Pune Institute of Computer Technology
Dhankawadi, Pune

HPC Project Report
ON

Image Compression

SUBMITTED BY

Abhishek Sawalkar 41402
Samarth Bhadane 41414
Sanket Bhatlawande 41415

Under the guidance of
Prof. Bhumesh Masram

DEPARTMENT OF COMPUTER ENGINEERING
Academic Year 2020-21

Title: IMAGE COMPRESSION

Problem Statement: Large amount of bandwidth is required for transmission or storage of images. This has driven the research area of image compression to develop parallel algorithms that compress images.

Software and Hardware Required: C++, OpenMp, OpenCV,64-bit OS, 8 GB RAM,500 GB HDD, Monitor, Keyboard.

Objective: Students will be able to compress RGB images

Concept Related Theory:

Image compression is the process of encoding or converting an image file in such a way that it consumes less space than the original file. It is a type of compression technique that reduces the size of an image file without affecting or degrading its quality to a greater extent. Image compression is typically performed through an image/data compression algorithm or codec. Typically, such codecs/algorithms apply different techniques to reduce the image size, such as by:

- Specifying all similarly colored pixels by the color name, code and the number of pixels. This way one pixel can correspond to hundreds or thousands of pixels.
- The image is created and represented using mathematical wavelets.
- Splitting the image into several parts, each identifiable using a fractal.

Some of the common image compression techniques are:

- Fractal
- Wavelets
- Chroma sub sampling
- Transform coding
- Run-length encoding

Image compression is significant because of the large image transfer happening over the internet and advancements in photography.Compression is useful because it helps reduce the consumption of expensive resources such as hard disk space or transmission bandwidth.The problem is to compress image files whilst maintaining the quality of the image.The JPEG algorithm takes advantage of the fact that humans can't see colors at high frequencies. These high frequencies are eliminated during the compression. Our project uses techniques of parallelization to compress multiple images simultaneously.

METHODS USED

1. **Reading the image**
    a. The pixel data is read from an image and is saved in 8 x 8 matrices
    b. As the pixel data for each block is independent of the other's, the data was read in parallel - SIMD
    c. The above is an application of data parallelism

2. **The image compression algorithm**
    a. Zero padding of the image
    b. Extracting image data (RGB Values)
    c. Parallel Discrete Cosine Transformation Algorithm

d. Parallel Quantization of all the blocks

e. Parallel Entropy coding

a.**Zero Padding**

i. If the image cannot be divided into 8-by-8 blocks, then we add in empty pixels around the edges, essentially zero-padding the image

b. **Extracting image data**

i. The algorithm works for grayscale images

ii. The RGB data is read and the average is stored in another matrix

c. **Parallel Discrete Cosine Transformation**

i. Conversion of matrix values using DCT function

ii. Complexity is O(n2 xm2)

iii. DCT is applied parallel to all the 8 x 8 blocks of the image

$$DCT(i, j) = \frac{1}{\sqrt{2N}} C(i)\, C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x, y)\, COS\left[\frac{(2x + 1)i\pi}{2N}\right] COS\left[\frac{(2y + 1)j\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \text{ if x is 0, else 1 if x} > 0$$

d. **Parallel Quantisation**

i. Process of reducing the number of bits needed to store an integer value by reducing the precision of the integer

ii. Multiplying DCT matrix with separate quantization matrix removes irrelevant frequencies

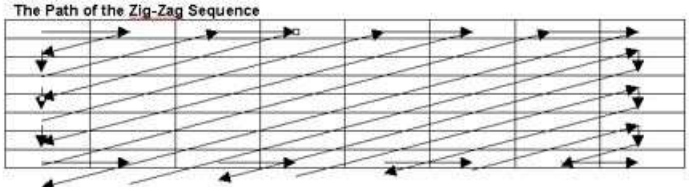
$$\text{Quantized Value}(i, j) = \frac{DCT(i, j)}{\text{Quantum}(i, j)} \text{ Rounded to the nearest integer}$$

e. **Parallel Entropy Encoding**

i. Convert the DC coefficient to a relative value

ii. Reorder the DCT block in a zig-zag sequence

iii. Entropy Encoding : Run length encoding and Huffman encoding



Conclusion:

- Varying the block size helped us improve performance. The block size is 8 x 8 because
  - i. Smaller block sizes make the compression more arduous
  - ii. Larger block sizes would result in data loss due to color gradients
- Using runtime scheduling gave the best performance
- The speedup increases as we increase the number of images
- Through this project we were able to apply the concepts of parallel computing to a real world scenario. It helped us gain practical knowledge in the field..

Sample Output:

Before

After

Code:

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <cmath>
#include "dequantization.h"
#include <C:\OpenCV\include\opencv2\highgui\highgui.hpp>
#include <fstream>
#include <string>
#ifndef M_PI
   #define M_PI 3.14159265358979323846
#endif

#define numThreads 8 //total number of threads

using namespace cv;
using namespace std;

int n,m;

void dct(float **DCTMatrix, float **Matrix, int N, int M);
void write_mat(FILE *fp, float **testRes, int N, int M);
void free_mat(float **p);
void divideMatrix(float **grayContent, int dimX, int dimY, int n, int m);


void print_mat(float **m, int N, int M){

  for(int i = 0; i < N; i++)
  {
   for(int j = 0; j < M; j++)
   {
     cout<<m[i][j]<<" ";
   }
   cout<<endl;
  }
  cout<<endl;
}

void divideMatrix(float **grayContent, int dimX, int dimY, int n, int m)
{
  float **smallMatrix = calloc_mat(dimX, dimY);
  float **DCTMatrix = calloc_mat(dimX, dimY);
  float **newMatrix = calloc_mat(dimX, dimY);
```

```
  #pragma omp parallel for schedule (runtime)
  for(int i = 0; i < n; i += dimX)
  {
    for(int j = 0; j < m; j += dimY)
    {
      for(int k = i; k < i + pixel && k < n; k++)
      {
        for(int l = j; l < j + pixel && l < m; l++)
        {
          smallMatrix[k-i][l-j] = grayContent[k][l];
        }
      }
      dct(DCTMatrix, smallMatrix, dimX, dimY);
      for(int k=i; k<i+pixel && k<n ;k++)
      {
        for(int l=j; l<j+pixel && l<m ;l++)
        {
          globalDCT[k][l]=DCTMatrix[k-i][l-j];

        }
      }
    }
  }
  delete [] smallMatrix;
  delete [] DCTMatrix;
  // free_mat(smallMatrix);
  // free_mat(DCTMatrix);
}

void dct(float **DCTMatrix, float **Matrix, int N, int M){

    int i, j, u, v;
    #pragma omp parallel for schedule(runtime)
    for (u = 0; u < N; ++u) {
        for (v = 0; v < M; ++v) {
        DCTMatrix[u][v] = 0;
            for (i = 0; i < N; i++) {
                for (j = 0; j < M; j++) {
                    DCTMatrix[u][v] += Matrix[i][j] *
cos(M_PI/((float)N)*(i+1./2.)*u)*cos(M_PI/((float)M)*(j+1./2.)*v);
                }
            }
        }
    }
    //print_mat(DCTMatrix, 8, 8);
```

```
 }

void compress(Mat img,int num)
{
    n = img.rows;
    m = img.cols;

    int add_rows = (pixel - (n % pixel) != pixel ? pixel - (n % pixel) : 0);
    int add_columns = (pixel - (m % pixel) != pixel ? pixel - (m % pixel) : 0) ;

    float **grayContent = calloc_mat(n + add_rows, m + add_columns);

    int dimX = pixel, dimY = pixel;

    #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < n; i++)
    {
      for(int j = 0; j < m; j++)
      {
        Vec3b bgrPixel = img.at<Vec3b>(i, j);
        grayContent[i][j] = (bgrPixel.val[0] + bgrPixel.val[1] + bgrPixel.val[2])/3;
      }
    }

    //setting extra rows to 0
    #pragma omp parallel for schedule(runtime)
    for (int j = 0; j < m; j++)
    {
      for (int i = n; i < n + add_rows; i++)
      {
        grayContent[i][j] = 0;
      }
    }

    //setting extra columns to 0
    #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < n; i++)
    {
      for(int j = m; j < m + add_columns; j++)
      {
        grayContent[i][j] = 0;
      }
    }

    n = add_rows + n; //making rows as a multiple of 8
    m = add_columns + m; //making columns as a multiple of 8
```

```
    divideMatrix(grayContent, dimX, dimY, n, m);


    quantize(n,m);
    dequantize(n,m);

    #pragma omp parallel for schedule(runtime)
    for (int i = 0; i < n; i++)
     {
       for(int j = 0; j < m; j++)
       {
         Vec3b bgrPixel;
         bgrPixel.val[0] = bgrPixel.val[1] = bgrPixel.val[2] = finalMatrixDecompress[i][j];
         img.at<Vec3b>(i,j) = bgrPixel;
       }
     }

    // free_mat(grayContent);
    //delete [] grayContent;
   free(grayContent);
}

int main(int argc, char **argv)
{

    FILE *fp;
    fp=fopen("./info.txt","a+");
    omp_set_num_threads(numThreads);
    string str="./images/";
    string ext=".jpg";
    string path=str+argv[1]+ext;
    Mat img=imread(path,IMREAD_GRAYSCALE);
    compress(img,0);
    cout<<path<<endl;
    int m;
    cin>>m;
return 0;
}
```