Parallelization of Image Compression Techniques

M. Abhijna Anand
(183079027)
Electrical Engineering
IIT Bombay

Neelam Sharma
(18307R030)
Electrical Engineering
IIT Bombay

Sana Naaz
(18307R001)
Electrical Engineering
IIT Bombay

***Abstract*—Image compression is a type of data compression applied to digital images for reduction in the size and faster transmission rate. It can also reduce the computational complexity for machine learning algorithms by reducing the number of features through dimensionality reduction. Image compression consists of the following two methods: Lossy and Lossless compression. There is a trade-off between the quality of an image and the size required to store it while using the above methods. The lossy compression technique does not restore the data in its original form after decompression because it eliminates the data, which is not noticeable. On the other hand, lossless compression restores data in its original state after decompression.**

The objective of the project is to implement a lossy image compression technique using the Principal component analysis (PCA) algorithm and a lossless image compression technique using the Huffman coding. Difference between the execution time of the serial and parallelized versions of both the algorithms is demonstrated by running many iterations to verify reproducibility of results on many images of different dimensions. Along with this, through timing analysis and profiling is done for both the algorithms.

***Index Terms*—PCA, Huffman encoding, Image compression, Parallelization, Timing analysis, Profiling**

I. Introduction

Image compression is the topic of importance in the field of image processing system and applications. Compression of an image plays a major role in reducing computational cost and storage cost. A simple passport size image of dimensions 600X600 represented by 24 bits per pixel requires about 1.03 MB of storage. It requires 2.2 minutes for transmitting 64 Kbits/sec. In today's growing age of digital transformation there is an inevitable demand of higher resolution of images and videos i.e., more storage requirement and a faster speed is required to reduce the image loading time and retrieval time, which makes image compression a necessity. **Image compression** is defined as reduction in the number of bits required to represent an image.

$$Image = Information + Redundant\ data \tag{1}$$

Eq. 1 explains the basic components of an image which are exploited by image compression techniques.There are two types of image compression techniques: 1) Lossy compression, which is not information preserving but can achieve high compression ratios. 2) Lossless compression, which removes the redundant data present in an image. Lossless compression is information preserving but cannot achieve high compression ratios.

In the era of Machine Learning and Computer Vision an image is nothing but data explained using different and large number of dimensions. As the size of image increases it's dimensions increases and we all know about the *Curse of dimensionality* [1] because it then becomes hard to visualize the data, might result in overfitting of model and redundant computations. Principal Component analysis is one such technique that finds out the correlation among the features and choose the features which can best explain maximum variation in the image.

This project demonstrates the serial and parallel implementation of two algorithms: Huffman coding and Principal Component Analysis (PCA) which can be parallelized because of their way of computation, which is discussed in detail in Section II and Section III. Parallelization is done using OpenMP and Hybrid parallelism (OpenMP and CUDA). OpenMP is executed on Intel(R) Core(TM) i7-7700 CPU which has 4 cores. OpenMP and CUDA code is executed on Intel(R) Core(TM) i7-7700 CPU and GK208B [GeForce GT 710] GPU. Huffman coding using OpenMP is executed on Intel(R) Core(TM) i7-8750 CPU which has 6 cores.

II. Lossy Compression: Image Dimensionalty reduction using Prinicipal Component Analysis

A. PCA

PCA is an unsupervised machine learning technique widely employed to reduce the dimensions of the dataset. It transforms the components of a dataset which are a set of values of correlated variables to a new dataset which is a set of values of uncorrelated variables. The Principal component is basically eigen vectors of the dataset. PCA algorithm choses the top k eigen values and using those eigen values most of the variation in a dataset is explained. Fig. 1 a) briefly explains the algorithm, for each image channel. The mean for a column is subtracted from the original value, to shift the mean to zero. Covariance of the mean shifted image channel is performed followed by finding and choosing the top 'k' eigen vectors. Finally, each channel's data is converted to the reduced dimensions using choosen eigen vectors.

B. Parallelization

This algorithm was parallelized using two parallelization methodologies:

1) OpenMP: From Fig. 1 b) Steps like subtraction from the mean value, covariance matrix compuation, finding

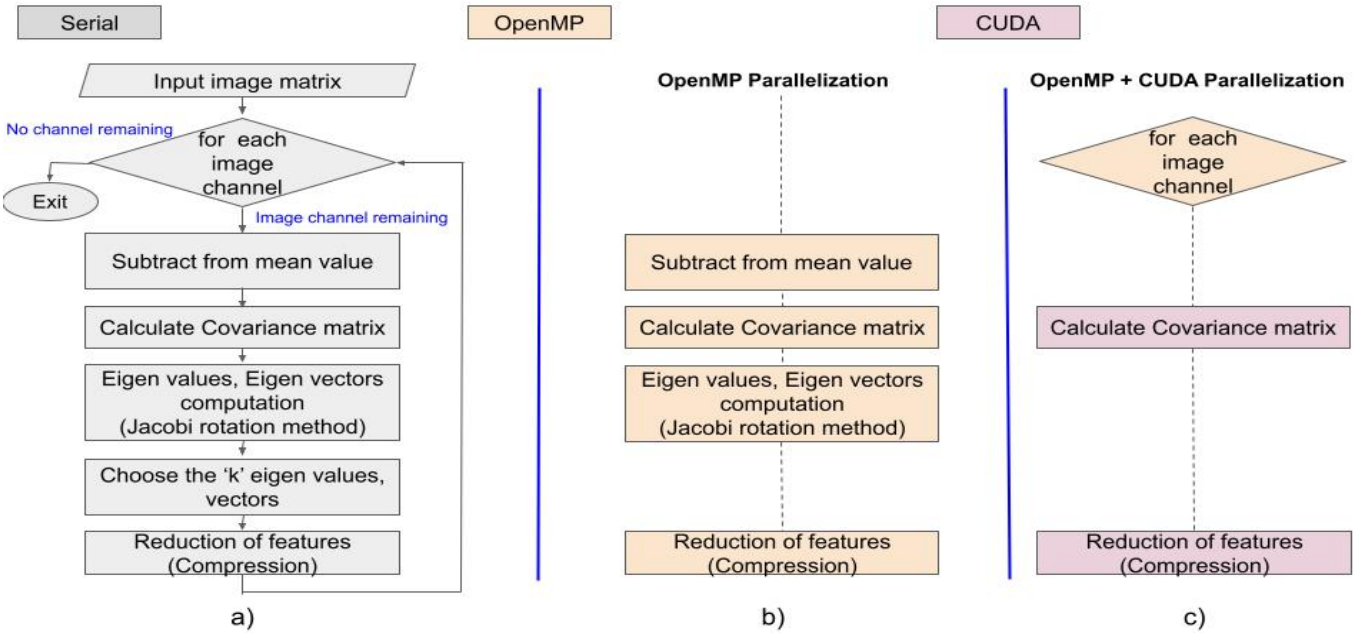

Fig. 1. a) Original Algorithm b) OpenMP parallelization strategy c) Hybrid parallellization using OpenMP and CUDA

eigen vectors and finally conversion of the data to reduced dimensions can work on independent data at the column of the matrix or element specific granularity.

2) Hybrid using OpenMP & CUDA: Reduction of data for each channel is independent of the other channel which is exploited for OpenMP parallelism. Covariance matrix computation and reduction of features step, as shown in Fig. 1 c) has a computational complexity of $O(n^3)$ which is suitable for implementation on GPU.

C. Results & Observations

1) Performance Analysis for OpenMP parallelization: Figures 2, 3 and 4 for 10, 50 and 100 PCA components respectively, corresponding to different images: image 1 (274x184), image 2 (280x280), image 3 (225x225), image 4 (194x260). It can be seen that linear speedup was observed only upto 4 threads, and for 6 and 8 threads the speedup almost saturated. In order to explain the overhead introduced by using larger number of threads, we have plotted $F_{parallel}$ and e (Experimentally determined serial fraction using the Karp-Flatt metric [2]) associated with each speedup result. With 6 and 8 threads, $F_{parallel}$ reduced or e is increased which shows that the overhead associated with creating large number of threads is dominating over the speedups achieved.
2) Performance Analysis for OpenMP + CUDA parallelization: Figures 5, 6 and 7 for 10, 50 and 100 PCA components respectively, corresponding to different images: image 1 (225x225), image 2 (280x280),image 3 (194x260),image 4 (280x280).The speedup saturated after 4 threads which is explained with the help of increasing e and reduction in $F_{parallel}$.

Profiling is done using VTune-Profiler [3] to determine the CPU Utilization achieved using OpenMP parallelization. The results in table I also explains that after 4 threads there is not much improvement in CPU Utilization due to which there is no improvement in speedup for 6 and 8 threads.

TABLE I
CPU UTILIZATION USING VTUNE FOR PCA

Threads	**1**	**2**	**4**	**6**	**8**
CPU Utilization	0.97	1.46	2.91	2.9	2.92

III. LOSSLESS COMPRESSION: HUFFMAN ALGORITHM

A. Huffman Algorithm

Huffman compression [4] is a lossless data compression algorithm. It makes use of variable length coding technique where each symbol or character is uniquely encoded by combination of bits of varying lengths. The length of the encoded symbol will depend on its frequency of occurrence in the input file. The compression methods consists of following steps:

1) Generating frequency histogram: In this step the input file needs to be scanned to find the number of times a unique symbol is repeated. Thus generating a frequency histogram of symbols. Consider the following example:
Image file contains - 1 6 1 8 7 8 3

2) Build Huffman tree: A Huffman tree follows the same structure as a normal binary tree, containing nodes and leafs. A leaf is a node which does not have any children (or any

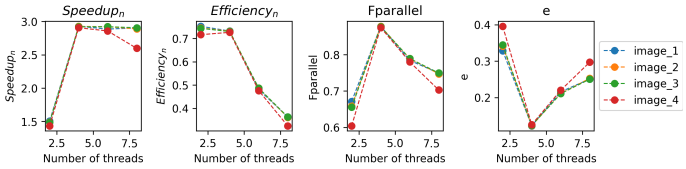

Fig. 2. Performance analysis with 10 PCA components using OpenMP

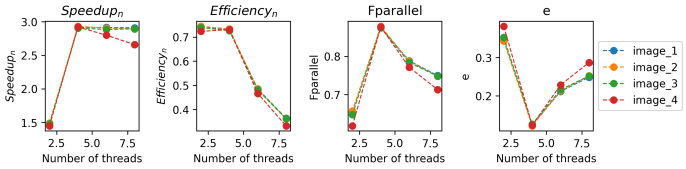

Fig. 3. Performance analysis with 50 PCA components using OpenMP

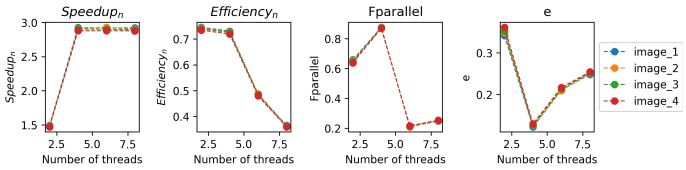

Fig. 4. Performance analysis with 100 PCA components using OpenMP

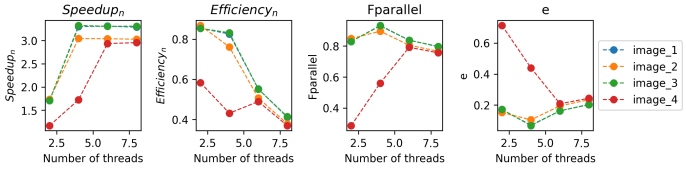

Fig. 5. Performance analysis with 10 PCA components using OpenMP and CUDA

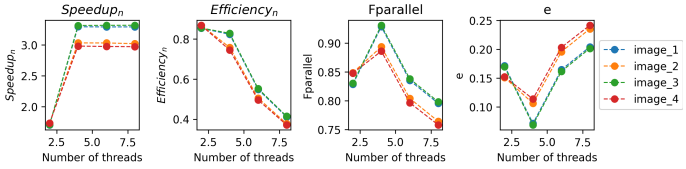

Fig. 6. Performance analysis with 50 PCA components using OpenMP and CUDA

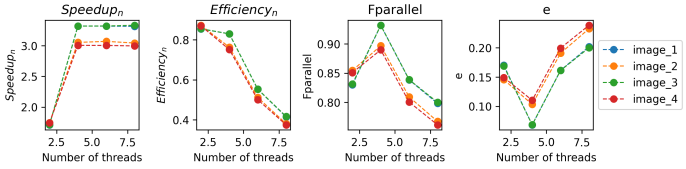

Fig. 7. Performance analysis with 100 PCA components using OpenMP and CUDA

TABLE II
SYMBOL FREQUENCY HISTOGRAM

Symbol	1	6	8	7	3
Frequency	2	1	2	1	1

outgoing edges). Following are the steps used to build the tree:

1) Create a leaf node for each symbol and add them to a sorted linked list in ascending order of the frequency.
2) While there is no more than one node in the list
   - remove the first two nodes in the list (with least frequencies)
   - create a new node with these two nodes as its children (left and right branch) and the frequency equal to sum of these two nodes' frequency
   - add this new node to the sorted linked list
3) The remaining node in the list is the root node of the tree.

- name - symbol or character
- frequency
- code – encoded string
- code length – the length of the encoded string
- left, right node pointers – used to point to its children

3) Character encoding: To find out the encoding of the symbol, the Huffman tree is traversed from the root to its leaves. For every left traversal of a node, bit 0 is appended to the code and simultaneously increment the code length.

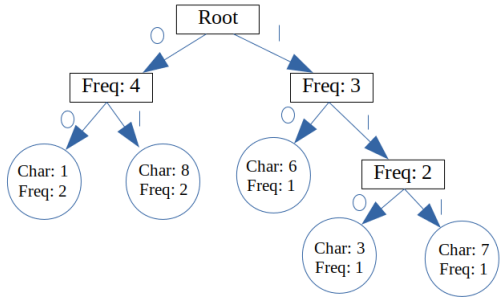

Fig. 8. Huffman tree for the example given

Similarly for every right traversal of a node, bit 1 is appended to the code. In the above example the encoded table becomes: With Huffman encoding the symbols with highest frequency

TABLE III
ENCODED SYMBOL TABLE

Symbol	1	6	8	7	3
Frequency	2	1	2	1	1
Encoded bits	00	10	01	111	110

is encoded with less number of bits, whereas the least frequent symbols are encoded with more number of bits. Total number of bits in compressed file will be = 2*2 + 1*2 + 2*2 + 1*3 + 1*3 = 16 bits, which is 0.285 times the original file size which is 8*7 = 56 bits.

4) Writing to the compressed file: After generating the encoded symbol table above, the last step is to write the

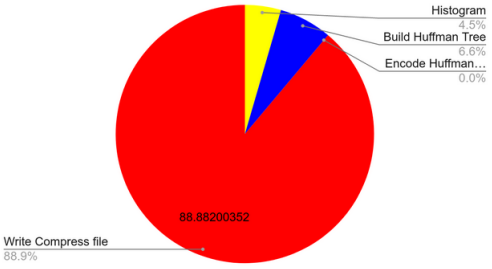

Fig. 9. Execution time breakdown of serial code for image 1

compressed file. A dictionary is implemented which is an array of 256 nodes with index corresponding to the symbol / character and the value is the leaf node of the Huffman tree. By pointing to the array index any node can be accessed to find its code, code length and frequency. In this step, the input file is scanned and for each input symbol its encoded value is written to the output file.

In order to recover the original image, the length (size) and the histogram of the original image is also written to the compressed file at the beginning.

5) Huffman decompression: Huffman decompression has been done to verify the compressed data and only the serial version has been implemented in this project. Decompression follows similar steps as compression. First it reads the original image size and its histogram from the compressed file. Then a Huffman tree is built from that histogram. In the end, the compressed file is scanned and original symbol is recovered by traversing the tree from the root.

Whenever input bit is '0', node pointer traverses to the left child of the current node pointer. Similarly for input bit '1' it moves to the right child node. If a leaf node is reached, that means the original symbol is present in the name variable of the node and that symbol is then written to the decompressed file. After writing the symbol reset the node pointer to point to the root of the tree.

B. Parallelization

It has been experimentally determined that writing to the compressed file consumes maximum amount of time up to 80-90% as shown in figure . And generating the histogram is second in most time consuming process. Also these processes can be highly parallelized as there is no inter dependence. Whereas building Huffman tree is inherently sequential and cannot be prallelized. Character encoding take minimal amount of time(0.001%) and thus parallelising will not help as overhead will become dominant. In this project, histogram generation and writing to the compressed file has been parallized by OpenMP.

- For histogram generation, the input file is divided into continuous chunks of data and each thread works on its own chunk of input file to calculate the frequency. These separate frequencies are combined serially to get the complete histogram.

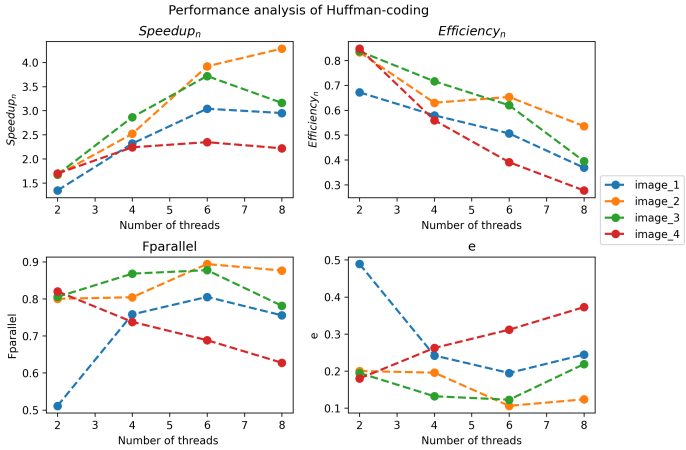

Fig. 10. Performance analysis of Huffman algorithm

- Similarly during writing the compressed file, input file is divided into continuous chunks of data and each thread encodes the corresponding input chunk. The offset address from which the thread should start writing to output is calculated by one thread.

C. Results & Observations

TABLE IV
COMPRESSION RESULTS OF SOME TEST IMAGES

Image	Input size (bytes)	Compressed size (bytes)	Compression ratio
artificial	18874385	14845564	0.786546
big_building	117158993	110609274	0.944095
flower_foveon	10287665	9170881	0.891444

Figure 10 shows the performance results obtained using OpenMP for parallelization. It is seen that a linear speedup was obtained only upto 6 threads, after that the speedup started declining. This is explained using the reduction in $F_{parallel}$ or increase in e, which indicates increasing overhead with creation and destruction of threads.

TABLE V
CPU UTILIZATION USING VTUNE FOR HUFFMAN CODING

Threads	1	2	4	6	8
CPU Utilization	0.97	1.55	2.95	3.33	3.21

The results in table V also shows that after 6 threads there is not much improvement in CPU Utilization due to which there is no improvement in speedup for 8 threads. This result is inline with the decrease in $F_{parallel}$ for 8 threads.

REFERENCES

[1] Curse of Dimensionality, https://en.wikipedia.org/wiki/Curse_of_dimensionality
[2] Karp-Flatt metric, https://en.wikipedia.org/wiki/Karp%E2%80%93Flatt_metric
[3] Intel VTune Profiler, https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html
[4] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9), 1098-1101.