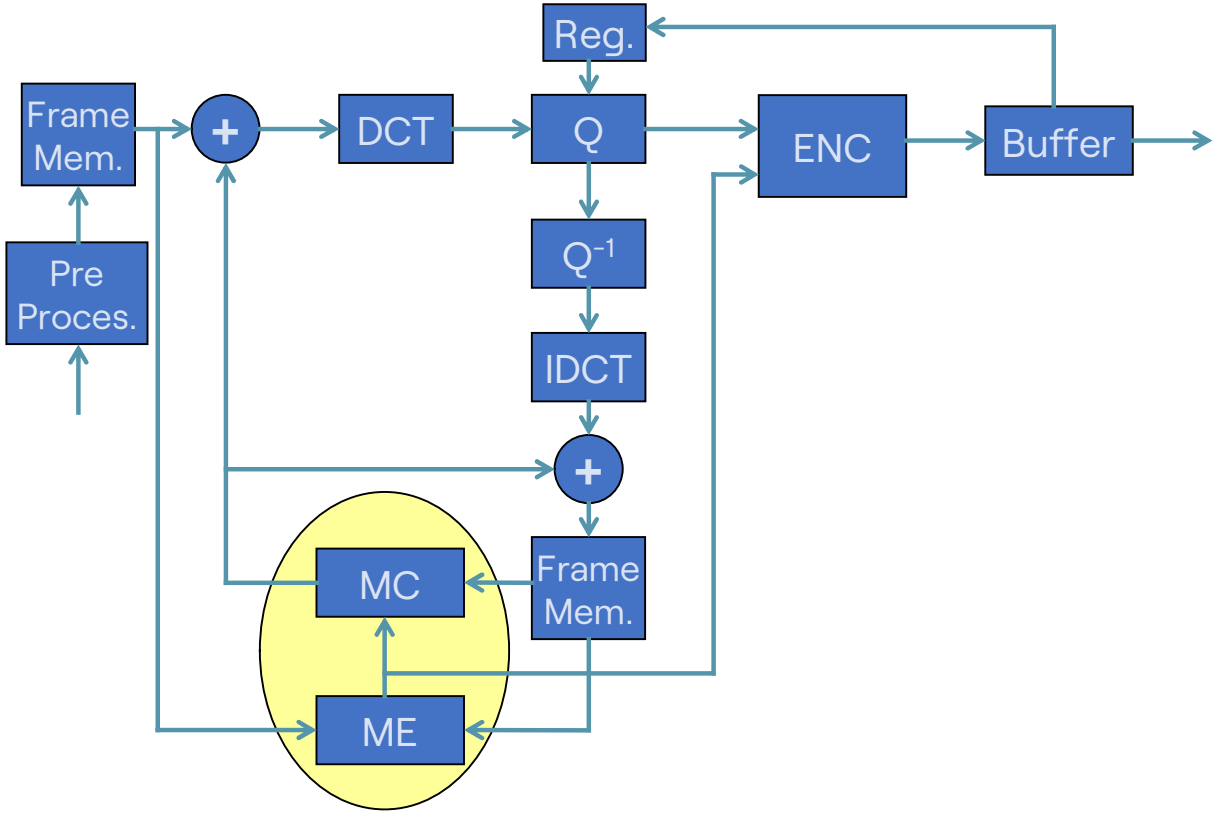
Multimedia Systems

Assistant professor Daniel Hofman, PhD

Associate professor Josip Knezović, PhD

MPEG coder


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

Types of redundance

- Spatial redundancy
  - Natural visual data (images) – correlation among neighbouring elements
- Temporal redundancy
  - Moving objects in adjacent visual images (frames)
  - Stationary scene
- Movements:
  - Magnifications/reductions (zoom)
  - Rotations
  - Translations

Intra and inter-block compression

- Intra-block compression
  - Removal of high frequency information that the human eye cannot recognize
  - Same as image compression (JPEG) as explained beforehand
- Inter-block compression
  - Reducing temporal redundancy
- Basis: Characteristics of the human visual system (HVS)

Interblock compression

- Often, individual parts of the image within a string change very little or are even unchanged
- Picture background often doesn't change

foreground (dynamic) background (static)

- There's not much change between adjacent sequence images
  - parts of the image are repeated – temporal redundancy
- Changing parts can be analysed as movements describing changes

Temporal redundancy

Temporal redundancy (2)

Temporal redundancy (3)

Estimation and compensation of movements

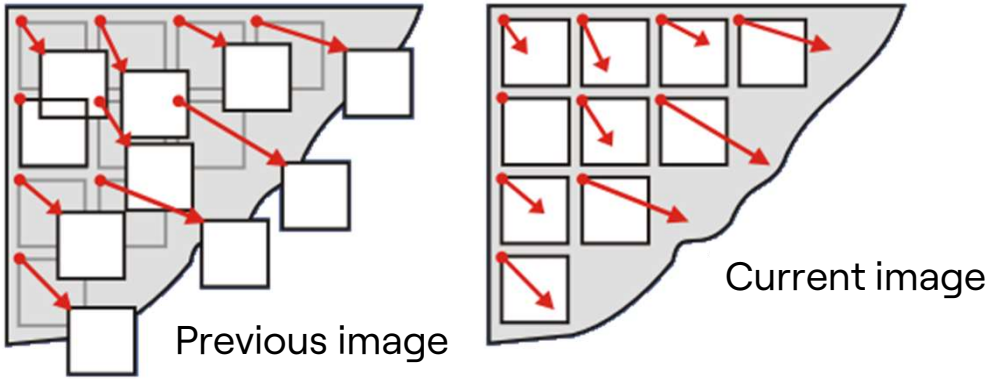
- Motion Estimation (ME):
  - Extract parts of a picture (motion segmentation)
  - Describing movements (estimate)
  - Running in encoder
- Motion Compensation (MC):
  - Use motion estimation results
  - Running in a decoder

Estimation and compensation of movements (2)

- Implementation:
  - Block: Segmentation into rectangular parts of pre-set dimensions (block-based motion estimation)
  - Each movement is translational (movement vector)

Estimation and compensation of movements (3)

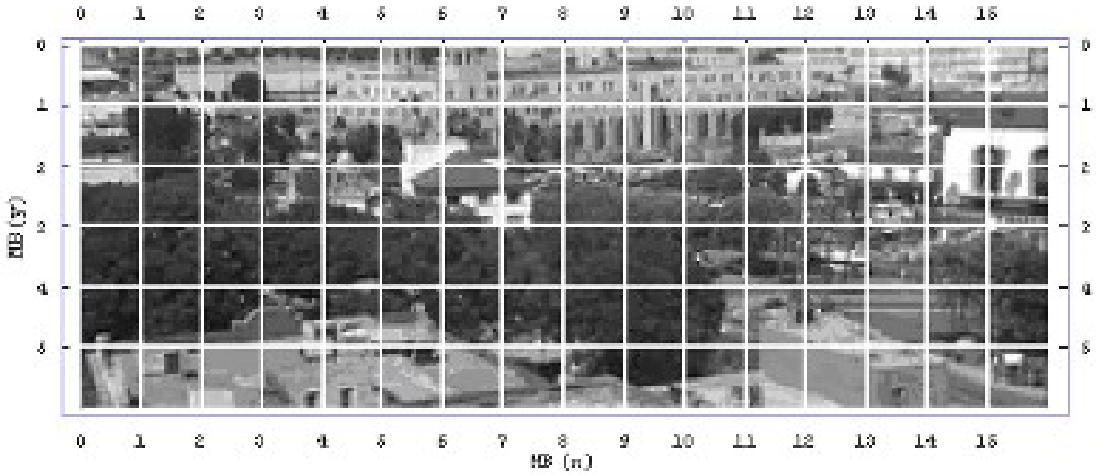
- Motion estimation algorithms for each block calculate the corresponding movement vector
- The vector shows the source of the block in the previous image before moving to the position in the current image



Estimation and compensation of movements (4)

Motion estimation

- For more efficient processing to assess movements, component Y is sufficient



Distance criterion

- The measure of similarity between the two blocks is the distance criterion
- Motion estimation is reduced to optimising the distance criterion as a criterion function
- Algorithms in a defined search area seek to find the minimum criterion function
  - Change of the picture in this location is the smallest
  - Movement of the reference block along the motion vector

Distance criterion (2)

- Some possible distance criterion of two blocks:
  - Mean Squared Error (MSE) – computationally demanding

$$MSE(x,y;d_x,d_y)=\frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left[Y(x+i,y+j,t_1)-Y(x+d_x+i,y+d_y+j,t_0)\right]^2$$

  - Mean Absolute Deviation (MAD) – widely accepted

$$MAD(x,y;d_x,d_y)=\frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left|Y(x+i,y+j,t_1)-Y(x+d_x+i,y+d_y+j,t_0)\right|$$

  - Number of matching picture elements (MPC)

$$MPC(x,y;d_x,d_y)=\frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}T\left[Y(x+i,y+j,t_1),Y(x+d_x+i,y+d_y+j,t_0)\right]$$

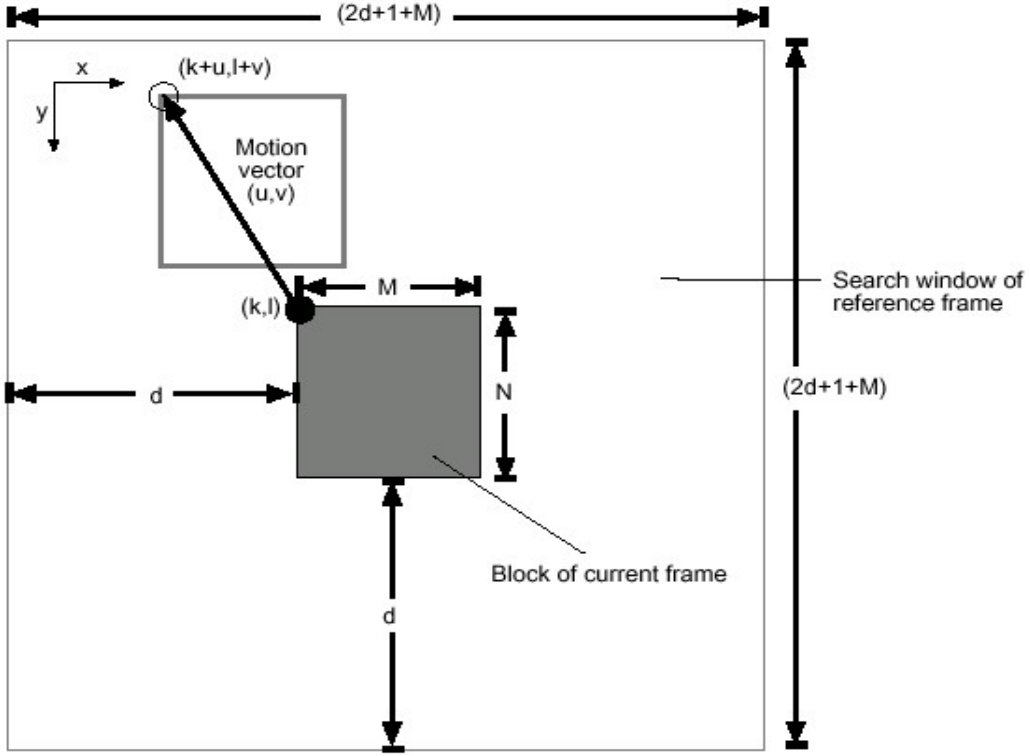
Motion estimation algorithms

- Depending on the search method, we distinguish between:
  - Full Search Algorithm
    - searches all points within the search area
    - computationally very demanding
    - gives the best possible result
  - Incomplete (fast, advanced) algorithms
  - The search area can be:
    - unalterable
    - adaptable

Full Search Algorithm

- Very demanding algorithm for computer resources:
  - Calculation of one Mean Absolute Deviation (MAD):
    - Data retrieval (very demanding)
    - For block 16x16: 256 subtractions + 1 offset (dividing by 256)
  - If the offset area is +-8 in both dimensions:
    - (2*8+1)*(2*8+1)*(256+retreival)=73984 + retrievals
    - Finding the minimum
  - For one picture 1280x1024
    - 5120*73984=378.798.080 + retrievals
  - For 30 pictures per second: 1,1*1010 additions/s

Full Search Algorithm (2)



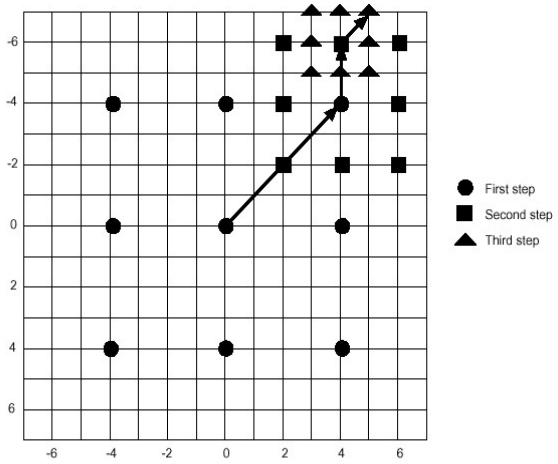
Full Search Algorithm (3)

- Optimal result but unfortunately in practice difficult to apply
- It is used mainly only by circuitry designs of the algorithm
- The need for fast algorithms
  - Benefits: less operations necessary
  - Disadvantages: larger differences to be encoded and thus the output amount of data is higher (lower degree of compression)
  - The quality of some algorithms is satisfactory and close to full searchalgorithm
  - Problems with movement types (explained later)

Fast algorithms

- Fast, advanced search algorithms – direct search depending on the value of the distance criterion
- Some examples:
  - Logarithmic Search (LOG)
  - Three-step search (3SS)
  - Orthogonal search (ORT)
  - Block-based gradient descent algorithm (BBGDS)

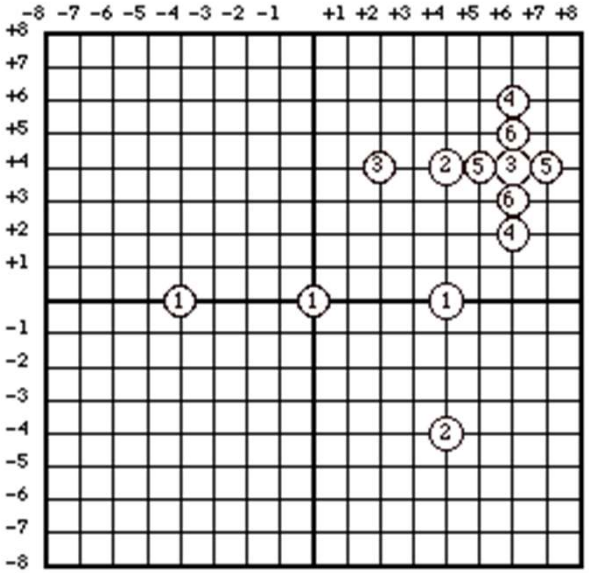
Three-step search (3SS)



0. N = 3
1. Compute SAE(0,0)
2. Set $S=2^{(N-1)}$ (step size)
3. Compute SAE on 2^N locs
4. Select loc with min SAE
5. Set S=S/2
6. Repeat [3:5] until S=1

- Each step – 9 points for calculations
- Number of points: 9+8+8 = 25 (FS: 15*15 = 225)

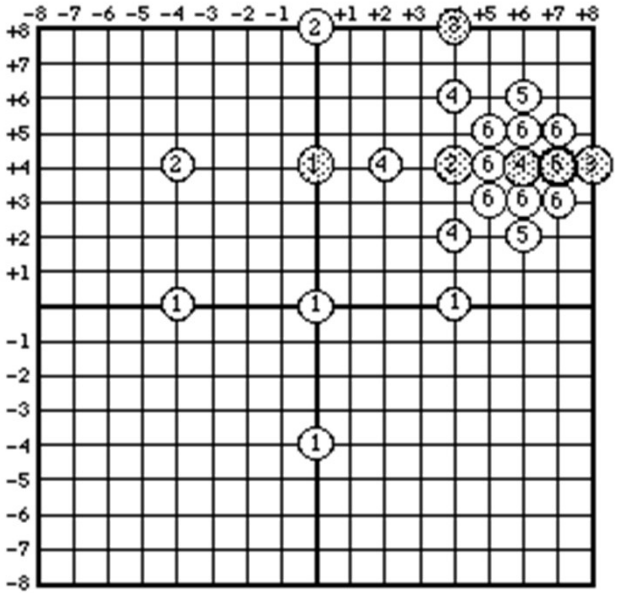
Orthogonal search (ORT)



0. S = Step size (d/2 = 8/2 = 4)
1. Compute SAE on 2 horizontal loc displaced S from origin
2. Select new origin
3. Repeat [1:2] vertically
4. Select loc with min SAE
5. Set S=S/2
6. Repeat [1:5] until S=1

Number of points: 3 + 2 + 2 + 2 + 2 = 13

Logarithmic Search (LOG)

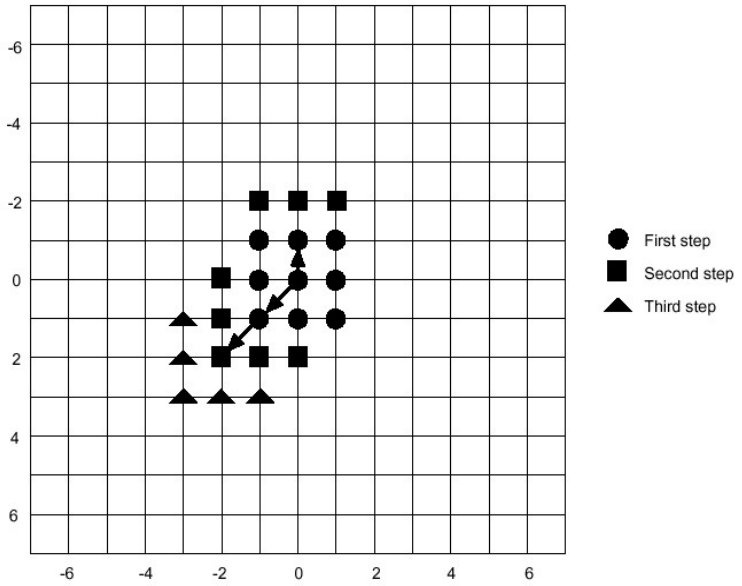


0. Step size = S (d/2) = 4
1. Search (0,0)
2. Search 4 loc S pixels away (horiz and vert.)
3. New origin = best match (BM)
4. If BM is in center: S = S/2
5. If S=1: goto 6, else 2
6. Search 8 loc arround BM and select BM

Number of searches: Depends on data

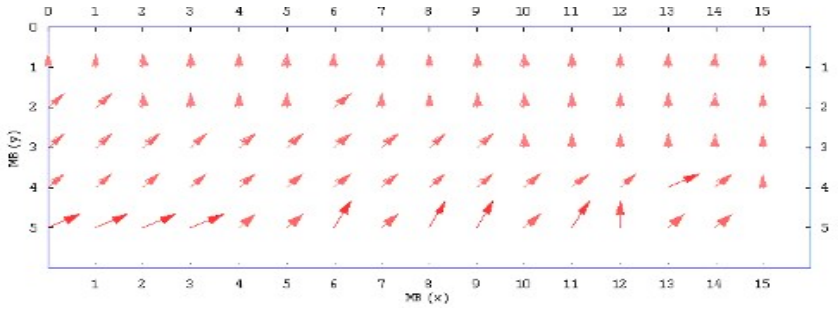
BBGDS algorithm (Gradient Descent Search)

- Centered search pattern with 9 dots with S=1 search step
- Number of searches not defined
  - End of Search:
    - BM is in the center
    - Window Edge
- Good for small movements



Distribution of movement vectors

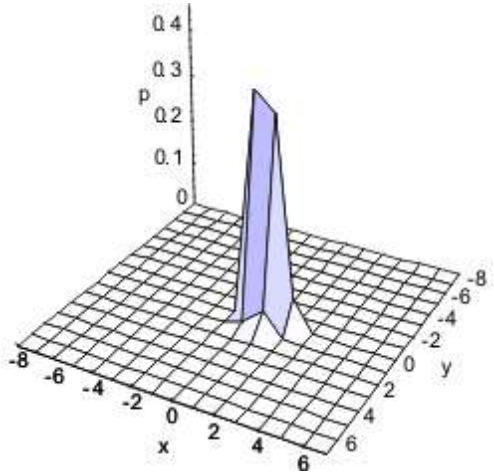
- The view indicates evenly distributed movement vectors between two sequence images
- Vectors are mostly short - centered vector distribution - typical for real-life sequences



Distribution of movement vectors (2)

- Counting vectors for all possible component pairs (x,y) throughout the sequence gives relative vector frequencies

- The efficiency of the algorithm can be increased if the observed centered probability distribution of vectors is taken into account, more attention should be paid to the center of the search area



Examples of algorithm comparison

- Parameters that determine the effectiveness of the algorithm:
  - the overall measure of disruption (distance criterium) after the movement assessment should be as small as possible (better compression)
  - the total number of test points should also be as low as possible (higher speed)
- We need an compromise between compression ↔ speed
- Example of three sequences containing different types of movements:
  - “City View” - evenly arranged and small movement
  - “Car Jump” - located and high movement
  - “Troops” - uniformly arranged and large movement

Comparison of search algorithms

- City view sequence – uniformly arranged small movements (throughout the frame)
- Car jump sequence – located and large movements (movements localized in part of the frame)
- Troops sequence – evenly distributed and large movements (all over the frame)

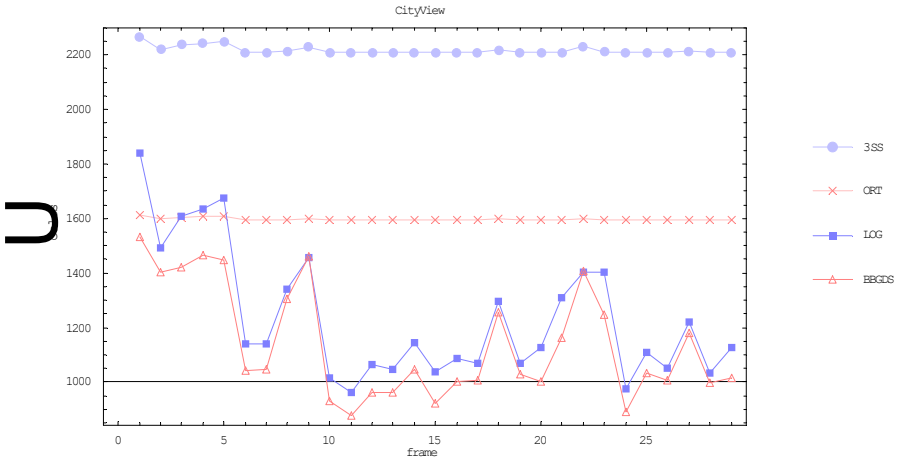
Implementation – Example

- The compression standard does not specify movement assessment procedures

- The quality of the application depends significantly on how the movement is assessed

Results (1)

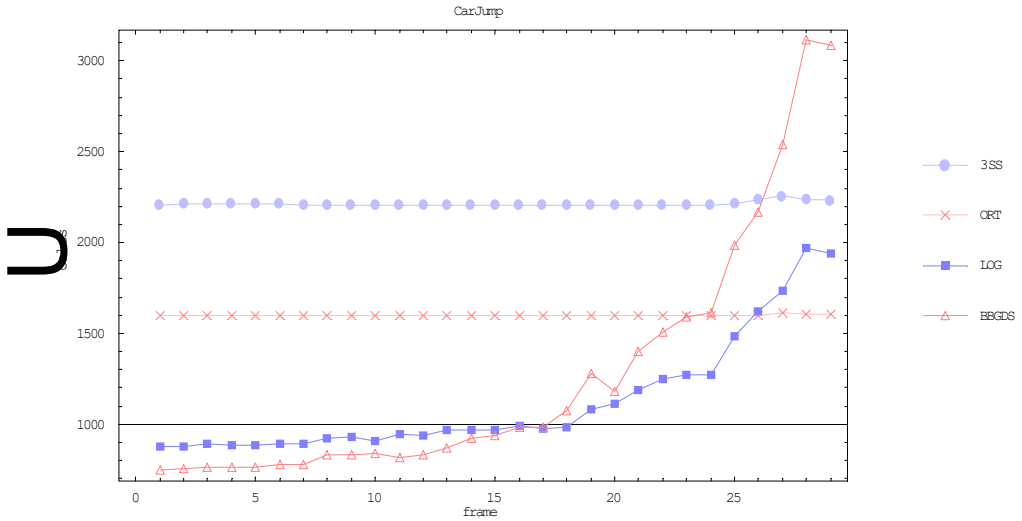
- "City View" sequence - number of test points:



- Centered Distribution: BBGDS Best
- ORT and 3SS constant and relatively large number of test points

Results (2)

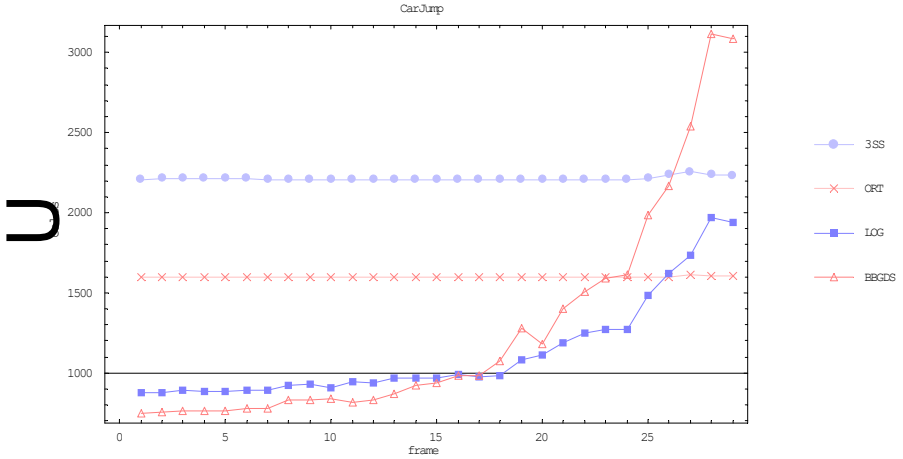
- "Car Jump" sequence - number of test points:



- Localized growing change: BBGDS and LOG produce underperformance as the disorder grows
- ORT better than 3SS, constant despite disruption

Results (3)

- "Troops" sequence – number of test points:



- Many complex movements: BBGDS bad for higher average displacement vector length
- LOG very good according to both criteria

Results (4)

- The effectiveness of a particular algorithm depends on the type of movement contained
- An effective procedure will heuristically select the most suitable algorithm based on the type of movement: adaptive algorithms

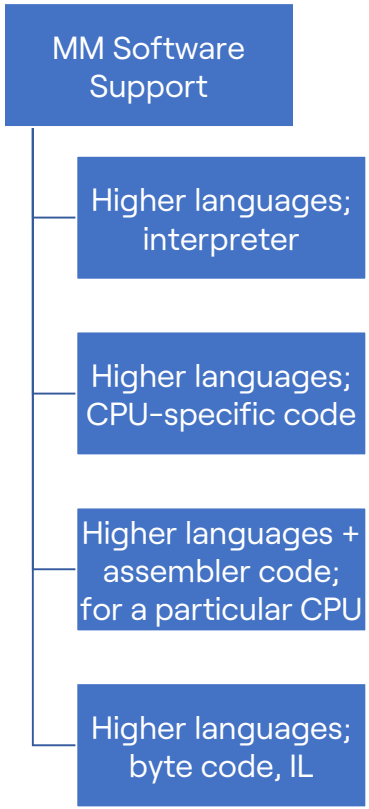
Performance

- With this, we briefly analyzed some of the most demanding algorithms when processing and compressing multimedia data
- We also saw the principle complexity of calculation without going into too much detail
- The question arises as to how to effectively perform such algorithms

Optimizations

- Let's remind ourselves from the last lecture to some core 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

Basic ways to SW support for MM

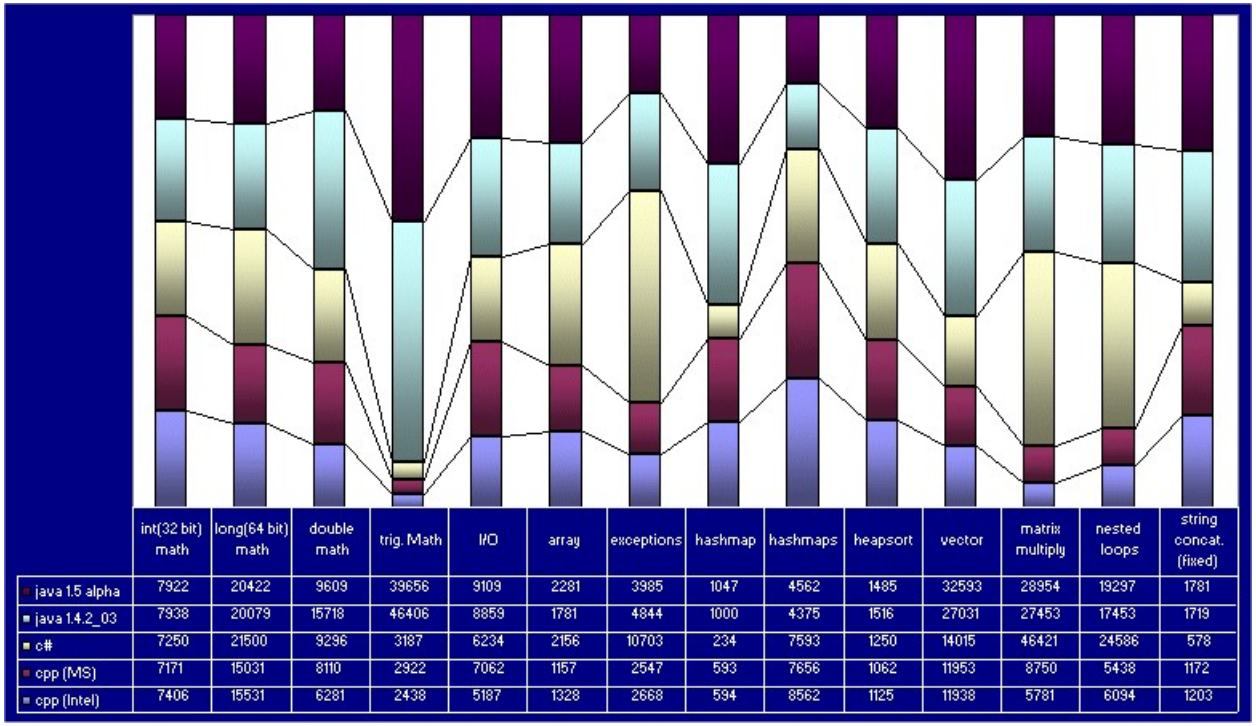


Performance

- You must be aware that a program written in Java or e.g. C# is running more slowly than programs that have been translated into target processor code

- Additionally: higher programming languages generate slower code than the one written in combination with the assembler

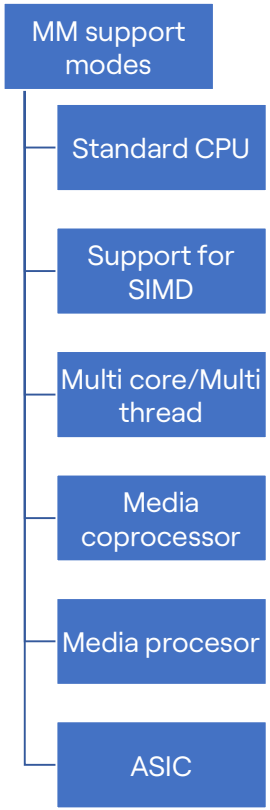
Comparison Java, C#, C++



	int(32 bit) math	long(64 bit) math	double math	trig. Math	I/O	array	exceptions	hashmap	hashmaps	heapsort	vector	matrix multiply	nested loops	string concat. (fixed)
java 1.5 alpha	7922	20422	9609	39656	9109	2281	3985	1047	4562	1485	32593	28954	19297	1781
java 1.4.2_03	7938	20079	15718	46406	8859	1781	4844	1000	4375	1516	27031	27453	17453	1719
c#	7250	21500	9296	3187	6234	2156	10703	234	7593	1250	14015	46421	24586	578
cpp (MS)	7171	15031	8110	2922	7062	1157	2547	593	7656	1062	11953	8750	5438	1172
cpp (Intel)	7406	15531	6281	2438	5187	1328	2668	594	8562	1125	11938	5781	6094	1203

www.tommti-systems.de/main-Dateien/reviews/languages/benchmarks.html

MM Basic CPU Support Modes



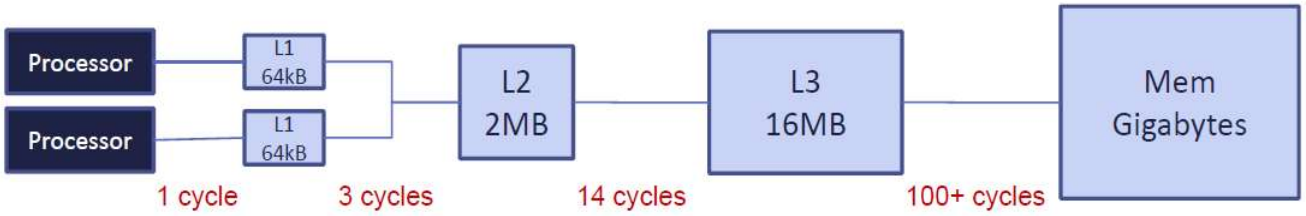
Standard CPU

- If we recall computer architecture then we know that ordinary processors can perform one ALU operation per period
- The operation is the width of the ALU

- In order to speed up performance, we need to pay a lot of attention to data retrieval and loop optimizations

Standard CPU – Memory System

- Memory Hierarchy
- Cache – smallest and fastest
- If the program access pattern corresponds to the heuristics of the cache access assembly: high speed
- If not: extremely slow!



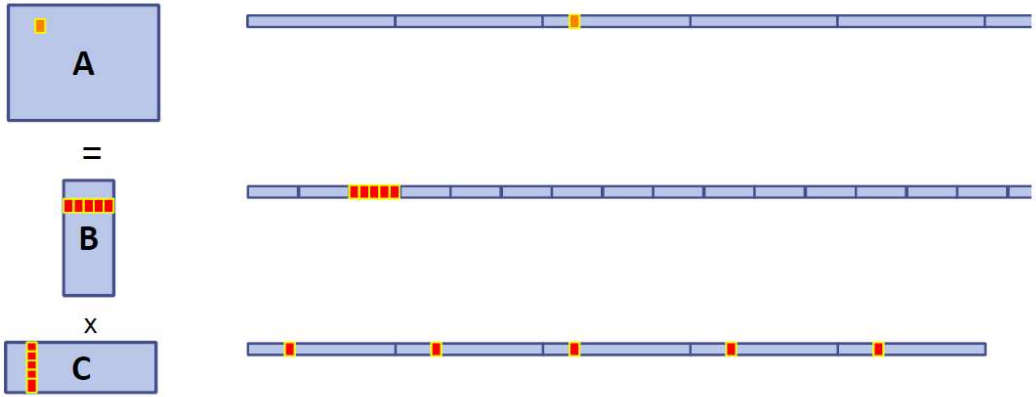
Standard CPU – Memory System (2)

- Let's consider multiplying the matrix: Multiplication of the matrix
- A = B x C
- Matrix B: Row Access
- Master C: Column Access

```
for(int i = 0; I < x; i++)
    for(int j = 0; j < y; j++)
        for(k=0; k < z; k++)
            A[i][j] += B[i][k] * C[k][j];
```

Standard CPU – Memory System (3)

- Sequential retrieves from cache memory are more efficient
- Example: x86 retrieval from cache memory reads 8 bytes (latency 1 cycle!!!)
- Access to matrix C does not match the organization of the cache
- Possible solution: Transpose matrix C, matrix multiplication loop will be much more effective
-



Memory system

```
#define READ(A, x, y, d) A[(x)*(d)+(y)]
    A = (double *)malloc(sizeof(double)*x*y);
    B = (double *)malloc(sizeof(double)*x*z);
    C = (double *)malloc(sizeof(double)*y*z);
    Cx = (double *)malloc(sizeof(double)*y*z);

    for(j =0; j < y; j++)
      for(k=0; k < z; k++)
        READ(Cx,j,k,z) = READ(C, k, j, y);

    for(i =0; i < x; i++)
      for(j =0; j < y; j++)
        for(k=0; k < z; k++)
          READ(A, i, j, y) += READ(B, i, k, z)*READ(Cx, j, k, z);
```

Acceleration approx. 3.5x

Standard CPU

- Performance-oriented libraries:
  - BLAS: Manually optimized library for basic vector and matrix operations (C/assembler)
    - http://www.netlib.org/blas/
  - Intel MKL: Math Kernel Library
    - Library of optimized mathematical functions for x86
    - BLAS, LAPACK, FFT, ..
  - Intel IPP: Integrated Performance Primitives
    - Optimized function library for signal processing, encoding, compression, multimedia
    - Optimized for individual architectures and their MM extension instructions (SSEx)

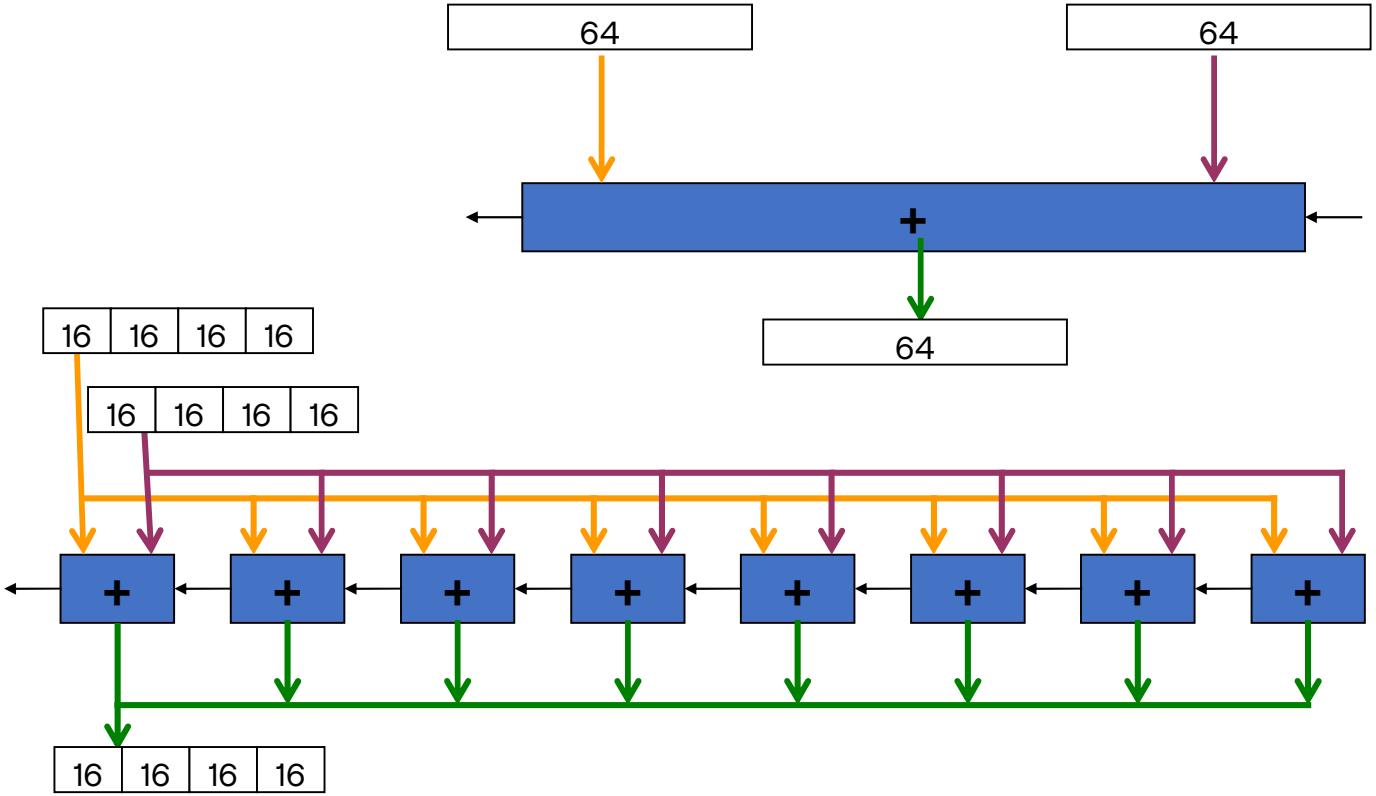
Standard CPU (2)

- For DSP operations (e.g. DCT), it is extremely useful if the processor has multiplication implemented in hardware and a related command
  - MUL
- Additional significant advantage if there is a multiplying command with addition
  - MLA
  - In examples of calculating DCT, DFT and similar we have most of the "butterfly" operations in which MLA is the basic link

SIMD Support

- SIMD (Single Instruction Multiple Data)
  - Cpu data path architecture that allows multiple data to be processed (less precision in principle) with a single command
  - Basic idea:
    - If we have e.g. 64 bit ALU then it is completely inefficient to process 8 bit data with it
    - Reorganise the ALU in such a way that the work on 8 bit data can be "divided" into multiple ALU

ALU – ordinary and SIMD



MM Extensions

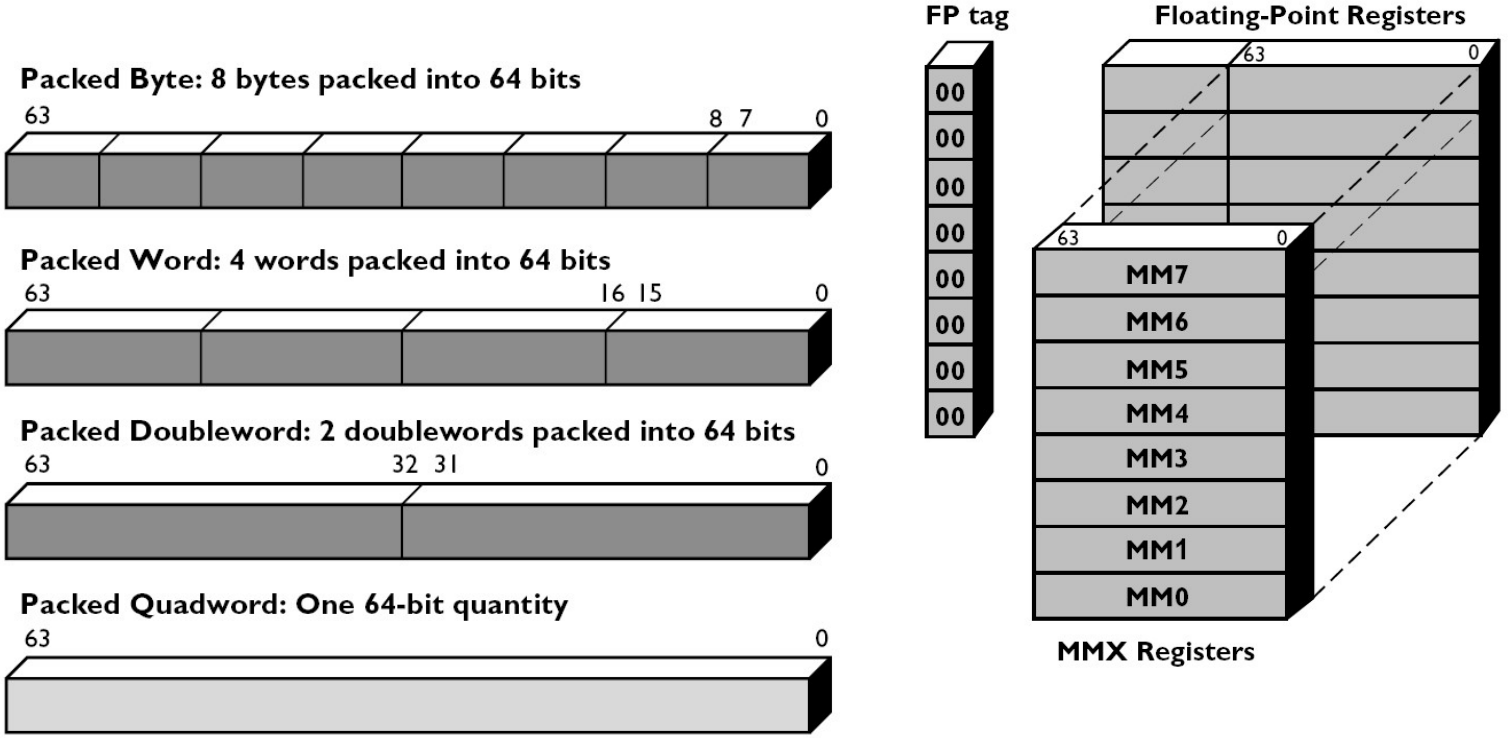
- This (basic) principle serves to significantly speed up the processing of multimedia algorithms

- Examples:
  - Old: VIS, PA-RISC
  - Not so old: MMX, 3DNow!
  - New: SSE4 (Streaming SIMD Extensions 4)

Example – MMX

- Put on the market in 1996 (Pentium with MMX and Pentium II)
- New data type introduced: Packed 64 bit
- Why 64 bits?
  - satisfactory acceleration
  - does not need major interventions on architecture

Example – MMX Data Types

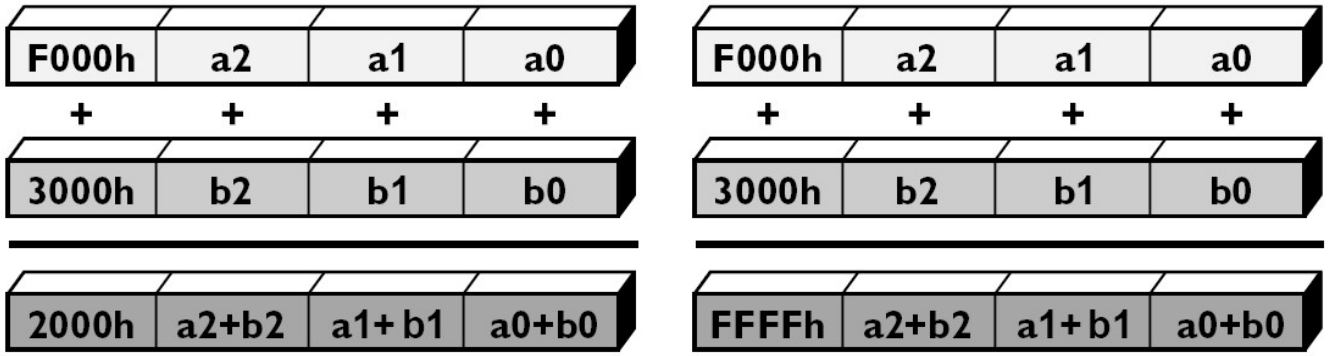


MMX instructions

- 57 new MMX instructions
- Arithmetic, logical, comparison, ...
- With saturation (signed and unsigned), no saturation
- All instructions except MOVE work on MMX registries
- All instructions work on 16 bit data and only some on 8 and 32 bit
- Disadvantage: Overlap with FP instructions

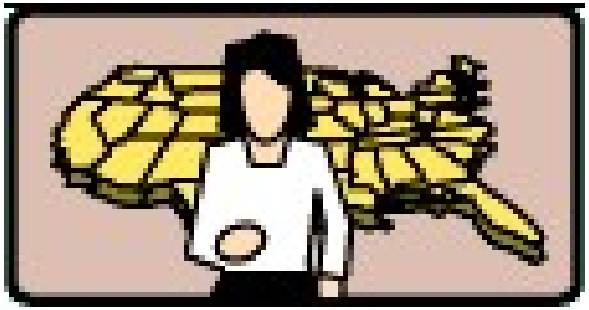
MMX instructions – Saturation

- Wrap-aroud or saturate
- Extremely useful with MM algorithms
-



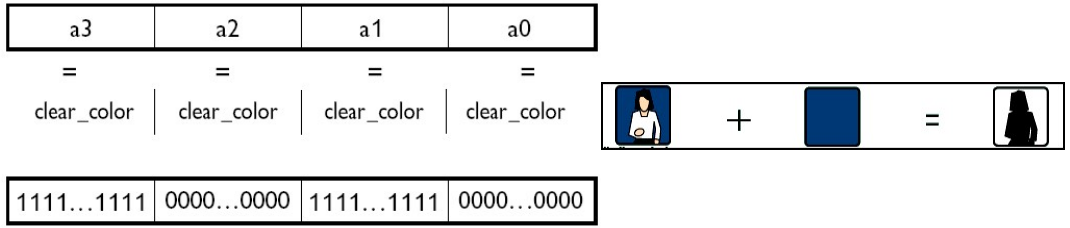
MMX instructions

- The standard function for TV applications is to overlap two images
- It is realized by putting a predefined background on the speaker that the computer replaces with another image
- By regularly calculating very slowly

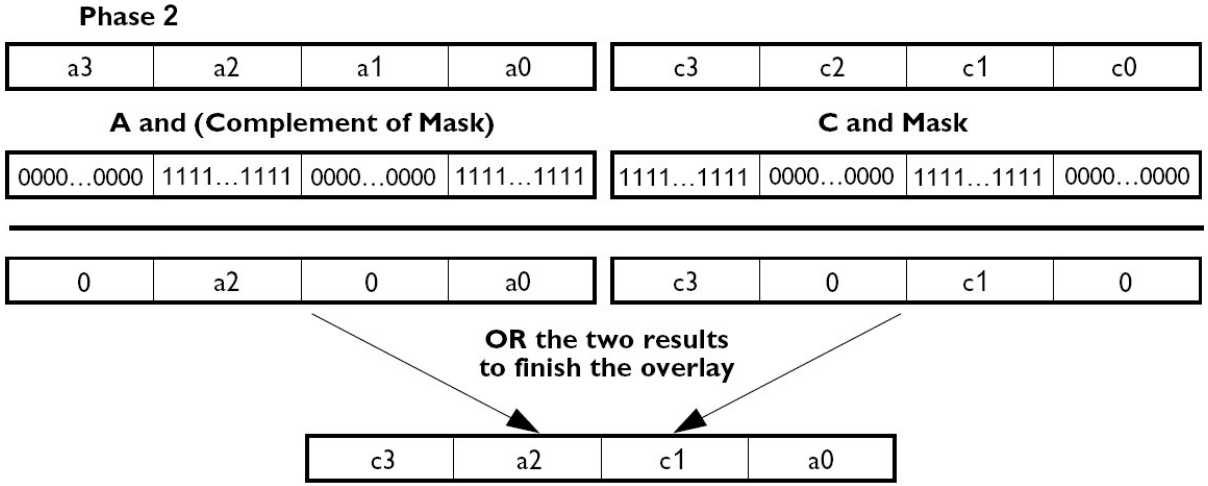


MMX – Picture Overlay

Phase 1



Phase 2



SSE – Streaming SIMD Extensions

- Presented in 1999 with Pentium III
- MMX's most important flaws fixed
- 8 brand new 128 bit registers XMM0..XMM7 (another 8, .. XMM15 in 64-bit)
- Enabled 128-bit (4x32bit) SIMD FP support:
  - arithmetic, comparison, shuffle, ...
  - Integer SIMD support over 64 bits
  - Over 60 new instructions

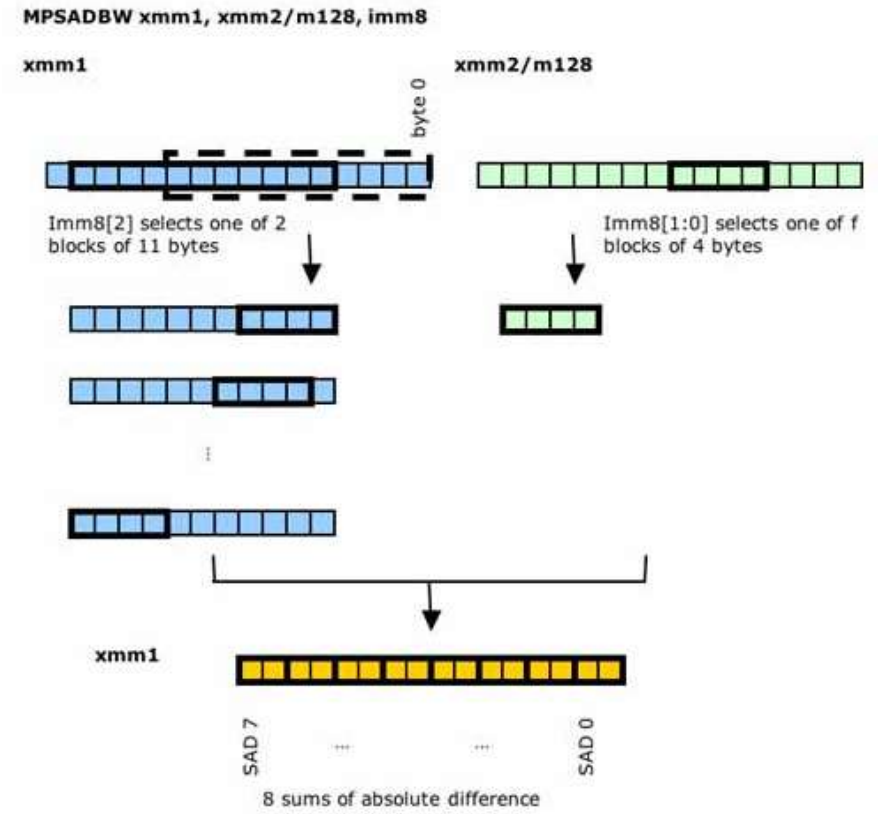
SSE4 part of a set of instructions ...

Instruction	Description
PEXTRB r32/m8, xmm, imm8	Extract Byte
PEXTRD r/m32, xmm, imm8	Extract Dword
PEXTRQ r/m64, xmm, imm8	Extract Qword
PEXTRW r/m16, xmm, imm8	Extract Word
PHMINPOSUW xmm1, xmm2/m128	Packed Horizontal Word Minimum
PINSRB xmm1, r32/m8, imm8	Insert Byte
PINSRD xmm1, r/m32, imm8	Insert Dword
PINSRQ xmm1, r/m64, imm8	Insert Qword
PMAXSB xmm1, xmm2/m128	Maximum of Packed Signed Byte Integers
PMAXSD xmm1, xmm2/m128	Maximum of Packed Signed Dword Integers
PMAXUD xmm1, xmm2/m128	Maximum of Packed Unsigned Dword Integers
PMAXUW xmm1, xmm2/m128	Maximum of Packed Unsigned Word Integers
PMINSB xmm1, xmm2/m128	Minimum of Packed Signed Byte Integers
PMINSD xmm1, xmm2/m128	Minimum of Packed Signed Dword Integers
PMINUD xmm1, xmm2/m128	Minimum of Packed Unsigned Dword Integers
PMINUW xmm1, xmm2/m128	Minimum of Packed Unsigned Word Integers
PMOVSXBD xmm1, xmm2/m32	Packed Move with Sign Extend - Byte to Dword
PMOVSXBQ xmm1, xmm2/m16	Packed Move with Sign Extend - Byte to Qword

Example: MPSADBW

- MPSADBW sums the absolute difference of 4 unsigned bytes, selected by bits [0:1] of the immediate byte (third operand), from the source (second operand) with sequential groups of 4 unsigned bytes in the destination operand. The first group of eight sequential groups of bytes from the destination operand (first operand) start at an offset determined by bit 2 of the immediate. The operation is repeated 8 times, each time using the same source input but selecting the next group of 4 bytes starting at the next higher byte in the destination. Each 16-bit sum is written to dest.

Example: MPSADBW (2)



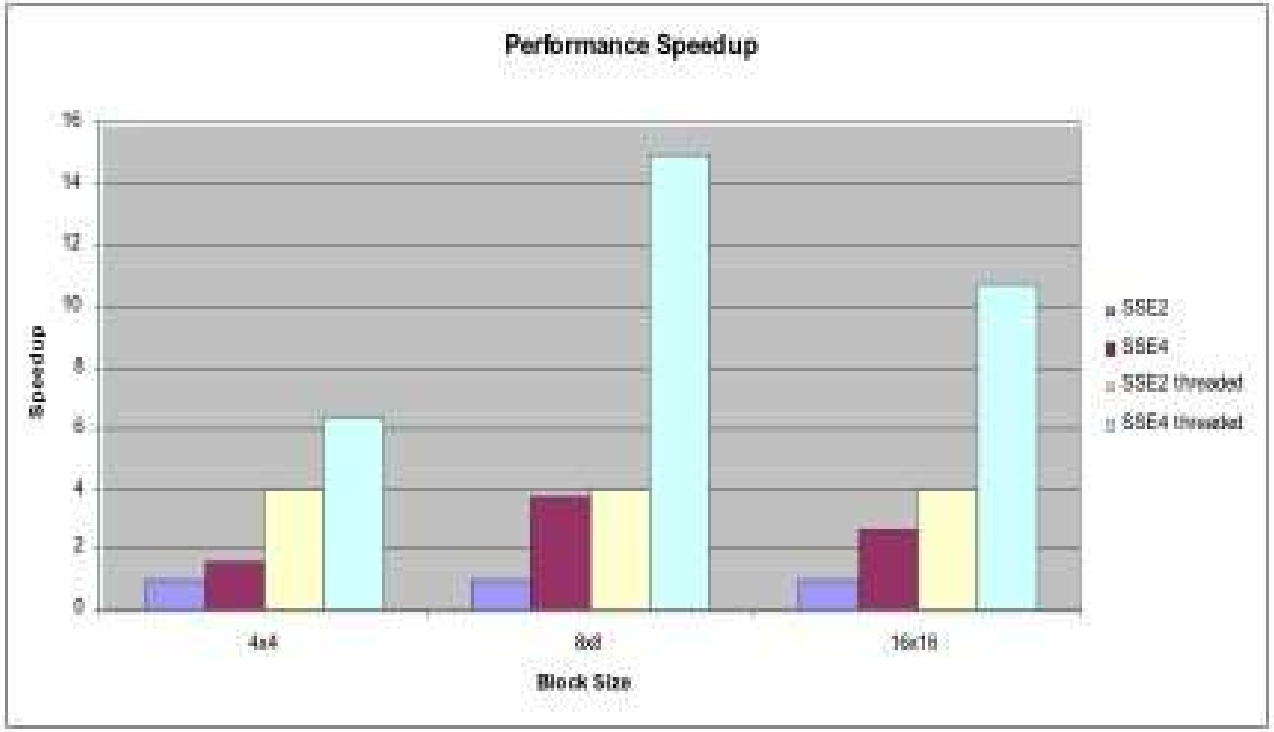
Optimization example (BlockMatch4x4)

- Unoptimized code
- SSE2 optimized
- SSE4 optimized

Example of acceleration

Code Sample	Cycles / Block SAD	Speedup
4x4 Block		
C++	54.84	1.0
SSE2	4.32	12.7
Intel SSE4	2.71	20.2
8x8 Block		
C++	180.55	1.0
SSE2	25.29	7.1
Intel SSE4	6.73	26.8
16x16 Block		
C++	173.01	1.0
SSE2	71.42	2.4
Intel SSE4	26.86	6.4

Acceleration



SIMD intrinsics

- Intel Intrinsics Guide
- Functions similar to C
  - Enable intel instructions to be performed
    - MMX, SSEx, AVXx, FMA, KNL, SVML
- Example

```
__m128d _mm_acos_pd (__m128d a)
#include "immintrin.h"
CPUID Flags: SSE
```

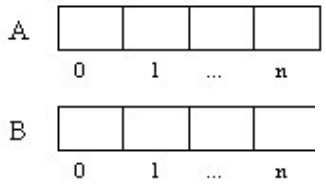
Compute the inverse cosine of packed double-precision (64-bit) floating-point elements in a expressed in radians, and store the results in dst.

SIMD intrinsics (2)

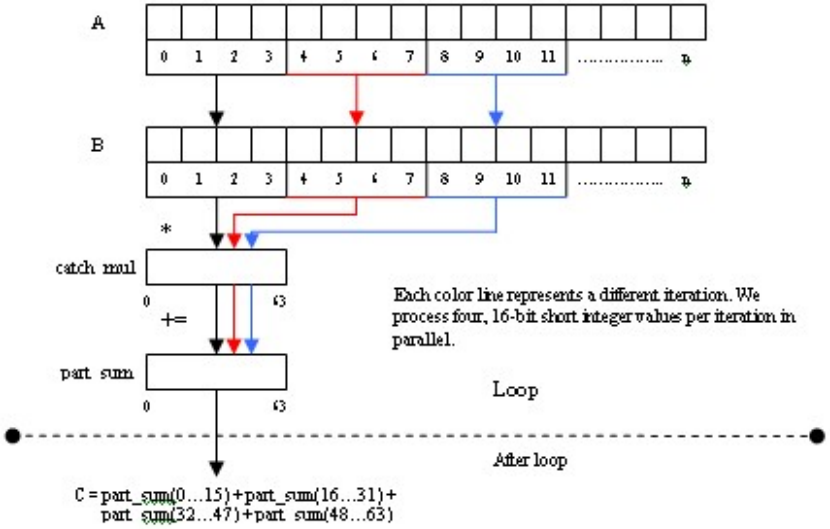
Technology	Header
MMX	mmintrin.h
SSE	xmmintrin.h
SSE2	emmintrin.h
Itanium	ia64intrin.h ia64regs.h

SIMD intrinsics (3)

$$\sum_{k=0}^{n} A[k] * B[k]$$



C = A[0] * B[0] + A[1] * B[1] + ... + A



SIMD conclusion

- SIMD extensions bring significant performance improvements
- Unfortunately, using SIMD requires tremendous knowledge and time but results in extremely efficient code
- New versions of SIMD lead to ever better results
- But SIMD also has its limits … What next?

Amdahl law

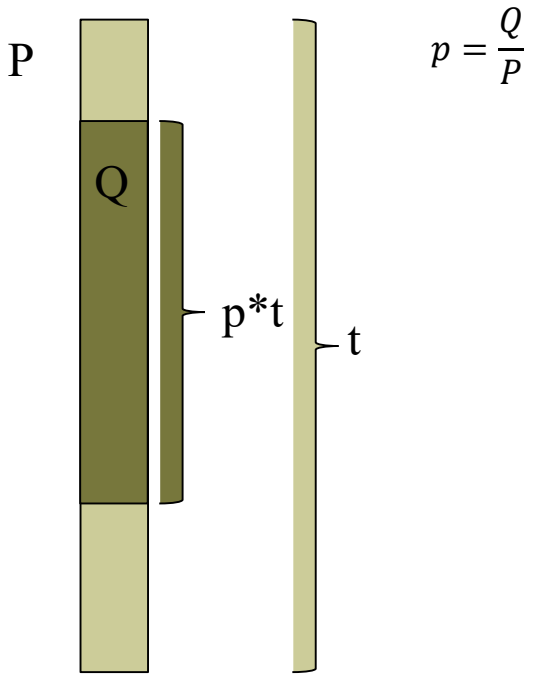
- Execution time of a program P is t
- Part of the program (marked with Q) with ratio p can be parallelized (speed-up) N times
- What is the speed-up of the whole program (U)?

Amdahl law:

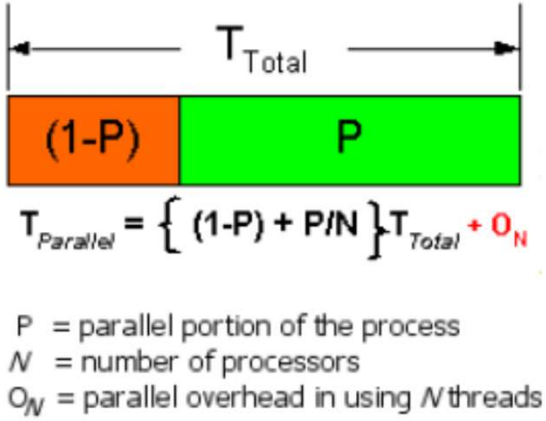
$$U = \frac{t}{t'} = \frac{t}{t\left((1-p) + \frac{p}{N}\right)} = \frac{1}{\left(1 - p + \frac{p}{N}\right)}$$

$$\text{If } N \longrightarrow \infty$$

$$U \approx \frac{1}{1-p}$$



Amdahl law (2)



- Speedup (in croatian Ubrzanje) $U(N) = T_{total}/T_{parallel}(N)$
  - Scalability $S = T_{total}/T_{parallel}(\infty)$
    - Assuming $N=\infty$ I $O_N=0$
  - Parallel efficiency $PE(N) = U(N)/N*100\%$