple processes, so that each process can work on its own list of files, independently of others. In this way, we can minimize communication and therefore latency and other slowdowns between the different components of the applications. Suppose we are dealing with $n$ process, $m$ threads, and an input folder.

1) First, the process with rank 0 (the main process) crawls all the files in the input directory recursively in all the subfolders.
2) Then the rank 0 process opens all the files and reads their size.
3) Once all files and their respective sizes are known, the main process creates a MPQ where each item represents a process and its priority is the size of files that have been assigned to it.
4) Process 0 can iteratively insert files in the priority queue and updates the priority of each process with the cumulative file size. The main idea is that we can ensure an equal work division among different processes.
5) When all this is done, the main process sends the list of files to each process and then each starts to work independently on its own list of files.

### C. Implementation details and other notes

Here follows a list of further comments and design choices for our specific Huffman implementation.

- We decided to apply Huffman coding to single bytes. As the C language does not allow a straightforward addressing technique for data at lower resolution than a byte, such an alphabet would not result in any performance benefit. We also considered encoding multiple bytes into a single Huffman code, but that results in worse compression as the resulting Huffman tree would have more leaves.
- We found that a chunk size of 4096 bytes has the best I/O times. This is probably due to the fact that 4096 bytes is the size of a page in most Linux based operating systems. Reading any other size at the time had resulted in significantly worse I/O times (both increasing and decreasing the chunk size). The last chunk of a file may be smaller.
- In general, all threads will finish processing their chunks at around the same time, but in some cases there are threads that can take longer. This is due to the very unlucky case where a whole chunk contains exclusively very infrequent bytes of data. Because all bytes are very infrequent, each one is encoded in extremely long sequences, up to 256 bits, resulting in a bigger encoded chunk than the original data. With our Huffman alphabet set to a single byte, the worst case results in having a compressed chunk being 32 times bigger than decompressed one.
- We tried to parallelize the I/O, by having each thread read its own chunk. We found that the standard file descriptors provided by C have a lock to guarantee thread safety. Unfortunately, this lock slows down the reads significantly when happening on the same node. We tried using multiple file descriptors to circumvent this limitation, but we found no improvement over a single-thread sequential read. This is likely because the O.S. schedules I/O requests and serves them one at the time, resulting in sequential I/O.
- We also tested an architecture where we had two dedicated threads to I/O, one for reading and one for writing chunks. The idea is that whenever a chunk is processed, the I/O threads would immediately write the processed chunk to the output file and similarly a new chunk would be read from the input file (very similar to a queue of jobs). Theoretically this would allow to parallelize I/O and computation operations: in this scenario we avoided concurrency problems by synchronizing the I/O with locks.
- Another detail we noticed on the implementation with two dedicated threads for I/O, is that if there are not enough cores on the CPU or if the operating system decides to allocate all the threads on a single processor, the encoding and decoding times are greatly affected by the scheduler. This is because it might be that the O.S.

gives priority to threads that are waiting (for example the writer cannot write any block until the first has finished, even if all others have finished). In the worst case scenario, the multithreading architecture could be even slower that the serial one.

- Our memory requirements do not scale upwards with the size of the files we need to encode, since at any given time the most we only store is the size of two buffers, each is a `unsigned char buffer[num_threads][4096]`. These buffers are emptied at each I/O cycle, therefore leading to a high memory efficiency. Only when processing multiple files our memory requirements scale up with the number of files, as each process needs to store information about the files is assigned to process.

## IV. PERFORMANCE AND BENCHMARKING

In this section we are going to discuss the performance evaluation of our algorithm. We start from evaluating some other implementations of Huffman algorithm and then focus on the analysis of our own implementation.

We include in this section some graphs and statistics, but all detailed results are included in the appendix at the end this report.

### A. Setup

All the tests were performed on the HPC2 cluster of the University of Trento. To reduce as much as possible the I/O timings, which are crucial of our application, jobs were run in a very specific setting:

- Threads are allocated in the same node and, if possible, on the same socket to guarantee fast communication between them. This has been achieved by using the MPI option `--map-by`.
- Processes are allocated in different nodes, by using the PBS directive `place=scatter:excl`. This because in our application different processes do not communicate very often between each other, but they just split their work and use the file-system. We also tested `place=pack:excl` but found worse results.

The provided timings are obtained by averaging the result of three runs with the exact same configurations: this helps to minimize the effect random variance between runs due to potential hardware congestion. Each job has been submitted to the cluster individually, waiting for the previous to finish preventing hiccups due to multiple instances of the program trying to access the same file-system resource, which we noticed to greatly affect I/O times.

We extensively verified the correctness of our algorithm by encoding and then decoding a file, then comparing the decoded result to the original file by using the `diff` command. Moreover, Valgrind has been used to spot any memory leaks that could cause runtime errors.

### B. Datasets

The evaluation dataset has been randomly generated by using a simple C program: we created text files with character frequencies that follow the occurrences of the letters in English language. Although for benchmarking we are only using ASCII characters, please note that our algorithm reads bytes of data and can therefore work with any file. The dataset is composed by files of increasing size: 1 MiB, 5 MiB, 10 MiB, 50 MiB, 100 MiB, 500 MiB, 1 GiB, 5 GiB, and 10 GiB. This should be enough range to understand how different algorithms scale with increasing file sizes.

Secondly, because our algorithm also works with many files at once, we chose some GitHub repositories as benchmark: Numpy [11], PyTorch [12] and Linux kernel [13]. They are respectively $\sim 30$ MiB, $\sim 200$ MiB and $\sim 1.3$ GiB of total size, with about $\sim 2.000$, $\sim 12.000$, and $\sim 80.000$ files, which on average are each 16MiB.

### C. Multithreading evaluation

In this section we evaluate our algorithm on single files, by using multithreading and testing our parallelization performances with an increasing number of threads.

With small files (i.e. less than a few MiBs) the algorithm is actually slower with more threads. This is likely because the data is not large enough to offset the cost of forking multiple threads. Considering the cost parallel processing, we highly suggest not doing parallelization for small files. However, when the data is large enough our multithreaded approach scales up. With the largest file (10 GiB) our algorithm has an efficiency of 93% when tested with 4 threads. This number steadily decreases with the number of threads, and we hit 16% with 64 threads.

The same issue is even more visible by looking at one implementation we found on GitHub that uses only MPI for parallelization [7]: with many processes, while with large files (even though it does not work with files bigger than $\sim 500$ MiB) it performs better than our implementation, with small files it is not able to scale. This is happening due to the overhead of splitting small files among too many processes.

Additionally, please note that our tool is heavily limited by the cluster I/O: especially writes to disk require a considerable amount of time. As a proof of this, if we consider only the processing section of our algorithm, and ignore the time required by the `fflush()` operation before `fclose()`, we see significantly decreased times. Although we start from similar efficiency at low threading, we achieve a 41% efficiency with 64 threads in this case.

We also noticed worse times in the decoding procedure with respect to encoding. This is probably due to the fact that the tool is reading fixed chunks of 4096 bytes in encoding, but during decoding the chunk can have a variable length. Moreover, by executing another implementation found on GitHub [3],we found that reading and writing buffers to the disk (and not single bytes) can obtain improvement very similar to one order of magnitude.

As already discussed previously, we also tested a version on a lock-based synchronization instead of barriers. Although theoretically we should get better results, in practice we obtain almost equal performances across the board, with only the
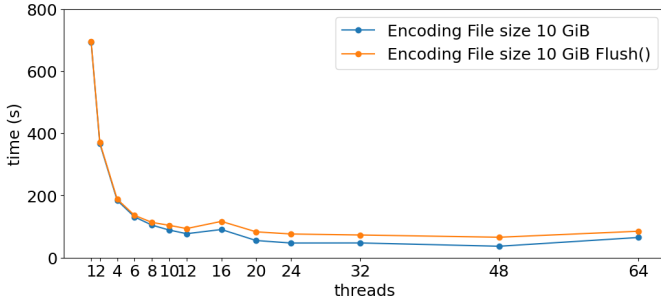
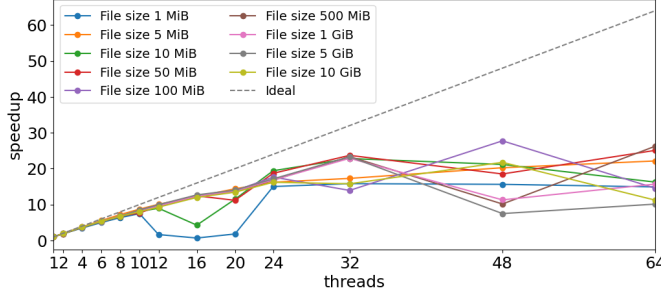Fig. 3: Encoding times with and without the flush()



Fig. 4: Encoding speedup

largest files having some significant difference. This difference is greater in decoding than in encoding.

### D. Multiprocessing

We tested on different folders the parallelization performances of our algorithm. Theoretically, it should scale very well with increasing number of processes and files, because once the main process sends the list of files to encode to each process there is no further communication between different processes. However, we find that real world results are different.

While we do not expect much performance improvement by the number of threads, because the files are on average very small, we should get better results by using many processes. In reality, our testing shows that this is not the case, probably due to the slow cluster I/O. When tested on a local machine (MacBook A2442) with one thread and 1,2,4 processes, we find that our algorithm does indeed scale with the number
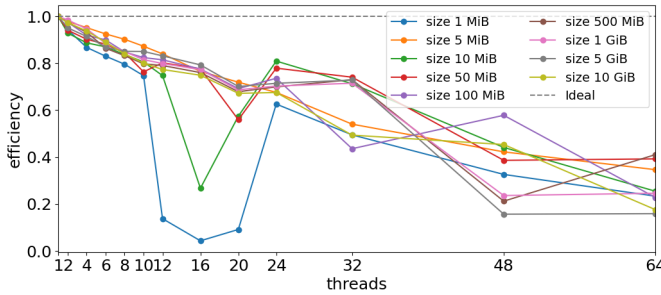


Fig. 5: Encoding efficiency
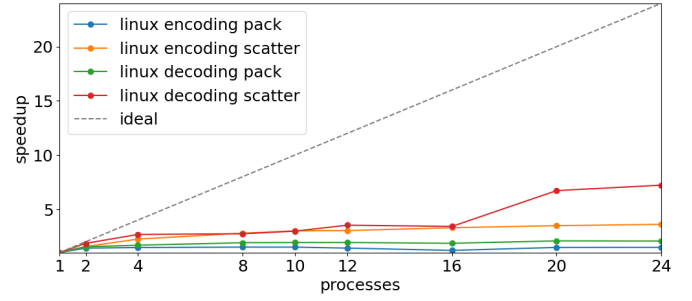


Fig. 6: Linux speedup for different placing strategies

of processes. We also find that `scatter` scales better than `pack`, probably because with `scatter` we are not limited by the I/O of a single node.

### REFERENCES

[1] YanaRGeorgieva, "Huffmanparallel," 2018. [Online]. Available: https://github.com/YanaRGeorgieva/HuffmanParallel

[2] oliviersm199, "Parallelizedtextcompressio," 2016. [Online]. Available: https://github.com/oliviersm199/Parallelized-Text-Compression

[3] spencerhuston, "Huffmanparallel2," 2018. [Online]. Available: https://github.com/spencerhuston/Huffman-Parallel

[4] S. Warne, "Huffman," 2017. [Online]. Available: https://github.com/Sean-Warne/Huffman

[5] patberry99, "Huffmancoding," 2022. [Online]. Available: https://github.com/patberry99/Huffman-Coding

[6] raemarks, "Parallelhuffmanencoding," 2015. [Online]. Available: https://github.com/raemarks/ParallelHuffmanEncoding

[7] smadhiv, "Huffmancodingmpicuda," 2017. [Online]. Available: https://github.com/smadhiv/HuffmanCoding_MPI_CUDA

[8] Billkyriaf, "pdsassignment4," 2022. [Online]. Available: https://github.com/Billkyriaf/pds_assignment_4

[9] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis, "A parallel priority queue with constant time operations," *Journal of Parallel and Distributed Computing*, vol. 49, no. 1, pp. 4–21, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731598914253

[10] A. Bertossi and A. Montresor, *Algoritmi e strutture di dati*. CittàStudi, 2010. [Online]. Available: https://books.google.es/books?id=WKWLSQAACAAJ

[11] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357362, 2020.

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[13] torvalds, "linux," 2022. [Online]. Available: https://github.com/torvalds/linux

## V. Appendix

### A. Encoding

**TABLE I: Overall encoding times**

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.108 | 0.103 | 0.083 | 0.091 | 0.072 | 0.110 | 0.134 | 0.211 | 0.170 | 0.030 | **0.025** | 0.027 | 0.026 |
| 5 MiB | 0.407 | 0.201 | 0.126 | 0.104 | 0.097 | 0.079 | 0.079 | 0.077 | 0.091 | 0.181 | 0.056 | 0.058 | **0.053** |
| 10 MiB | 0.761 | 0.474 | 0.266 | 0.208 | 0.180 | 0.186 | 0.220 | 0.335 | 0.243 | 0.102 | **0.089** | 0.096 | 0.170 |
| 50 MiB | 3.924 | 2.138 | 1.279 | 1.020 | 0.857 | 0.759 | 0.777 | 0.658 | 0.752 | 0.424 | 0.537 | 0.417 | **0.394** |
| 100 MiB | 7.727 | 4.068 | 2.403 | 1.860 | 1.554 | 1.432 | 1.299 | 1.247 | 1.231 | 0.856 | 0.945 | **0.720** | 0.888 |
| 500 MiB | 35.775 | 18.383 | 11.496 | 8.351 | 7.559 | 5.755 | 6.086 | 5.570 | 5.143 | 3.800 | 3.669 | 5.378 | **3.538** |
| 1 GiB | 73.112 | 38.168 | 22.507 | 16.336 | 14.213 | 13.756 | 12.620 | 10.826 | 10.335 | 7.810 | **7.468** | 10.505 | 8.655 |
| 5 GiB | 359.849 | 186.629 | 99.804 | 77.462 | 65.655 | 67.306 | 52.909 | 48.459 | 48.013 | **36.637** | 38.346 | 59.600 | 47.238 |
| 10 GiB | 694.975 | 370.549 | 188.228 | 136.172 | 113.588 | 104.025 | 93.723 | 116.825 | 83.208 | 76.289 | 73.011 | **65.807** | 85.032 |

**TABLE II: Pure encoding times**

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.064 | 0.034 | 0.018 | 0.013 | 0.010 | 0.009 | 0.039 | 0.094 | 0.035 | **0.004** | **0.004** | **0.004** | **0.004** |
| 5 MiB | 0.298 | 0.152 | 0.078 | 0.054 | 0.041 | 0.034 | 0.030 | 0.024 | 0.021 | 0.018 | 0.017 | 0.015 | **0.013** |
| 10 MiB | 0.655 | 0.352 | 0.184 | 0.125 | 0.098 | 0.081 | 0.073 | 0.153 | 0.057 | 0.034 | **0.029** | 0.031 | 0.040 |
| 50 MiB | 3.303 | 1.755 | 0.915 | 0.624 | 0.488 | 0.433 | 0.343 | 0.266 | 0.295 | 0.177 | 0.139 | 0.178 | **0.132** |
| 100 MiB | 6.721 | 3.446 | 1.827 | 1.245 | 0.990 | 0.815 | 0.687 | 0.542 | 0.486 | 0.380 | 0.482 | **0.242** | 0.462 |
| 500 MiB | 32.383 | 16.605 | 8.682 | 6.246 | 4.862 | 4.059 | 3.413 | 2.662 | 2.385 | 1.929 | 1.386 | 3.187 | **1.233** |
| 1 GiB | 67.176 | 34.097 | 17.735 | 12.655 | 9.879 | 8.237 | 7.004 | 5.431 | 4.892 | 3.982 | **2.936** | 5.939 | 4.277 |
| 5 GiB | 331.253 | 174.089 | 90.562 | 63.491 | 48.645 | 38.946 | 33.185 | 26.100 | 23.623 | 19.312 | **14.208** | 44.206 | 32.709 |
| 10 GiB | 639.337 | 328.580 | 170.409 | 119.682 | 95.123 | 79.805 | 68.875 | 53.342 | 47.573 | 39.444 | 40.491 | **29.375** | 56.942 |

**TABLE III: Overall encoding efficiency**

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.527 | 0.328 | 0.200 | 0.189 | 0.099 | 0.068 | 0.032 | 0.032 | 0.151 | 0.135 | 0.083 | 0.064 |
| 5 MiB | 1.000 | 1.014 | 0.811 | 0.654 | 0.525 | 0.518 | 0.430 | 0.332 | 0.225 | 0.094 | 0.229 | 0.146 | 0.119 |
| 10 MiB | 1.000 | 0.803 | 0.716 | 0.609 | 0.527 | 0.409 | 0.288 | 0.142 | 0.156 | 0.311 | 0.267 | 0.165 | 0.070 |
| 50 MiB | 1.000 | 0.918 | 0.767 | 0.641 | 0.572 | 0.517 | 0.421 | 0.373 | 0.261 | 0.386 | 0.229 | 0.196 | 0.156 |
| 100 MiB | 1.000 | 0.950 | 0.804 | 0.693 | 0.621 | 0.539 | 0.496 | 0.387 | 0.314 | 0.376 | 0.255 | 0.224 | 0.136 |
| 500 MiB | 1.000 | 0.973 | 0.778 | 0.714 | 0.592 | 0.622 | 0.490 | 0.401 | 0.348 | 0.392 | 0.305 | 0.139 | 0.158 |
| 1 GiB | 1.000 | 0.958 | 0.812 | 0.746 | 0.643 | 0.531 | 0.483 | 0.422 | 0.354 | 0.390 | 0.306 | 0.145 | 0.132 |
| 5 GiB | 1.000 | 0.964 | 0.901 | 0.774 | 0.685 | 0.535 | 0.567 | 0.464 | 0.375 | 0.409 | 0.293 | 0.126 | 0.119 |
| 10 GiB | 1.000 | 0.938 | 0.923 | 0.851 | 0.765 | 0.668 | 0.618 | 0.372 | 0.418 | 0.380 | 0.297 | 0.220 | 0.128 |

**TABLE IV: Pure encoding efficiency**

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.943 | 0.868 | 0.830 | 0.796 | 0.747 | 0.137 | 0.042 | 0.091 | 0.626 | 0.495 | 0.326 | 0.233 |
| 5 MiB | 1.000 | 0.979 | 0.952 | 0.925 | 0.902 | 0.873 | 0.840 | 0.765 | 0.719 | 0.676 | 0.540 | 0.422 | 0.346 |
| 10 MiB | 1.000 | 0.929 | 0.887 | 0.870 | 0.837 | 0.806 | 0.748 | 0.267 | 0.574 | 0.810 | 0.713 | 0.441 | 0.254 |
| 50 MiB | 1.000 | 0.941 | 0.903 | 0.882 | 0.845 | 0.764 | 0.802 | 0.777 | 0.559 | 0.779 | 0.741 | 0.386 | 0.392 |
| 100 MiB | 1.000 | 0.975 | 0.920 | 0.900 | 0.848 | 0.824 | 0.815 | 0.776 | 0.692 | 0.736 | 0.436 | 0.578 | 0.227 |
| 500 MiB | 1.000 | 0.975 | 0.932 | 0.864 | 0.832 | 0.798 | 0.791 | 0.760 | 0.679 | 0.699 | 0.730 | 0.212 | 0.410 |
| 1 GiB | 1.000 | 0.985 | 0.947 | 0.885 | 0.850 | 0.816 | 0.799 | 0.773 | 0.687 | 0.703 | 0.715 | 0.236 | 0.245 |
| 5 GiB | 1.000 | 0.951 | 0.914 | 0.870 | 0.851 | 0.851 | 0.832 | 0.793 | 0.701 | 0.715 | 0.729 | 0.156 | 0.158 |
| 10 GiB | 1.000 | 0.973 | 0.938 | 0.890 | 0.840 | 0.801 | 0.774 | 0.749 | 0.672 | 0.675 | 0.493 | 0.453 | 0.175 |

## B. Encoding with Locks

TABLE V: Overall encoding times

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.088 | 0.084 | 0.073 | 0.065 | 0.072 | 0.072 | 0.067 | 0.067 | 0.075 | 0.071 | 0.031 | **0.027** | 0.033 |
| 5 MiB | 0.425 | 0.406 | 0.275 | 0.172 | 0.167 | 0.164 | 0.148 | 0.139 | 0.139 | 0.136 | **0.058** | 0.064 | 0.059 |
| 10 MiB | 0.713 | 0.678 | 0.242 | 0.194 | 0.176 | 0.147 | 0.168 | 0.129 | 0.140 | 0.157 | 0.102 | **0.094** | 0.102 |
| 50 MiB | 3.585 | 3.371 | 1.141 | 0.838 | 0.748 | 0.698 | 0.961 | 0.968 | 0.733 | 0.895 | 0.430 | **0.388** | 0.573 |
| 100 MiB | 6.805 | 6.601 | 2.159 | 1.622 | 1.359 | 1.200 | 1.098 | 0.974 | 0.888 | 0.842 | 0.771 | **0.722** | 0.735 |
| 500 MiB | 33.464 | 31.995 | 10.564 | 7.854 | 6.683 | 6.041 | 5.518 | 5.931 | 4.577 | 4.248 | 4.140 | **3.551** | 3.811 |
| 1 GiB | 68.489 | 65.942 | 21.173 | 16.425 | 13.648 | 12.077 | 10.966 | 9.795 | 9.036 | 8.342 | 8.007 | 8.001 | **7.690** |
| 5 GiB | 367.228 | 370.850 | 110.299 | 84.128 | 75.319 | 66.408 | 59.666 | 59.479 | 51.181 | 48.904 | 38.824 | **38.677** | 40.676 |
| 10 GiB | 676.529 | 642.922 | 187.620 | 135.991 | 123.822 | 109.770 | 96.643 | 93.226 | 90.825 | 88.154 | 63.886 | **61.847** | 64.598 |

TABLE VI: Pure encoding times

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.061 | 0.062 | 0.036 | 0.028 | 0.030 | 0.022 | 0.024 | 0.024 | 0.030 | 0.016 | 0.011 | **0.009** | 0.011 |
| 5 MiB | 0.361 | 0.363 | 0.209 | 0.109 | 0.103 | 0.092 | 0.079 | 0.066 | 0.053 | 0.050 | 0.023 | 0.023 | **0.017** |
| 10 MiB | 0.617 | 0.607 | 0.169 | 0.117 | 0.090 | 0.076 | 0.078 | 0.054 | 0.053 | 0.059 | 0.039 | **0.030** | 0.035 |
| 50 MiB | 3.131 | 3.057 | 0.847 | 0.595 | 0.464 | 0.416 | 0.671 | 0.652 | 0.435 | 0.500 | 0.186 | **0.145** | 0.160 |
| 100 MiB | 6.079 | 6.037 | 1.664 | 1.151 | 0.891 | 0.735 | 0.642 | 0.521 | 0.438 | 0.386 | 0.327 | 0.285 | **0.249** |
| 500 MiB | 31.001 | 30.471 | 8.287 | 5.810 | 4.519 | 3.778 | 3.226 | 2.610 | 2.227 | 1.946 | 1.814 | 1.430 | **1.404** |
| 1 GiB | 62.748 | 62.355 | 17.230 | 11.939 | 9.153 | 7.582 | 6.505 | 5.363 | 4.612 | 4.101 | 3.698 | 2.966 | **2.833** |
| 5 GiB | 343.876 | 351.067 | 99.909 | 73.825 | 62.694 | 54.050 | 47.849 | 42.794 | 33.944 | 29.952 | 18.749 | 14.847 | **14.376** |
| 10 GiB | 629.449 | 623.710 | 172.829 | 120.552 | 94.623 | 78.728 | 68.025 | 55.215 | 46.723 | 41.474 | 32.493 | 27.176 | **23.935** |

TABLE VII: Overall encoding efficiency

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.526 | 0.300 | 0.224 | 0.152 | 0.122 | 0.109 | 0.082 | 0.059 | 0.051 | 0.089 | 0.067 | 0.042 |
| 5 MiB | 1.000 | 0.523 | 0.386 | 0.410 | 0.317 | 0.259 | 0.239 | 0.191 | 0.153 | 0.130 | 0.229 | 0.138 | 0.113 |
| 10 MiB | 1.000 | 0.525 | 0.737 | 0.613 | 0.506 | 0.485 | 0.354 | 0.346 | 0.254 | 0.189 | 0.219 | 0.158 | 0.109 |
| 50 MiB | 1.000 | 0.532 | 0.786 | 0.713 | 0.599 | 0.513 | 0.311 | 0.231 | 0.245 | 0.167 | 0.261 | 0.192 | 0.098 |
| 100 MiB | 1.000 | 0.515 | 0.788 | 0.699 | 0.626 | 0.567 | 0.516 | 0.437 | 0.383 | 0.337 | 0.276 | 0.196 | 0.145 |
| 500 MiB | 1.000 | 0.523 | 0.792 | 0.710 | 0.626 | 0.554 | 0.505 | 0.353 | 0.366 | 0.328 | 0.253 | 0.196 | 0.137 |
| 1 GiB | 1.000 | 0.519 | 0.809 | 0.695 | 0.627 | 0.567 | 0.520 | 0.437 | 0.379 | 0.342 | 0.267 | 0.178 | 0.139 |
| 5 GiB | 1.000 | 0.495 | 0.832 | 0.728 | 0.609 | 0.553 | 0.513 | 0.386 | 0.359 | 0.313 | 0.296 | 0.198 | 0.141 |
| 10 GiB | 1.000 | 0.526 | 0.901 | 0.829 | 0.683 | 0.616 | 0.583 | 0.454 | 0.372 | 0.320 | 0.331 | 0.228 | 0.164 |

TABLE VIII: Pure encoding efficiency

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.493 | 0.425 | 0.367 | 0.252 | 0.283 | 0.209 | 0.160 | 0.101 | 0.161 | 0.172 | 0.138 | 0.085 |
| 5 MiB | 1.000 | 0.497 | 0.431 | 0.551 | 0.438 | 0.391 | 0.380 | 0.344 | 0.341 | 0.300 | 0.491 | 0.331 | 0.334 |
| 10 MiB | 1.000 | 0.508 | 0.913 | 0.881 | 0.852 | 0.816 | 0.655 | 0.714 | 0.583 | 0.436 | 0.495 | 0.426 | 0.275 |
| 50 MiB | 1.000 | 0.512 | 0.924 | 0.878 | 0.843 | 0.753 | 0.389 | 0.300 | 0.360 | 0.261 | 0.525 | 0.449 | 0.306 |
| 100 MiB | 1.000 | 0.503 | 0.914 | 0.880 | 0.853 | 0.827 | 0.789 | 0.729 | 0.693 | 0.656 | 0.581 | 0.444 | 0.381 |
| 500 MiB | 1.000 | 0.509 | 0.935 | 0.889 | 0.857 | 0.821 | 0.801 | 0.742 | 0.696 | 0.664 | 0.534 | 0.452 | 0.345 |
| 1 GiB | 1.000 | 0.503 | 0.910 | 0.876 | 0.857 | 0.828 | 0.804 | 0.731 | 0.680 | 0.638 | 0.530 | 0.441 | 0.346 |
| 5 GiB | 1.000 | 0.490 | 0.860 | 0.776 | 0.686 | 0.636 | 0.599 | 0.502 | 0.507 | 0.478 | 0.573 | 0.483 | 0.374 |
| 10 GiB | 1.000 | 0.505 | 0.911 | 0.870 | 0.832 | 0.800 | 0.771 | 0.712 | 0.674 | 0.632 | 0.605 | 0.483 | 0.411 |

## C. Decoding

### TABLE IX: Overall decoding times

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.130 | 0.091 | 0.079 | 0.099 | 0.091 | 0.097 | 0.118 | 0.115 | 0.119 | 0.104 | 0.045 | **0.026** | 0.030 |
| 5 MiB | 0.422 | 0.253 | 0.177 | 0.166 | 0.141 | 0.156 | 0.151 | 0.139 | 0.130 | 0.121 | 0.087 | 0.087 | **0.082** |
| 10 MiB | 0.871 | 0.517 | 0.384 | 0.296 | 0.293 | 0.433 | 0.372 | 0.356 | 0.368 | 0.355 | 0.171 | **0.129** | 0.161 |
| 50 MiB | 4.108 | 2.277 | 1.511 | 1.252 | 1.276 | 1.382 | 1.412 | 1.409 | 1.434 | 1.328 | 0.693 | 0.594 | **0.544** |
| 100 MiB | 7.961 | 4.507 | 2.915 | 2.275 | 1.884 | 2.418 | 2.400 | 2.422 | 2.343 | 2.251 | 4.392 | 1.858 | **1.722** |
| 500 MiB | 37.619 | 22.511 | 13.501 | 11.334 | 10.035 | 9.326 | 9.640 | 11.182 | 11.069 | 10.021 | 19.667 | **5.680** | 7.006 |
| 1 GiB | 76.679 | 43.252 | 27.585 | 23.271 | 18.590 | 17.824 | 18.532 | 22.027 | 19.813 | 17.128 | 37.634 | **12.183** | 14.964 |
| 5 GiB | 361.871 | 193.112 | 108.812 | 96.790 | 81.066 | 84.897 | 79.142 | **73.725** | 79.374 | 75.976 | 155.945 | 111.347 | 81.739 |
| 10 GiB | 759.072 | 391.202 | 212.381 | 156.396 | 135.844 | 135.401 | 134.098 | 138.000 | 134.006 | 132.255 | 253.313 | **106.227** | 119.210 |

### TABLE X: Pure decoding times

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.083 | 0.058 | 0.052 | 0.074 | 0.069 | 0.071 | 0.092 | 0.090 | 0.092 | 0.076 | 0.015 | **0.005** | **0.005** |
| 5 MiB | 0.340 | 0.188 | 0.123 | 0.101 | 0.084 | 0.094 | 0.089 | 0.079 | 0.073 | 0.062 | 0.020 | 0.018 | **0.016** |
| 10 MiB | 0.752 | 0.414 | 0.267 | 0.191 | 0.191 | 0.265 | 0.267 | 0.246 | 0.282 | 0.249 | 0.059 | 0.032 | **0.029** |
| 50 MiB | 3.634 | 1.888 | 1.058 | 0.745 | 0.670 | 0.976 | 0.954 | 0.943 | 0.966 | 0.857 | 0.271 | 0.148 | **0.132** |
| 100 MiB | 7.132 | 3.613 | 2.012 | 1.388 | 1.123 | 1.540 | 1.520 | 1.541 | 1.478 | 1.468 | 2.433 | **0.299** | 0.511 |
| 500 MiB | 36.395 | 18.594 | 9.874 | 6.980 | 5.458 | 4.707 | 5.739 | 6.970 | 6.499 | 5.473 | 16.056 | **1.514** | 2.716 |
| 1 GiB | 74.764 | 38.408 | 19.764 | 13.840 | 10.908 | 10.242 | 9.231 | 12.504 | 12.060 | 9.825 | 29.059 | **3.072** | 5.671 |
| 5 GiB | 359.883 | 188.886 | 97.788 | 82.355 | 56.999 | 59.936 | 50.887 | 39.291 | 48.562 | 48.877 | 141.437 | **15.227** | 55.262 |
| 10 GiB | 754.842 | 385.021 | 199.850 | 138.739 | 113.105 | 95.262 | 81.112 | 108.727 | 94.317 | 81.262 | 245.681 | **32.584** | 75.318 |

### TABLE XI: Overall decoding efficiency

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.719 | 0.411 | 0.220 | 0.179 | 0.134 | 0.092 | 0.071 | 0.055 | 0.052 | 0.090 | 0.103 | 0.067 |
| 5 MiB | 1.000 | 0.835 | 0.595 | 0.423 | 0.374 | 0.270 | 0.233 | 0.189 | 0.163 | 0.145 | 0.152 | 0.101 | 0.080 |
| 10 MiB | 1.000 | 0.842 | 0.567 | 0.491 | 0.371 | 0.201 | 0.195 | 0.153 | 0.118 | 0.102 | 0.159 | 0.140 | 0.085 |
| 50 MiB | 1.000 | 0.902 | 0.680 | 0.547 | 0.402 | 0.297 | 0.242 | 0.182 | 0.143 | 0.129 | 0.185 | 0.144 | 0.118 |
| 100 MiB | 1.000 | 0.883 | 0.683 | 0.583 | 0.528 | 0.329 | 0.276 | 0.205 | 0.170 | 0.147 | 0.057 | 0.089 | 0.072 |
| 500 MiB | 1.000 | 0.836 | 0.697 | 0.553 | 0.469 | 0.403 | 0.325 | 0.210 | 0.170 | 0.156 | 0.060 | 0.138 | 0.084 |
| 1 GiB | 1.000 | 0.886 | 0.695 | 0.549 | 0.516 | 0.430 | 0.345 | 0.218 | 0.194 | 0.187 | 0.064 | 0.131 | 0.080 |
| 5 GiB | 1.000 | 0.937 | 0.831 | 0.623 | 0.558 | 0.426 | 0.381 | 0.307 | 0.228 | 0.198 | 0.073 | 0.068 | 0.069 |
| 10 GiB | 1.000 | 0.970 | 0.894 | 0.809 | 0.698 | 0.561 | 0.472 | 0.344 | 0.283 | 0.239 | 0.094 | 0.149 | 0.099 |

### TABLE XII: Pure decoding efficiency

| Size \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.720 | 0.401 | 0.188 | 0.150 | 0.117 | 0.075 | 0.058 | 0.045 | 0.046 | 0.170 | 0.338 | 0.245 |
| 5 MiB | 1.000 | 0.902 | 0.690 | 0.558 | 0.504 | 0.361 | 0.317 | 0.269 | 0.233 | 0.227 | 0.519 | 0.404 | 0.324 |
| 10 MiB | 1.000 | 0.909 | 0.705 | 0.655 | 0.493 | 0.284 | 0.234 | 0.191 | 0.133 | 0.126 | 0.397 | 0.490 | 0.402 |
| 50 MiB | 1.000 | 0.962 | 0.859 | 0.813 | 0.678 | 0.372 | 0.317 | 0.241 | 0.188 | 0.177 | 0.419 | 0.511 | 0.429 |
| 100 MiB | 1.000 | 0.987 | 0.886 | 0.856 | 0.794 | 0.463 | 0.391 | 0.289 | 0.241 | 0.202 | 0.092 | 0.497 | 0.218 |
| 500 MiB | 1.000 | 0.979 | 0.922 | 0.869 | 0.834 | 0.773 | 0.528 | 0.326 | 0.280 | 0.277 | 0.071 | 0.501 | 0.209 |
| 1 GiB | 1.000 | 0.973 | 0.946 | 0.900 | 0.857 | 0.730 | 0.675 | 0.374 | 0.310 | 0.317 | 0.080 | 0.507 | 0.206 |
| 5 GiB | 1.000 | 0.953 | 0.920 | 0.728 | 0.789 | 0.600 | 0.589 | 0.572 | 0.371 | 0.307 | 0.080 | 0.492 | 0.102 |
| 10 GiB | 1.000 | 0.980 | 0.944 | 0.907 | 0.834 | 0.792 | 0.776 | 0.434 | 0.400 | 0.387 | 0.096 | 0.483 | 0.157 |

## D. Decoding with Locks

### TABLE XIII: Overall decoding times

| Sizes \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.272 | 0.212 | 0.116 | 0.101 | 0.070 | 0.068 | 0.066 | 0.069 | 0.075 | 0.093 | 0.095 | **0.036** | **0.036** |
| 5 MiB | 0.635 | 0.915 | 1.265 | 0.655 | 0.684 | 0.289 | 0.482 | 0.298 | 0.330 | 0.180 | 0.562 | **0.124** | 0.254 |
| 10 MiB | 0.861 | 0.501 | 0.333 | 0.271 | 0.269 | 0.258 | 0.236 | 0.217 | 0.190 | **0.183** | 0.604 | 0.440 | 0.404 |
| 50 MiB | 3.991 | 2.248 | 1.462 | 1.176 | 1.160 | 1.088 | 1.017 | 0.913 | 0.837 | 0.775 | 1.409 | 0.761 | **0.726** |
| 100 MiB | 7.615 | 33.645 | 15.250 | 11.628 | 13.018 | 10.761 | 9.192 | 3.661 | 5.691 | 4.475 | 2.175 | **1.312** | 1.464 |
| 500 MiB | 37.082 | 20.422 | 13.236 | 11.344 | 10.774 | 10.441 | 9.767 | 8.661 | 7.387 | 7.051 | 10.581 | 9.984 | **6.591** |
| 1 GiB | 73.855 | 41.185 | 26.929 | 23.051 | 17.977 | 18.781 | 19.731 | 17.837 | 16.171 | 15.437 | 19.766 | 14.432 | **12.636** |
| 5 GiB | 352.386 | 475.087 | 265.894 | 204.661 | 156.227 | 139.894 | 127.649 | 109.245 | 100.918 | 89.890 | 104.173 | 65.082 | **63.837** |
| 10 GiB | 714.907 | 376.029 | 202.712 | 190.071 | 435.224 | 145.673 | 133.319 | 122.774 | 117.529 | **115.262** | 158.281 | 150.057 | 148.834 |

### TABLE XIV: Pure decoding times

| Sizes \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 0.116 | 0.064 | 0.056 | 0.056 | 0.041 | 0.041 | 0.038 | 0.041 | 0.038 | 0.035 | 0.061 | **0.006** | 0.007 |
| 5 MiB | 0.565 | 0.859 | 1.202 | 0.598 | 0.627 | 0.234 | 0.425 | 0.244 | 0.269 | 0.113 | 0.019 | 0.017 | **0.016** |
| 10 MiB | 0.715 | 0.399 | 0.234 | 0.174 | 0.169 | 0.159 | 0.136 | 0.119 | 0.095 | 0.085 | 0.132 | 0.031 | **0.028** |
| 50 MiB | 3.531 | 1.848 | 1.034 | 0.742 | 0.730 | 0.656 | 0.620 | 0.486 | 0.417 | 0.352 | 0.703 | 0.143 | **0.117** |
| 100 MiB | 6.874 | 33.366 | 14.884 | 10.786 | 12.377 | 9.907 | 8.348 | 2.879 | 5.046 | 3.695 | 0.968 | **0.285** | 0.510 |
| 500 MiB | 35.450 | 18.238 | 10.009 | 7.179 | 6.792 | 6.317 | 5.622 | 4.667 | 3.831 | 3.280 | 6.218 | 1.394 | **1.212** |
| 1 GiB | 72.114 | 36.926 | 20.113 | 14.503 | 11.600 | 12.692 | 11.302 | 9.281 | 7.654 | 6.478 | 9.440 | 2.841 | **2.705** |
| 5 GiB | 349.890 | 471.340 | 257.714 | 197.061 | 140.154 | 124.853 | 110.634 | 91.401 | 74.582 | 62.539 | 73.525 | 14.510 | **12.708** |
| 10 GiB | 711.985 | 370.520 | 191.792 | 182.3175 | 421.918 | 126.678 | 113.482 | 92.082 | 76.309 | 64.483 | 100.849 | **29.389** | 54.947 |

### TABLE XV: Overall decoding efficiency

| Sizes \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.642 | 0.585 | 0.450 | 0.486 | 0.402 | 0.346 | 0.245 | 0.182 | 0.122 | 0.089 | 0.157 | 0.119 |
| 5 MiB | 1.000 | 0.347 | 0.126 | 0.162 | 0.116 | 0.220 | 0.110 | 0.133 | 0.096 | 0.147 | 0.035 | 0.107 | 0.039 |
| 10 MiB | 1.000 | 0.860 | 0.647 | 0.529 | 0.400 | 0.334 | 0.303 | 0.248 | 0.227 | 0.196 | 0.045 | 0.041 | 0.033 |
| 50 MiB | 1.000 | 0.888 | 0.683 | 0.566 | 0.430 | 0.367 | 0.327 | 0.273 | 0.238 | 0.215 | 0.089 | 0.109 | 0.086 |
| 100 MiB | 1.000 | 0.113 | 0.125 | 0.109 | 0.073 | 0.071 | 0.069 | 0.130 | 0.067 | 0.071 | 0.109 | 0.121 | 0.081 |
| 500 MiB | 1.000 | 0.908 | 0.700 | 0.545 | 0.430 | 0.355 | 0.316 | 0.268 | 0.251 | 0.219 | 0.110 | 0.077 | 0.088 |
| 1 GiB | 1.000 | 0.897 | 0.686 | 0.534 | 0.514 | 0.393 | 0.312 | 0.259 | 0.228 | 0.199 | 0.117 | 0.107 | 0.091 |
| 5 GiB | 1.000 | 0.371 | 0.331 | 0.287 | 0.282 | 0.252 | 0.230 | 0.202 | 0.175 | 0.163 | 0.106 | 0.113 | 0.086 |
| 10 GiB | 1.000 | 0.951 | 0.881 | 0.625 | 0.205 | 0.491 | 0.447 | 0.364 | 0.304 | 0.258 | 0.141 | 0.099 | 0.075 |

### TABLE XVI: Pure decoding efficiency

| Sizes \ Threads | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 16 | 20 | 24 | 32 | 48 | 64 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 MiB | 1.000 | 0.901 | 0.515 | 0.349 | 0.357 | 0.285 | 0.255 | 0.176 | 0.152 | 0.138 | 0.060 | 0.428 | 0.260 |
| 5 MiB | 1.000 | 0.329 | 0.118 | 0.157 | 0.113 | 0.242 | 0.111 | 0.145 | 0.105 | 0.208 | 0.918 | 0.703 | 0.551 |
| 10 MiB | 1.000 | 0.897 | 0.764 | 0.684 | 0.529 | 0.449 | 0.437 | 0.376 | 0.377 | 0.351 | 0.169 | 0.486 | 0.402 |
| 50 MiB | 1.000 | 0.955 | 0.854 | 0.793 | 0.605 | 0.538 | 0.474 | 0.454 | 0.423 | 0.418 | 0.157 | 0.516 | 0.471 |
| 100 MiB | 1.000 | 0.103 | 0.115 | 0.106 | 0.069 | 0.069 | 0.069 | 0.149 | 0.068 | 0.078 | 0.222 | 0.502 | 0.210 |
| 500 MiB | 1.000 | 0.972 | 0.885 | 0.823 | 0.652 | 0.561 | 0.525 | 0.475 | 0.463 | 0.450 | 0.178 | 0.530 | 0.457 |
| 1 GiB | 1.000 | 0.976 | 0.896 | 0.829 | 0.777 | 0.568 | 0.532 | 0.486 | 0.471 | 0.464 | 0.239 | 0.529 | 0.417 |
| 5 GiB | 1.000 | 0.371 | 0.339 | 0.296 | 0.312 | 0.280 | 0.264 | 0.239 | 0.235 | 0.233 | 0.149 | 0.502 | 0.430 |
| 10 GiB | 1.000 | 0.9606 | 0.928 | 0.651 | 0.211 | 0.562 | 0.523 | 0.483 | 0.467 | 0.460 | 0.221 | 0.505 | 0.202 |

*E. numpy encoding pack*

TABLE XVII: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 39.668 | 39.316 | **38.853** |
| 2 | **28.366** | 29.230 | 29.015 |
| 4 | 25.048 | 25.980 | **24.888** |
| 8 | 26.048 | **25.678** | 25.688 |
| 10 | **25.551** | 25.874 | 25.738 |
| 12 | 26.608 | 26.560 | **24.980** |
| 16 | 26.316 | **24.455** | 46.843 |
| 20 | 233.430 | **68.885** | 122.285 |
| 24 | **57.244** | 59.275 | 69.871 |

TABLE XVIII: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.504 | 0.255 |
| 2 | 0.699 | 0.339 | 0.171 |
| 4 | 0.396 | 0.191 | 0.100 |
| 8 | 0.190 | 0.097 | 0.048 |
| 10 | 0.155 | 0.077 | 0.039 |
| 12 | 0.124 | 0.062 | 0.033 |
| 16 | 0.094 | 0.051 | 0.013 |
| 20 | 0.008 | 0.014 | 0.004 |
| 24 | 0.029 | 0.014 | 0.006 |

*F. numpy encoding scatter*

TABLE XIX: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | **39.346** | 42.755 | 39.929 |
| 2 | 28.056 | 27.240 | **26.568** |
| 4 | 20.548 | **19.755** | 33.501 |
| 8 | 16.588 | 15.468 | **15.198** |
| 10 | 27.227 | 21.108 | **20.999** |
| 12 | 26.551 | 25.964 | **25.659** |
| 16 | 20.122 | **19.575** | 19.687 |
| 20 | 62.409 | **26.710** | 27.580 |
| 24 | 25.184 | 28.774 | **20.043** |

TABLE XX: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.460 | 0.246 |
| 2 | 0.701 | 0.361 | 0.185 |
| 4 | 0.479 | 0.249 | 0.073 |
| 8 | 0.296 | 0.159 | 0.081 |
| 10 | 0.145 | 0.093 | 0.047 |
| 12 | 0.123 | 0.063 | 0.032 |
| 16 | 0.122 | 0.063 | 0.031 |
| 20 | 0.032 | 0.037 | 0.018 |
| 24 | 0.065 | 0.028 | 0.020 |

*G. numpy decoding pack*

TABLE XXI: Overall decoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 43.866 | **42.586** | 43.507 |
| 2 | **31.064** | 33.149 | 36.228 |
| 4 | 34.853 | 33.162 | **30.350** |
| 8 | 30.040 | 27.924 | **27.418** |
| 10 | 28.016 | 28.269 | **27.062** |
| 12 | **28.995** | 29.148 | 29.472 |
| 16 | 30.624 | **30.456** | 32.091 |
| 20 | 38.500 | **34.945** | 39.356 |
| 24 | 34.141 | 33.487 | **32.558** |

TABLE XXII: Overall decoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.515 | 0.252 |
| 2 | 0.706 | 0.331 | 0.151 |
| 4 | 0.315 | 0.165 | 0.090 |
| 8 | 0.183 | 0.098 | 0.050 |
| 10 | 0.157 | 0.078 | 0.041 |
| 12 | 0.126 | 0.063 | 0.031 |
| 16 | 0.090 | 0.045 | 0.021 |
| 20 | 0.057 | 0.031 | 0.014 |
| 24 | 0.054 | 0.027 | 0.014 |

*H. numpy decoding scatter*

TABLE XXIII: Overall decoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 46.248 | 44.085 | **43.422** |
| 2 | 30.331 | **28.251** | 28.600 |
| 4 | 22.234 | 21.866 | **21.471** |
| 8 | 18.648 | **16.757** | 16.952 |
| 10 | 16.364 | **15.773** | 15.839 |
| 12 | 15.574 | 15.720 | **15.148** |
| 16 | 22.756 | **22.066** | 22.733 |
| 20 | 30.425 | **14.413** | 15.643 |
| 24 | 19.983 | 19.048 | **15.158** |

TABLE XXIV: Overall decoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.525 | 0.266 |
| 2 | 0.762 | 0.409 | 0.202 |
| 4 | 0.520 | 0.264 | 0.135 |
| 8 | 0.310 | 0.172 | 0.085 |
| 10 | 0.283 | 0.147 | 0.073 |
| 12 | 0.247 | 0.123 | 0.064 |
| 16 | 0.127 | 0.065 | 0.032 |
| 20 | 0.076 | 0.080 | 0.037 |
| 24 | 0.096 | 0.051 | 0.032 |

*I. pytorch encoding pack*

TABLE XXV: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 247.430 | 229.251 | **225.37**0 |
| 2 | 174.185 | **168.884** | 168.979 |
| 4 | 157.589 | **155.282** | 157.478 |
| 8 | 161.628 | 158.789 | **156.443** |
| 10 | 188.020 | 167.726 | **158.378** |
| 12 | 161.222 | 158.014 | **154.675** |
| 16 | 159.562 | 153.257 | **148.363** |
| 20 | 236.616 | **216.872** | 221.219 |
| 24 | 237.359 | 210.131 | **202.445** |

TABLE XXVI: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.540 | 0.274 |
| 2 | 0.710 | 0.366 | 0.183 |
| 4 | 0.393 | 0.199 | 0.098 |
| 8 | 0.191 | 0.097 | 0.049 |
| 10 | 0.132 | 0.074 | 0.039 |
| 12 | 0.128 | 0.065 | 0.033 |
| 16 | 0.097 | 0.050 | 0.026 |
| 20 | 0.052 | 0.029 | 0.014 |
| 24 | 0.043 | 0.025 | 0.013 |

*J. pytorch encoding scatter*

TABLE XXVII: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 317.663 | 268.323 | **265.387** |
| 2 | 184.717 | 179.515 | **173.750** |
| 4 | 139.132 | **123.383** | 126.569 |
| 8 | 105.950 | 102.119 | **100.966** |
| 10 | 98.206 | 94.839 | **91.242** |
| 12 | 95.794 | 96.229 | **90.712** |
| 16 | 87.427 | 84.979 | **84.262** |
| 20 | 95.613 | **87.998** | 91.617 |
| 24 | 85.077 | **84.196** | 84.829 |

TABLE XXVIII: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.592 | 0.299 |
| 2 | 0.860 | 0.442 | 0.229 |
| 4 | 0.571 | 0.322 | 0.157 |
| 8 | 0.375 | 0.194 | 0.098 |
| 10 | 0.323 | 0.167 | 0.087 |
| 12 | 0.276 | 0.138 | 0.073 |
| 16 | 0.227 | 0.117 | 0.059 |
| 20 | 0.166 | 0.090 | 0.043 |
| 24 | 0.156 | 0.079 | 0.039 |

*K. pytorch encoding pack*

TABLE XXIX: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 247.430 | 229.251 | **225.37**0 |
| 2 | 174.185 | **168.884** | 168.979 |
| 4 | 157.589 | **155.282** | 157.478 |
| 8 | 161.628 | 158.789 | **156.443** |
| 10 | 188.020 | 167.726 | **158.378** |
| 12 | 161.222 | 158.014 | **154.675** |
| 16 | 159.562 | 153.257 | **148.363** |
| 20 | 236.616 | **216.872** | 221.219 |
| 24 | 237.359 | 210.131 | **202.445** |

TABLE XXX: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.540 | 0.274 |
| 2 | 0.710 | 0.366 | 0.183 |
| 4 | 0.393 | 0.199 | 0.098 |
| 8 | 0.191 | 0.097 | 0.049 |
| 10 | 0.132 | 0.074 | 0.039 |
| 12 | 0.128 | 0.065 | 0.033 |
| 16 | 0.097 | 0.050 | 0.026 |
| 20 | 0.052 | 0.029 | 0.014 |
| 24 | 0.043 | 0.025 | 0.013 |

*L. pytorch encoding scatter*

TABLE XXXI: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 317.663 | 268.323 | **265.387** |
| 2 | 184.717 | 179.515 | **173.750** |
| 4 | 139.132 | **123.383** | 126.569 |
| 8 | 105.950 | 102.119 | **100.966** |
| 10 | 98.206 | 94.839 | **91.242** |
| 12 | 95.794 | 96.229 | **90.712** |
| 16 | 87.427 | 84.979 | **84.262** |
| 20 | 95.613 | **87.998** | 91.617 |
| 24 | 85.077 | **84.196** | 84.829 |

TABLE XXXII: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.592 | 0.299 |
| 2 | 0.860 | 0.442 | 0.229 |
| 4 | 0.571 | 0.322 | 0.157 |
| 8 | 0.375 | 0.194 | 0.098 |
| 10 | 0.323 | 0.167 | 0.087 |
| 12 | 0.276 | 0.138 | 0.073 |
| 16 | 0.227 | 0.117 | 0.059 |
| 20 | 0.166 | 0.090 | 0.043 |
| 24 | 0.156 | 0.079 | 0.039 |

## M. linux encoding pack

### TABLE XXXIII: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 169.893 | 162.189 | **146.131** |
| 2 | 120.387 | 116.712 | **115.819** |
| 4 | 116.030 | **111.948** | 112.458 |
| 8 | 113.034 | **111.988** | 113.310 |
| 10 | **113.167** | 115.369 | 121.193 |
| 12 | **120.443** | 121.303 | 145.063 |
| 16 | 141.595 | **123.439** | 126.436 |
| 20 | 115.303 | **114.752** | 118.585 |
| 24 | 114.763 | 114.116 | **112.077** |

### TABLE XXXIV: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.524 | 0.291 |
| 2 | 0.706 | 0.364 | 0.183 |
| 4 | 0.366 | 0.190 | 0.094 |
| 8 | 0.188 | 0.095 | 0.047 |
| 10 | 0.150 | 0.074 | 0.035 |
| 12 | 0.118 | 0.058 | 0.024 |
| 16 | 0.075 | 0.043 | 0.021 |
| 20 | 0.074 | 0.037 | 0.018 |
| 24 | 0.062 | 0.031 | 0.016 |

## N. linux encoding scatter

### TABLE XXXV: Overall encoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 189.447 | 150.256 | **147.334** |
| 2 | 122.175 | 118.490 | **117.44**0 |
| 4 | 84.408 | 81.996 | **81.776** |
| 8 | 68.110 | 67.038 | **43.153** |
| 10 | 63.032 | 61.393 | **61.122** |
| 12 | 62.554 | **61.992** | 62.346 |
| 16 | 57.539 | **56.922** | 56.938 |
| 20 | 54.402 | 51.974 | **51.381** |
| 24 | 52.580 | 51.944 | **51.464** |

### TABLE XXXVI: Overall encoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.630 | 0.321 |
| 2 | 0.775 | 0.400 | 0.202 |
| 4 | 0.561 | 0.289 | 0.145 |
| 8 | 0.348 | 0.177 | 0.137 |
| 10 | 0.301 | 0.154 | 0.077 |
| 12 | 0.252 | 0.127 | 0.063 |
| 16 | 0.206 | 0.104 | 0.052 |
| 20 | 0.174 | 0.091 | 0.046 |
| 24 | 0.150 | 0.076 | 0.038 |

## O. linux decoding pack

### TABLE XXXVII: Overall decoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 194.257 | 184.958 | **170.975** |
| 2 | 126.599 | **125.535** | 130.044 |
| 4 | 115.689 | 103.416 | **101.447** |
| 8 | 101.767 | **100.484** | 104.773 |
| 10 | **101.006** | 102.295 | 102.599 |
| 12 | **100.869** | 116.434 | 102.087 |
| 16 | **104.753** | 114.768 | 173.998 |
| 20 | 93.438 | **92.893** | 125.593 |
| 24 | 94.186 | **93.018** | 122.240 |

### TABLE XXXVIII: Overall decoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.525 | 0.284 |
| 2 | 0.767 | 0.387 | 0.187 |
| 4 | 0.420 | 0.235 | 0.120 |
| 8 | 0.239 | 0.121 | 0.058 |
| 10 | 0.192 | 0.095 | 0.047 |
| 12 | 0.160 | 0.070 | 0.040 |
| 16 | 0.116 | 0.053 | 0.017 |
| 20 | 0.104 | 0.052 | 0.019 |
| 24 | 0.086 | 0.044 | 0.017 |

## P. linux decoding scatter

### TABLE XXXIX: Overall decoding times

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 299.416 | 277.318 | **216.163** |
| 2 | 162.069 | 129.983 | **123.616** |
| 4 | 112.011 | 103.347 | **98.043** |
| 8 | 109.391 | **97.004** | 97.808 |
| 10 | 100.648 | 93.448 | **86.719** |
| 12 | 84.940 | 83.006 | **81.729** |
| 16 | 87.747 | 80.180 | **79.871** |
| 20 | 44.576 | 41.677 | **41.459** |
| 24 | 41.485 | 40.884 | **40.618** |

### TABLE XL: Overall decoding efficiency

| Processes \ Threads | 1 | 2 | 4 |
|---|---|---|---|
| 1 | 1.000 | 0.540 | 0.346 |
| 2 | 0.924 | 0.576 | 0.303 |
| 4 | 0.668 | 0.362 | 0.191 |
| 8 | 0.342 | 0.193 | 0.096 |
| 10 | 0.297 | 0.160 | 0.086 |
| 12 | 0.294 | 0.150 | 0.076 |
| 16 | 0.213 | 0.117 | 0.059 |
| 20 | 0.336 | 0.180 | 0.090 |
| 24 | 0.301 | 0.153 | 0.077 |

*Q. Local testing*

TABLE XLI: Overall encoding times for different folders on local machine

| Source | Processes Threads | 1 1 | 2 1 | 4 1 |
|---|---|---|---|---|
| Numpy | | | | |
| | Encoding times | 2.0731 | 1.0857 | 0.6210 |
| | Encoding efficiency | 1.0 | 0.9547 | 0.8345 |
| | Decoding times | 1.8602 | 1.0178 | 0.5757 |
| | Decoding efficiency | 1.0 | 0.9138 | 0.8078 |
| Pytorch | | | | |
| | Encoding times | 15.1102 | 7.7887 | 5.4025 |
| | Encoding efficiency | 1.0 | 0.9701 | 0.6992 |
| | Decoding times | 14.6897 | 7.7248 | 5.6166 |
| | Decoding efficiency | 1.0 | 0.9508 | 0.6538 |
| Linux | | | | |
| | Encoding times | 85.0387 | 49.5736 | 30.6647 |
| | Encoding efficiency | 1.0 | 0.8577 | 0.6933 |
| | Decoding times | 80.3765 | 47.8856 | 29.6687 |
| | Decoding efficiency | 1.0 | 0.8393 | 0.6773 |