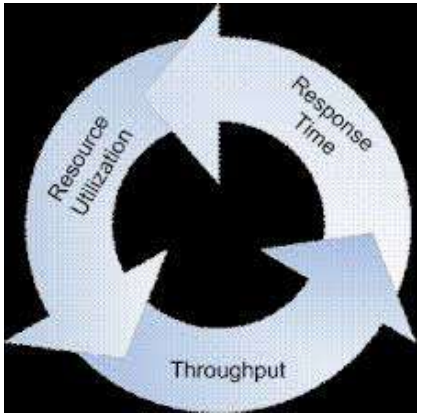
Multimedia Systems

Assistant professor Daniel Hofman, PhD

Associate professor Josip Knezović, PhD

What are we going to study?

- Implementation
- Analysis
- Performance
  - Processing time
  - Throughput
  - Resources



Why?

- Analyses and examples will be done on image and video data because they are the most demanding
- Why computer system performance is essential:
  - Acceptable response time (mpeg2 decoder – 20 fps, Google search: 1s, …)
  - Option to add new functionality
  - Scaling option (increase of workload)
  - Use fewer resources:
    - energy consumed in built-in systems
- Portability!

Data sizes (Images)

Acronym	Image resolution	Element Precision	Size
VGA	640x480	8	307 kB
WSXGA	1280x1024	24	3,9 MB
SXGA+	1400x1050	24	~4,4 MB
WUXGA	1920x1200	24	~7 MB
WHUXGA	7680x4800	24	~110 MB

Data sizes (Video)

Acronym	Image Resolution	Frames/s	Bandwidth (MB/s)
VGA	640x480x8 bpp	10	~3
WXGA (720p)	1280x720x24 bpp	30	~83
SXGA	1280x1024x24 bpp	30	~120
Full HD	1920x1080x24 bpp	30	~186
8K UHD (4320p)	7680x4320x24 bpp	30	~3000

Network technologies

Acronym	Bandwidth	Video	Time
Ethernet	12.5 MB/s	FHD 30fps, 30 min	7,4 h
10 Gigabit eth.	1.25 GB/s	FHD 30fps, 30 min	4,4 min
HSPA+	5.25/1.5 MB/s	FHD 30fps, 30 min	17,7 h
LTE	21.6/7.25 MB/s	FHD 30fps, 30 min	4,3 h
ADSL2+	3/0.41 MB/s	FHD 30fps, 30 min	31 h

Data sizes (Images)

Image resolution	Bits/pixel	Size
640x480	8	307kB
1280x1024	24	3,9MB

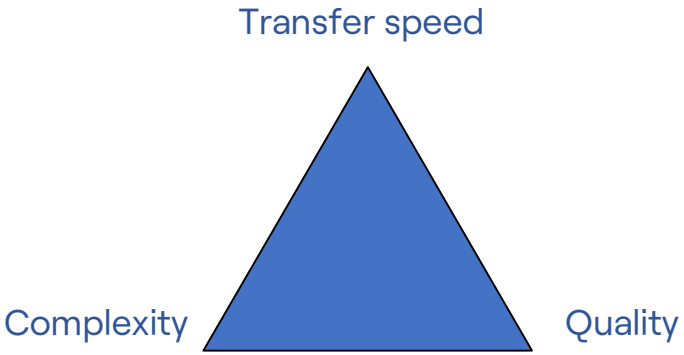
Video resolution	Frames/s	Size/s
640x480 (8bpp)	10	3 MB/s
1280x1024 (24bpp)	30	120 MB/s

Compression

- Compression of multimedia data
  - Reducing the required memory capacity
  - Reducing the bandwidth of the communication channel
  - Data Processing Speed > Memory System Speed
  - Scalability (video telephony, HD, …)

Compression of MM data (2)

- Performance dictates execution:
  - Asymmetric algorithms
    - MPEG coder/decoder
  - Calculation Precision
    - Approximate computing (memoization, loop perforation, task skipping)
  - Computational complexity – quality – bit rate



Introductory Example

- Let's generally see how fast they really are...
- Let's make a simple program:
  - To each image element in the input image (paradise.raw, 1280x1024x24bpp) we will ad a constant

- Example

Introductory Example (2)

- We can see what the basic speed of performing such a simple program is
- We are more interested in whether we are satisfied with the achieved performance:
  - Run Time
  - Bandwidth
  - Resource usage
  - Energy consumed
- A detailed analysis could determine the precise consumption of resources
- Experimental analysis of program execution
- Profiling

Ways to optimize algorithms

- Some general optimization groups:
  - Reduce calculation accuracy
  - Development of equivalent mathematical algorithms with less complexity
  - Change the program's development environment
  - Change the program's performance environment
  - Change the system architecture on which the algorithms are run

Ways to optimize algorithms (2)

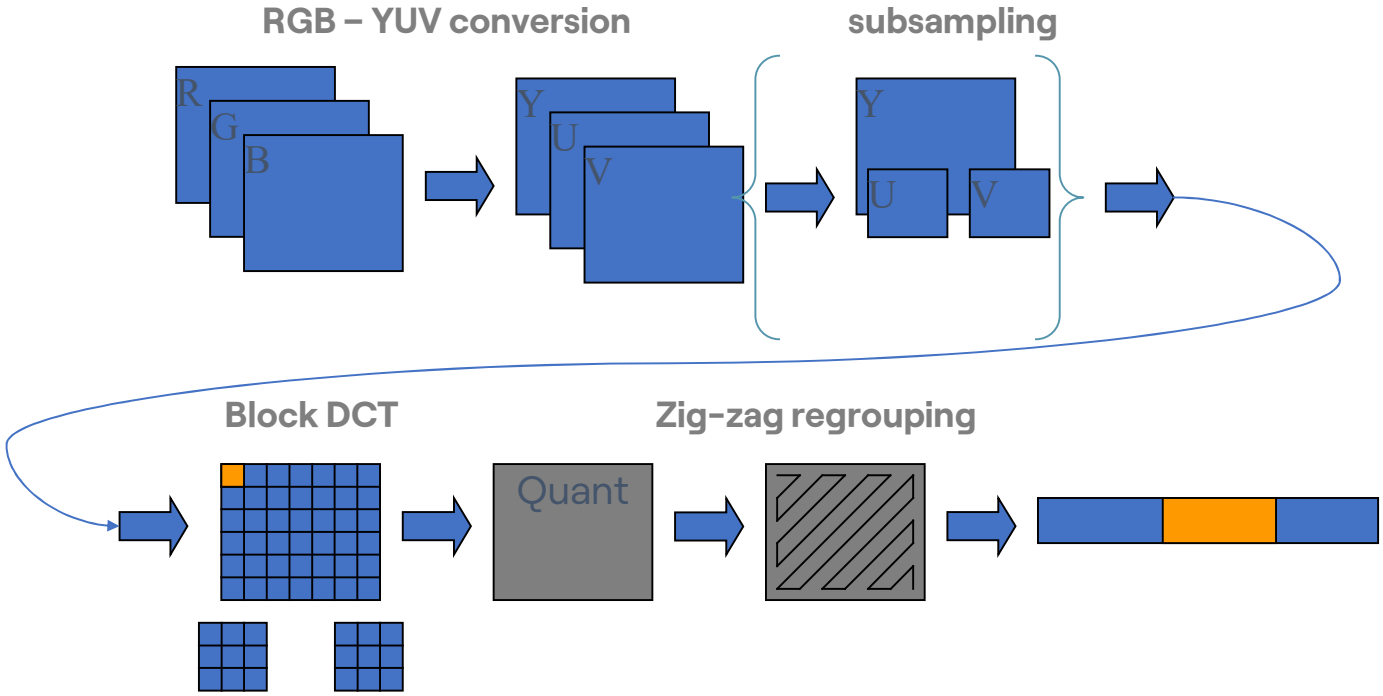
- In the design of high-end products we will very often have to use ALL available methods and their combinations

- In the continuation we will study how to approach the optimization and execution of some key parts of multimedia algorithms

Ways to optimize algorithms (3)

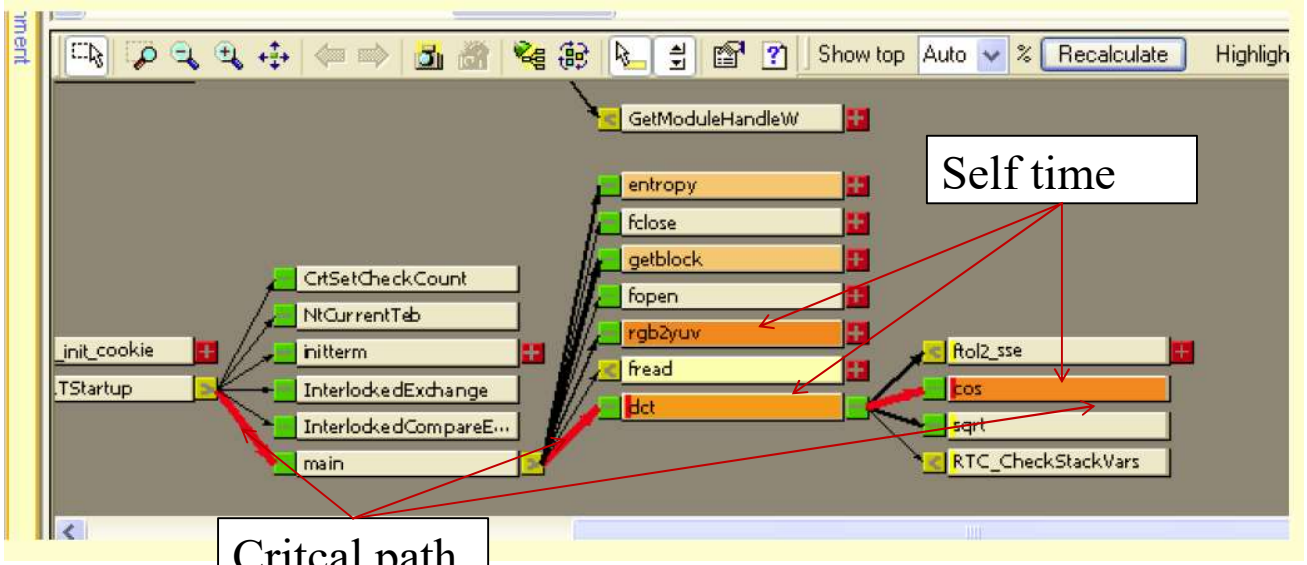
- In order to be able to consider possible ways of optimizing, WE MUST KNOW WELL:
  - Algorithms we optimize
  - Software solutions we use
  - The architecture of the system on which the algorithms run
- We are about to try to consider multimedia algorithms according to this division

JPEG encoder



JPEG Profile – VTune

Intel VTune Performance Analyzer:



JPEG Profile – gprof

- Unix/Linux distributions
  - gcc –pg –O2 jpeg_encode.cpp –o jpeg_encode
  - ./jpeg_encode
  - gprof jpeg_encode > jpeg_encode.prof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 92.63      2.75     2.75     4096   671.68   671.68  dct()
  2.69      2.83     0.08    12288     6.51     6.51  pipe(rlstruct*, pestruct*)
  1.68      2.88     0.05     4096    12.21    12.21  rgb2yuv()
  1.35      2.92     0.04    12288     3.26     3.26  runlen(int*, rlstruct*, int)
  0.67      2.94     0.02   823296     0.02     0.04  codeinsert(int)
  0.34      2.95     0.01   823296     0.01     0.01  status(pestruct*, int*, int)
  0.34      2.96     0.01   113555     0.09     0.09  order(char*, int*, int, int, int*)
  0.34      2.97     0.01     4096     2.44     2.44  getblock(int)
  0.00      2.97     0.00    12288     0.00     0.00  zigzag(int*)
  0.00      2.97     0.00    12288     0.00    13.03  entropy(int*, int)
```

JPEG Profile – gprof (2)

```
Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.34% of 2.97 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    2.97                 main [1]
                2.75    0.00    4096/4096        dct() [2]
                0.00    0.16   12288/12288       entropy(int*, int) [3]
                0.05    0.00    4096/4096        rgb2yuv() [5]
                0.01    0.00    4096/4096        getblock(int) [10]
                0.00    0.00       1/113555      order(char*, int*, int, int, int*)
                0.00    0.00       1/1           std::basic_string<char, std::char_t
                0.00    0.00       1/1           std::basic_string<char, std::char_t
-----------------------------------------------
                2.75    0.00    4096/4096        main [1]
[2]     92.6    2.75    0.00    4096         dct() [2]
-----------------------------------------------
                0.00    0.16   12288/12288       main [1]
[3]      5.4    0.00    0.16   12288         entropy(int*, int) [3]
                0.08    0.00   12288/12288       pipe(rlstruct*, pestruct*) [4]
                0.04    0.00   12288/12288       runlen(int*, rlstruct*, int) [6]
                0.02    0.01  823296/823296      codeinsert(int) [7]
                0.01    0.00  823296/823296      status(pestruct*, int*, int) [8]
                0.00    0.00   12288/12288       zigzag(int*) [17]
-----------------------------------------------
```

Implementation

- Hotspot – System parts (process, thread, part of code, memory address) where there is a distinct activity

- Critical parts of the JPEG encoder (computational):
  - DCT transformation
  - RGB to YUV conversion
  - Entropy coding

JPEG – DCT

- 2D-DCT, 2D-IDCT

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N)$$

$$f(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v) C(u,v) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N)$$

- Compaction of energy into low-frequency components
- Computational complexity:
  - Floating point data operations

JPEG – DCT (2)

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \quad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

- For each element:
  - α()α() ΣΣ f*cos()*cos()
  - In principle, 64 multiplications, 63 additions (assuming flattening of cos coefficients and coefficients α)
  - O(n^4)

JPEG – DCT description: https://www.math.cuhk.edu.hk/~lmlui/dct.pdf

JPEG – DCT (3)

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \qquad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

- HDTV (1080p, 1920x1080)

- First optimization: Calculation in integer arithmetic (fixed point, embedded computer systems)
  - Computation Precision

JPEG – DCT (4)

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \qquad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

- There are many different mathematical and computational methods for how to efficiently calculate such a function
- We'll look at some of the most essential:
  - Precalculated tables (lookup tables)
  - Separability

2D DCT: Lookup tables

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \qquad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

- Due to the discretized values of cos() parameters, factors are NOT calculated but precalculated values stored in the table (lookup table) are used.
- Combining with parameters α
- Same complexity O(n4), reduced processing time

2D DCT

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \qquad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

- Same complexity (two double loops):
- $O(n^4)$
  - 2D DCT for block 8x8:
    - For one element: 64 multiplications, 63 additions
    - For block 8x8: 4096 multiplications, 4032 additions
  - For only one image in resolution 1280x1024
    - 83886080 multiplications (~83M), 82575360 (~82M) additions

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \qquad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

2D DCT: Separability

$$C(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) cos(((2x+1)u\pi)/2N) cos(((2y+1)v\pi)/2N) \qquad \alpha(u) = \begin{cases} \sqrt{(1/N)} & \text{for } u = 0 \\ \sqrt{(2/N)} & \text{for } u = 1, 2, ..., N-1 \end{cases}$$

- 2D DCT can be calculated using 1D DCT, which is calculated by rows and then by columns

- 1D DCT:

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) cos(((2x+1)u\pi)/2N)$$

1D DCT

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) cos(((2x+1)u\pi)/2N)$$

- Complexity:
  - α() Σ f*cos()
  - Theoretical 8 multiplications, 7 additions
  - 8 calculations of cos() function (which has 1 division in parameters, 3 multiplications, 1 summing up)
  - 1D DCT: $O(n^2)$
  - 2D DCT using 1D DCT: $O(n^3)$

2D DCT: Separability with 1D DCT

$$C(u) = \alpha(u) \sum_{x=0}^{N-1} f(x) cos(((2x+1)u\pi)/2N)$$

- Using the separability property
- 1D DCT za 1 data: 8 multiplications, 7 additions
  - 1D DCT for 8 data
    - 64 multiplications, 56 additions
  - 2D DCT using 1D DCT
    - 8 rows and 8 columns: 16x1D DCT
    - 1024 multiplications, 896 additions
- For only one image in resolution 1280x1024
    - 20971520 multiplications, 18350080 additions

2D DCT: Separability with 1D DCT (2)

- Insufficient for systems with limited resources (built-in, battery-powered, etc.)

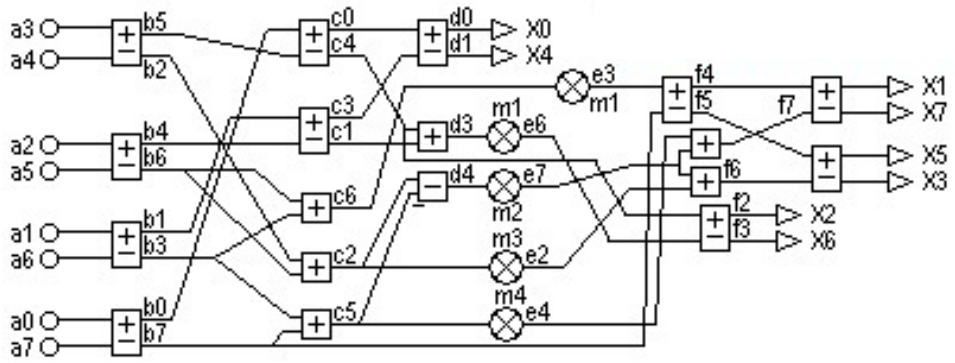
- Therefore, 2D DCT is accessed through the calculation of scaled fast DFT (the separability property is reused)
  - Modification of the algorithm
- Fast algorithms for calculating DCT

2D DCT: Fast algorithms

- Reducing the number of multiplications
  - Feig/Winograd: fastest 2D DCT
  - 2D DCT on 8x8: 94 multiplications, 454 additions
- LLM Loeffler/Ligtenberg/Moschytz: 1D DCT
  - 11 multiplications, 28 additions
  - 2D DCT on 8x8: 176 multiplications, 448 additions
- AAN (Arai/Agui/Nakajima) algorithm (1988) for scaled 1D DCT (8 points):
  - 5 multiplications, 29 additions (16 double complements)
  - 2D DCT on 8x8: 144 multiplications, 464 additions
- Kovač, Ranganathan algorithm (1995) for scaled 1D DCT (8 points):
  - 5 multiplications, 29 additions (12 double complements)

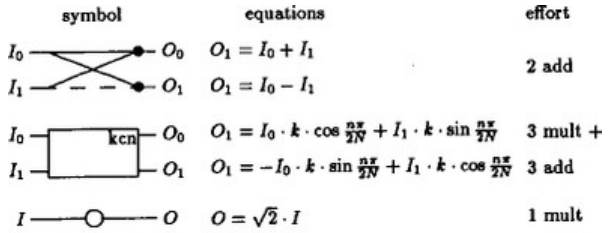
2D DCT: Feig & Winograd

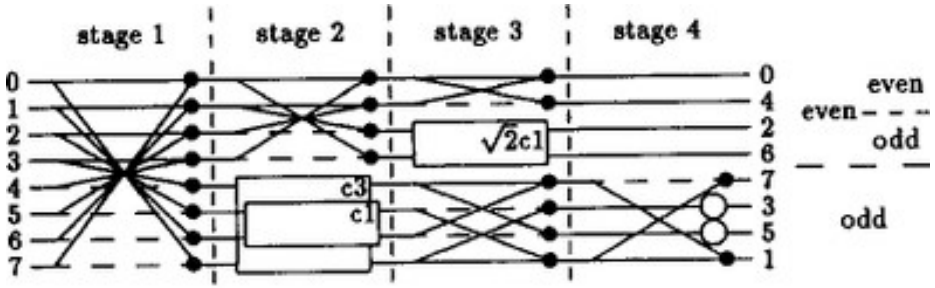
- Fastest 2D DCT
- 2D DCT on 8x8:
  - 94 multiplications,
  - 454 additions
  - Note: #additions> #multiplications



2D DCT: Algorithm LLM

- Loeffler/Ligtenberg/Moschytz:
  - 1D DCT (1989)
    - 11 multiplications, 28 additions
  - 2D DCT on 8x8:
    - 176 multiplications
    - 448 additions





2D DCT: Algorithm AAN

- Arai/Agui/Nakajima (1988)
  - algorithm (1988) for scaled 1D DCT:
    - 5 multiplications, 29 additions (16 double complements)
  - 2D DCT on 8x8:
    - 144 multiplications, 464 additions

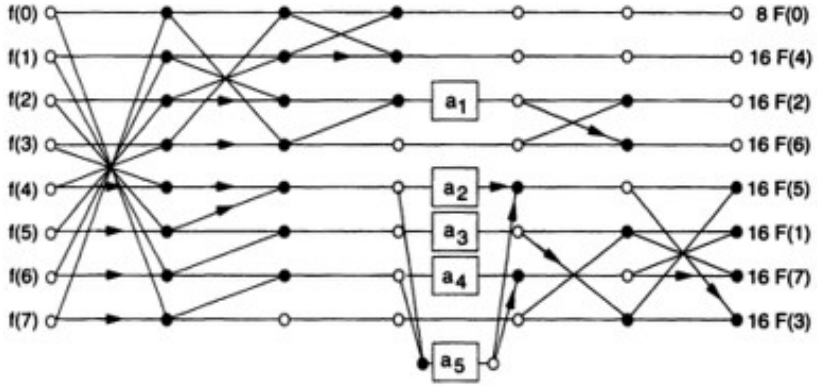


Figure 4-8. Flowgraph for 1-D DCT adapted from Arai, Agui, and Nakajima. $a_1 = 0.707$, $a_2 = 0.541$, $a_3 = 0.707$, $a_4 = 1.307$, and $a_5 = 0.383$.

Brzi algoritmi AAN

- DFT symetry

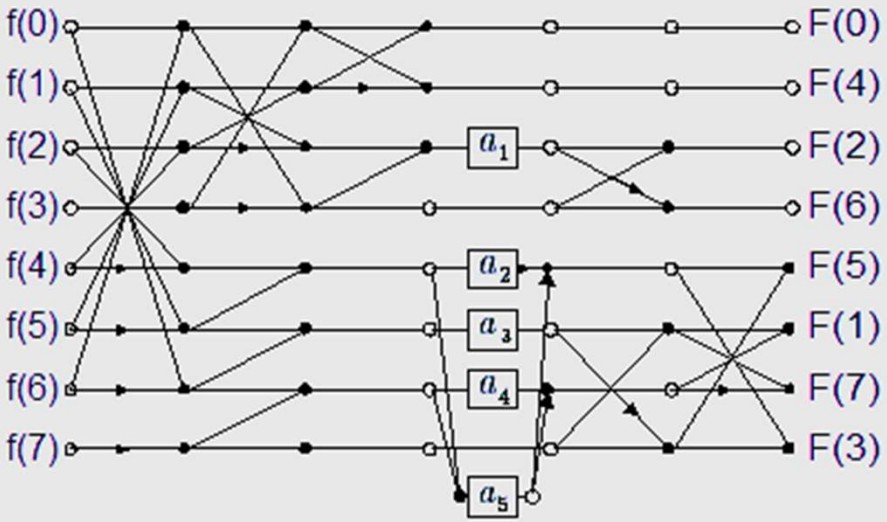


Figure 1. Flowgraph for 8-point DCT adapted from Arai, Agui, and Nakajima.
a1 = 0:707; a2 = 0:541; a3 = 0:707; a4 = 1:307; and a5 = 0:383

2D DCT: Fast algorithms (honorable mentions)

- Kovač, Ranganathan algorithm (1995) for scaled 1D DCT (8 points):
  - Based on the AAN
  - 5 multiplications, 29 additions, 12 double complements
  - Tailored for execution in VLSI hardware

JAGUAR: A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard

MARIO KOVAC AND N. RANGANATHAN, SENIOR MEMBER, IEEE

In this paper, we describe a fully pipelined single chip VLSI architecture for implementing the JPEG baseline image compression standard. The architecture exploits the principles of pipelining and parallelism to the maximum extent in order to obtain high speed and throughput. The architecture for discrete cosine transform and the entropy encoder are based on efficient algorithms designed for high speed VLSI implementation. The entire architecture can be implemented on a single VLSI chip to yield a clock rate of about 100 MHz which would allow an input rate of 30 frames per second for 1024 × 1024 color images.

Keywords—*data compression, DCT, JPEG, parallel processing, VLSI*

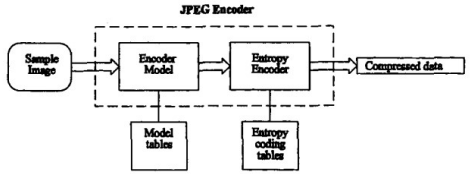


Fig. 1. JPEG Encoder.

Exercise Task (Optional)

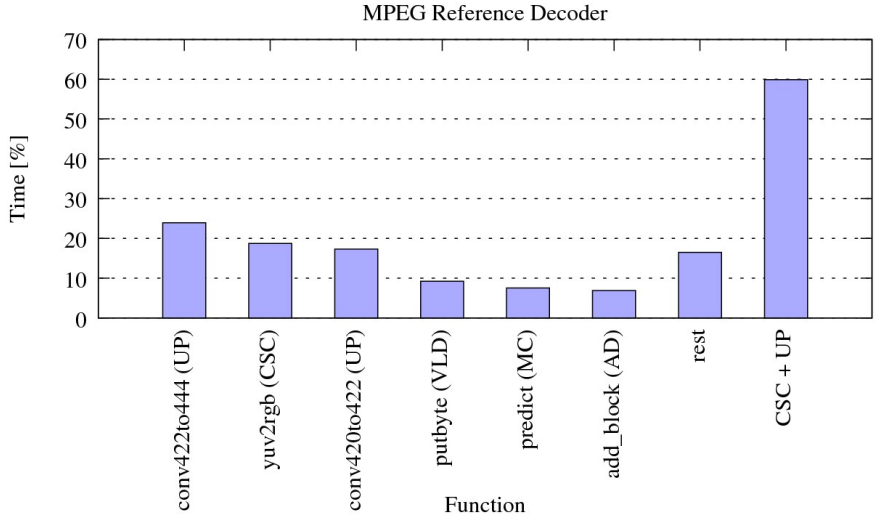
- Write a program that will count 2D DCT for an input image using a basic (theoretical) algorithm and using detachability and 1D DCT

Color space conversion

- HVS (Human Visual System)
  - Sensitivity to brightness
- RGB color space (display on screen)
- YUV color space (compression methods)
- Before starting compression:
  - RBG -> YUV
- Inverse transformation is performed after decoding, and before displaying on the screen
  - YUV -> RGB

MPEG-2 Decoder (MediaBench) – run analysis

- Optimized DCT
- UP – upsampling
- CSC – Color Space Conversion



Color space conversion (RGB2YUV v1)

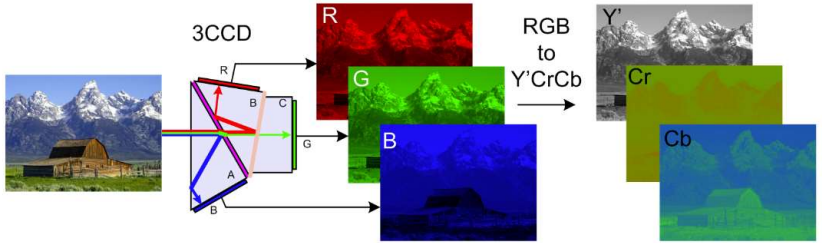
- RGB–YUV conversion can easily be done with a simple matrix operation:

- Y = (0,257*R) + (0,504*G) + (0,098*B) + 16
- U =-(0,148*R) – (0,291*G) + (0,439*B) + 128
- V = (0,439*R) – (0,368*G) – (0,071*B) + 128

- Image element:
  - 9 multiplications and 9 additions (+retrieval, storing)

Color space conversion

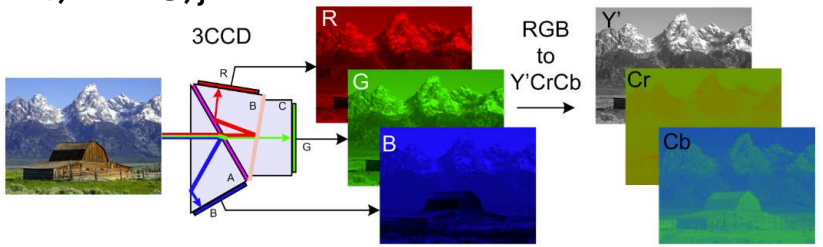
- Although mathematically trivial, color conversion is one of the more computer-demanding tasks
- Image of resolution 1280x1024:
  - 11,9M multiplications
  - 11,9M additions
  - +retrieval, storing
- Let's look at the example to see how long it takes:
  - Example
  - RGB2YUV v.1.



Color space conversion (2)

- Version 2: basic approximation
  - Y = (0.257 * R) + (0.504 * G);
  - U = – (0.291 * G) + (0.439 * B) + 128;
  - V = (0.439 * R) – (0.368 * G) + 128;
  - Example
  - RGB2YUV v.2.

- Version 3: improved approximation
  - data[i++] = (byte)(R/4 + G/2);
  - data[i++] = (byte)(-(G/4) + B/2 + 128);
  - data[i++] = (byte)(R/2 – (G /4) + 128);
  - Example
  - RGB2YUV v.3.



Analyzes

- For one very demanding operation, we were able to significantly reduce the processing requirements
- However, we achieved this while degrading the quality of data calculations
- Whether the quality is satisfactory or not is very difficult to define empirically and therefore objective/subjective quality tests should be done

Exercise Task (Optional)

- Use any tool (Visual Studio, Matlab, Mathematica, ...) to compare the quality of the three color space conversion methods described earlier
- Procedure:
  - Write a function that loads a color picture into RGB
  - Write three different color transformation functions (one of which is in full formulas)
  - Calculate MSE for each component (Y,U,V) for one test image for two approximative solutions
  - Calculate PSNR for all solutions
  - You can download the test image (paradise.raw, 24bpp RGB, 1280x1080) from the WEB

Entropy coding – Huffman

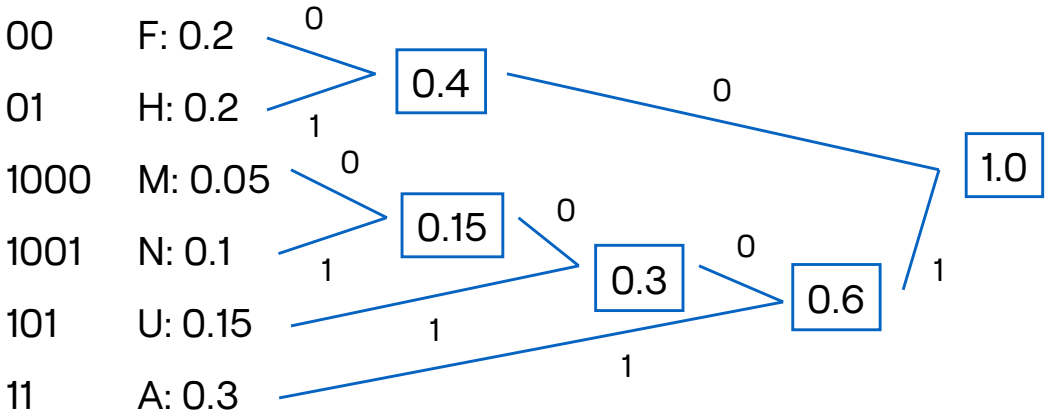
- Huffman D., method published in 1952.
  - Prefix codes
  - Optimal codes for a given probability set
  - Shortest variable length code (integer length)

$$H(S) <= l < H(S) + 1$$

  - Today, the most commonly used algorithm for entropy coding

Huffman algorithm

Encoding example:



HUFFMAN = 01101000010001111001

56 bits 19 bits

Huffman algoritm – Executions

- Executions:
  - Static execution
    - Separate passage through data with the aim of assessing statistical properties
    - Predetermined Tables (JPEG)
    - Fast and easy
  - Dynamic execution (adaptive)
    - One pass
    - Adaptive building of Huffman tree
    - Demanding (algorithmic: requires restructuring of Huffman tree)

Huffman excutions: Compression

- Compression performance:
- “Model cost”
  - Static execution
    - Additional information for the decoder
      - Statistics Table
      - Code Words
  - Dynamic execution (adaptive)
    - "Learning" effect
    - Non-optimal codes at the initial stage of construction of the statistical model
    - Context dilution

Golomb-Rice

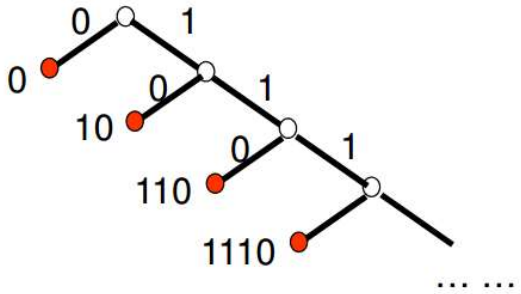
- Golomb-Rice codes:
  - Special case of Huffman codes
  - Coding non-negative integers, larger numbers are less likely to occur
  - Parameter: $m=2^k$, k = 0, 1, 2, …
    - Symbol n >= 0

$$n \rightarrow Un\left(\left\lfloor \frac{n}{m} \right\rfloor\right) \circ Bin(n\%m); n \geq 0$$

Golomb-Rice (2)

- Optimal code:
  - When the probabilities of symbols are $1/2^k$

Symbol	Code word
0	0
1	10
2	110
3	1110
4	11110
5	111110



Golomb-Rice (3)

Golomb / Rice	$m=1$ $k=0$	$m=2$ $k=1$	$m=3$	$m=4$ $k=2$	$\cdots$	$m=6$	$\cdots$	$m=8$ $k=3$
$n=0$	0·	0·0	0·0	0·00		0·00		0·000
1	10·	0·1	0·10	0·01		0·01		0·001
2	110·	10·0	0·11	0·10		0·100		0·010
3	1110·	10·1	10·0	0·11		0·101		0·011
4	11110·	110·0	10·10	10·00		0·110		0·100
5	111110·	110·1	10·11	10·01		0·111		0·101
6	1111110·	1110·0	110·0	10·10		10·00		0·110
7	11111110·	1110·1	110·10	10·11		10·01		0·111
8	111111110·	11110·0	110·11	110·00		10·100		10·000
9	1111111110·	11110·1	1110·0	110·01		10·101		10·001
⋮	⋮	⋮	⋮	⋮		⋮		⋮

$$n \rightarrow Un\left(\left\lfloor \frac{n}{m} \right\rfloor\right) \circ Bin\,(n\,\%\,m\,);n \geq 0$$

Golomb-Rice: Implementation

- Coding

```
UnaryEncode(n) {
  while (n > 0) {
    WriteBit(1);
    n--;
  }
  WriteBit(0);
}
```

- Decoding

```
UnaryDecode() {
  n = 0;
  while (ReadBit(1) == 1) {
    n++;
  }
  return n;
}
```

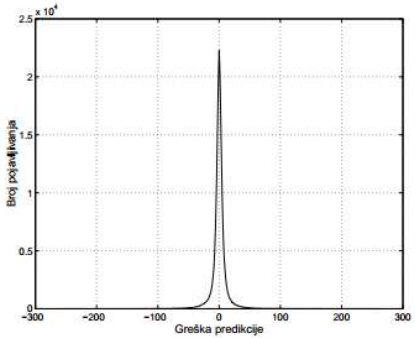
Golomb-Rice: Negative values

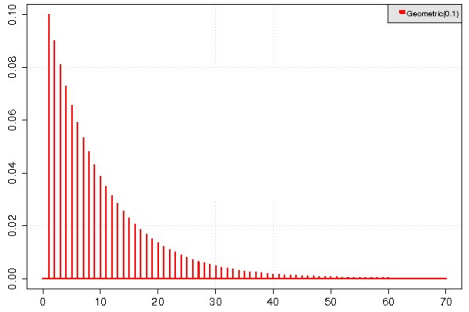
- Mapping

  0, 1, -1, 2, -2, 3, -3, 4, -4

  0, 1, 2, 3, 4, 5, 6, 7, 8

```
if (s< 0) {
    ms= 2 * (-s) - 1;
} else {
    ms= 2 * s;
}
```



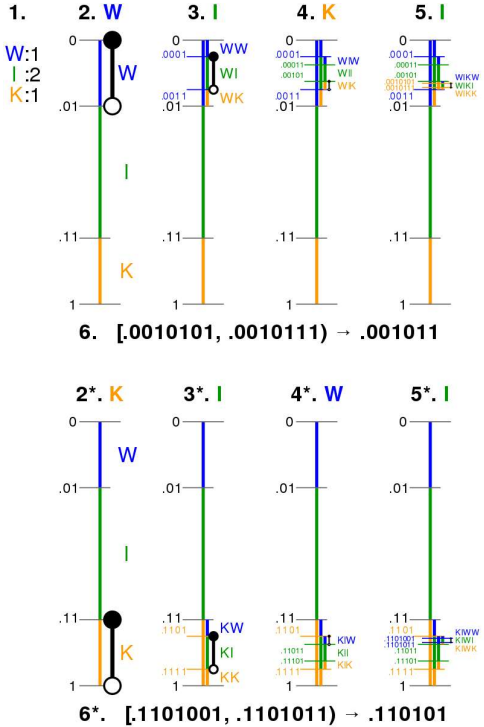


Arithmetic coding

- Does not generate prefix codes
- The entire input data series is encoded in a single code word
- Advantages: flexibility (separation of model and encoder), theoretical optimality with regard to the length of the code, practical realizations closest to optimal
- Shortcomings: Speed, poor fault resistance, naïve implementation in a full-across high-resolution arithmetic

Arithmetic coding – Execution

- Incremental transmission
  - As the interval decreases, greater precision is required
  - The encoder ejects the leading bit as soon as it is fixed
- Fixed arithmetic (implementation of integer algorithm)
- Various algorithm designs (hardware, programmatic): QM-coder, reduced-precision, binary arithmetic coding, ...



Arithmetic coding – JPEG Execution

- JPEG version with s arithmetic encoder QM-coder (Patent: IBM, Mitsubishi, Lucent)
  - Protected by patent rights
  - Rarely used
  - Not part of the basic profile due to patent rights
- JPEG2000 – arithmetic MQ-coder (IBM, Mitsubishi)
  - Partially the reason of the poor reception of the norm

Arithmetic coding – patents

- U.S. Patent 4,122,440 — (IBM) Filed 4 March 1977, Granted 24 October 1978
- U.S. Patent 4,286,256 — (IBM) Granted 25 August 1981
- U.S. Patent 4,467,317 — (IBM) Granted 21 August 1984
- U.S. Patent 4,652,856 — (IBM) Granted 4 February 1986
- U.S. Patent 4,891,643 — (IBM) Filed 15 September 1986, granted 2 January 1990
- U.S. Patent 4,905,297 — (IBM) Filed 18 November 1988, granted 27 February 1990
- U.S. Patent 4,933,883 — (IBM) Filed 3 May 1988, granted 12 June 1990
- U.S. Patent 4,935,882 — (IBM) Filed 20 July 1988, granted 19 June 1990
- U.S. Patent 4,989,000 — Filed 19 June 1989, granted 29 January 1991
- U.S. Patent 5,099,440 — (IBM) Filed 5 January 1990, granted 24 March 1992
- U.S. Patent 5,272,478 — (Ricoh) Filed 17 August 1992, granted 21 December 1993

Run-Length algorithm

- Based on successive occurrence of input symbols
- Very easy to perform
- Used for example in the compression of an image in binary image data (fax transmission)

Run-Length algorithm (2)

- Example:
  - Lines read from a document in a fax machine
  - 1 line (75 dpi, 8”)=600 bits=75 B
  - 000000000000000011111111111111111111111100000000000.....000111111111.....11111000000
  - 17,24,3,211,22,188,77,54,4 = 9 B

Example: Stock market price problem

- Example: Stock price status on the stock exchange

- Model: Forecast Function
  - $x_t = x_{t-1} + (x_{t-1} - x_{t-2})$

Prediction

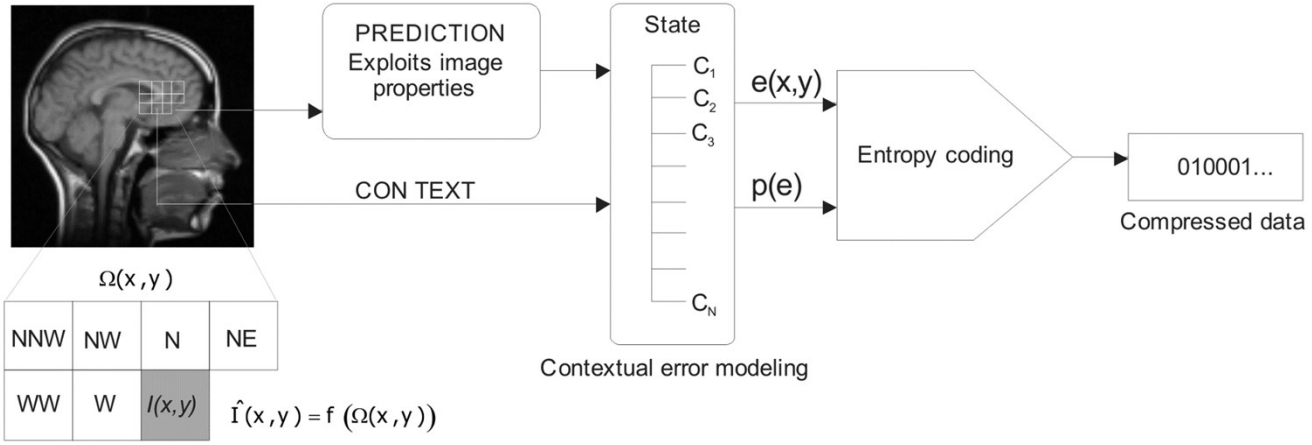
Stock price	Predicted	Difference
134	0	134
141	0	141
145	148	-3
150	149	1
153	155	-2
152	156	-4
150	151	-1
...		

Prediction + Huffman

- Prediction:
  - a new, very good distribution is obtained (frequent appearance of small values, rare appearance of large ones)
  - Arithmetic coding
  - Huffman
    - good compression for distribution similar to the above
- Golomb-Rice

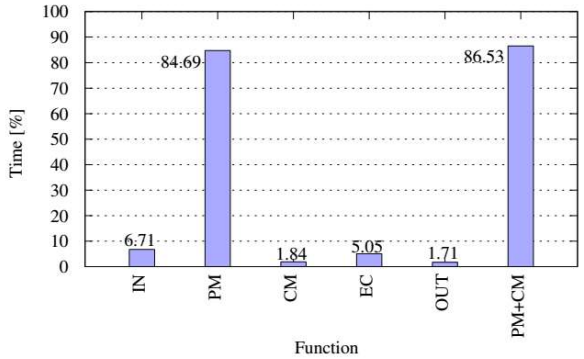
Compression of image data without losses

- PM Predictive Model
- CM Contextual Model
- EC Entropy Coding

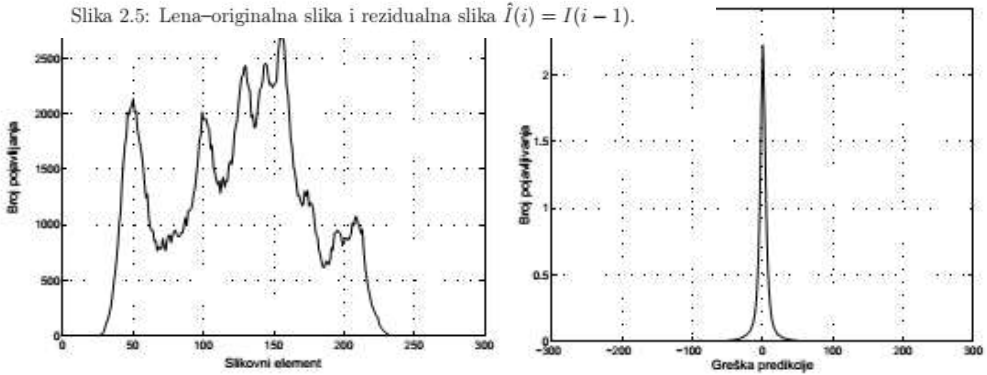


Compression of image data without losses (2)

(a) Originalna slika (b) Rezidualna slika

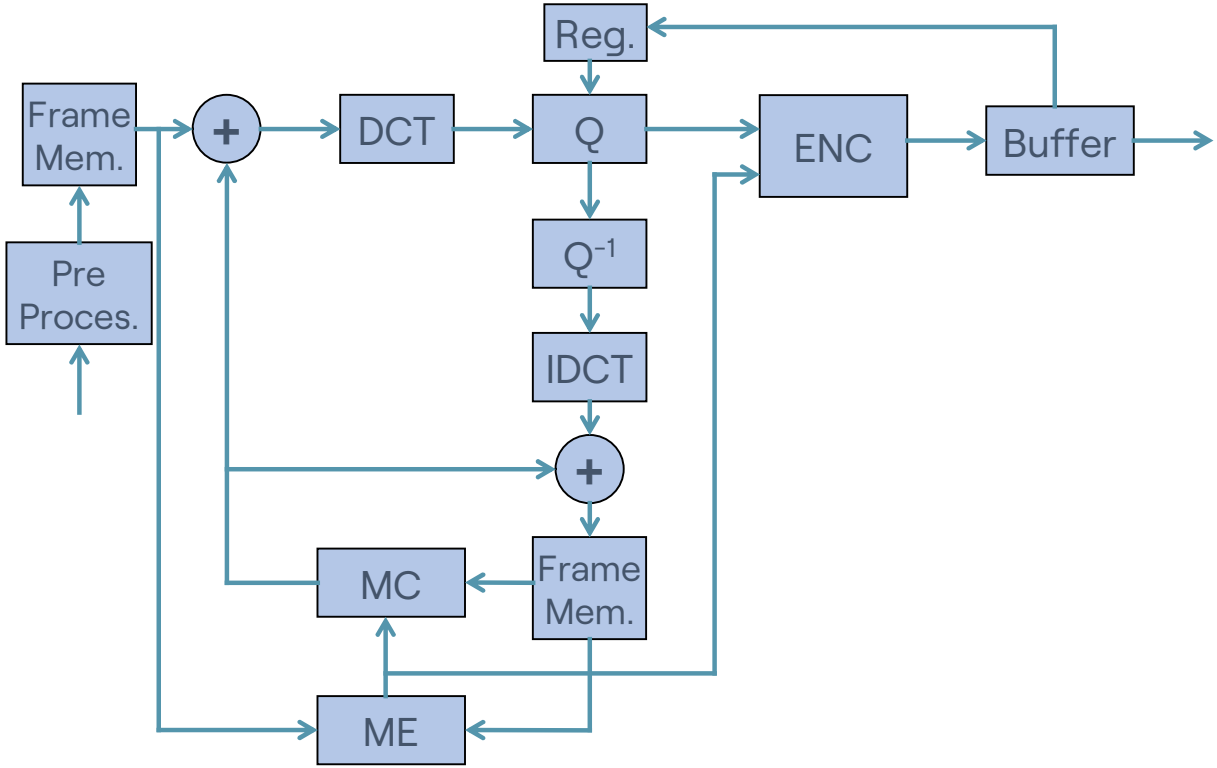


Slika 2.5: Lena–originalna slika i rezidualna slika $\hat{I}(i) = I(i-1)$.

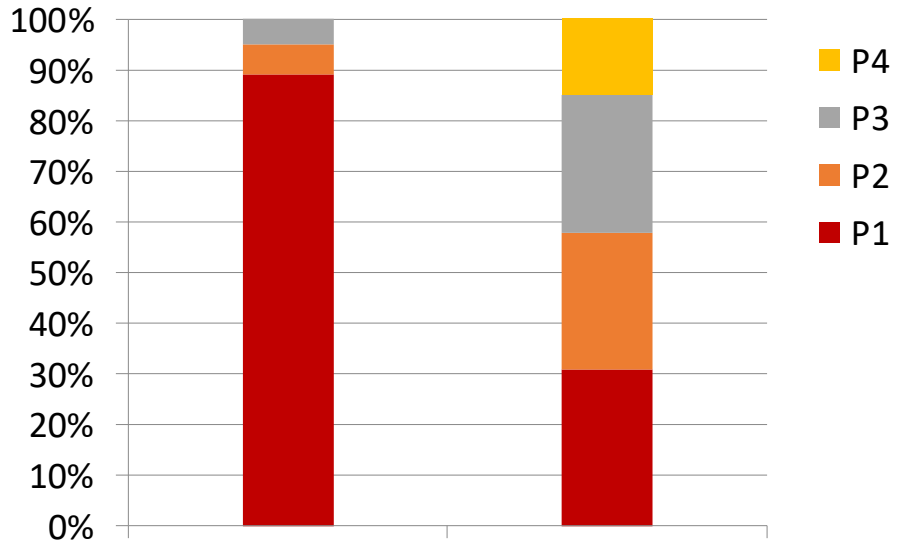


(a) Entropija modela = 7.45 bps (b) Entropija modela = 5.06 bps

MPEG coder


Reg.
Frame Mem.
+
DCT
Q
ENC
Buffer
Pre Proces.
Q-1
IDCT
+
MC
Frame Mem.
ME

MPEG-2: Run Analysis



P1 – ME, FDCT, MC
P2 – Q, VLC, Rate Control
P3 – IQ, IDCT
P4 – Other

P1 – IDCT
P2 – MC
P3 – VLD
P4 – Other

Iwata, Olokutun: Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm