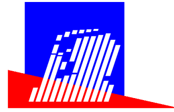
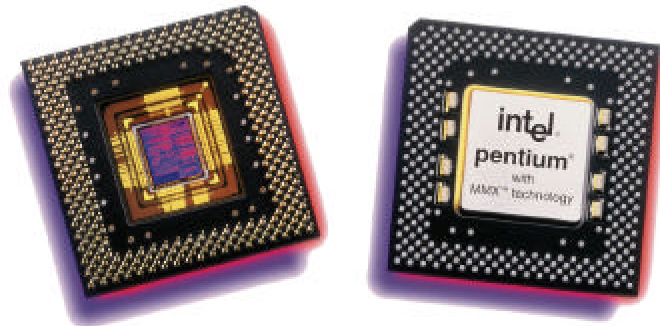


NOORDELIJKE HOGESCHOOL
LEEUWARDEN

# Using MMX Technology
## for image processing





Student: Andreas Wierda
Date: June 18, 1998
Company: Océ Technologies B.V.
P.O. box 101
5900 MA Venlo
Study: Noordelijke Hogeschool Leeuwarden
Engineering department
Technical Computer Science
Supervisors: ir. R.J.P. Rutten (Océ Technologies B.V.)
ir. G.J. van Woudenberg
drs. P.J.A. van Eijk.

## Preface

This report describes the results of my graduation project at the Research & Development (R&D) department of Océ Technologies B.V. in Venlo.

It is assumed that the reader is slightly familiar with computer architectures, assembly and C++ programming, but not with MMX Technology. Furthermore, it is assumed that the reader is not familiar with image processing.

During my period at Océ I have worked in pleasant way with a number of people. I specially would like to thank my supervisor, Rob Rutten. Furthermore I would like thank Marco Krom, Jan Jacobs and my fellow graduate students for the pleasant corporation. Especially Arjan v.d. Ven and Jurgen Rosendahl, who where researching familiar subjects. Finally, I would like to thank Klaas Jan Wierda for introducing me at Océ.

Venlo, June 18 1998,

Andreas Wierda

## Summary

This report describes the usage of Intel's MMX Technology in image processing applications. The primary goal is to provide the reader with knowledge on the possible problems when using MMX for image processing algorithms. Additionally an indication of the possible performance improvement is given for a number of algorithm types (compared to a C++ implementation).

Before discussing implementations of specific algorithms the basic concepts of digital copying are explained. After that the Pentium processor architecture is discussed. Instead of describing the hardware this discussion focuses on the consequences of hardware features for the programmer. The most important issues to take in account when writing code are the superscalar architecture and optimal usage of the memory bandwidth (caches). Finally, MMX Technology is discussed by explaining each of the MMX instruction types.

The following algorithms have been implemented:
- Smooth
- Sharpen
- Color conversion (lookup table with interpolation)
- Halftoning (error diffusion)

The discussion of the algorithm implementations starts with an introduction to each of the implemented algorithms. This is followed by a discussion of the main considerations made during the implementation of the algorithm in C++ and with MMX. The main considerations fall in the field of optimal image traversing and the way pixels can be processed in parallel. Next, the performance of the C++ and MMX versions are compared. Finally, an analysis of the processing time of the MMX version is made to determine the amount of processing time required for each part of the algorithm.

In the next chapter the impact of recent CPU developments on the way image processing algorithms can be implemented is discussed. Additionally the possibilities of a number of alternative CPU's are discussed (Intel Pentium II, AMD K6-2, Cyrix 6x86MX, IDT Winchip C6 and Motorola AltiVec).

In the last chapter it is concluded that the main problems encountered during the implementation of an algorithm are caused by shortcomings of the Pentium processors architecture. Additionally for some algorithms it is difficult to process pixels in parallel and very few MMX development tools are available. Using MMX results in a performance gain of a factor two to five, depending on the algorithm type.

*Océ Technologies B.V. enabled the author of this report to do the research that formed the basis for this report. Océ-Technologies B.V. is not liable for the correctness of the contents and the conclusions in this report. The author is solely responsible for the contents of this report.*

# Contents

**1 Introduction ........................................................................................ 7**
1.1 Background ........................................................................................ 7
1.2 Project definition ........................................................................................ 7
1.3 Plan of action ........................................................................................ 7

**2 Project settings ........................................................................................ 9**
2.1 Image processing for digital copiers ........................................................................................ 9
2.1.1 Digital copying ........................................................................................ 9
2.1.2 Color ........................................................................................ 9
2.2 The processing path ........................................................................................ 9
2.3 The test system ........................................................................................ 10
2.4 The Pentium Processor with MMX Technology ........................................................................................ 11
2.4.1 General features ........................................................................................ 11
2.4.2 The superscalar architecture ........................................................................................ 11
2.4.3 Instruction cycle times ........................................................................................ 14
2.4.4 Caches ........................................................................................ 14
2.4.5 MMX Technology ........................................................................................ 16
2.5 Performance measurement ........................................................................................ 19

**3 Algorithm implementation ........................................................................................ 21**
3.1 Introduction ........................................................................................ 21
3.2 RGB Color plane separation ........................................................................................ 21
3.2.1 The algorithm ........................................................................................ 21
3.2.2 The implementation ........................................................................................ 21
3.3 Adding borders ........................................................................................ 22
3.3.1 The algorithm ........................................................................................ 22
3.3.2 The implementation ........................................................................................ 22
3.4 The smooth algorithm ........................................................................................ 23
3.4.1 The algorithm ........................................................................................ 23
3.4.2 C++ implementation ........................................................................................ 23
3.4.3 MMX implementation ........................................................................................ 25
3.4.4 The basic loop structure ........................................................................................ 30
3.5 The sharpen algorithm ........................................................................................ 40
3.5.1 The algorithm ........................................................................................ 40
3.5.2 C++ implementation ........................................................................................ 40
3.5.3 MMX implementation ........................................................................................ 42
3.6 RGB to CMYK conversion ........................................................................................ 52
3.6.1 The algorithm ........................................................................................ 52
3.6.2 C++ implementation ........................................................................................ 53
3.6.3 MMX implementation ........................................................................................ 56
3.7 The halftoning algorithm ........................................................................................ 70
3.7.1 The algorithm ........................................................................................ 70
3.7.2 C++ implementation ........................................................................................ 74
3.7.3 MMX implementation ........................................................................................ 78
3.8 Bits to Bytes conversion ........................................................................................ 89
3.8.1 The algorithm ........................................................................................ 89
3.8.2 The implementation ........................................................................................ 89
3.9 Removing borders ........................................................................................ 90
3.9.1 The algorithm ........................................................................................ 90
3.9.2 The implementation ........................................................................................ 90
3.10 The total print processing path ........................................................................................ 90
3.10.1 ‘Warming up’ effects ........................................................................................ 90
3.10.2 Algorithm performance ........................................................................................ 91
3.10.3 Improvements of the processing path ........................................................................................ 92

**4 Architecture developments ........................................ 95**
4.1 The IA32 architecture ........................................ 95
4.1.1 Intel Pentium Pro ........................................ 95
4.1.2 Intel Pentium II ........................................ 96
4.1.3 AMD ........................................ 98
4.1.4 Cyrix ........................................ 98
4.1.5 IDT ........................................ 99
4.2 The PowerPC architecture ........................................ 99

**5 Evaluation & conclusion ........................................ 103**
5.1 Implementation issues ........................................ 103
5.2 Performance ........................................ 105
5.3 Recommendations ........................................ 106

**6 Literature ........................................ 107**

**Appendix A: MMX instruction set summary ........................................ 109**

# 1 Introduction

## 1.1 Background

Océ Technologies B.V. is one of the worlds leading producers of copiers, printers and plotters. In order to further expand Océ's market position, the Research & Development (R&D) department continuously improves existing products and develops new ones.

GRT

Within the R&D department, the Group Research and Technology (GRT) focuses on long term subjects. It investigates possibilities of new technologies for Océ and searches for emerging markets which might be interesting for Océ.

Image processing

The last few years the well known analog copiers are gradually being replaced by digital copiers, which achieve a much better image quality. This image quality is, among others, achieved by applying digital image processing algorithms. The hard- and software required for this image processing accounts for a considerable amount of the cost price. Therefore continuous research is done on how to do more and better image processing with less cost.

Since an image processing device, such as a digital copier, usually contains a personal computer (PC), a considerable cost reduction could be achieved when this PC is used for the required image processing. However, currently the image processing performance of PC's is not enough to satisfy the large demands made. According to Intel MMX Technology can improve the image processing performance of PC's significantly.

## 1.2 Project definition

Problems

Performance

The primary goal of this project is to investigate the problems encountered when implementing image processing algorithms with MMX Technology. Additionally, the possible performance gain when using MMX must be determined for a number of algorithm classes.

The following algorithms will be implemented in a demo program:

- Smoothing
- Sharpening
- Color conversion
- Halftoning

## 1.3 Plan of action

Before implementing any algorithms the Pentium architecture will be studied. Additionally general knowledge of digital copying and its specific properties has to be acquired.

In the main phase of the project the four algorithms will be implemented. The implementation of an algorithm starts with an investigation of the way the algorithm operates. Next, a reference C++ implementation of the algorithm will be built. This is done to understand the operation of the algorithm better before implementing the algorithm with MMX. After this an MMX version will be implemented. In this report the entire

development phase will be described to give the reader maximum insights in the problems encountered.
Finally the performance of the C++ and the MMX version of the algorithm will be compared. To gain insights in the weaknesses and bottlenecks of MMX an analysis of the processing time will be made.

During the final phase of the project a number of alternative architectures will be investigated, focusing on the differences with MMX.

# 2 Project settings

## 2.1 Image processing for digital copiers

### 2.1.1 Digital copying

A digital copier consists essentially of an expensive scanner, a computing-core for image processing and a fast print engine. Although desktop scanners and printers might seem to have ideal characteristics from the users point of view, real world scanners

Image processing

and printers are far from ideal and a lot of image processing has to be performed to obtain an acceptable image quality.

### 2.1.2 Color

Color can be represented using several methods. Each of these methods has its own advantages and is often specific for a certain application. The demo application uses two different color representations: RGB and CMYK.

#### 2.1.2.1 RGB

Human eye

RGB (Red, Green, Blue) is a well known color representation which is among others used for television and scanning applications. The RGB representation is based on the way the human eye perceives color. The inner back of the human eye is covered with a lot of small sensors, called 'cones'. There are four types of cones; one type is sensitive to all light, the other three to Red, Green and Blue light respectively. To represent a color in for example a scanner, the Red, Green and Blue color-bases are sufficient.
If all three color planes of a pixel are 255 (the upper limit), the pixel will be white. If all three planes are zero the pixel will be black.

#### 2.1.2.2 CMYK

Print engine

The RGB representation causes problems when printing, because on the white paper a white pixel should be represented with no ink. Therefore in print engines the complementary CMY (Cyan, Magenta, Yellow) color representation is used. A white pixel can then be represented with no ink, a black pixel with ink of all three colors. Because of the non-ideal characteristics of the ink however, the black pixel created this way has a somewhat brownish color. Since this is unacceptable for text, blacK is added, forming the CMYK color representation.

## 2.2 The processing path

There are many variations in processing-paths. The structure which depends on both the desired image quality and restrictions on the computational resources and architecture. Likewise, many variations exist for the individual algorithms.

In order to implement a wide range of algorithm classes, it was chosen to implement both a scan (processing) path and a print path. For algorithms such as the halftoning a number of different version of varying complexity exist. It was chosen to implement

one of the versions, based on which the problems expected when implementing the others are predicted.

Both the sequence of the algorithms as well as the specific choices of algorithms of both paths are typical for Océ.
Figure 1 shows the sequence of filter algorithms for both paths, where the right path represents the print path and the left the scan path. The italic algorithms are necessary to be able to show the processed images on a PC screen. In the processing path of a digital copier these algorithms are not necessary.

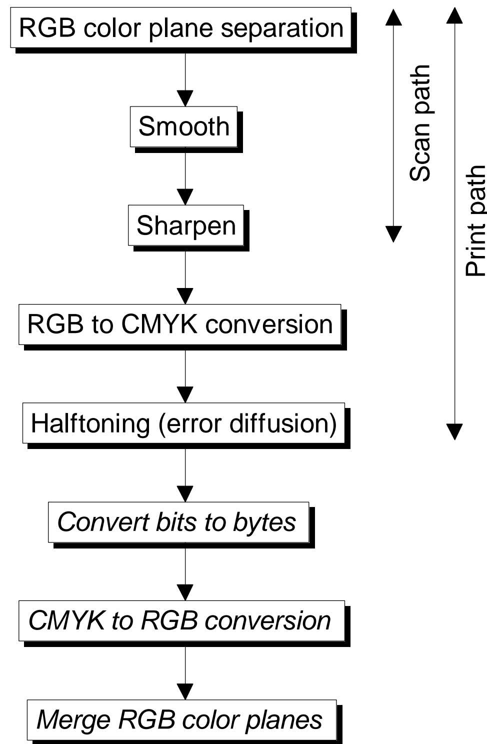



Figure 1: Processing path

Before implementing each of these algorithms a short description of them will be given, including example images.

## 2.3 The test system

The development platform was a Compaq Deskpro 4000 which was also used for the performance measurements. The relevant characteristics of this PC are:

Pentium 166 with MMX

- Pentium 166 MHz Processor with MMX Technology (See section 2.4).
- An internal memory of 64 MB.
- A 256 KB level two cache.
- 2.4 GB hard disk.
- Windows NT 4.0 Workstation (Service Pack 3).

Windows NT 4.0

The default parameters of Windows NT were not changed, unless mentioned otherwise. The measures taken to achieve a more constant performance are described in section 2.5.

## 2.4 The Pentium Processor with MMX Technology

### 2.4.1 General features

In 1997 Intel introduced the Pentium processor with MMX[1]. There were several changes that made the Pentium considerably faster than its predecessor, the 486:

- A dual pipeline architecture vs. a single pipeline on the 486. This so called superscalar architecture is described in section 2.4.2.
- MMX Technology[2] allows parallel data processing. MMX is described in section 2.4.5.
- The reduction of cycle times, especially of floating point instructions. See section 2.4.3 for more information on cycle times.
- A new branch prediction feature allows the processor to predict the destination of a branch, eliminating branch delays.
- A 64-bit databus, improving databus bandwidth (32 bit on the 486).
- 16 KB code- and 16KB datacache vs. 8K combined on the 486 (2x8 KB on the Pentium without MMX). The cache architecture and its influence on the performance is discussed in section 2.4.4.
- The branch prediction logic allows the processor to predict the destination of a branch, eliminating branch delays.
- Higher clock speeds further increased the CPU speed. On the 486 clock speeds varied from 25 MHz to 100 MHz. At its introduction the Pentium was available in a 60 and a 66 MHz version. Later versions reached clock speeds up to 233 MHz.

Some other features of the Pentium with MMX are:

- Dual processing configuration
- Fractional bus support, allowing the CPU to run at an higher clock speed than the external databus. The external databus of a Pentium CPU always runs at 66 MHz, while versions with internal clock speeds of 166, 200 and 233 MHz are available.

### 2.4.2 The superscalar architecture

On the predecessor of the 486, the 386 Intel introduced a pipelined architecture. Such an architecture refers to an architecture where the CPU executes each portion of an instruction in different stages. When a stage is completed, another one begins executing in the first stage, while the previous instruction moves to the second stage.

[1] There are a number of differences between the conventional Pentium and the Pentium with MMX. The most important one obviously is MMX. Additionally, the Pentium with MMX has an improved branch prediction and larger caches. The conventional Pentium is not discussed further.

[2] MMX is sometimes believed to stand for MultiMedia eXtensions or Matrix Math eXtensions. According to Intel however MMX is not an acronym.

Pipelining

A five stage pipeline might be divided into the following stages:

1. Fetch instruction opcode from memory
2. Decode instruction opcode[1]
3. Calculate operand[2] address and read operand from memory
4. Execute instruction
5. Write instruction result

If each stage takes 1 cycle this architecture might reach execution times of 1 cycle per instruction, compared to 5 cycles per instruction for the non-pipelined equivalent.

Pipeline delays

There are two situations where there is a significant delay in the pipeline.
First, if a jump is encountered, the entire pipeline has to be flushed and reloaded.
Second, if one of the instructions takes more than one cycle to read the operands, while the other stage still require one cycle, the preceding stages of the pipeline will stall, since they have to wait for the next stage to complete the previous instruction.

'u' and 'v'-pipe

The superscalar architecture introduced with the Pentium basically consists of two parallel pipelines, each divided into five stages. When running at maximum speed, two instruction results per clock cycle can be produced. The pipelines of the Pentium are called the 'u'-pipe and the 'v'-pipe.

Unfortunately only the 'u'-pipe can execute all instructions. The 'v'-pipe can only execute 'simple' integer instructions, floating point and MMX instructions. Simple integer instructions are for example move instructions, the various additions/subtractions and logical instructions.
Memory accesses can only be made from the 'u'-pipe. Another exception is made for read-write dependencies; if the first instruction changes an operand of the second instruction they cannot be executed parallel, and the second pipeline will stall until the first instruction finishes.

Pairing

When two instructions can be executed simultaneously, i.e. the first in the 'u'-pipe, and the second in the 'v'-pipe they are said to pair. A pairing rate of for example 50% indicates that half of the instructions can be paired with another instruction, thus reducing execution times by 25% (assuming that all instruction take the same number of cycles).
The complete list of paring rules is described in "Pentium Processor Family Developers Manual" [5], chapter two.
Examples and further information on paring issues can be found in "Pentium Processor Optimization Tools" [9], chapter nine.

Pairing a filter

After a filter has been programmed the code has to be paired. When pairing code the statements are re-ordered in such a way that that the CPU can execute two instructions parallel.
Intel provides the tuning program VTune. This program simulates the behaviour of a CPU, and logs the required processing time. VTune also indicates why two instructions will not pair.

[1] The hexadecimal code of the executed statement.
[2] The data element the operation is performed on or with.

**General pairing rules**

Although it is not necessary to learn the complete list of pairing rules, a few basic rules are still necessary:

1. If the first instruction modifies a register that is used by the second they will not pair.
2. Only one MMX shift unit is available. This means only one MMX shift instruction can be performed, either in the 'u'- or in the 'v'-pipe. The pack and unpack instructions also use the shift unit.
3. MMX memory moves can only be issued in the 'u'-pipe.
4. Add instructions do not pair with MMX memory moves.
5. When writing data to memory the data has to be ready one cycle in advance before the memory move.

These are a few of the paring rules for MMX instructions. To indicate the complexity of the total set of pairing rules; the complete list by Intel is 11 pages long.

It is important to keep the pairing in mind when programming the filter; use all registers instead of using one register multiple times for a number of tasks. This allows two (independent) blocks of code to be merged, running the first block in the 'u'-pipe, the second in the 'v'-pipe. Care also has to be taken not to use non-pairable instructions, especially for non-mmx instructions.

**Perfect pairing**

A pairing rate of 100% is possible by moving read and calculate operations from the beginning of the loop to the end of the loop[1]. This is necessary because when pairing the code, instructions that can be delayed are shifted down, and instructions that use recourses not used by other instructions are moved up. At the end of the loop however, it is not possible to move instructions down, commonly resulting in a number of not pairable instructions. If some instructions from the top of the loop are moved to the end, it is possible to pair them with the instructions that otherwise would not be pairable.

It was mentioned before to take care not to switch related statements. However, especially memory dependencies are easily overseen. It is therefore inevitable that mistakes will be made. If these mistakes are not noticed the consequences are disastrous; it is virtually impossible to find bugs in paired code because the logic between statements is completely lost[2].

[1] And of course just before the beginning of the loop. If this is skipped, the first loop cycle will not perform all the required operations.

[2] While pairing the smooth algorithm a mistake was made. The mistake was made in a group of about ten statements, but undoing the moves was not possible because Visual C++ had been shut down. After spending three hours searching for the misplaced statement. it was decide to start pairing all over again with a non-paired version of the filter.

Guidelines for safe pairing

In this paragraph some guidelines are given that help limit the consequences of errors made while pairing the code:

1. Back up previous versions.
2. Number each statement before starting to pair the code.
3. After pairing about ten to twenty statements, test the filter code.
4. Test the filter by subtracting the filters result byte-wise from the result of the unpaired code previously stored, for example with Paint Shop Pro. Errors can cause subtle changes in the image.
5. Do not close Visual C++ before the new version of the code passed a thorough test. If Visual C++ is closed, it is not possible to undo actions.
6. Do not use test images that contain large areas of the same color. Such images are sometimes processed correct, although the code is corrupted.

### 2.4.3 Instruction cycle times

One of the major improvements of the Pentium compared to the 486 is the reduction of instruction cycle times. For example a floating point add takes 23 to 72 cycles on a 386, 8 to 32 on a 486 and 1 to 3 on a Pentium. Besides the floating point instructions, quite a number of other instructions were also improved. The return instruction for example takes 11 cycles on a 386, 5 on a 486 and just 2 on a Pentium.

Memory accesses

The cycle times for instructions that read and write to memory are based upon the assumption that the data can be read (or written) from the level one cache. In most situations however, that will not be the case, and severe delays will occur. For more information on cache and memory issues see section 2.4.4.

### 2.4.4 Caches

Code and data cache

The first series of the Pentium had a separate code and data cache of each 8 KB, vs. 8 KB combined on the 486. There are some advantages in having the code and data cache separated. This separation generates fewer internal bus conflicts that could cause delays. But more important, this allows the code cache to contain additional information about each byte in the cache, for example instruction pairing information.

Cache lines

The later versions of the Pentium, those who where equipped with MMX Technology, have a separate code- and data cache of 16 KB each. The cache consists of 512 lines of 32 bytes each. Every time a data item is read which is not cached, the processor will read an entire cache line from memory. The cache lines are always aligned to a physical address divisible by 32. When a data element is spread over multiple cache lines, a read takes at least 3 clock cycles extra because the CPU can not read data spread over multiple cache lines. (The read is split up in two reads, after which the data is merged). This is called a Data Cache Unit (DCU) Split.

Misaligned data

Another delay of 3 clock cycles occurs when reading misaligned data. Intel recommends data to be aligned on the boundaries shown in the list below for optimal performance[1]:

[1] Source: ”Intel Architecture Optimization Manual “ [8], section 3.4.2.

- Bytes (8 bit): On any boundary.
- Words (16 bit): Align data to be contained within an aligned 4-byte doubleword.
- Doublewords (32 bit): On any boundary which is a multiple of four.
- Quadwords (64 bit): On any boundary which is a multiple of eight.

For example, a quadword read from address 0x02F01 is not aligned (light gray in Figure 2), and takes three clock cycles more to read than a quadword on address 0x02F08 (dark gray).

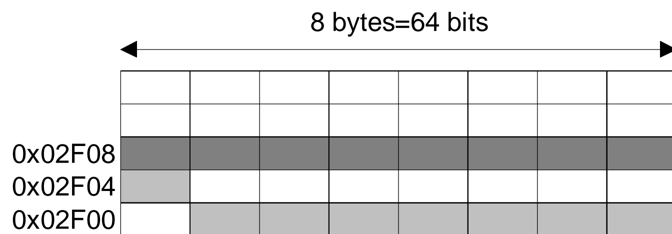


Figure 2: Misaligned vs. aligned memory accesses

Set associative

The cache is set-associative. This means that a cache line can not be assigned to arbitrary memory addresses. To determine the cache line and the position within a cache line, a physical address is split up like shown in Figure 3.

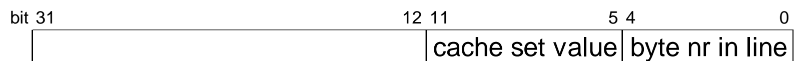


Figure 3: Mapping a physical address to a cache line

The seven bits for the cache set value are used to address 128 set values. The four least significant bits indicate the byte number of the physical address within the cache line.

For each of the 128 set values, four cache lines are available. Thus, a cache line address is composed as shown in Figure 4.

Figure 4: Cache line address

The seven cache set value bits are the same bits as in the physical address, allowing 128 set values. The two least significant bits allow four cache lines per set value.

The consequence of this is that the cache can hold no more than four different data blocks with the same set value (bits 5 to 11) of the address.

Level one cache

The cache previously described is called the level one cache, and is located on the same chip as the processor itself. The level one cache is very fast; data can be read or written in just one clock cycle. If data can not be read from the level one cache, it has to be read from the level two cache. The cache is located on the motherboard, and is much slower than the level one cache. A read from the level two cache takes 50 to 100 ns.

Level two cache

For a 166 MHz CPU (the one we use) this means a delay of 8 to 16 clock cycles.

Memory

If the data is not available in the level two cache, it has to be read from memory. This takes between 200 and 300 ns. For our 166 MHz CPU this would be 33 to 50 clock cycles.

When a write is performed to an address which is not in the level one cache, then the value will go right through to the level cache or to the RAM, depending upon the implementation of the motherboard. This takes approximately 100 ns, causing a delay of 17 cycles (on our 166 MHz system).
The Pentium with MMX has four write buffers. This means that up to four unfinished writes to uncached memory can be done without delaying the subsequent instructions.

Address generation

The last cache related issue we will discuss here is the so called Address Generation Interlock (AGI). This delay is caused by the fact that it takes one clock cycle to calculate the address needed by an instruction which accesses memory. Normally, this calculation is done in a separate stage of the pipeline, while the preceding instruction is executing. But if the address depends on the result of an instruction executing in the preceding clock cycle, an extra cycle is needed in order to calculate the address. This is called an AGI stall.

### 2.4.5 MMX Technology

In 1996 Intel introduced MMX Technology. Basically MMX provides a set of SIMD (Single Instruction Multiple Data) instructions, allowing processing of multiple data elements in one instruction.

Data types

MMX Technology introduces 57 new instructions, 4 new data types and 8 MMX registers. The basic data type is the packed fixed point integer, which can be used in the following configurations:

- Packed byte. Eight bytes combined in a 64 bit quantity.
- Packed word. Four words packed in a 64 bit quantity.
- Packed doubleword.
- Quadword. One 64 bit quantity.

The first three data types are called packed data. The MMX instructions operate on groups of eight bytes, four words two doublewords or one quadword, as shown in Figure 5.

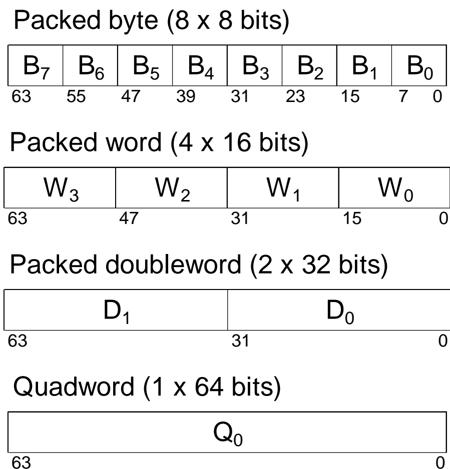



Figure 5: MMX data types

### MMX Registers

The MMX registers MM0 to MM7 are aliased on the general purpose floating point registers R0 to R7 and are used in the same way.

### Saturation arithmetic

MMX Technology introduces a new arithmetic mode known as saturation arithmetic. Saturation can best be explained by contrasting it with wraparound mode. In wraparound mode results that over- or underflow are truncated and only the least significant bits are returned. In saturation mode the results are clipped to the data range limits shown in Table 1.

| | **Lower limit** | | **Upper limit** | |
|---|---|---|---|---|
| | **(hex)** | **(decimal)** | **(hex)** | **(decimal)** |
| Signed byte | 0x80 | -128 | 0x7F | 127 |
| Signed word | 0x800 | -32,768 | 0x7FFF | 32,767 |
| Unsigned byte | 0x00 | 0 | 0xFF | 255 |
| Unsigned word | 0x0000 | 0 | 0xFFFF | 65,536 |

Table 1: Data type ranges

To perform actions on packed data types, Intel provides the following types of instructions:

- Arithmetic instructions
- Comparison instructions
- Conversion instructions
- Logical instructions
- Shift instructions
- Data transfer instructions
- The “Empty-MMX-State” instruction (EMMS)

Since limitations of the MMX instruction set have serious consequences on the way algorithms must be implemented, we will discuss some details of the MMX instruction set here. For a complete list of all MMX instructions, see appendix A.

### Arithmetic instructions

MMX provides most common arithmetic instructions, such as addition, subtraction and multiplication. The latter can only be performed on packed words, the other on packed words and packed bytes. The packed add, subtract and the packed multiply and add can also be performed on packed doublewords. However, none of the arithmetic instructions operates on a

quadword. A packed divide instruction is not available. Division can be made either with the shift-right instruction (divide by a power of 2) or by multiplying with the reciprocal.

Comparison instructions

Since a packed data type consists of multiple data elements, a compare instruction could not be implemented with a single zero flag as is common for normal compare instructions. Instead, the compare instruction generates a bit mask as shown[1] in Figure 6.

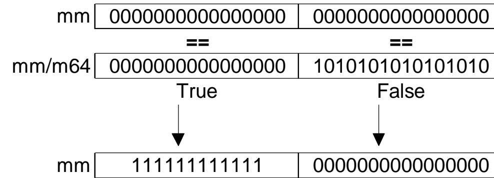



Figure 6: Packed compare (PCMPEQD MM, MM/M64)

Conversion instructions

The conversion instructions are used to convert packed data from one type to another. To perform an extend operation, for example to convert packed bytes to words, the packed unpack instruction must be used. If in the example in Figure 7 the mm/m32 operand is filled with zeroes, the mm operand is extended from the packed byte data type to packed words.

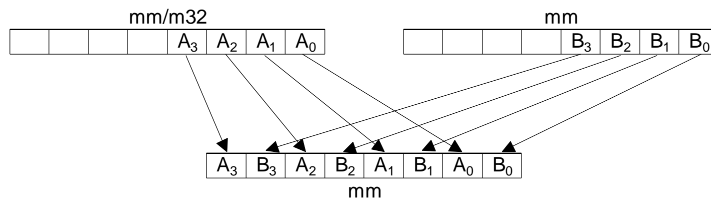



Figure 7: Unpack operation (punpcklbw mm, mm/m32)

The complement of the unpack is the pack instruction (see Figure 8). This instruction can be used to shrink packed data types, for example from packed words to bytes.

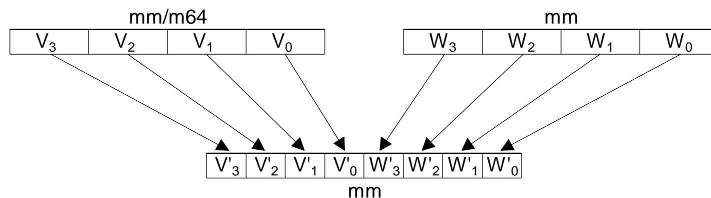



Figure 8: Pack operation (PACKUSWB MM,MM/M64)

PACKUSWB stands for “Pack Unsigned with Saturation Words to Bytes”. Thus, when the value of a unsigned packed word exceeds the range of the unsigned packed byte, the result is saturated.

Logical instructions

The packed logical instructions are similar to those that operate on non-packed data types, except that the result is placed in a bit mask in the first (=destination) operand, similar to the packed compare instructions.

[1] mm = MMX register.
m32 = Memory location (32 bits).
m64 = Memory location (64 bits).

**Shift instructions**

The MMX instruction set has three shift instructions; logical shift left and right and arithmetic shift right. The arithmetic shift right instruction is needed to divide -by a power of 2-. The difference between the arithmetic shift and the logical shift is that the latter also shifts the sign bit, unlike the arithmetic shift which skips this bit. The arithmetic shift left can be done with a logical shift left combined with some code to preserve the sign bit.

**Data transfer instructions**

Data transfers can performed with the move instructions. When using these instructions to move packed data between registers an exact copy of the destination register is made.
Care has to be taken when moving packed data from memory to an MMX register; the byte on the memory location with the highest address is placed in the most significant byte of the packed data type, as show in Figure 9.

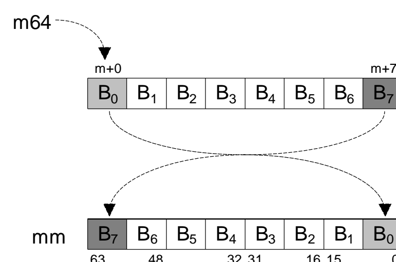



Figure 9: MMX memory move operation (MOVQ MM,M64)

**Immediate operands**

Most instructions do not accept immediate operands, for example to add ‘1’ to all of the eight packed bytes. They only accept MMX registers or memory operands. An exception is made for the shift instructions, which accept an immediate operand for the number of bits to shift.

**‘EMMS’ instruction**

A special MMX instruction is the EMMS instruction. This instruction clears the internal state registers of the MMX unit. The EMMS instruction is necessary because the MMX registers are mapped to the floating point registers. The next floating point instruction after the EMMS instruction takes 57 cycles extra to execute, thus mixing floating point and MMX instructions must be avoided at all costs.

**Cycles times**

As far as the cycle times of regular MMX instructions are concerned; they all take one cycle, except for instructions that use the MMX multiply unit[1]. The results of these instructions are available three cycles after the instruction has been started. An average throughput of one instruction per cycle can be reached by starting a number of result-independent multiplies consecutively.

## 2.5 Performance measurement

The time necessary to apply each filter is measured by reading the time stamp counter register of the Pentium CPU before and after the call to the filter function. The time stamp counter register is a 64 bits wide register that is incremented one each clock cycle.

[1] The multiply unit is used by the packed multiply and the packed multiply-and-add instructions.

Cycles/pixel

The time needed to process one pixel is calculated by dividing the number of clock cycles that passed by the number of pixels processed.

Multitasking

Windows NT is a multitasking operating system. This implies that an applications performance also depends the other processes running. Even if all other applications are shut down, in the background Windows NT still runs internal processes, called services, such as the spooler service and the TCP/IP service. To reach constant performance of the filter, all services possible where shut down. Two services can not be shut down:

1. The eventlog service .
2. The RPC service.

It is expected that the influence of these services on the filter applications performance will be minimal, because they are just two of the nine services running normally. Besides shutting down most services, the test application has been given maximum scheduling priority[1].

Disk swap

Windows NT uses, like all modern operating systems, virtual addressing. Virtual addressing allows the usage of more memory than physical available in the computer. This is done by mapping the virtual address either to a physical memory address (the data is loaded into the physical memory), or to a location on the hard disk. In the latter situation the data is said to be ‘swapped’ to the disk. Windows NT manages address mapping in such a way that a data block requested by an application is loaded in the physical memory. This so called ‘disk swap’ dramatically influences the filter applications performance. By closing all other application, most of the disk swapping can be prevented. However, if the filter application allocates more memory than physically available in the PC, Windows NT will have to swap data to the hard disk.

Allocating memory

Therefore the filter application has to allocate the required memory in such a way that no swapping will occur in critical sections:

1. Minimise the amount of allocated memory.
2. Allocate memory outside the (timed) filter functions.

Further research is required on how to achieve constant processing performance on a Windows NT platform.

Fastest reproducible run

When measuring filter performance a number of measurements where performed, out of which the fastest that could be reproduced was documented. It was chosen to record the fastest reproducible measurement instead of the average because the slower measurements are heavily influenced by the time Windows NT needs to swap data to the hard disk[2].

---

[1] This is done with the Windows NT Task Manager.

2 A disk access requires several milliseconds, while a CPU clock cycle is about 6 nanoseconds (based on a clock speed of 166 MHz).

# 3 Algorithm implementation

## 3.1 Introduction

The application framework needed to test the algorithms is built using the MFC-Application wizard provided with Microsoft Visual C++ 5.0. The wizard generates a customised application template, containing for example menu's and toolbars. Reading and writing of images has been implemented using the Image Gear function library. The image is read by Image Gear from the harddisk into memory as a linear sequence of pixels. Each pixel is represented by 24 bits. For each of the Red, Green and Blue color planes 8 bits are used, giving a Red, Green and Blue range from 0 to 255. Within each pixel the planes are placed in Blue - Green - Red order.

Image Gear

In the test application two versions of the processing paths are implemented; one written in standard C++, another using in-line assembler with MMX Technology. It has been decided to implement the filters in C++ for two reasons:

C++ vs. MMX

1. To find out each filters typical difficulties.
2. To be able to compare the performance of the MMX implementation to a conventional C++ implementation.

A test session is started by choosing either the "C processing path" or the "MMX processing path" option from the image menu. Then the required main processing function ('FilterC' or 'FilterMmx') is called. These two functions are the base of the processing path. They allocate all the required storage space and call the specific filter functions. The required processing time
is measured by reading the time stamp counter register before and after the call to the filter function (see section 2.5).

Time stamp counter

When the users starts a test session, a dialog box is displayed in which the user can select the filters to execute. When the image processing is completed, another dialog box is displayed containing the number of cycles per pixel and the number of milliseconds for each of the applied filters.

## 3.2 RGB Color plane separation

### 3.2.1 The algorithm

The filters in the processing path (smooth, sharpen) will be applied to each of the individual color planes. In the image however, the pixels are stored in a 3x8 bits block, where the first eight bits represent the Blue value, the middle eight Green and the last eight Red.
For an easier and faster[1] implementation of the filters, a function is built that separates the three color planes, as well as its complement, a function that merges the three planes.

### 3.2.2 The implementation

The implementation of the separate function is easy. A loop runs through the array containing the image and puts of every group

[1] Faster because memory reads are less cluttered up.

of three bytes the first byte in the Blue color plane array, the second into the Green, and the third into Red color plane array.

C++ version only

This function, which is written in C++, is used for the C++ as well as for the MMX implementation of the processing path. However, it is expected that the use of MMX technology will lead to a performance improvement because an MMX implementation can separate multiple color pixels parallel, where the C++ implementation processes the pixels sequentially.

## 3.3 Adding borders

### 3.3.1 The algorithm

Smooth and sharpen

In section 3.4.3.5 and 3.5.3.5 it can be read that both the smooth and the sharpen filter require an image width which is a multiple of eight. Since not all images have such a width, the image width has to be extended.

On page 81 it can be seen that the MMX implementation of the halftoning algorithm does not process all the border pixels. Both on the left and the right a seven pixels wide column is not completely processed. Therefore it has been decided to add two, eight pixel wide borders to the left and right edge of the image.

Separated color planes

The border adding function can be applied to the merged color planes or to the separated color planes. Since the function that separates the color planes does not require an image width which is a multiple of eight, it has been chosen to apply the border adding function to the three separated color planes.

### 3.3.2 The implementation

The implementation of the border adding function is not difficult. Two loops are used to traverse the image; one to process the image horizontally (the rows), another to process all the rows.

The inner loop, which processes one row, simply copies all the pixels from the source array to the destination array. After all the pixels from one row of the source image have been copied, a number of pixels is added to reach an image width of 8n.
The two eight pixel wide borders which have to be added for the halftoning algorithm are copied before respectively after the row is copied.

MMX version only

It has been chosen to implement this function with MMX Technology for two reasons:

1. By using MMX eight pixels can be copied simultaneously, compared to one of a C++ implementation.
2. This function is only used by the MMX version of the processing path.

Since the implementation of this function is straightforward, implementation details will not be discussed here.

## 3.4 The smooth algorithm

### 3.4.1 The algorithm

The smooth algorithm is used to suppress disturbances added to a scanned image because of the non-ideal characteristics of scanners. A scanner always adds some electronic noise to the scanned image, and scanning of regular patterns, such as color photos, can result in moiré like effects. A smoothing operation is performed on the separated Red, Green and Blue planes of the image to reduce these artefacts.

In Figure 10 the smooth filter is applied to the squared area.

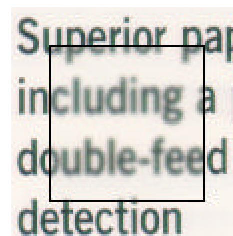



Figure 10: Effect of a smooth operation

Weighted mean

The smoothing operation is a low-pass filter. Its calculated by replacing every pixel by a weighted mean of the pixels in the surrounding. All calculations are performed on the original pixels; the calculated values are written to a destination array. A much used version of the smooth filter uses the 3x3 area printed in Figure 11 to calculate a pixel. The pixel for which the new value is calculated is the pixel in the middle (the one weighted by 4/16). This is called the centerpixel.

| 1/16 | 2/16 | 1/16 |
|---|---|---|
| 2/16 | 4/16 | 2/16 |
| 1/16 | 2/16 | 1/16 |

Figure 11: Smoothing kernel

Kernel operation

This kind of operation, where every pixel is replaced by a linear combination of pixel in its neighbourhood is called a kernel-operation.

### 3.4.2 C++ implementation

#### 3.4.2.1 Traversing the image array

When implementing the smooth filter in C++ the most important question is how to process the source image array (horizontal, vertical or perhaps diagonal).

Cache line

If a single byte (=one pixel) is read from memory, the Pentium processor reads an entire cache line of 32 bytes into the level one cache. When the image would be divided in a number of one byte wide columns, it would be possible to run vertically through the image. However, since each block of 2048 bytes has the same cache set value[1] and there are four cache lines available for each of the set values, a point will be reached at

[1] There are 128 cache lines of 32 bytes each. See section 2.4.4.

which the bytes previously read will be overwritten[1]. This means that the bytes needed for the next one byte column have to be read again from the slow main-memory.
When traversing the image array diagonally, of the 32 bytes read into the level one cache, the same (small) number of bytes would be used as when we would run vertically through the array. Because at a certain point cache lines still needed are overwritten, performance will be far from optimal.

Horizontal traversing

Because traversing image array horizontally results in the most efficient use of the level one cache, this way has been chosen. The direction -from left to right or vice versa- has no influence on the performance, so it has been decided to run from left to right, since the bytes are stored in memory from left to right.

#### 3.4.2.2 Re-using calculated values

Calculation of the new pixel value could be done in one single calculation where all the pixel values are added (with their own weight) and the sum is divided by 16.

Three kernel columns

Such an implementation will result in poor performance because the kernel symmetry enables the calculation of row or column (weighted) sums. Because we run from left to right through the image, it is possible to calculate the weighted sum of each of the three kernel columns, add them and divide by 16.

By storing the sum of the rightmost column in a buffer the number of uncached memory accesses can be reduced by 66%[2]. In the next loop cycle the previously buffered result can be multiplied by two to calculate the weighted sum of the middle column. One loop cycle later the value stored two cycles ago can be used as the weighted sum of the left column of the kernel.
The example in Figure 12 shows how the same column-result is calculated three times when calculating three new pixel values. Suppose at a certain moment the memory looks like shown in Figure 12. Pixel '5' is currently processed.

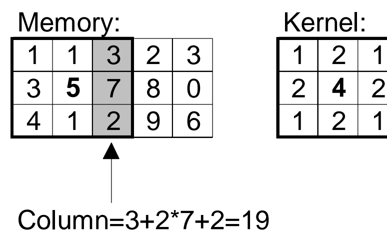



Figure 12: Calculating the right kernel column

At the next iteration of the loop (when calculating the new value for the '7') the same column is used.

---

[1] Since there are a source and destination array this will occur approximately after 2*2048=4096 byes have been read.
[2] Assuming that the buffer is loaded into the level one cache, or placed in registers.

| 1 | 1 | 3 | 2 | 3 |
|---|---|---|---|---|
| 3 | 5 | **7** | 8 | 0 |
| 4 | 1 | 2 | 9 | 6 |

Column=2*3+4*7+2*2=38=2*19

Figure 13: Calculating the middle kernel column

The third time, when calculating the new value for the ‘8’, it looks like shown in Figure 14.

| 1 | 1 | 3 | 2 | 3 |
|---|---|---|---|---|
| 3 | 5 | 7 | **8** | 0 |
| 4 | 1 | 2 | 9 | 6 |

Column=3+2*7+2=19

Figure 14: Calculating the left kernel column

#### 3.4.2.3 The final C++ implementation

With this buffering implemented the filter loop final C++ implementation looks like shown in the pseudocode in Code example 1. The buffer consists of two storage spaces, A and B. The function Weighted_Sum calculates the (1, 2, 1) weighted sum of the first three bytes of a column, each with the appropriate weight.

```
Dst.Rows[0] = Src.Rows[0]        // Copy top and bottom row
Dst.Rows[Height] = Src.Rows[Height]
RowCounter = 1
Do While (RowCounter < Height-1)
       {
         Dst.Pixels[RowCounter,0] = Src.Pixels[RowCounter,0]
         A = Weighted_Sum(Src.Columns[0])
         B = Weighted_Sum(Src.Columns[1])
         ColCounter = 2
         Do While(ColCounter < Width-1)
              {
               Temp=Weighted_Sum(Src.Columns[ColCounter])
               Dst.Pixels[RowCounter,ColCounter-1]=(A+2*B+Temp )/16
               A = B
               B = Temp
               ColCounter = ColCounter+1
              }
       Dst.Pixels[RowCounter,Width]=Src.Pixels[RowCounter,Width]
       RowCounter = RowCounter+1
       }
```

Code example 1: C++ implementation of the smooth

Performance

The performance of the code is discussed in section 3.4.4.7, where it is compared to that of the MMX implementation.

### 3.4.3 MMX implementation

#### 3.4.3.1 Parallelability of the algorithm

When implementing a filter with MMX Technology, the first question to ask is:

“Is it possible to process pixels simultaneously ?”

If not, using MMX will not improve performance because MMX's strength is the fact that it can perform an operation on multiple pixels simultaneously.
The smooth filter applies a kernel to a number of pixels of an image, resulting in the new pixel value. Since the output of the filter has no influence on the input, parallel processing of pixels is possible without any limitations.

Unlimited parallelability

#### 3.4.3.2 Pixel block width

Next it must be decided how many pixels can be processed simultaneously. The answer to this question is basically provided by the way the data to calculate one pixel is loaded into the MMX registers. To understand this, it must be clear how the pixels are
stored in memory; the pixels of each row are written sequentially from left to right. The first (=leftmost) pixel of the next row is stored in memory right after the last (=rightmost) pixel of the row last written. Thus, the pixels of the image shown in Figure 15 are written in the same order as they are numbered; pixel '1' is stored on the lowest address in memory, pixel '80' on the highest. Pixel '21' is stored one location higher than pixel '20'. As a result of this, all the pixels of the top row are placed in adjacent memory locations, and can be read with one MMX memory read.

Pixels written sequentially

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 |

Figure 15: Reading the source pixels

Thus when calculating the new value for pixel 22 the pixels of the top kernel-row can be read by reading pixel 1 with an MMX memory read. However, the MMX memory read loads eight pixels (=bytes) from memory. This enables the calculation of six pixels, 22 to 27[1], shown in the squared area. It is obvious that this way of implementing the filter uses the memory bandwidth more efficient than an implementation that calculates only one pixel per loop cycle. But how many pixels should be processed each loop
cycle ?

The answer to this question is provided by the number of data-elements calculated in parallel. When processing packed bytes eight bytes are processed parallel[2]. As far as the other data types are concerned; four packed words or two packed double-words can be processed parallel.
These considerations lead to the conclusion that it would be very inefficient to process six pixels per loop cycle; such an implementation only uses a part of the available processing power.

Since the pixels are stored in memory as bytes it is obvious that packed bytes are used for pixel calculations. This implies that each loop cycle eight new pixels are calculated.

Eight pixels per loop cycle

[1] The kernel requires two extra -border- pixels.
[2] Because a packed byte contains eight bytes.

### 3.4.3.3 Re-usage of calculated values

The choice made in section 3.4.3.2 does not determine how to run through the image array. Although the example runs through the image array vertically, it would also be possible to run horizontally, as with the C++ implementation. The third possibility is to run through the image diagonal.

Let us suppose the filter runs horizontally through the image array. Data would then be processed as show in Figure 16; in the first loop cycle the pixels 22-29 are calculated requiring the grayed pixels of the source image. In the next loop cycle the pixels 30-37 are calculated, requiring the squared area of the source image.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |

Figure 16: Traversing the image array horizontally

Thus, when running horizontally through the image, only the weighted sum of two columns can be re-used to calculate the next group of pixels. Now suppose the loop runs through the array vertically. The second cycle of the loop calculates the new values for pixels 42- 49, requiring the squared area of the source array in Figure 17.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 |

Figure 17: Traversing the image array vertically

In this situation, the weighted sum of the row 21-30 and of the row 41-50 can be re-used. The re-usage of two previously calculated results is possible because the kernel is vertically symmetric and because the middle row is a multiple of the bottom row, as shown in Figure 18.

| 1 | 2 | 1 |
|---|---|---|
| 2 | 4 | 2 |
| 1 | 2 | 1 |

Figure 18: Smoothing kernel

The third option was to run through the image diagonal. Such an implementation would result in poor performance because calculated values could not be re-used efficiently.

Vertical traversing

From the previous it is clear that optimal usage of previously calculated weighted sums can only be achieved when running vertically through the image array.

The values to be re-used have to be stored somewhere. If registers would be used, this would require four[1] of the eight

[1] Four instead of two because the data has to be extended from bytes to words. The reasons it has been decided to extend data from bytes to words are discussed later.

available MMX registers, leaving not enough registers free to do the calculations. Thus, these values are stored in an array. Because the array is accessed frequently, the data will always be in the level one cache, allowing data to be read in one cycle.

#### 3.4.3.4 Optimal image traversing

From the previous paragraph it is clear that as far as the re-usage of calculated values is concerned the most efficient way of processing the image would be by running vertically through the image. But in the discussion of the C++ implementation[1] it was concluded that from the caches point of view, an optimal implementation would run horizontally through the image.

Cache lines filled

An implementation that combines these two demands would run vertically through the image to enable optimal re-usage of calculated values, but would not overwrite cache lines still needed. This can be achieved by running down until all the cache lines are filled, instead of running down until the bottom of the image. This way of running through the image is shown in Figure 19.

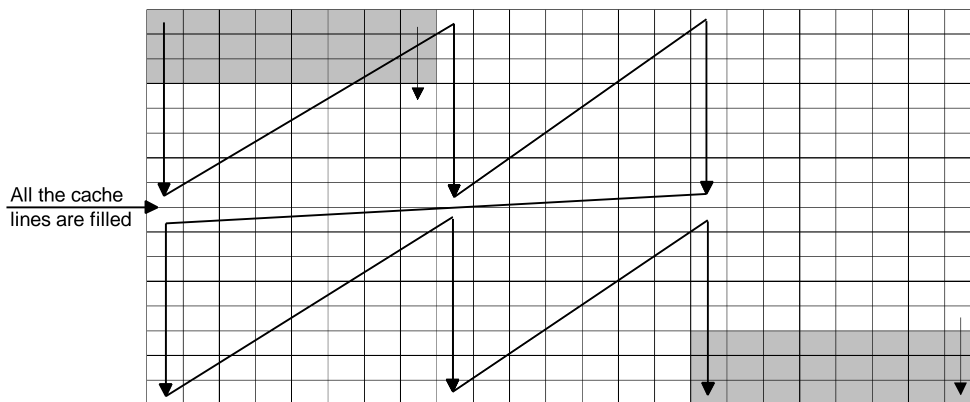



Figure 19: Traversing the image array using rowsets

Row-set size

Since this way of processing the image promises optimal results it has been chosen to implement the filter in this way.
A key issue in this implementation is the moment where all the cache lines are filled. The level one cache consists of 512 lines of 32 bytes each. When the loop runs down the image, each loop cycle one cache line is filled[2]. This means that after running 256 lines[3] down, the entire cache is filled. At this point, the filter starts reading at the top of a new column. The number of rows to run down before starting at the top of the next column is called the row-set size.
If the rowsets are too small performance is decreased because of the overhead. If the rowsets are too big cache lines needed later are overwritten, reducing performance.

The optimal rowset size is further influenced by other data that is read from memory. Because it is hard to predict the right rowset

[1] See section 3.4.2.1.
[2] When re-using the inner products only one 10 byte-wide row is needed to calculate 8 new pixels.
[3] 512/2 because the source and destination both require cache space. The cache space used for overhead (for example Windows NT ) is neglected.

size, performance is tested with various rowset sizes. The results of these tests are described in section 3.4.4.5.

#### 3.4.3.5 Border pixels

When processing pixels like shown in section 3.4.3.4, the leftmost, top and bottom pixels of the image require special treatment, because the complete kernel can not be applied to them. Either a special kernel for border pixels is created, or the border pixels are not processed.

A kernel to apply to the left-border pixels is shown in Figure 20.

| | |
|---|---|
| 2/12 | 1/12 |
| 4/12 | 2/12 |
| 2/12 | 1/12 |

Figure 20: Smooth kernel for left-border pixels

The pixel weighted 4/12 in the kernel shown in Figure 20 is the centerpixel. An additional difficulty when implementing special border kernels is that instead if dividing by 16, a division by 12 has to be made. Because of the absence of a packed divide instruction, the division by 12 is very complex (the division by 16 can be made by shifting three bits to the right). Instead of dividing by 12 it is also possible to assign different weights to the pixels to allow a division by 16.

Copy border pixels

Other algorithms, such as halftoning, add noise to the border of the image, requiring a border to be cut off. Therefore it is generally not necessary to pay special attention to the border pixel. Because of the influence they have on their neighbour pixels when the next filter is applied they are not skipped entirely (resulting in random values) but copied from the source image. The top, bottom and leftmost pixels of the image are copied like this, but a problem arises for the rightmost pixels. The processing loop of the filter calculates eight new pixels simultaneously. Since images will seldom have a width of (8n+2)[1] pixels, the source image will have to be extended horizontally. At the end of the processing path, the border added to the right of the image will have to be removed.

Image width 8n

If the filter would be implemented this way, of each row a number of pixels would be processed that would be deleted from the image later anyway (on the right edge).
If the image width would be extended to a multiple of eight (8n), the seven rightmost pixels would have to be copied. Because the pixels are placed sequentially in the image array, the leftmost pixel of the next row are placed in memory right after the seven rightmost pixels of the previous, allowing the MMX move instruction to copy all eight pixels with two instructions. This requires half the time needed to copy a single leftmost and rightmost pixel.

The top and bottom rows of the image are copied at the beginning respectively the end of the processing loop.

[1] Pixels are processed in 8 pixel wide groups (=8n), two border pixels are copied (=+2) resulting in an image width of 8n+2.

Because the next filter function in the processing path (sharpen) is similar to the smooth filter, an extra function that extends the image width is created and called from the “FilterMmx” function.

### 3.4.4 The basic loop structure

The implementation of the basic loop structure designed in the previous sections is shown in Code example 2.

```
NrOfRowsInSet=256
NrOfColSets=Width/8
ColCounter = Height
Do While (ColCounter > 0)
       {
        ColSetCounter=NrOfColSets
        Do While (ColSetCounter > 0)
              {
               RowCounter= NrOfRowsInSet
               Do While (RowCounter > 0)
                     {
                      Process_Pixels()
                      RowCounter=RowCounter-1
                     }
               ColSetCounter=ColSetCounter-1
              }
        ColCounter = ColCounter-NrOfRowsInSet
        If (ColCounter < 256) Then
              NrOfRowsInSet=ColCounter
       }
```

Code example 2: The basic loop structure of the smooth

In the inner loop the actual processing is done. Each loop cycle an eight pixel wide row is calculated, running down the image array (thus forming an eight pixel wide column). The middle loop (Do While (ColSetCounter > 0) ) determines the number of those columns horizontally processed.

The outer loop counts the number of rowsets to process. If the number of rows left to process is less than the default number of rows in a rowset (256) the default number is adjusted. At the end of the next loop cycle, the ColCounter will be decreased to zero and the filter will exit from the loop.

Avoid ‘if-then’

Note that the required ‘if-then’ would normally be programmed with a compare instruction followed by a jump. However a significant delay occurs if the jump is predicted incorrect[1].

[1] See section 2.4.2.

### 3.4.4.1 Avoiding a jump

Minimum

The jump can be avoided with the following trick; basically, the minimum of the rowset-size and the number of rows left to process has to be taken. Code example 3 takes the minimum of ebx and eax, and places it in eax.

```
sub ebx, eax
sbb ecx, ecx
and ecx, ebx
add eax, ecx
```

Code example 3: Minimum of eax and ebx

The first instruction determines the difference between ebx and eax. If this difference is negative (eax > ebx) the carry-flag is set, otherwise it is cleared. The next instruction copies the carry bit to all bits of ecx, after which the 'AND' instruction masks the calculated difference. If the difference was positive (ebx > eax) the 'AND' instruction will clear ecx. If ebx was less than eax, ecx will be loaded with the (either zero or negative) difference. Finally, ecx is added to eax, causing eax to be decreased if ebx was less. Note that the initial value in ebx is lost.

### 3.4.4.2 Register allocation

The next step in the implementation of the smooth filter is determining the register allocation for calculating the new pixels. Basically, all the pixels required to calculate a new pixel are loaded into MMX registers in such a way that they can be added each with its own weight, after which the result is divided by 16. However, suppose a pixel in a gray plane like shown in Figure 21 is calculated.

Pixels:

| | | |
|---|---|---|
| 192 | 192 | 192 |
| 192 | 192 | 192 |
| 192 | 192 | 192 |

Kernel:

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

Figure 21: Calculating a pixel in a gray plane

Bytes saturate

When calculating the new pixel value for the grayed centerpixel, a number of registers would be used, each containing one of the source pixels. Next, each of the pixels would be multiplies width its weight, one two or four. However, the pixels multiplied with a weight of two or four, would saturate to 255 because the calculated values exceed the range of an unsigned byte (0 to 255), resulting in the calculation shown in [ 1 ] (the italic values are saturated pixels).

$$\text{new pixel} = \frac{4 \times 192 + (4 \times \mathit{255}) + \mathit{255}}{16} = 128 \qquad [\ 1\ ]$$

In [ 2 ] the calculation without the saturation of the pixels is shown.

$$\text{new pixel} = \frac{4 \times 192 + 4 \times (2 \times 192) + 1 \times (4 \times 192)}{16} = 192 \qquad [\ 2\ ]$$

The two resulting pixels are shown[1] in Figure 22.

Figure 22: Difference between pixel value 128 and 192

Obviously these changes are unacceptable. Besides this, it is difficult to divide packed bytes because the packed shift, as well as the packed multiply instruction do not operate on packed bytes[2]. For these two reasons the pixels have to be extended from packed bytes to packed words before calculations can be performed.

Extend to packed words

Suppose in the memory shown in Figure 23, the new value for the squared pixels has to be calculated.

Image:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 |

Kernel:

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

Figure 23: Calculating four pixels

Kernel row

Instead of calculating the new pixel value immediately, first the weighted sum of each of the kernel rows will be calculated, after which these will be added and divided by 16. To calculate the weighted sum of the top row registers would be loaded as shown in Figure 24. Register esi points to the memory location where pixel 01 is stored.

| Instruction | Register | | | | |
|---|---|---|---|---|---|
| movq mm0, [esi] | mm0= | 04 | 03 | 02 | 01 |
| movq mm1,[esi+2] | mm1= | 05 | 04 | 03 | 02 |
| psllw mm1,1 | mm1= | 2x05 | 2x04 | 2x03 | 2x02 |
| movq mm2,[esi+4] | mm2= | 06 | 05 | 04 | 03 |
| paddusw mm0,mm1 | mm0= | 04+2x05 | 03+2x04 | 02+2x03 | 01+2x02 |
| paddusw mm0,mm2 | mm0= | 04+2x05+06 | 03+2x04+05 | 02+2x03+04 | 01+2x02+03 |

Figure 24: Calculating the top row of the kernel

The example in Figure 24 assumes that the source image esi points to consists of words. Therefore the second instruction reads from location esi+2. In the real implementation of the filter, data will be read from memory as bytes, after which the data is extended to words.

Reverse order

Note that the pixels in the register are reverse ordered compared to the pixels in memory. On Intel Architecture microprocessors this is caused by the way data is copied from memory to a register. The byte on the lower memory address is placed in the least significant byte of the register. For more information on the way data is copied see section 2.4.5.

Multiplying by two

The third instruction shown (PSLLW = packed shift left logical word) shifts the words in mm1 one bit to the left, thus multiplying

[1] This is for one color plane .
[2] With a workaround it is possible to shift bytes. This can be done by shifting words after which the bits shifted out of the most- to the least significant byte of a word have to be masked out with an and-instruction.

them by two. Care has to be taken not to use PSLLQ (= packed shift left logical quadword), which will shift the entire 64-bit register instead of the four packed words, resulting in incorrect values. The two PADDUSW (= packed add unsigned words) add the three weighted pixels, forming the result of the top row.

Bytes to words

The extension of the bytes read from memory can be implemented by using the PUNPCKLBW instruction (= Packed UNPaCK Lower Byte to Word). This instruction merges the four least significant bytes of two quadwords into one quadword, as shown in Figure 25.

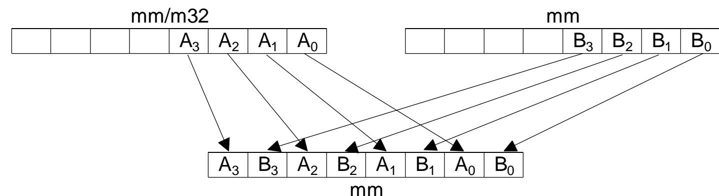



Figure 25: Unpack operation (PUNPCKLBW MM, MM/M32)

If the second register (mm/m32) is filled with zeros, the lower four bytes of the first (mm) are extended from bytes to words. The counterpart of the unpack is the pack instruction, which is described in section 2.4.5.

Eight pixels per loop cycle

After the extension of the bytes to words, two sets of four pixels are processed. After they have been calculated, the new pixel values have to be shrunk from packed words to packed bytes, after which they can be written to memory. Instead of processing each loop cycle eight pixels in two portions of four, it is also possible to process one portion of four pixels. However, this was not done because of two reasons:

1. Other algorithms that do not require the bytes to be extended to words require an image width of 8n pixels so the image width has to be extended anyway.
2. The sharpen filter, which is similar to the smooth, requires an image width of 8n. It is preferred to keep the loop structure of the sharpen and the smooth identical, because only the kernels differ.

Processing each loop cycle multiple sets of pixels instead of one is called ‘loop unrolling’. Loop unrolling is sometimes applied to achieve better pairing rates (see section 2.4.2).

#### 3.4.4.3 The inner loop structure

The register allocation described in section 3.4.4.2 results in the inner loop structure shown in Code example 4. The code resembles the Process_Pixels function in the pseudocode shown in Code example 2on page 30. Register esi and edi point to the lower-left pixels of the data-block required to process eight pixels. esi points to the source array, edi to the destination. The registers mm1 to mm7 are MMX registers, ResultBuffer is a four quadword sized buffer used to store the weighted sums used later.

```
                    //Least significant 4 pixels
mm1=ResultBuffer[0]       //Upper row
mm2=Resultbuffer[3]       //Middle row
mm2=ShiftLeft(mm2,1)      //mm2=2*mm2
mm3=Read8Pixels(esi)      //Lower row
mm3=UnpackData(mm3)

mm3=CalculateRow(mm3)
mm0=(mm1+mm2+mm3)/16      //Calc. new pixels

ResultBuffer[0]=ResultBuffer[3]  //Shift buffer.
Resultbuffer[3]=mm3       //Store res. of lower row

                    //Most significant 4 pixels
mm1=ResultBuffer[1]       //Upper row
mm2=Resultbuffer[4]       //Middle row
mm2=ShiftLeft(mm2,1)      //mm2=2*mm2
mm3=Read8Pixels(esi+4)    //Lower row
mm3=UnpackData(mm3)

mm3=CalculateRow(mm3)
mm1=(mm1+mm2+mm3)/16      //Calc. new pixels

ResultBuffer[1]=ResultBuffer[4]  //Shift buffer
Resultbuffer[4]=mm3       //Store res. of lower row


mm0=PackData(mm0,mm1)
Write8Pixels(edi+1)
```

Code example 4: The inner loop structure of the smooth

#### 3.4.4.4 Image size

Before the performance for various rowset sizes can be determined, first the resolution of the images which will be processed has to be determined. Figure 26 shows the relation between the number of dots per inch (dpi) and the number of pixels for two resolutions.

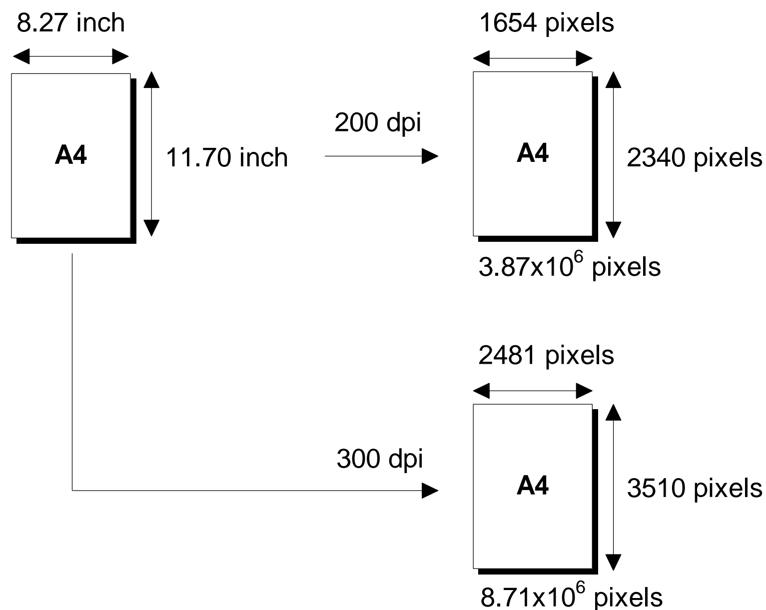



Figure 26: Relation between the resolution and the image size

Note that each pixel requires 3 bytes, resulting in an image size of 11.6 MB for the 200 dpi scan, and 26.1 MB for the 300 dpi scan. The image size was the primary reason why it was chosen

to use 200 dpi images instead of 300. If a 300 dpi image would be used, the minimum amount of storage space to be allocated by the program would be 52 MB[1]. On a PC with a total physical memory of 64 MB this would lead to large portions of the image being

Minimum physical memory

swapped to and from the hard disk. Generally, the minimal size of the physical memory should be about twice the size of the image plus 25 MB, which is required for Windows NT.

The test image scanned at 200 dpi has a total pixel size of 1580x2176. This is smaller than expected because the scanner removes 0,42 inch from the top and bottom edges, as well as

200 dpi: 10MB

0.19 inch from the left and right edge. The 200 dpi scan with the borders cut of has a total size of 10 MB.

#### 3.4.4.5 The optimal rowset size

Now the final code has been built and it has been decided which resolution images will be processed, the optimal rowset size can be determined.

150 dpi A4
200 dpi

Performance is measured for two different images, an A4 scanned at 150 dpi, another scanned at 200 dpi resulting in a resolution of 1210x1627 for the first image and 1580x2176 for the second (both images with 16 million colors).The performance for different rowset sizes is shown in Table 2 (in cycles per processed pixel).

| Rowset size | 1210x1627x16 milj (cycles/pixel) | 1580x2176x16 milj (cycles/pixel) |
|---|---|---|
| 1 | 48 | 50 |
| 2 | 35 | 37 |
| 4 | 29 | 31 |
| 8 | 26 | 28 |
| 16 | 24 | 26 |
| 32 | 23 | 25 |
| 48 | 23 | 25 |
| 64 | 23 | 25 |
| 96 | 23 | 33 |
| 128 | 29 | 34 |
| 256 | 29 | 34 |
| 512 | 35 | 40 |
| 1024 | 37 | 40 |

Table 2: Performance for various rowset sizes

For small rowset sizes performance is sub-optimal. After reaching a minimum, the number of cycles per pixel increases. The latter is caused by cache lines which are overwritten although they are still needed. For rowset sizes which are a bit too small performance decreases marginally. For *very* small rowsets the performance is poor. This is caused by the overhead required to process the rowsets.

The performance measurements for various rowset sizes clearly show the influence of the image size on the optimal rowset size.

[1] Assuming the source an destination array both require 26 MB.

The optimal rowset size *seems* to be determined by the size of the level two cache and the image width in pixels[1].
The test PC has a level two cache of 256 KB. Assuming that both the source and destination array require 125 KB, the optimal rowset size for the 200 dpi scan can be determined with the

Calculated rowset size

calculation shown in [ 3 ].

$$\frac{\frac{1}{2}\times \text{L2 cache size}}{\text{Image width}} = \frac{\frac{1}{2}\times 256\times 1024}{1580} = 83 \text{ rows / rowset} \qquad [3]$$

Test results

Tests show the optimal rowset size is between 32 and 64 rows/rowset (200 dpi scan). The reason the value calculated in [ 3 ] results in a value slightly too large is that it ignores the cache space used for other variables, as well as the cache space used by Windows NT.
From the previous it can be concluded that the optimal rowset size can be estimated be dividing half the level two cache size by the image width (in pixels), but that the accurate size can only be determined by measuring performance with various rowset sizes.

Rowsets of 64

It has been decided to use a rowset size of 64[2], because this size gives optimal performance for both 150 and 200 dpi (A4) scans. For smaller images the performance will be sub-optimal.

38% faster by rowsets

In Table 3 the performance improvement of an optimal rowset implementation has also been compared to an implementation that does not use rowsets (for the 150 dpi scan).

| | |
|---|---|
| Without rowsets: | 37 cycles/pixel |
| Rowsets of 64: | 23 cycles/pixel |
| Improvement: | 38 % |

Table 3: Performance gain by using rowsets

#### 3.4.4.6 Pairing the code

The final step in implementing the smooth filter is pairing the code. Because most of the processing time required is used for the inner loop, most attention is paid to the inner loop.

Re-order statements

When pairing the code, program statements are re-ordered in such a way that the CPU can execute two instructions parallel. Care has to be taken not to switch related statements, since this will cause the filter to malfunction.

#### 3.4.4.7 Filter performance

After the code has been paired performance has been tested again, resulting in the cycle times shown in Table 4. In this table the performance of the paired version is compared to the unpaired version[3].

[1] Not the image with in bytes, because the filter operates on the separated color planes. Each pixel is represented with one byte in each color plane.
[2] The final step in the implementation of the filter (pairing the code) has no influence on the optimal rowset size because pairing only influences the code, which is loaded into a separate code cache.
[3] The improvement is calculated by dividing the difference in number of cycles/pixel by the new number of cycles/pixel.

| Image size (pixels) | Cycle times | | Improvement (%) |
|---|---|---|---|
| | unpaired (cycles/pixel) | paired (cycles/pixel) | |
| 290x509 | 23 | 20 | 15,0 |
| 1580x2176 | 25 | 19 | 31,6 |

Table 4: Performance improvement by pairing the code

The difference in performance improvement is caused by the fact that for the larger image the memory bandwidth has more influence on the performance than for the smaller image.
In Table 4 it can be seen that pairing the code improves performance of the smooth operation on the 200 dpi A4 image with 32%.

32% faster by paring

In Table 5 the performance of the paired MMX code is compared to that of the C++ code for several images.

| Image size (pixels) | Cycle times | | Improvement | |
|---|---|---|---|---|
| | C++ version (cycles/pixel) | MMX version (cycles/pixel) | (%) | (factor) |
| 87x16 | 67 | 13 | 80,6 | 5,2 |
| 63x96 | 73 | 16 | 78,1 | 4,6 |
| 127x96 | 83 | 17 | 79,5 | 4,9 |
| 255x320 | 94 | 19 | 79,8 | 4,9 |
| 290x509 | 94 | 20 | 78,7 | 4,7 |
| 1580x32 | 89 | 19 | 78,7 | 4,7 |
| 1580x2176 (1) | 96 | 19 | 80,2 | 5,1 |
| 1580x2176 (2) | 95 | 19 | 80,0 | 5,0 |

Table 5: Performance of the smooth filter

Influence of clipping

The last two images are A4 pages scanned at 200 dpi. The performance of the C++ implementation differs because the images differ; in the first image more clipping[1] occurs than in the second, thus requiring more calculations. The MMX implementation has a constant performance because no ‘if-then’ is needed[2] for clipping.

MMX 5x faster than C++

From Table 5 it can be concluded that the MMX implementation is about a factor five faster than the C++ implementation. Processing the scanned A4 images with the C++ implementation requires 96 cycles/pixel (1.99 seconds). The MMX implementation requires 19 cycles/pixel (0.39 seconds).

Small images

Notice that the number of cycles/pixel decreases for smaller images. This is caused by the fact that a (very) small image can be kept entirely in the level one cache[3]. A slightly larger image can still be kept into the level two cache, but the 200 dpi scanned A4 images (of approximately 10 MB) can not.

Added border

Because the MMX implementation of the processing path adds a border to the image to extend the image width to a multiple of eight (8n), performance will be optimal for images where the extra pixels do not have to be added because the image width

[1] If the pixel clips has to be tested with an “If-Then” statement.
[2] See section 2.4.5.
[3] It is loaded into the level one cache when performing the color plane separation.

already is 8n[1]. When implementing a complete processing path using MMX technology, the width of the scanned image would have to be chosen a multiple of eight.

Finally, a it must be remarked that the number of cycles/pixel discussed will differ if the code is run on a system with a different

Different CPU speed

CPU clock speed. This is caused by the fact that the time required to read data from memory is constant, while the cycle time changes. Thus for a CPU with a higher clock speed, the number of cycles/pixel will increase.

#### 3.4.4.8 Categorising the processing time

In Figure 27 the processing time required for the inner loop of the unpaired smooth filter is shown categorised into five categories:

1. Read pixels
2. Buffer (management)
3. Write pixels
4. Calculate
5. Overhead

Cycle count method

If two instructions pair both are counted as using cycles. Alternative counting methods for paired instructions are:

1. Count both half; when counting this way, a one cycle add instruction pairing with a ten cycle memory move, is counted as a five cycle instruction.
2. If both instructions take the same amount of cycles, count both half. Otherwise count the one taking most cycles. Since memory accesses are often paired with arithmetic instructions, this way of counting cycles favours the arithmetic instructions.
3. Weigh each instructions cycles with its proportion to the total number of cycles the pair takes.

The first two alternatives give a wrong impression of the partition of the cycle times. The third is the best way of counting the cycle times, but takes a lot of time to be applied. When counting all instructions (with a pairing rate of 100%) the total number of cycles will be twice the amount when counted according to method three, but the partitioning of the processing time will be the same. Therefore this method of counting the cycle times has been chosen.

For 200 dpi A4

The processing time is measured for an A4 image scanned at 200 dpi. However, not the entire image is processed because it would take VTune a number of days to do this.

[1] When calculating the number of cycles/pixel the total number of cycles is devided by the number of pixels in the original image.

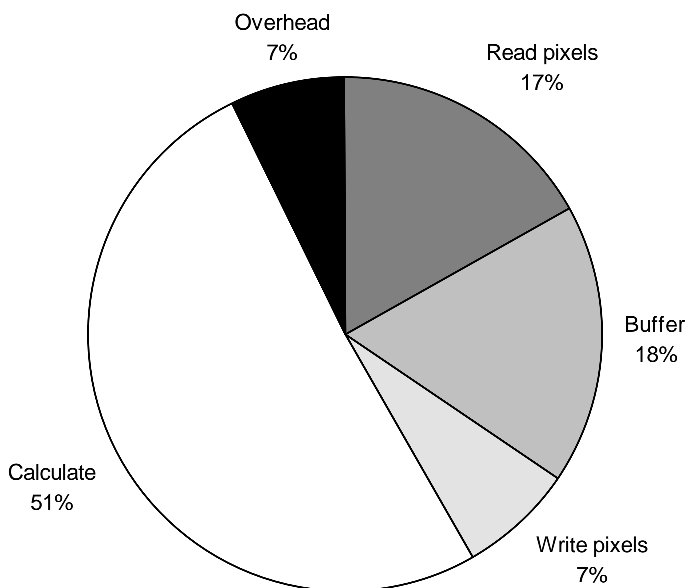



Figure 27: Categorised processing time of the smooth

40% memory

51% calculations

Figure 27 shows that only half of the processing time is required for calculations. More than 40% of the processing time is needed for memory related operations (buffer management, read and write pixels). 51% of the processing time is required for calculations.

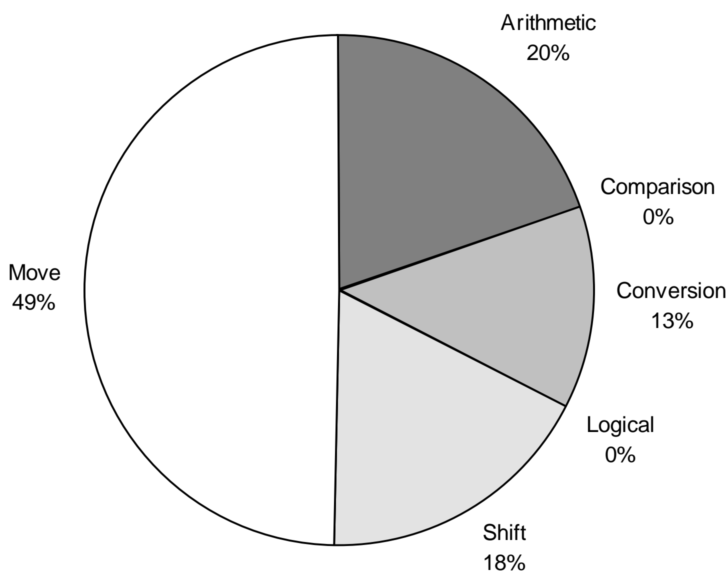



Figure 28: Processing time per instruction type

In Figure 28 the processing time categorised per instruction type is shown. The cycle times are counted in the same way as in Figure 27. Notice that the amount of time used for move

operations (including memory moves) is about the same as the amount of memory related operations in Figure 27.

Figure 28 shows that only 20% of the processing time is used for arithmetic instructions. When taking in account that some of these are required for the overhead, this is surprisingly low compared to the 51% required for calculations in Figure 27. When adding the processing time of the conversion and shift instructions[1] however, this value[2] is also reached.

## 3.5 The sharpen algorithm

### 3.5.1 The algorithm

During the smooth operation discussed in the previous chapter, sharp edges are blurred. This is unwanted for text and other sharp object. This effect is compensated by enhancing the edges using a sharpen operation. Thus, after performing a smooth and a sharpen the scanner interference is suppressed, but edges are still sharp.

In Figure 29 the sharpen filter is applied to the squared area.

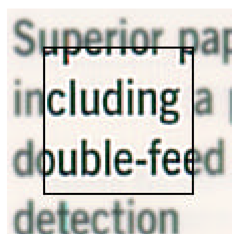



Figure 29: Effect of sharpening

The sharpen operation is implemented as a 3x3 kernel operation, with the coefficients shown in Figure 30.

| -1/4 | | -1/4 |
|---|---|---|
| | 8/4 | |
| -1/4 | | -1/4 |

Figure 30: Sharpening kernel

### 3.5.2 C++ implementation

#### 3.5.2.1 Traversing the image array

Basically the only difference between the smooth and the sharpen filter is that other coefficients are applied. Therefore the loop structure of the C++ implementation of both filters is identical. Thus the loop runs horizontally from left to right through the image.

Horizontal traversing

[1] These instructions are counted as calculate-instructions in Figure 27 because they are necessary to enable the calculations.

[2] 20%+18%+13% = 51%

#### 3.5.2.2 Re-using calculated values

The limited kernel symmetry allows re-usage of calculated values only for the left and right column of the kernel. A calculated weighted sum of the right row can not be used in the next loop cycle, but in the loop cycle after that. This requires a buffer.

#### 3.5.2.3 Clipping

Negative pixels

The negative values of the kernel can cause the new pixel value to be negative (if the centerpixel is zero and the other pixels are not). Negative pixels are invalid and should be corrected to zero[1].
Now suppose the center-source-pixel is 255, while the other pixels are all zero. The new pixel value calculated will be 510 (255*8/4). This value is too large and should be corrected to 255. The described adjustment of a pixel value because it exceeds the pixels upper or lower limit is called clipping.

Clipping requires the value of the calculated pixel to be tested, and if necessary, adjusted. Normally, this would be done in the way shown in Code example 5.

```
If (New_Pixel < 0) Then
       ImageArray[Row,Col]=0
Else If (New_Pixel > 255) Then
       ImageArray[Row,Col]=255
Else
       ImageArray[Row,Col]=New_Pixel
```

Code example 5: Non-optimised clipping code

This will result in non optimal performance because the compiler will translate this using four jumps[2] (one for each 'if', one for each 'else'). The code in Code example 6 uses three jumps.

```
ImageArray[Row,Col]=New_Pixel
If (New_Pixel < 0) Then
       ImageArray[Row,Col]=0
Else If (New_Pixel > 255) Then
       ImageArray[Row,Col]=255
```

Code example 6: Optimised clipping code

Optimised clipping code

This version of the clipping code will use less time because it contains less jumps; a jump which is mispredicted requires a lot of time because the pipeline has to be reloaded (see section 2.4.2).

#### 3.5.2.4 The final C++ implementation

Taking the issues described in the previous sections in account, the final structure of the C++ implementation of the sharpen filter is shown in Code example 7.

[1] If a pixels value exceeds the upper limit (255) the pixel should be saturated to 255.
[2] With the compiler option for optimisation set to off.

Code example 7: C++ implementation of the sharpen

```
Dst.Rows[0] = Src.Rows[0]        // Copy top and bottom row
Dst.Rows[Height] = Src.Rows[Height]
RowCounter = 1
Do While (RowCounter < Height-1)
        {
          Dst.Pixels[RowCounter,0] = Src.Pixels[RowCounter,0]
          A = Src.Pixels[RowCounter-1,0] + Src.Pixels[RowCounter+1,0]
          B = Src.Pixels[RowCounter-1,1] + Src.Pixels[RowCounter+1,1]
          ColCounter = 1
          Do While(ColCounter < Width-1)
                {
                 Temp=Src.Pixels[RowCounter-1,ColCounter+1] +
                       Src.Pixels[RowCounter+1,ColCounter+1]
                 New_Pixel=(-A+8*Src.Pixels[ColCounter,RowCounter]-Temp)/4

                 Dst.Pixels[RowCounter,ColCounter]=New_Pixel
                 If (New_Pixel < 0) Then
                       Dst.Pixels[RowCounter, ColCounter]=0
                 Else If (New_Pixel > 255) Then
                       Dst.Pixels[RowCounter, ColCounter]=255

                 A = B              //Shift buffer contents
                 B = Temp
                 ColCounter = ColCounter+1
                }
         Dst.Pixels[RowCounter,Width]=Src.Pixels[RowCounter,Width]
         RowCounter = RowCounter+1
        }
```

Performance

The performance of the code is discussed in section 3.5.3.11, where it is compared to that of the MMX implementation.

## 3.5.3 MMX implementation

### 3.5.3.1 Parallelability of the algorithm

Similar to the smooth filter, there are no limitations as far as the parallelability of the sharpen filter is concerned.

### 3.5.3.2 Pixel block width

Next it has to be decided how many pixels to process in parallel. This is determined by the number of data elements processed parallel; eight packed bytes, four packed words or two packed double-words.

Smooth filter

The smooth filter read pixels as packed bytes from memory, after which they are extended to packed words for accuracy reasons. Each loop cycle eight new pixels are calculated. Because the ten-pixel wide rows of input pixels required to calculate eight new pixel values have to be read with two MMX memory reads. Besides this the pixels have to be extended to packed words (see section 3.4.4.2). Therefore instead of calculating eight pixels per loop cycle, the smooth filter could also be implemented calculating four pixels per loop cycle.

Sharpen filter

Now, lets look at the input pixels required for the sharpen algorithm to calculate eight new pixels. In Figure 32 the new values for the pixels 22 to 29 are calculated, requiring input from the grayed area.

| -1/4 | | -1/4 |
|---|---|---|
| | 8/4 | |

| -1/4 | | -1/4 |
|---|---|---|

Figure 31: Sharpening kernel

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 |

Figure 32: Source pixels for the sharpen algorithm

Notice that the pixels 21 and 30, are not used for the calculation of 22-29. This is caused by the fact that the kernel does not use the pixel to the left and the pixel to the right of the centerpixel of the kernel.

Similar to the smooth, the sharpen requires ten pixels of the top and bottom row. But for the middle row only eight pixels are needed. These pixels can be read with one MMX memory read, instead of two for the top and bottom row (each).
If the filter processes eight pixels per loop cycle five memory reads are required to calculate eight pixels. When processing four pixels per loop cycle, three reads are required for four pixels (=six reads for eight pixels).

Eight pixels per loop cycle

Because a memory access requires relatively many cycles, it has been decided to process eight pixels per loop cycle.

### 3.5.3.3 Re-usage of calculated values

Suppose the image is traversed vertically, and that the total calculation is split into three calculations (one for each of the kernel-rows). The calculated values for the bottom row can then be stored to be used for the top row in the loop cycle after the next loop cycle, like shown in the following example, with the image shown in Figure 34 (the sharpen kernel is shown in Figure 33).

| -1 | | -1 |
|---|---|---|
| | 8 | |
| -1 | | -1 |

Figure 33: Sharpen kernel

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |

Figure 34: The intermediate result of the bottom row

For each of the three image rows the intermediate result of the kernel coefficients can be calculated. In the next loop cycle the pixels 42 to 49 are calculated.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |

Figure 35: The middle row

Because the middle row of the kernel is completely different than the bottom row, the intermediate result of the bottom row cannot be re-used. In the next loop cycle however, the stored intermediate result can be re-used for the top row.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |

Figure 36: Re-using the bottom row

Vertical traversing

Obviously re-using calculated values in this way is only possible when running vertically through the image. If the image is traversed horizontally, only the two rightmost edge values can be re-used, similar to the smooth algorithm (see section 3.4.3.3).

#### 3.5.3.4 Optimal image traversing

Using rowsets

The fact that the re-usage of calculated values only is possible when running vertically through the image, combined with the cache related factors described in section 3.4.2.1 leads to the conclusion that optimal image traversing can be reached by using rowsets in the same way as with the smooth filter (see section 3.4.3.4).

The optimal rowset size has to be determined again, because the way the sharpen filter reads the data is slightly different than the way the smooth does. After the sharpen filter has been implemented completely the optimal rowset size is determined (see section 3.5.3.9).

#### 3.5.3.5 Border pixels

Similar to the smooth, the sharpen algorithm requires special treatment of border pixels. For example, the left-border pixels would has to be calculated with the kernel shown in Figure 37.

| | |
|---|---|
| | -1/6 |
| 8/6 | |
| | -1/6 |

Figure 37: Kernel for left-border pixels

Because a border of the image is cut off anyway it is decided to skip the border pixels (top-, bottom-, left- and right edge).

With the smooth filters MMX implementation it was decided to extend the image's width to a multiple of eight (=8n)[1]. Not

[1] See section 3.4.3.5.

surprisingly, this is an optimal width for the sharpen filter too.

Copy borders

Because the pixels are calculated in groups of eight, one MMX memory move can be used to copy one pixel of the left border as well as seven pixels of the right border.

#### 3.5.3.6 The basic loop structure

From the previous section it is clear that the loop structure used for the smooth filter can be used also for the sharpen filter, as shown in Code example 8.

```
NrOfRowsInSet=256
NrOfColSets=Width/8
ColCounter = Height
Do While (ColCounter > 0)
        {
         ColSetCounter=NrOfColSets
         Do While (ColSetCounter > 0)
                {
                 RowCounter= NrOfRowsInSet
                 Do While (RowCounter > 0)
                        {
                         Process_Pixels()
                         RowCounter=RowCounter-1
                        }
                 ColSetCounter=ColSetCounter-1
                }
         ColCounter = ColCounter-NrOfRowsInSet
         If (ColCounter < 256) Then
                NrOfRowsInSet=ColCounter
        }
```

Code example 8: The basic loop structure of the sharpen

The number of rows in a rowset temporarily is set to 256. After the filter has been fully implemented the optimal rowset size has to be determined (section 3.5.3.9).

Avoiding ‘if-then’

The jump normally required for the ‘if-then’ at the end of the outer loop can be avoided as described in section 3.4.4.1.

#### 3.5.3.7 Register allocation

Next the register allocation has to be determined. Basically, all the pixels needed to calculate a new pixel are loaded into MMX registers. After the centerpixels are multiplied with their weight (8) the four pixels of the upper and lower corners are subtracted. Finally the result is divided by four.

Now consider a pixel in a white plane like shown in Figure 38 is calculated.

Pixels:

| | | |
|---|---|---|
| 255 | 255 | 255 |
| 255 | 255 | 255 |
| 255 | 255 | 255 |

Kernel:

| | | |
|---|---|---|
| -1 | | -1 |
| | 8 | |
| -1 | | -1 |

Figure 38: Calculating a pixel in a white plane

Suppose the new pixel value is calculated using packed bytes. Signed packed bytes are required because of the subtraction of the corner-pixels, resulting in a range of -128 to +127. The calculation of the new pixel is shown in [ 4 ] (saturated values are italic).

[ 4 ]

$$\text{new pixel} = \frac{-255-255+255-255-255}{4} = -191 = 0 \ \text{(clipped)}$$

Bytes saturate

After the multiplication with eight the centerpixel saturates to 255. In [ 4 ] the corner pixels are subtracted unsigned with saturation, leading to the result shown[1]. Because negative pixels are not allowed, this value has to be clipped to 0 (see section 3.5.2.3).

The calculation without the (unwanted) saturation is shown in [ 5 ].

$$\text{new pixel} = \frac{-255-255+(8\times 255)-255-255}{4} = 255 \qquad [\ 5\ ]$$

The two resulting pixels are shown in Figure 39.

Figure 39: Difference between pixel value 0 and 255

Packed signed words

Because these changes are unacceptable it has been decided to extend the packed unsigned bytes of the source image to packed
signed words before performing calculations.

Suppose with the memory shown in Figure 40 the new values for the pixels 12 to 15 have to be calculated.

Image:

| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |
|---|---|---|---|---|---|---|---|---|
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |

Kernel:

| -1 | | -1 |
|---|---|---|
| | 8 | |
| -1 | | -1 |

Figure 40: Calculating four pixels

Top kernel row

The calculation is divided into four parts; one part for each row, and a fifth to merge the three rows. The example in Figure 41 shows how the intermediate result for the top row is calculated. Similar to the smooth, register esi points to the source array.

| | | | | | |
|---|---|---|---|---|---|
| movq mm0, [esi] | mm0= | 04 | 03 | 02 | 01 |
| movq mm1[esi+4] | mm1= | 06 | 05 | 04 | 03 |
| paddusw mm0,mm1 | mm0= | 04+06 | 03+05 | 02+04 | 01+03 |

Figure 41: Calculating an intermediate result

The example assumes the pixels are written in the array as signed words; in the real implementation the pixels are written as unsigned bytes. The second instruction then reads data from (esi+2) instead of (esi+4).

Notice that the pixels are reverse ordered compared to the pixels in memory. This is caused by the way multi-byte data is read from memory in Intel Architecture microprocessors (see section 2.4.5).

[1] Other implementation of the calculation have been investigated, but they all produce incorrect results.

Merge kernel rows

In the same way like shown in Figure 41 the intermediate values for the bottom row is calculated. The code in Figure 42 merges the three intermediate results, and writes the resulting pixels. The intermediate results are assumed to be loaded into mm0 (top-), mm1 (middle-) and mm2 (bottom row). In the final implementation the top row's intermediate value is re-used from the bottom row of previous pixels.

| | | | | | |
|---|---|---|---|---|---|
| | mm0= | 04+06 | 03+05 | 02+04 | 01+03 |
| | mm1= | 15 | 14 | 13 | 12 |
| | mm2= | 24+26 | 23+25 | 22+24 | 21+23 |
| psubw mm1,mm0 | mm1= | -04-06+15 | -03-05+14 | -02-04+13 | -01-03+12 |
| psubw mm1,mm2 | mm1= | -04-06+15-24-26 | -03-05+14-23-25 | -02-04+13-22-24 | -01-03+12-21-23 |

Figure 42: Merging the three intermediate results

Arithmetic vs. logical shift

Next, the value in mm1 has to be divided by four to complete the operation. This is done with the instruction 'PSRAW MM1, 2' (Packed Shift Right Arithmetic Words) which shifts the words in mm1 two bits to the right. A common mistake is that instead of the arithmetic shift the logical shift instruction is used. For negative values this will produce incorrect results because then the (leftmost) sign-bit is also shifted, as illustrated in the example in Figure 43 where a byte is shifted.

| | Binary | Decimal |
|---|---|---|
| Start value: | 11111100 | -4 |
| After shift right logical 1 bit: | 01111110 | 129 |
| After shift right arithmetic 1 bit: | 11111110 | -2 |

Figure 43: Shift logical vs. shift arithmetic

Notice that the logical shift inserts zeroes at the left side of the byte, while the arithmetic shift inserts the sign bit.

Similar to the smooth, the extension from packed bytes to packed words is done with the PUNPCKLBW (Packed UNPaCK Lower Bytes to Words) instruction.

#### 3.5.3.8 The inner loop structure

The previous section describes how four pixels are calculated with input from packed words. In the final implementation the input pixels are read as packed bytes, and each loop cycle eight pixels are calculated.

This results in the inner loop pseudocode shown in Code example 9. This code resembles the Process_Pixels function in the pseudocode shown on page 45.

```
                    //Least significant 4 pixels
mm1=ResultBuffer[0]       //Upper row
mm2=Read8Pixels(esi+1)    //Middle row
mm6=mm2                   //Copy
mm2=UnpackData(mm2)
mm3=Read8Pixels(esi+Width)//Lower row
mm3=UnpackData(mm3)

mm3=CalculateRow(mm3)
mm0=(8*mm2-mm1-mm3)/4     //Calc. new pixels

ResultBuffer[0]=ResultBuffer[3] //Shift buffer.
Resultbuffer[3]=mm3       //Store res. of lower row

                    //Most significant 4 pixels
mm1=ResultBuffer[1]       //Upper row
mm2=UnpackData(mm2)       //Middle row
mm3=Read8Pixels(esi+Width+2)//Lower row
mm3=UnpackData(mm3)

mm3=CalculateRow(mm3)
mm1=(8*mm2-mm1-mm3)/4     //Calc. new pixels

ResultBuffer[1]=ResultBuffer[4] //Shift buffer
Resultbuffer[4]=mm3       //Store res. of lower row


mm0=PackData(mm0,mm1)
Write8Pixels(edi+1)
```

Code example 9: Inner loop structure of the sharpen

In the pseudocode in Code example 9 register esi points to the source array; edi points to the destination array like shown in Figure 44.

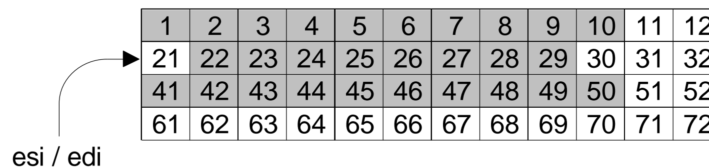



Figure 44: Pointer positioning

Automatic clipping

Notice that in the pseudocode no statements are placed to perform clipping of pixels whose values lie outside the range 0 to 255. This code can be omitted because the pack instruction used to pack the two packed words into a packed byte automatically clips the pixels[1]. This is a huge advantage compared to the C++ implementation, because the 'if-then' required there consumes a lot of processing time.

### 3.5.3.9 The optimal rowset size

In section 3.5.3.4 it was decided to use rowsets for optimal image traversing. Now the entire filter has been implemented the optimal rowset size has to be determined. Performance is measured for two A4 images; the first scanned at 150 dpi, the second at 200.

[1] See section 2.4.5.

| Row set size | 1210x1627x16 milj (cycles/pixel) | 1580x2176x16 milj (cycles/pixel) |
|---|---|---|
| 1 | 44 | 45 |
| 2 | 35 | 37 |
| 4 | 31 | 33 |
| 8 | 30 | 31 |
| 16 | 28 | 29 |
| 32 | 28 | 29 |
| 48 | 28 | 29 |
| 64 | 27 | 29 |
| 96 | 27 | 36 |
| 128 | 33 | 37 |
| 256 | 34 | 37 |
| 512 | 37 | 41 |
| 1024 | 38 | 41 |

Table 6: Sharpen performance for various rowset sizes

Notice that the range of optimal rowset sizes is much larger as with the smooth filter[1].
For the reference image of 200 dpi (right column) the optimal rowset size is between 16 and 64. Because smaller images, such as shown in the left column, seem to prefer larger rowset sizes, it has been chosen to use rowsets of 64.

Rowsets of 64

It is expected that for images much smaller than the 200 dpi A4 scan performance will be far from optimal, because the 150 dpi already shows optimal performance with 64 rows/rowset.

### 3.5.3.10 Performance gain caused by re-using results

26% gain by re-using

Before paring the code it was measured if the re-using results causes a performance improvement. It appeared that re-using results causes a performance improvement of 26% for the sharpen algorithm. Since the smooth kernel is symmetrical, there the performance gain will be even more.

### 3.5.3.11 Filter performance

After the inner loop of the filter has been paired, performance is compared to that of the unpaired code. The results are shown in Table 7.

| Image size | Cycle times | | Improvement |
|---|---|---|---|
| | unpaired | paired | |
| (pixels) | (cycles/pixel) | (cycles/pixel) | (%) |
| 290x509 | 27 | 20 | 25,9 |
| 1580x2176 | 29 | 20 | 31,0 |

Table 7: Performance gain caused by pairing

31% gain by pairing

In Table 7 it can be seen that pairing improves the performance of the sharpen filter significantly. The improvement for the 200 dpi A4 is 31%. This is slightly less than for the smooth filter because the amount of (unpairable) memory accesses of the sharpen is higher.

In Table 8 the performance of the paired version is compared to that of the C++ version for several images.

[1] See section 3.4.4.5.

| Image size | Cycle times | | Improvement | |
|---|---|---|---|---|
| (pixels) | C++ version (cycles/pixel) | MMX version (cycles/pixel) | (%) | (factor) |
| 87x16 | 55 | 12 | 78,2 | 4,6 |
| 63x96 | 63 | 16 | 74,6 | 3,9 |
| 127x96 | 75 | 18 | 76,0 | 4,2 |
| 255x320 | 85 | 19 | 77,6 | 4,5 |
| 290x509 | 88 | 20 | 77,3 | 4,4 |
| 1580x32 | 83 | 18 | 78,3 | 4,6 |
| 1580x2176 (1) | 90 | 20 | 77,8 | 4,5 |
| 1580x2176 (2) | 88 | 20 | 77,3 | 4,4 |

Table 8: Performance of the sharpen algorithm

The lower two images are A4 images scanned at 200 dpi. The difference between the cycle times is caused by the fact that in the first image more clipping[1] occurs than in the second. In the C++ implementation this causes a slight performance reduction because of the additional code processed.

MMX 4x faster than C++

From Table 8 it can be concluded that the MMX implementation is
about a factor 4.4 faster that the C++ implementation. Processing 200 dpi A4 images with the C++ implementation takes 90 cycles/pixel (1.86 seconds. The MMX implementation requires 20 cycles/pixel (0.41 seconds).

Similar to the smooth, the number of cycles/pixel for the sharpen generally decreases when processing smaller images. As discussed in section 3.4.4.7 this is caused by caching effects.

Borders

Performance of the MMX implementation could be increased with 0.5%[(2)] by removing the function that increases the width of the image to a multiple of eight. Note that such a version of the processing path can only process image with a width of a multiple of eight.

#### 3.5.3.12 Categorising the processing time

Figure 45 shows the categorised processing time of the sharpen algorithm. The cycle times are counted in the same way as for the smooth filter[3].

43% calculations

49% memory

Notice that, compared to the smooth filter[4], less time is required for calculations (43% vs. 51%) this is caused by the fact that the sharpen kernel requires less calculations. The time required to read the pixels has increased from 17% to 26% because the sharpen filter requires the middle kernel row to be read form the source image, whereas the smooth filter can re-use the bottom kernel row. Because of this the time required for memory operations has increased to 49%[(5)].

---

1 See section 3.5.2.3.
2 8 / 1580*100%.
3 See section 3.4.4.8.
4 See Figure 27.
5 26%+15%+8% = 49%

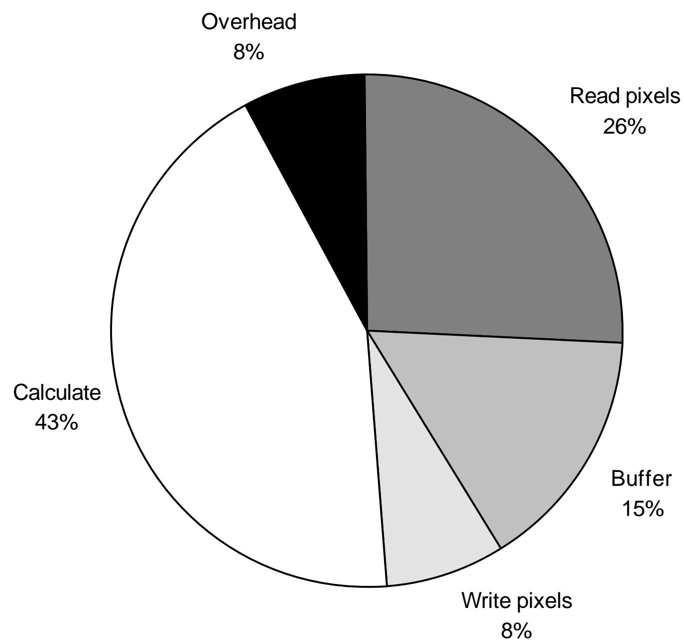



Figure 45: Categorised processing time of the sharpen

In Figure 46 the processing time usage is shown categorised according to the instruction type. This chart also shows that the number of memory operations has increased.

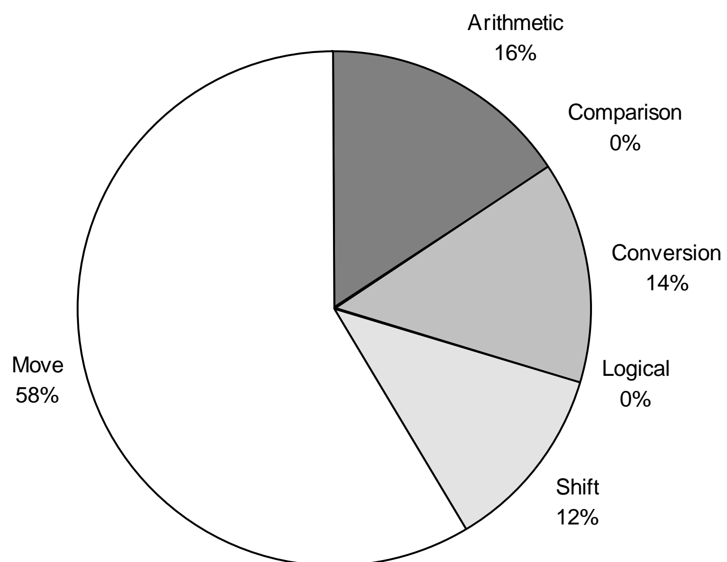



Figure 46: Processing time per instruction type

## 3.6 RGB to CMYK conversion

### 3.6.1 The algorithm

#### 3.6.1.1 The basic conversion

After the smoothing and sharpening the image has to be prepared for printing. This is primarily done by the halftoning algorithm discussed in section 3.7.

Prepare for halftoning

Before the halftoning algorithm can be applied the image has to be converted from the RGB color representation to the CMYK[1] representation. This is necessary for two reasons:

1. Print engines print from the CMYK[2] color domain.
2. When halftoning RGB images around black areas, such as black text, color is created.

Complement

Since the CMYK color representation basically is the complement of the RGB representation, the corresponding CMYK value can be calculated from an RGB value.

This calculation is performed in three steps:

1. Cyan = 255-Red.
   Magenta = 255-Green
   Yellow = 255-Blue
2. Black = Min(Cyan, Magenta, Yellow)
3. Cyan = Cyan-Black
   Magenta = Magenta-Black
   Yellow = Yellow-Black

#### 3.6.1.2 The interpolation

The conversion described in section 3.6.1.1 is not used in Océ processing paths. Instead the conversion is performed by using a 3D lookup table. The RGB values are used to determine the position in the table from where the CMYK value can be read. The conversion is performed this way to enable corrections in the processing path. These correction are required to correct the non-linear toner characteristics.

Lookup table

Corrections

Unfortunately, if all RGB values would be placed in the lookup table, it would have a size of 64 MB[3]. Since this is much too large, the table is made smaller by placing every eighth R, G and B entry on the axis', instead of all R, G and B values.

3D interpolation

Since most CMYK values cannot be read directly from the table a 3D interpolation has to be performed. Figure 47 and [ 6 ] show how this interpolation is performed. The value of point x is calculated by weighed adding the eight angular points that lie around it. The weights are represented by three fractions $\alpha$, $\beta$ and $\gamma$. If the interpolated point lies at angular point zero for example, $\alpha$, $\beta$ and $\gamma$ are zero. In [ 6 ] all except point zero will be weighed zero and x will have the same value as point zero.

---

[1] Cyan, Magenta, Yellow, Black.
[2] See section 2.1.2.2.
[3] 256*256*256*4 = 67,108,864 B = 64MB.

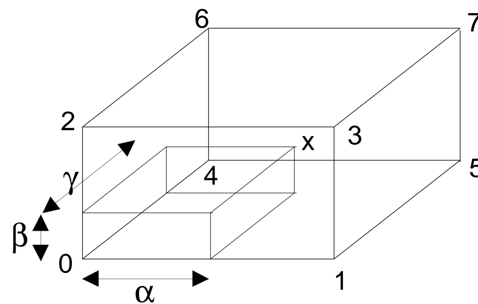



Figure 47: 3D interpolation

$$
\begin{aligned}
X \approx\ & (1-\boldsymbol{a})(1-\boldsymbol{b})(1-\boldsymbol{g})\cdot f(0) \\
+\ & \boldsymbol{a}\cdot(1-\boldsymbol{b})(1-\boldsymbol{g})\cdot f(1) \\
+\ & (1-\boldsymbol{a})\cdot \boldsymbol{b}\cdot(1-\boldsymbol{g})\cdot f(2) \\
+\ & \boldsymbol{a}\cdot \boldsymbol{b}\cdot(1-\boldsymbol{g})\cdot f(3) \\
+\ & (1-\boldsymbol{a})(1-\boldsymbol{b})\cdot \boldsymbol{g}\cdot f(4) \\
+\ & \boldsymbol{a}\cdot(1-\boldsymbol{b})\cdot \boldsymbol{g}\cdot f(5) \\
+\ & (1-\boldsymbol{a})\cdot \boldsymbol{b}\cdot \boldsymbol{g}\cdot f(6) \\
+\ & \boldsymbol{a}\cdot \boldsymbol{b}\cdot \boldsymbol{g}\cdot f(7)
\end{aligned}
\qquad [\,6\,]
$$

## 3.6.2 C++ implementation

### 3.6.2.1 Traversing the image arrays

Horizontal traversing

The first question to ask when implementing the conversion is how to traverse the image array(s). Since none of the directions offers special advantages (such as the re-use of intermediate results) the direction that uses the caches optimally can be chosen. Based on the arguments described in section 3.4.3.4 the horizontal direction is chosen. This way of traversing the image can be implemented with one loop.

Since the input of the conversion function consists of three RGB arrays and the output of four CMYK arrays, a total of seven arrays have to be processed. Obviously, all the arrays are traversed in the same way.

### 3.6.2.2 Reading the lookup table

3D lookup table

The three dimensional lookup table can be stored in memory in two ways:

1. As a three dimensional array containing 32 bit values (each containing the packed CMYK value. To be able to read the individual values the 32 bit quantity is copied into an array of four bytes.
2. As a four dimensional array. This way the fourth dimension can be used to select the individual color components of the angular points.

Copy 32 bit value

A disadvantage of the first method is that the 32 bit quantity must be copied into an array of four bytes. A disadvantage of the second is that an additional multiply operation, which costs many cycles, is used by the compiler to calculate the address of the

element[1]. Since a multiply operation is a very time consuming operation the first method of storing the lookup table is chosen, but both methods seem to require about the same amount of time.

Another method of addressing the color components in the lookup table is overlaying the lookup table with an array of four bytes. This method seems to be the most elegant one, but the required pointer manipulation will cost the same amount of time as needed to copy the 32 bit value to another array (the pointer is also stored as a 32 bit variable in the memory).

#### 3.6.2.3 Implementation of the interpolation

Basically the interpolation can be devided into:

1. Calculate the fractions ($\alpha$, $\beta$, $\gamma$) that determine the weight of the eight angular points.
2. Multiply each of the angular points values with its appropriate weight for all four color planes.
3. Add the eight weighted angular points.

Fractions $\alpha$, $\beta$ and $\gamma$

The fractions $\alpha$, $\beta$ and $\gamma$ would normally have a value between zero and one. These values would be calculated by dividing the *distance* from point zero (which is zero to seven) by eight (the distance between two R, G, or B entries in the lookup table).
To avoid rounding errors the division by eight is performed after each points' values have been multiplied with the distance to point zero. Obviously, fractions like ($1-\alpha$) should be converted to ($8-\alpha$).

Multiply operations

Since multiply operations required require much more cycles (10) than other operations, it is important to keep the number of multiplies as low as possible.
When implementing the interpolation straightforward, 21 multiplies are required (see [ 6 ]). If the gammas are ignored, the weights of the points zero to three are the same as the weights of the points four to seven (see [ 7 ]). Computing these four different weights requires four[2] multiplies. After the points zero to three have been weighed (four multiplies), the sum of those values is multiplied with 8-gamma.

Re-using weights

Since the alpha and beta weights of point zero to three can be re-used to weigh the points four to seven, this only requires five multiplies, resulting in a total of 14 multiplies.

[1] The address of an element in a multidimensional array is calculated by the compiler using a multiply for each dimension (the index times the size of the lower dimension). For more information on how the elements in a multidimensional arrays are addressed see section 3.6.3.4.
[2] One multiply per weight.

$$
\begin{aligned}
X \approx\ & (8-\mathbf{a})(8-\mathbf{b})\,f(0) \\
+\ & \mathbf{a}\cdot(8-\mathbf{b})\,f(1) \\
+\ & (8-\mathbf{a})\cdot\ \mathbf{b}\cdot f(2) \\
+\ & \mathbf{a}\cdot\ \mathbf{b}\cdot f(3) \\
+\ & (8-\mathbf{a})(8-\mathbf{b})\,f(4) \\
+\ & \mathbf{a}\cdot(8-\mathbf{b})\,f(5) \\
+\ & (8-\mathbf{a})\cdot\ \mathbf{b}\cdot f(6) \\
+\ & \mathbf{a}\cdot\ \mathbf{b}\cdot f(7)
\end{aligned}
\qquad [7]
$$

Since this reduces the number of multiplies significantly, it has been decided to calculate the four alpha and beta weights separately. The calculated values are stored in an array.

Four interpolations

Notice that the above describes the interpolation of one value. Four of these interpolations are required for the entire RGB to CMYK conversion.

32-bit arithmetic

Before the interpolations can be implemented, it has to be chosen which data type to use when performing the calculations. It is not possible to use eight bit arithmetic because of the multiplications[1].
Sixteen bit arithmetic would be sufficient, but since 32-bit arithmetic is faster[2] it has been chosen to use 32-bit arithmetic (integers).

#### 3.6.2.4 Reading and writing the pixels

Reading angular points

Reading the required eight angular points from the lookup table is not difficult; the R, G and B values divided by eight can be used as indexes. At position (R/8, G/8, B/8) point zero is located, point one is loaded from position (R/8+1, G/8, B/8) and so on. The individual color components are selected by mapping an array of four bytes over the position in the lookup table indexed by the RGB value.

Reading RGB

The R, G and B values are read from the input arrays. Their position is indicated by the loop index.

Writing CMYK

Write to *source* arrays

Writing the interpolated CMYK values to the four respective arrays is not be a problem either. However, it would not be very efficient to write the CMYK values to four new arrays. Since the RGB values read are no longer needed, they can be overwritten by three of the calculated values. By overwriting the source pixels by calculated values, the memory bandwidth can be reduced drastically, resulting in better performance.

#### 3.6.2.5 The final C++ implementation

α, β and γ: remainder

The issues described in the previous sections lead to the implementation shown in Code example 10. The fractions α, β and γ are calculated by taking the remainder of the division of R, G and B by eight. Of course the added neighbours have to be divided three times by eight[3] to compensate this.

---

[1] This would cause overflow.
[2] Source: "How to optimize for the Pentium Processor" [11], section 20.
[3] One time for α, β and γ each.

```
Integer x, y, z            // Position of neighb. 0
Integer alpha, beta, gamma
Integer Weights[4]         // Weights of neighb. (ignore gamma)
Byte Neighbours[8][4]      // 8 neighbours, each CMYK

Integer i=0
Do While (i < Width*Height)
        {
         alpha=cArrayR[i]%8
         beta =cArrayG[i]%8
         gamma=cArrayB[i]%8
         Weights=Calculate4Weights(alpha,beta)

         x=cArrayR[i]/8
         y=cArrayG[i]/8
         z=cArrayB[i]/8
         Neighbours=Read8Neighbours(x,y,z)

         cArrayR[i]=(Weights[0 to 3]*                 // Cyan
                           Neighbours[0 to 3][0]*(8-gamma) )+
                    (Weights[0 to 3]*
                           Neighbours[4 to 7][0]*gamma) )/8/8/8
         cArrayG[i]=(Weights[0 to 3]*                 // Magenta
                           Neighbours[0 to 3][1]*(8-gamma) )+
                    (Weights[0 to 3]*
                           Neighbours[4 to 7][1]*gamma) )/8/8/8
         cArrayB[i]=(Weights[0 to 3]*                 // Yellow
                           Neighbours[0 to 3][2]*(8-gamma) )+
                    (Weights[0 to 3]*
                           Neighbours[4 to 7][2]*gamma) )/8/8/8
         cArrayK[i]=(Weights[0 to 3]*                 // Black
                           Neighbours[0 to 3][3]*(8-gamma) )+
                    (Weights[0 to 3]*
                           Neighbours[4 to 7][3]*gamma) )/8/8/8
         i++
        }
```

Code example 10: The RGB to CMYK conversion

Performance

The performance of the code is discussed in section 3.6.3.9, where the performance is compared to the MMX implementation.

### 3.6.3 MMX implementation

#### 3.6.3.1 Parallelability of the algorithm

**Parallel pixel processing**

Since no dependencies between the pixels exist, the basic RGB to CMYK conversion does not limit the number of pixels to process in parallel.

Lookup table

By using a lookup table however the number of pixel processed in parallel is limited to one, since the lookup operation can only be performed for one pixel. It is possible to create a lookup table in which multiple pixels can be looked up, but such a table would be very large.

Parallel interpolation

To perform the interpolation for multiple pixels in parallel, the pixels have to be packed manually, since they have to be looked up one at the time. Obviously, packing the pixels manually will decrease performance significantly. In the next two paragraphs

two other ways to implement parallelism in the algorithm will be discussed.

Four weights

Both methods perform the interpolation in parallel. A key factor in the choice of one of these methods is the way the weights are calculated. Obviously the four weights required (ignoring gamma) are calculated in parallel. The resulting weights are loaded in a register as shown in Figure 48 (the weights could be reverse ordered). The weights are numbered according to the angular point they are applied to.

| $(8-\alpha)(8-\beta)$ | $\alpha\,(8-\beta)$ | $(8-\alpha)\,\beta$ | $\alpha\,\beta$ |
|---|---|---|---|
| $=W_0$ | $=W_1$ | $=W_2$ | $=W_3$ |

Figure 48: Calculated weights in register

Re-ordering

Figure 49 shows the four color components of angular point zero loaded into a register[1]. Notice that these registers cannot be multiplied without some re-ordering because all the color components of a point have to multiplied with the same weight (in this case $W_0$). Since re-ordering a register requires quite an amount of computations, the choice on how to perform the interpolation in parallel primarily depends on the amount of re-ordering required.

| $C_0$ | $M_0$ | $Y_0$ | $K_0$ |
|---|---|---|---|

Figure 49: Loaded angular point in register

Four words in parallel

Before the two methods of parallel pixel processing can be discussed, the number of pixels calculated in parallel has to be determined. It might be tempting to try to interpolate eight pixels in parallel. However, this is not possible because the multiply operations required operate on packed words only. Therefore four words are multiplied in parallel,

### The four planes in parallel

Re-order four weights

The first method of parallel processing is to interpolate the four color planes in parallel. The neighbour points read from the lookup table are loaded in the right format (see Figure 49) but the
weights (see Figure 48) have to be re-ordered. Since four different weights are used[2], four registers have to be re-ordered. To be more precise; four weights have to be duplicated to four words.

After the weights have been duplicated correctly, they can be multiplied with the angular points. Finally the multiplied angular points can be added and divided without any re-ordering.

### Process four points in parallel

Re-order eight points

The second method processes the input of four angular points in parallel. An advantage of this method is that the weights do not have to be re-ordered in any way. The eight angular points however have to be re-ordered completely. They are read from the lookup table as shown in Figure 49, but in order to multiply

[1] The four components are extended to words because the multiplication can only be performed in packed words.
[2] Each is used for two of the eight angular points.

them correctly they have to be re-ordered to the format shown in Figure 50.

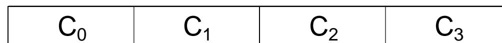


Figure 50: Format to process four pixels in parallel

After the multiplications have been performed, the input of the four angular points has to added. Since the packed add operation

Shift before addition

cannot add the four words straightway, they first have to be shifted to the format shown in Figure 51. Notice that this addition uses only one of the four datapaths available.

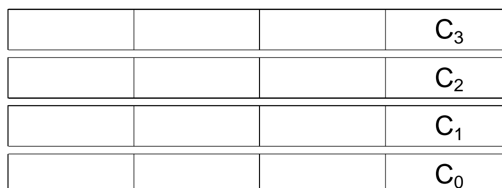


Figure 51: Format to add the pixels

**Four points or four planes ?**

Processing four *planes* in parallel requires four weights to be re-ordered. The angular points read from the lookup table do not have to be re-ordered. Processing four angular *points* in parallel requires eight angular points to be re-ordered. Additionally the results have to be shifted before the weighed points can be added.

Four planes in parallel

Obviously the first method requires less operations than the second. Therefore it has been chosen to process the four planes in parallel.

### 3.6.3.2 Optimal image traversing

Horizontally

In section 3.6.2.1 it can be read that the RGB to CMYK conversion sets no limitations on the way the image is traversed. Therefore it has been decided that the MMX implementation will traverse the image horizontally, since this direction uses the caches optimally[1].

Normally horizontal traversing means the image is traversed from the top-left to the down-right pixel of the image (each row from left to right). Such an implementation generally requires three registers;

1. A counter of the number of pixels to process (or the number of pixels processed).
2. One or more pointers to the position of the input array(s) currently processed.
3. One or more pointers to the location in the output array(s) where the calculated result has to be stored.

Since the RGB to CMYK conversion requires multiple pointers to source and destination registers, there are not enough registers available for such an implementation.

[1] See section 3.4.3.4.

Pixel addressing

Therefore a different way of addressing the pixels has been chosen. One register is used to count the number of pixels left to process[1], while the addresses of the input and output pixels are determined by adding the base address of the arrays to the register. Since the register is decremented the image array is processed backwards.

This way of indexed addressing is commonly used in C++ code. A disadvantage of this method is that the base address of the array has to be read from memory, but since this memory location is frequently accessed it generally will be loaded into the cache.

#### 3.6.3.3 Determining a, b and g

Now that it has been decided how to implement parallelism, the way to calculate the distances of the interpolated point to neighbour zero has to be determined. These three values ($\alpha$, $\beta$ and $\gamma$) are used to determine the weights of the points zero to three (ignoring gamma). These values are loaded into a register as shown in Figure 48.

Remainder

Alpha, beta and gamma are determined by taking the remainder of the division by eight of the R, G and B values.

#### 3.6.3.4 Absolute addresses of angular points

**Point zero**

Absolute address point zero

The next preparation necessary before the actual interpolation can be performed is the calculation of the absolute position of point zero. This is necessary because addressing a three dimensional array in assembly is not as easy as in C++. The addresses of the other angular points are calculated by adding an offset to the address of point zero.

2D array in memory

To understand the way this address is calculated, it is necessary to understand the way multidimensional arrays are placed in memory. Figure 52 shows how a two dimensional array A is placed in memory. Notice that the rightmost dimension is placed linear in the memory.

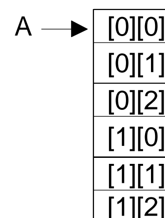



Figure 52: Memory for array A[2][3]

Base address

In [ 8 ] the calculation of the absolute position of an element in an m-dimensional array is shown. 'S' represents the size of the dimension, 'I' the index. A is the base address of the array.

[1] This way, after the register has been decremented, a jump-non-zero can be used. When counting the number of pixels already processed, the register has to be compared to the total number of pixels, after which a jump-equal can be used. This method is slower because an additional compare is necessary.

$$pos = A + I_{[0]} + \sum_{p=1}^{n-1} \left( \left( \prod_{q=1}^{p} S_{[q-1]} \right) \cdot I_{[p]} \right) \qquad [\ 8\ ]$$

The lookup table is a three dimensional array with three axis' of 33[1] elements. The position of element [x][y][z] can be calculated as shown in [ 9 ].

$$\begin{aligned} pos &= A + 4 \cdot \left( I_{[0]} + \sum_{p=1}^{n-1} \left( \left( \prod_{q=1}^{p} S_{[q-1]} \right) \cdot I_{[p]} \right) \right) \qquad [\ 9\ ] \\ &= A + 4 \cdot \left( I_{[0]} + \left( S_{[0]} \cdot I_{[1]} + S_{[1]} \cdot S_{[0]} \cdot I_{[2]} \right) \right) \\ &= A + 4 \cdot \left( z + 33 \cdot y + 33 \cdot 33 \cdot x \right) \end{aligned}$$

32 bit array

Notice that the calculated index value is multiplied by four. This is necessary because the lookup table consists of 32 bit doublewords.

**The offset of the other points**

Relative to point zero

The offset of the other angular points relative to point zero can be determined in the same way as the physical address of point zero is calculated. Figure 53 shows the offset of the angular points relative to point zero. Point two for example can be read from the memory location 4x33 bytes after point zero. The offset of point seven is (4x1+4x33+4x33x33).

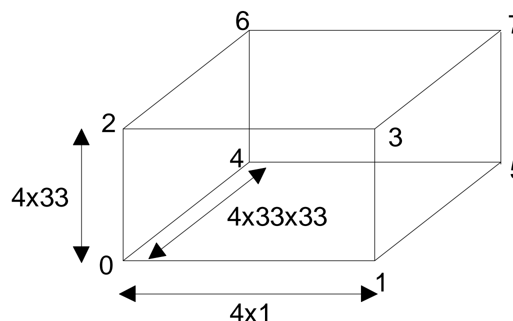



Figure 53: Physical addresses of angular points

### 3.6.3.5 Weighed addition of the angular points

The weighed addition of the angular points basically consists of six steps. The first four steps have to be performed for each angular point, the last two are performed once for all the points.

1. Duplicate the weight.
2. Read the angular point and extend the point to packed words.
3. Multiply the point with the weight.
4. Add the result of the multiplication to the intermediate result.
5. Multiply the (total) intermediate result with the appropriate gamma factor.
6. Add intermediate results and divide by a factor to compensate the too large fractions α, β and γ.

[1] One would expect 32 elements. The last is added for interpolation purposes.

**Duplicating the weight**

Duplicating the weight can be done straightforward. Code example 11 shows how. Notice that the other weights are lost. Therefore a copy of the original weights has to be made before starting the duplication.

```
                        ;mm4=[  W0   W1   W2   W3  ]
psllq  mm4,   16        ;mm4=[  W1   W2   W3   00  ]
psrlq  mm4,   48        ;mm4=[  00   00   00   W1  ]
movq   mm5,   mm4       ;mm5=[  00   00   00   W1  ]
psllq  mm5,   32        ;mm5=[  00   W1   00   00  ]
por    mm4,   mm5       ;mm4=[  00   W1   00   W1  ]
movq   mm5,   mm4       ;mm5=[  00   W1   00   W1  ]
psllq  mm5,   32        ;mm5=[  W1   00   W1   00  ]
por    mm4,   mm5       ;mm4=[  W1   W1   W1   W1  ]
```

Code example 11: Duplicating Weights

Two angular points

On page 55 it can be read that, when ignoring gamma, a weight can be used for two angular points. To minimise the number of registers used these two points are processed right after each other.

Duplicate via memory

An alternative method of duplicating the weight is to write the non-duplicated value four times to adjacent memory locations, after which the MMX register can be read from memory. This method has not been chosen because of the required memory bandwidth.

**Read the angular point**

Lookup table

After the weights have been duplicated the angular point has to be read from the lookup table. The way the address of the point is determined is described in section 3.6.3.4. After a point has been read from the calculated address, the unpack instruction (PUNPCKLBW) is used to extend it to packed words.

**Multiply with the weight**

The multiplication of the pixels with the weights is done with the PMULLW instruction. Theoretically, the multiplication of two 16 bit values normally requires 32 bits to store. Obviously four 32 bits values cannot be stored in a 64 bit MMX register. To solve this problem two multiply instructions are available: one storing the lower 16 bits of the result (PMULLLW), another storing the higher 16 bits (PMULHW). In our algorithm 16 bits should be sufficient to store all possible results[1].

**Add to intermediate result**

Two intermediate results

After the multiplication is performed, the result is added to the previously multiplied angular points. This way two intermediate results (for each color plane) are calculated; one that must be multiplied with gamma, another to be multiplied with 8-gamma.

But can this result always be stored in a 16 bit quantity ? The angular points read from the lookup table are eight bit values. Figure 54 shows the four weights. Notice that the net result when adding these weights always is 64, as shown in [ 10 ].

[1] The maximum values multiplied are 255 (pixel) and 64 ($\alpha$ and $\beta$ weight). The result (16320) can easily be stored in a 16 bits quantity.

| (8-α)(8-β) | α (8-β) | (8-α) β | α β |
|---|---|---|---|
| $=W_0$ | $=W_1$ | $=W_2$ | $=W_3$ |

Figure 54: Calculated weights in register

$$\begin{aligned} sum &= (8-\boldsymbol{a})(8-\boldsymbol{b}) + \boldsymbol{a}(8-\boldsymbol{b}) + (8-\boldsymbol{a})\boldsymbol{b} + \boldsymbol{ab} \\ &= 64 - 8\boldsymbol{b} - 8\boldsymbol{a} + \boldsymbol{ab} + 8\boldsymbol{a} - \boldsymbol{ab} + 8\boldsymbol{b} - \boldsymbol{ab} + \boldsymbol{ab} \\ &= 64 \end{aligned} \quad [\ 10\ ]$$

15 bit value

Since 64 is a seven bit number the result of the weighed addition of four angular points can be stored in a 15 bit quantity.

**Multiply with gamma**

When multiplying the two intermediate results (each produced by weighed adding four angular points), these 15 bit values are multiplied with a four bit gamma (ranging from zero to eight).

19 bit value

This results in a 19 bit value. Since the register contains (16 bit) packed words the result will saturate, causing severe rounding errors. To avoid this, the intermediate result is shifted[1] three[2] bits to the right before multiplying with gamma. This results in a 16 bit value.

**Add and divide**

17 bit value

Finally, the two intermediate results have to be added and divided. The addition of the two 16 bits values would result in a 17 bit value, causing saturation. Therefore the intermediate results are shifted one[3] bit to the right before the addition is performed.

Now all the weighed angular points have been added, the sum would have to be divided three by times by eight (/8/8/8). This would be done by shifting three times three bits to the right (=9 bits). Since the value already has been shifted four bits to the right, the sum is shifted five[4] bits to the right.

**Rounding errors**

During the interpolation the divisions cause rounding errors. Three divisions are performed:

1. The two intermediate results are divided by eight before multiplying them with gamma.
2. After they have been multiplied they are divided by two before the are added.
3. After the have been added the result is divided by 32.

When dividing by eight the maximum rounding error is 7[(5)]. After this, these errors (in the two intermediate results) are multiplied with γ respectively (8-γ) resulting in the errors shown in [ 11 ] and [ 12 ].

---

[1] Shift right arithmetic has to be used. If shift right logical is used incorrect results will be produced (see page 47).
[2] Equivalent to a division by eight.
[3] Equivalent to a division by two.
[4] Equivalent to a division by 32.
[5] Fractions are *always* rounded down.

$$\Delta_1 \leq \Delta_1 \times g_{max} \leq 7 \times 7 \leq 49 \qquad [\ 11\ ]$$

$$\Delta_2 \leq \Delta_2 \times (8 - g_{min}) \leq 7 \times 8 \leq 56 \qquad [\ 12\ ]$$

Next, a division by two is performed to prevent saturation when adding the two intermediate values. Since this division also rounds the value, the error values in [ 13 ] and [ 14 ] are rounded up.

$$\Delta_1 \leq \left\lceil \frac{\Delta_1}{2} \right\rceil \leq \left\lceil \frac{49}{2} \right\rceil \leq 25 \qquad [\ 13\ ]$$

$$\Delta_2 \leq \left\lceil \frac{\Delta_2}{2} \right\rceil \leq \left\lceil \frac{56}{2} \right\rceil \leq 28 \qquad [\ 14\ ]$$

Finally, the two intermediate results are added and divided by 32.

$$\Delta \leq \left\lceil \frac{\Delta_1 + \Delta_2}{32} \right\rceil \leq \left\lceil \frac{25+28}{32} \right\rceil \leq 2 \qquad [\ 15\ ]$$

Rounding error

These rounding errors are considered acceptable.

**Cycle times of the multiply**

Pipelining

Interleaving

The multiply operation normally takes three cycles to complete. When pairing the code the time necessary for this operation can be reduced in two ways. First multiplies can be pipelined. This means each cycle one multiply operation is started and one is completed. Second, by moving two pairs of non-related statements between the multiplication and the instruction that uses the result of the multiplication, two of the three cycles needed by the multiply unit can be used to perform other actions, as shown in Code example 12.

```
pmullw mm0,   mm1    <--+
                        |
pand   mm3,   mm4       |
por    mm5,   mm6       |
                        |
mov    mm4,   mm3       |
mov    mm5,   mm6       |
                        |
padd   mm7,   mm0    <--+
```

Code example 12: Adding non-related statements

## 3.6.3.6 Reading and writing the pixels

**Reading the source pixels**

Base pointer

In section 3.6.3.2 it can be seen that pixels are read from the input arrays by means of the base pointer to the array and an index register.

The base pointer to the array is stored at a memory location, similar to common C++ code. The required memory space is allocated by declaring a pointer variable in the C++ section of

the filter implementation. This variable is declared ‘static’ to ensure that it is aligned correctly[1].

Addressing mode

The pixel cannot be read by simply adding the declared variable and the index register, since there is no such addressing mode. Therefore the variable containing the base address of the array is first loaded into a general purpose register, after which the pixel can be read.

Because the memory location where the base address is stored is accessed each loop cycle, it will be loaded into the level one cache. Therefore reading the base address takes only one cycle. Reading the pixel with the more complex addressing mode does not take any extra cycles, so this way of reading the source pixels does not decreases performance significantly.

**Writing the resulting pixels**

Write to *source* array

After the interpolation is completed the calculated pixels have to be written to the destination arrays. Similar to the C++ version (see section 3.6.2.4) the three source arrays are used to store the calculated pixels. The pixels are addressed in the same way as the source pixels.

A problem is caused by the fact that the four color components, which have to be written to four separate arrays, are loaded in a MMX register as packed words in the format shown in Figure 55.

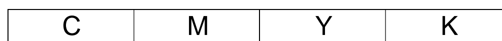



Figure 55: Register format after the interpolation

Write *one* byte

Because the calculated values are written over the *source* pixels, only *one* byte must be written. This is only possible by using a general purpose register. The data is copied to this register with the MOVD instruction. Since this instruction only copies the least significant 32 bits, the MMX register has to be shifted to copy the right color component.
After the 32 bits have been copied into the general purpose register, the least significant byte can be written to memory.

Code example 13 shows how the Yellow component is saved to ArrayB[esi]. Notice that the least significant byte of eax is moved by applying a conventional move instruction to register AL. Notice also that the blacK component is lost when shifting the MMX register 16 bits to the right, so color components have to be saved from right to left.

```
                                ;mm1=[  C    M    Y    K  ]
pslrq  mm1,   16                ;mm1=[  0    C    M    Y  ]
movd   eax,   mm1               ;eax=[  M    Y  ]
mov    edi,   ArrayB
mov    [edi+esi], al                    ;ArrayB[esi]=Y
```

Code example 13: Writing the Yellow component

[1] Static variables are aligned automatically by the compiler to the appropriate boundary. For more information on data alignment see page 15.

#### 3.6.3.7 Border pixels

The C++ implementation processes all pixels, since all of them contain valuable information. The MMX version of the processing

White borders

path however, has an eight pixel wide, white border[1] on both edges (see section 3.3). Since a white pixel will result in a CMYK value with all components zero, these pixels do not have to be interpolated. Because most of the time required to convert a pixel is needed for the interpolation, it might be a good idea not to interpolate these border pixels.

Let us take a look at the amount of pixels skipped this way. An A4 image scanned at 200 dpi has a width of 1580 pixels. After adding two, eight pixel wide borders, the image width is 1596

Borders are 1%

pixels. The 16 pixels only make one percent of this[2].

The performance gain possible when not interpolating these pixels will be less than one percent because the memory accesses required to read and write the *pixels* are still necessary[3].

Additional overhead

The overhead caused by the added complexity of the loop structure will be:

- Two loop counter registers are required instead of one. This makes the usage of the loop counter to address the pixels much more complex (and therefore costs time).
- One or more jumps will be required. Since these jumps will be hard to predict for the Pentium's branch prediction they will consume many cycles.

Because this will probably cost more cycles than the performance gain possible, it has been decided not to skip the border pixels.

#### 3.6.3.8 The final MMX implementation

Code example 14 shows the final structure of the RGB to CMYK color conversion in pseudocode.

The functions, such as "DuplicateWeight()", represent a number of actions. For the sake of clarity they are represented through a function call. In the implementation however they are implemented in-line with a number of statements.

---

[1] The border added to extend the image width to a multiple of eight is not always white, and can therefore not be skipped.
[2] 16/1596*100 = 1,0%
[3] After the conversion has been implemented it turned out that 91% of the processing time is needed for the interpolation (and preparations for it; see page 68). From this it can be concluded that skipping the border pixels can maximally result in a performance gain of 0.9%.

```
Static PIXEL *ArrayR, ArrayG, ArrayB, ArrayK // Ptrs to arrays

esi=Width*Height
Do While (esi >= 0)
       {
                                 // Alpha and beta weights
        mm0=CalcWeights(ArrayR[esi],ArrayG[esi])
        eax=CalcPosOfPoint0(ArrayR[esi],ArrayG[esi],
                                 ArrayB[esi], LTable)

        mm2=mm0
        DuplicateWeight(mm2,W0)
        mm6=ReadPoint(eax,0)     // Point 0
        mm6=Pmul(mm6,mm2)        // Interm. result 1 in mm6
        mm1=ReadPoint(eax,4)     // Point 4
        mm1=Pmul(mm1,mm2)        // Interm. result 2 in mm1

        mm2=mm0
        DuplicateWeight(mm2,W1)
        mm5=ReadPoint(eax,1)     // Point 1
        mm5=Pmul(mm5,mm2)
        mm6=mm6+mm5              // Add to result 1
        mm3=ReadPoint(eax,5)     // Point 5
        mm3=Pmul(mm3,mm2)
        mm1=mm1+mm3              // Add to result 2

        mm2=mm0
        DuplicateWeight(mm2,W2)
        mm5=ReadPoint(eax,2)     // Point 2
        mm5=Pmul(mm5,mm2)
        mm6=mm6+mm5              // Add to result 1
        mm3=ReadPoint(eax,6)     // Point 6
        mm3=Pmul(mm3,mm2)
        mm1=mm1+mm3              // Add to result 2

        DuplicateWeight(mm0,W3)
        mm5=ReadPoint(eax,3)     // Point 3
        mm5=Pmul(mm5,mm0)
        mm6=mm6+mm5              // Add to result 1
        mm3=ReadPoint(eax,7)     // Point 7
        mm3=Pmul(mm3,mm0)
        mm1=mm1+mm3              // Add to result 2

        ebx=CalcGamma(ArrayB[esi]
        mm5=Duplicate(ebx)
        mm1=ShiftRightArithmetic(mm1,3)
        mm1=mm1*mm5              // Result 2 * gamma

        mm5=8-mm5
        mm6=ShiftRightArithmetic(mm6,3)
        mm6=mm6*mm5              // Result 1 * (8-gamma)

        mm1=ShiftRightArithmetic(mm1,1)
        mm6=ShiftRightArithmetic(mm6,1)
        mm1=mm1+mm6              // Add interm. results
        mm1=ShiftRightArithmetic(mm1,5)

        WritePixels(mm1, ArrayR[esi],
                          ArrayG[esi],
                          ArrayB[esi],
                          ArrayK[esi])

        esi--                    // Next pixel
       }
```

Code example 14: MMX version of the conversion

### 3.6.3.9 Filter performance

40% faster by pairing

After the code has been paired the performance is compared to the performance of the unpaired version. Table 9 shows that a performance improvement of about 40% is achieved.

Notice that this is more than for the kernel operations (31%). This is primarily caused by the fact that the interpolation requires many register-register operations. Code containing many register-register operations will benefit more from pairing than code with many register-memory operations, because two operations that access memory cannot pair.

| Image size | Cycle times | | Improvement |
|---|---|---|---|
| | unpaired | paired | |
| (pixels) | (cycles/pixel) | (cycles/pixel) | (%) |
| 290x509 | 208 | 125 | 39,9 |
| 1580x2176 | 201 | 123 | 38,8 |

Table 9: performance gain by pairing

Table 10 lists the performance of the C++ and the paired MMX version of the RGB to CMYK conversion. The two images of 1580x2176 pixels are A4 images scanned at 200 dpi.

MMX 5x faster than C++

For those images the MMX implementation of the conversion is 5.2 times faster than the C++ version. The performance gain is not only reached by processing four data elements in parallel, but also by using a much more efficient multiplier[1].

| Image size | Cycle times | | Improvement | |
|---|---|---|---|---|
| | C++ version | MMX version | | |
| (pixels) | (cycles/pixel) | (cycles/pixel) | (%) | (factor) |
| 87x16 | 628 | 130 | 79,3 | 4,8 |
| 63x96 | 639 | 143 | 77,6 | 4,5 |
| 127x96 | 642 | 133 | 79,3 | 4,8 |
| 255x320 | 640 | 125 | 80,5 | 5,1 |
| 290x509 | 641 | 125 | 80,5 | 5,1 |
| 1580x32 | 639 | 118 | 81,5 | 5,4 |
| 1580x2176 (1) | 646 | 123 | 81,0 | 5,3 |
| 1580x2176 (2) | 646 | 124 | 80,8 | 5,2 |

Table 10: Performance of the RGB to CMYK conversion

#### 3.6.3.10 Categorising the processing time

Figure 56 shows the processing time of the paired version divided into the categories discussed in the previous sections. The cycle times are counted as described in section 3.4.4.8. Notice that

50% for weighed addition

almost 50% of the processing time is used to weighed-add the angular points.

Only 8% to write pixels

Notice also that only eight percent of the processing time is used to write the pixels, although the unpacking operations require quite an amount of computations. This is primarily caused by the fact that the memory locations the pixels are written to are still loaded into the level one cache (the source pixels are read from the same location).

[1] The conventional multiplier takes 10 cycles to complete. The MMX multiplier takes three but an average throughput of one per cycle can be reached by pipelining the multiplies.

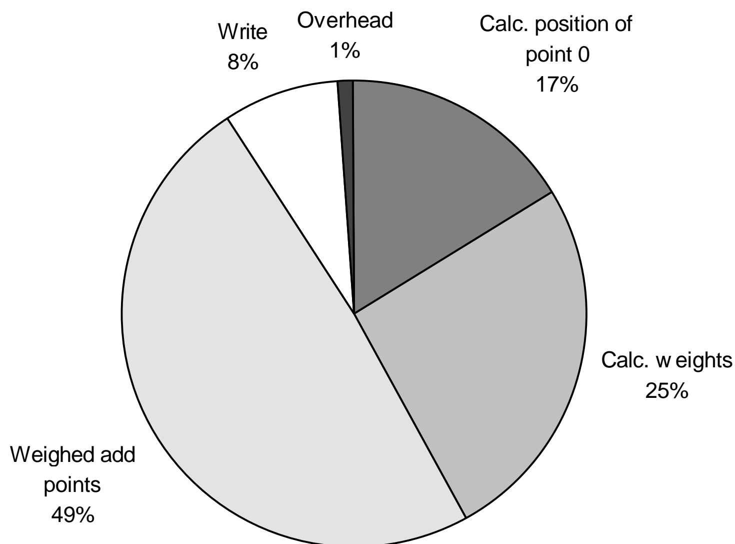



Figure 56: Categorised processing time of the entire RGB to CMYK conversion

In Figure 56 it can be seen that 17% of the processing time is used to calculate the position of neighbour zero. During this calculation, the R, G and B color components can be added. By testing the result of this addition, it can be checked if the pixel is has a certain color. Generally, scans contain a lot of white pixels -for example around text-, so by testing if a pixel is white, a fixed value[1] can be written to the output array, instead of performing the interpolation. This is called 'white detection'.

White detection

Since writing the CMYK value to the output array also takes eight percent of the processing time, a white pixel could be written in 25% percent of the time needed to calculate another pixel.
The actual performance gain will be less because of the additional overhead required. Especially the required jump will take many cycles. This overhead will cause the performance to decrease for images containing no white pixels.

It has been decided not to apply white detection because then the performance would depend on the test image used to measure the performance. Besides this, the peak performance of an implementation using white detection can be predicted quite
accurately to be a factor three to four times the performance of the implemented algorithm. When processing images containing no white pixels the performance will decrease slightly.

Peak: 4x faster

The white detection has no influence on the proportion of the performance of the C++ implementation and the MMX implementation. The MMX implementation has no advantages over the C++ version because both the calculation of point

[1] For a white pixels all the CMYK components will have to be made zero.

zeroes position and writing the calculated pixels to the arrays is done in conventional assembly code.

Figure 57 shows the processing time needed to weighed add two angular points (point five and one). This represents the “Weighed add points” section in Figure 56.

Duplicate weight 41%

Notice that duplicating the weights costs 41% of the processing time required to weighed add a point. With this amount of time in mind it might be a good idea to duplicate the weight by writing the individual value four times to memory, as discussed on page 61. If duplicating the weights would be entirely omitted the weighed addition of an angular points would be 41% faster. Since the weighed addition of the angular points takes 49% of the total processing time (see Figure 56), this would improve the overall performance with 20%[(1)].

Duplicate via memory

When duplicating the weights via the memory, the calculated weights first have to be unpacked[2]. After this, they have to be moved four times to the memory. Obviously this requires some processing time, but remember that the memory locations were the weights are stored will generally be loaded into the level one cache. Therefore it is reasonable to expect a performance gain of about ten percent.

About 10% faster

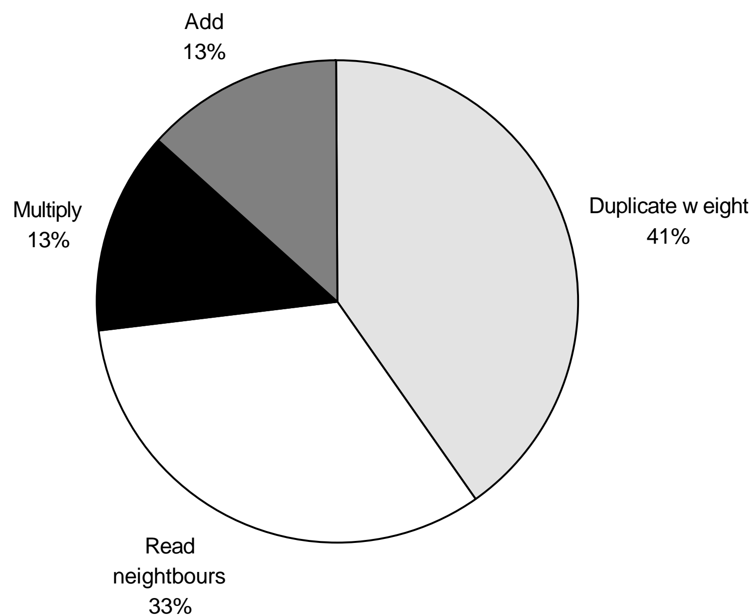



Figure 57: Categorised processing time of the weighed add section

17% arithmetic

Figure 58 shows the processing time categorised according to the instruction type. The amount of time required for arithmetic operations is surprisingly low; only 17%. This is, among others, caused by the fact that the multiplies are pipelined and interleaved with other instructions.

20% shift

The reason the number of shift operations is relatively high (20%) is that duplicating the weights, as well as (un)packing, requires many shift operations.

---

[1] 0.41x0.49 = 0.20

[2] The package format is shown in Figure 48.

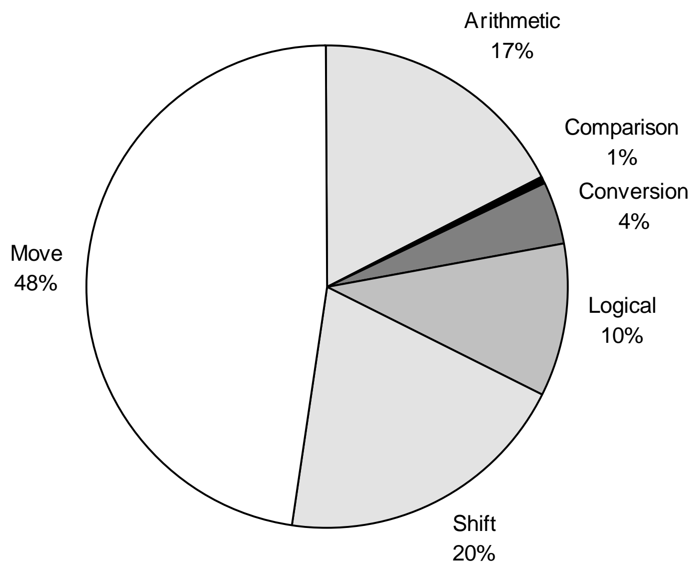



Figure 58: Processing time per instruction type

## 3.7 The halftoning algorithm

### 3.7.1 The algorithm

Decrease color depth

Halftoning is used in every printing device to decrease the number of colors in the image to a number of colors printable by the print engine. This is necessary because a single-color pixel printed by a print engine can only have two values, on ('1') or off ('0')[1]. Since print engines normally use four base colors (CMYK [2]), a total of nine[3] different colors can be printed.

CMYK color planes

The pixels in an image are normally represented with 3*8 bits in the RGB color representation, allowing 16 million[4] colors. Before applying the halftoning algorithm, the RGB image is converted to four separate CMYK color planes, after which the individual planes are halftoned.

Bitwise pixels

Note that because the halftoned pixels in a single color plane can only have two values a single bit is sufficient for each pixel. Because the pixels in the source image -with separated color planes- are represented with eight bits, generally the halftoning generates pixels which are either 0 (all bits '0') or 255 (all bits '1'), after which only one bit of the calculated pixel value is written to the destination image.

---

[1] Analog print engines can print gray scale pixels but the image processing discussed in this report is designed for digital print engines.
[2] Cyan, Magenta, Yellow and Black. See section 2.1.2.2 for more information on color representation.
[3] Cyan, Magenta and Yellow allow $2^3$=8 different colors. Black is only used to print black pixels, resulting in a total of nine different colors.
[4] $2^{(8+8+8)}$ = 16,777,216

Variations

Halftoning algorithms with varying image quality and algorithm complexity exist. In the following sections some of them are shortly discussed. To compare the quality of the processed images[1], a (256 gray scale) test image called 'Lena' is printed[2] in Figure 59 at 28% of its original size. This image is used to compare the image quality of the various halftoning algorithms.

Figure 59: Original 'Lena'

### 3.7.1.1 Thresholding

Fixed threshold

Thresholding is the most basic form of halftoning. Each pixel in each of the separated color planes is compared to a fixed threshold; if the pixel's value exceeds the threshold the pixel is rounded to 255 (=on='1'), otherwise it is made 0 (=off='0').

In Figure 60 the thresholded 'Lena' is printed. A threshold of 128 has been used.

Figure 60: Thresholded 'Lena'

Figure 60 shows that thresholding gives reasonable quality for black text, but unacceptable quality for pictures. For this reason, thresholding is not often used.

### 3.7.1.2 Dithering

Dither matrix

Dithering is a halftoning variant where the threshold depends on the pixel's position. The threshold is read from the dither matrix, which is layed-over the image like shown in Figure 61.

[1] Because 24-bit color images are separated into four color planes, the individual planes can be thought to be 256 gray scales images.

[2] Because the printer this report is printed on also halftones the image, this is not the exact original. The image quality however is sufficient to compare the halftone algorithms.

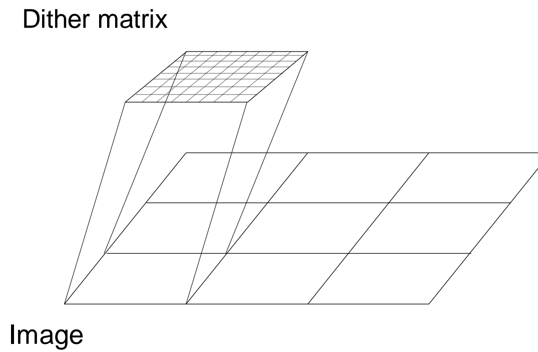



Figure 61: Dithering

In Figure 62 an 8x8 dither matrix is shown.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 251 | 235 | 187 | 155 | 123 | 91 | 59 | 51 |
| 2 | 243 | 227 | 179 | 135 | 115 | 83 | 43 | 35 |
| 3 | 219 | 211 | 171 | 115 | 107 | 75 | 27 | 11 |
| 4 | 203 | 195 | 163 | 100 | 99 | 67 | 19 | 3 |
| 5 | 123 | 91 | 59 | 51 | 251 | 235 | 187 | 155 |
| 6 | 115 | 83 | 43 | 35 | 243 | 227 | 179 | 135 |
| 7 | 107 | 75 | 27 | 11 | 219 | 211 | 171 | 115 |
| 8 | 99 | 67 | 19 | 3 | 203 | 195 | 163 | 100 |

Figure 62: An 8x8 dither matrix

In Figure 63 the dithered ‘Lena’ is shown.

Figure 63: Dithered 'Lena'

For photos dithering gives acceptable results. When dithering text however, the edges of the letters are frayed as shown in Figure 64. Because this is very disturbing when reading the text, pure dithering is not often used for images containing text.

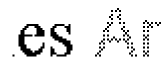



Figure 64: Dithered text

### 3.7.1.3 Error diffusion

Error diffusion uses a fixed threshold, similar to thresholding. The difference is that with error diffusion the error made when rounding a pixel is passed to the neighbouring pixels.

The error is calculated by subtracting the new pixel value from the original pixel value. When calculating the black pixel in Figure 65

Error passing

the error is added to four of the neighbouring pixels with the factors shown. The gray pixels are processed previously, the black pixel is the one currently processed.

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | | 7/16 | |
| | 3/16 | 5/16 | 1/16 | |
| | | | | |

Figure 65: Floyd-Steinberg error passing

“Floyd-Steinberg”

This technique of passing the error called the “Floyd-Steinberg” algorithm. In Figure 66 the error diffused ‘Lena’ is shown.

Figure 66: Error diffused 'Lena'

‘Worms’

Figure 67 shows an error diffused gray plane (real size). Notice the little ‘worms’. These are created when applying an error passing algorithm to very light or very dark planes.

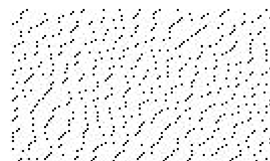


Figure 67: Error diffused gray plane

Good quality

Because error diffusion produces good results for text and for (most) photos, this is a much used halftoning variant. A disadvantage of error diffusion are the ‘worms’ that sometimes are created in images containing large areas of the same color.

#### 3.7.1.4 Other variants

In the previous sections the most common halftoning algorithms are discussed. Because most algorithms produce optimal results only for certain image types (text or photos), mixtures of the discussed variants are used. These mixtures commonly use dithering for photos and error diffusion for text areas.

Implement error diffusion

It is expected that when implementing an algorithm, passing the error to the neighbouring pixels will cause most problems. For this reason it has been chosen to implement the error diffusion algorithm. Based on the problems encountered there, an

indication of the difficulties expected when implementing other algorithms can be given.

An implementation of an error dithering algorithm with MMX or C++ will only differ from error diffusion in the way the threshold is determined. A mixture of error diffusion and dithering will probably be more complex because switching between the two algorithm types is difficult to implement.

### 3.7.2 C++ implementation

#### 3.7.2.1 Traversing the image array

Similar to the algorithms previously implemented, the first question to ask when implementing the error diffusion algorithm is how to traverse the image array; horizontally, vertically or diagonally.

No re-usage

When taking a closer look at the algorithm it is noticed that it is not possible to re-use intermediate results as with the smooth and the sharpen algorithm. Only the error passing algorithm might prefer some way of traversing the image.

Figure 68 shows the way the error is passed to the four neighbour pixels; the black pixel is the one currently processed.

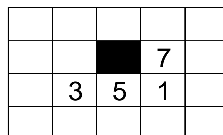



Figure 68: Floyd-Steinberg error passing

From Figure 68 it can be concluded that two ways of traversing the image array are possible; horizontally from left to right or diagonally from the left top to the right bottom. In both situations one of the error values has to be passed to the next pixel processed. The other error values are needed when processing the next row or (diagonal) column[1].

Horizontal traversing

In section 3.4.2.1 it was concluded that optimal caching can be achieved when traversing the image horizontally. Additionally, a loop that traverses an image diagonally, is more difficult to implement than a loop that traverses an image horizontally[2]. Therefore it is decided to process the image array horizontally.

#### 3.7.2.2 The basic inner loop structure

Basically error diffusing one pixel consists of the following steps:
1. Receive the error from other pixels
2. Determine the new pixel value
3. Pass the error to other pixels

Step one and three are described in section 3.7.2.3.

[1] Lacking a correct name diagonal column is used.
[2] Vertical image traversing would be possible when using other Floyd-Steinberg error passing algorithms. The variant shown in Figure 68 has been chosen because it is most often used. Besides this, vertical image traversing does not offer any advantages over horizontal image traversing (for as far as the implementation is concerned).

Determining the new pixel value is not complex; after the error passed from other pixels is added, the pixel is compared to the threshold. If the pixel is smaller the new pixel value is zero, otherwise it is 255. After the new pixel value has been determined the error can be determined by subtracting the new pixel value from the old one.

#### 3.7.2.3 Passing the error

When processing the image horizontally, the error weighed 7/16 has to be added to the next pixel processed. The other three error values have to be stored in an array for later usage.

Array

Since the image is traversed horizontally, the array where the error values are stored must have enough positions to store errors for an entire row of the image. Figure 69 shows how this array is filled.

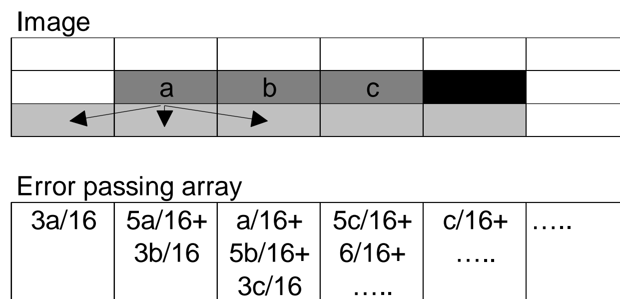



Figure 69: Using an error passing array

In Figure 69 the way the errors have to be passed from the three dark-grayed pixels to the light grayed pixels is shown. The black pixel, which is processed the next loop cycle, receives an error from pixel c. This error value can be stored in a local variable, the light gray errors have to be stored in the error passing array. As shown in Figure 69 the middle of the light gray pixels receives

Receive three errors

three errors passed from above. Only edge pixels receive less error values, all the other pixels in an image receive three error values from above (and one from the right).

It has been chosen to store the errors which have to be passed to the lower row in the array. It is also possible to store the rounding errors that have been made in the array and calculate the error to pass to the lower pixels later. This way, passing the error requires less calculations to store the error values but more to calculate the error to add to a pixel. For the entire image the total number of calculations is the same. Both implementations benefit from the caches in the same way, so any one of them can be chosen. As mentioned before it has been chosen to use the error passing array in the way shown in Figure 69.

When storing the error values in the array, care has to be taken not to overwrite values needed later.

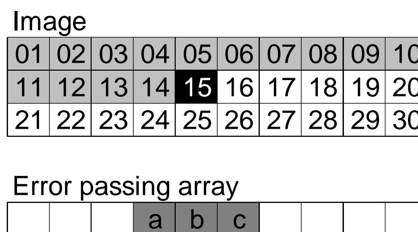



Figure 70: Error passing via array

Suppose (in Figure 70) pixel 15 is currently processed. After the error has been determined, the weighed error has to be saved to the error passing array in order to pass it to the pixels 24, 25 and 26. Note that the positions a and b in the array contain errors passed previously from the pixels 13 and 14. The error made with pixel 15 has to be added to those errors. Position c of the error passing array contains the error that has to be passed to pixel 16 in the next loop cycle. Unfortunately this position is also needed

Overwrite next error

for the error passed from pixel 15 to 26. This problem has been solved by reading position c from the error passing array into a variable called "NextError", after which the error to pass to pixel 26 can be written to the array.

Negative error values

Now the basic way of passing the error is designed the data type for the positions in the error passing array have to be chosen. The pixels in the image are written as unsigned bytes, so it would be obvious to create an error passing array of this type. Because negative error values are also possible a sign bit is required. Since it is not possible to multiply the unsigned bytes read from the source image with the signed bytes, both the pixels and the error values have to be extended to signed words. An additional advantage of this extension is that rounding errors are avoided, because the division by sixteen can be delayed until the addition of the error value and the pixel value read from the image array is performed. A disadvantage of the extension is that instead of processing eight pixels per loop cycle, now only four pixels are processed.

Because the top row of the image does not receive error values from above, the error passing array has to be cleared before the actual filter loop is started. This is necessary because C++ does not clear storage space when allocated by an application.

#### 3.7.2.4 Border pixels

The Floyd-Steinberg error passing shown in Figure 68 requires special attention for border pixels, since either the left or the right neighbour pixel is missing.

Skip borders

To avoid problems, a special error passing algorithm can be implemented for border pixels, or the border pixels can be skipped entirely when processing the image. Since applying different error passing mechanisms to border pixels requires four additional error passing versions, it has been decided not to process border pixels.

#### 3.7.2.5 Writing the resulting pixels

Error diffusion produces pixels which are either zero or 255. Their values can therefore be represented with one bit.

Compared to the eight bit (= one byte) pixels read from the input file the one bit data requires much less memory bandwidth. For this reason it is

Bitwise

common practice to write the halftoned pixels as bits[1] instead of bytes.

Collecting bits

Since bitwise addressing is not possible, eight bits have to be collected, after which they are written as a one byte to the destination array.
If it was chosen not to traverse the image horizontally in section 3.7.2.1 this way of storing the bits would re-order the pixels.
If the image would be traversed vertically for example, the resulting bits (which are collected into one byte) are automatically written horizontally. Thus the pixels would be rotated 90° !

Collecting the bits into a byte is implemented as shown in Code example 15.

```
byte PackedBits = 0       // Initialisation
byte Temp
int  BitCounter = 0

Do While (…………)          // The loop...
       {
        Temp = GetPixel()
                          // Store bit
        Temp = And(Temp,0x01)
        PackedBits = Or(Temp,PackedBits)
        ShiftLeft(PackedBits,1)
        BitCounter++
        If (BitCounter==8) Then
              {
               StoreByte(PackedBits)
               PackedBits = 0
               BitCounter = 0
              }
       }
```

Code example 15: Collecting bits into a byte

Skipping border pixels

Caution has to be taken when skipping the border pixels. When storing the pixels as bytes a single byte can be skipped by simply incrementing the index to the array by one. However when writing the pixels as bits it is not sufficient to increment the BitCounter. The previously packed bits have to be shifted one bit to the left too. Additionally, if BitCounter has reached the value eight, the byte containing the packed bits has to be stored.

Since a pixel value only depends on the value of one source pixel and the error passed from other pixels, it is possible to overwrite

Overwrite source pixels

the source pixels with the calculated values[2]. Because the memory location is loaded into the cache when reading the source pixel this could reduce the time necessary to write the pixel. However since the resulting pixels are written as bits, the calculated pixels require eight times less storage space than the original pixels. This causes the pointer to the currently processed

[1] Obviously, when halftoning to more than one level more bits are required. Halftoning to, for example four levels produces pixels which can be represented with two bits.
[2] For the smooth and the sharpen algorithm this is not possible because a source pixel is input for nine pixel calculations.

source pixel to 'run away' from the pointer to the storage space for the calculated value, eliminating the caching advantage. Therefore it has been decided to store the calculated values in a separate array.

#### 3.7.2.6 The final C++ implementation

The complete C++ implementation of the error diffusion algorithm is shown in Code example 16 (in pseudocode).

```
Clear(ErrorArray)
BitCounter = 0
PackedBits = 0

RowCounter = 1
Do While (RowCounter < Height-1)
       {
        SkipLeftPixel()
        ColCounter = 1
        NextError = ErrorArray[ColCounter]    // Error from above
        Do While (ColCounter < Width-1)
              {
                                              // Get input value
               Old_Pixel = Src.Pixels[RowCounter,ColCounter]
               Old_Pixel = Pixel + (NextError/16)
                                              // Get new pixel
               If (Old_Pixel < Treshold) Then
                      New_Pixel=0
               Else
                      New_Pixel=255
               PackBits()
               If (BitCounter==8) Then StoreBits()
                                              // Error passing
               Error=Old_Pixel-New_Pixel
               NextError= (7*Error) + ErrorArray[ColCounter+1]
               ErrorArray[ColCounter+1]=1*Error
               ErrorArray[ColCounter] +=5*Error
               ErrorArray[ColCounter] +=3*Error

               ColCounter++
              }
        SkipRightPixel()
        RowCounter ++
       }
```

Code example 16: The final C++ implementation

Nested loops

It has been chosen to implement two nested loops because the border pixels require special attention. The functions SkipLeftPixel() and SkipRightPixel() represent the actions necessary to skip the border pixels. The PackBits() and StoreBits() functions[1] pack the bits into a byte respectively stores the collected bits. All four functions are described in section 3.7.2.5.

### 3.7.3 MMX implementation

#### 3.7.3.1 Parallelability of the algorithm

The parallelability of the error diffusion algorithm is limited by the way the error is passed to the neighbour pixels.
In Figure 71 the way the error is spread across the neighbours is shown.

[1] In the C++ implementation these sub-functions are not used because of the extra overhead required; all the code is placed in one function.

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | ■ | 7 | |
| | 3 | 5 | 1 | |
| | | | | |

Figure 71: Floyd-Steinberg error passing

Pixel dependencies

Obviously, pixels that receive an error from other pixels can not be processed before the error is known. These dependencies highly limit the possibilities for parallel pixel processing. In Figure 72 the dependencies are shown. The pixel currently processed is black. This pixel receives errors from the light gray pixels. The dark gray pixels are influenced by the error passed from the black pixel.

| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 |

Figure 72: Pixel dependencies

From Figure 72 it can be concluded that it is not possible to process a number of pixels placed next to each other, as with for example the smooth algorithm. Worse, it seems to be impossible to process pixels in parallel. However, there are two ways to process pixels in parallel; skewing and tiling. In the next two paragraphs these two alternatives will be compared.

Horizontal traversing

Both ways can be implemented with horizontal as well as diagonal image traversing. Since diagonal image traversing requires a more complex loop structure, and gives no advantage over horizontal traversing[1], it has been chosen to traverse the image horizontally.

**Skewing**

Independent pixels

Basically skewing avoids the dependencies between pixels by processing independent pixels. Figure 73 shows the dependencies for a pixel in the middle of the image.

| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 |

Figure 73: Dependencies

The dependencies are limited to the areas to the upper left and lower right of the pixel processed. The pixels in the light gray area pass an error *to* the pixel, the pixels in the dark gray area receive an error *from* the pixel.

From Figure 73 it can be concluded that parallel pixel processing is possible for pixel 12, 30 48 and so on. This is shown in Figure 74.

[1] See section 3.7.2.1.

| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 |
| 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 |

Figure 74: Skewed processing

A disadvantage of skewed processing is that the pixels to be processed can not be read into an MMX register with one memory read, as done for example with the smooth algorithm.

Manual packing

The pixels have to be read one at a time, after which they can be packed manually into an MMX register to allow parallel processing.

Figure 75 shows the required error passing when processing a number of pixels skewed. The black pixels are the pixels currently processed. The resulting error values a, b and c have to be passed as shown.

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | a | 7a/16 |
| | | | b | 3a/16+ 7b/16 | 5a/16 | a/16 |
| | c | 3b/16+ 7c/16 | 5b/16 | b/16 | | |
| 3c/16 | 5c/16 | c/16 | | | | |

Figure 75: Error passing with skewing

Notice that, when traversing the image array horizontally from left to right, some of the error values are required for the next pixel processed (dark gray pixels). The errors passed to the light gray pixels are required in the two next loop cycles. These three groups of error values can be stored in register, instead of an array to improve performance. The errors passed to the lower row cannot, because they have to be stored until the next row is processed.

The array required to store these error values can be implemented in a similar way as the error passing array of the C++ version of the error diffusion. In section 3.7.2.3 it was concluded that it was necessary for the C++ implementation to use signed words for the error passing array, and to extend the pixels from unsigned bytes to signed words.

Four pixels/loop cycle

For a skewed MMX implementation this is also required. This implies that each loop cycle only four pixels can be processed.

Now let us take a look at the number of memory accesses required to pass the error values when processing four pixels in a skewed implementation (see Figure 76). Assuming that the error passing array contains the error value received by a pixel (instead of the error value to pass from a pixel[1]), one memory read is required to receive an error from above to add to pixel a.

[1] In section 3.7.2.3 it was concluded that both ways of storing the errors require the same amount of computations.

One memory operation

The error passed from pixel d to pixel a of the lower row has to be stored with three memory operations; two memory adds, one memory move. Thus, four memory operations are required to process four pixels (one per pixel).

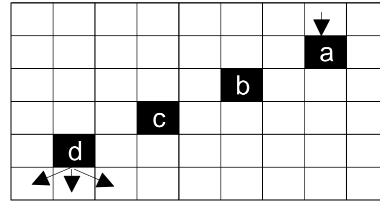


Figure 76: Error passing via array

A disadvantage of skewing is that seven of the border pixels can not be processed in the same way the other pixels are, as can be seen in Figure 77.

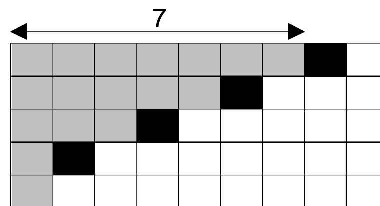


Figure 77: Skewed edge pixels

Six pixels have to be skipped because of the sloped processing. The leftmost pixel has to be skipped because the error passed from that pixel to another cannot be passed since there is no left neighbour. This is an inevitable side effect of skewing that can be avoided by temporarily adding two, seven pixels wide borders to the left and right of the image. This causes an extra overhead of

Overhead of 0.9%

0.9%[1] for A4 image scanned at 200 dpi.

**Tiling**

Overlapping tiles

Another method of processing pixels in parallel is tiling. This methods divides an image into a number of overlapping[2] tiles, after which each loop cycle one pixel of each tile is processed. In Figure 78 for example the first iteration of the loop processes the four light gray pixels. The next processes the dark gray pixels and so on.

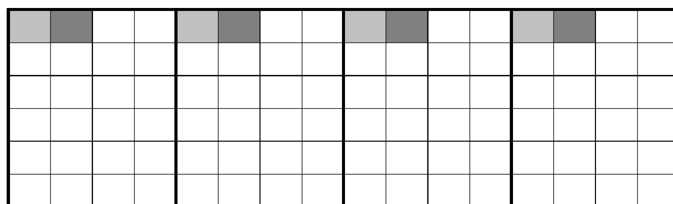


Figure 78: Tiling

Dividing the image into tiles is only allowed when the tiles do not influence each other. For error diffusion this is not true, but by creating an overlap of minimal 32 pixels[3] an acceptable result can be obtained. When dividing an A4 image scanned at 200 dpi

[1] (2*7)/(2*7+1580)*100% = 0.9%
[2] To eliminate the influence of one tile on another.
[3] Source: "Image processing with SIMD architectures" [10].

7.5% overhead

(1580 pixels wide) into four tiles, each 395 pixels wide, 7.5%[(1)] redundant pixels have to be processed.

Four pixels/loop cycle

Similar to a skewed implementation, the negative error values that have to be passed necessitate the extension of the pixels from unsigned bytes to signed words. This automatically reduces the number of pixels to process in parallel to four.

Now lets take a look at the way the error has to be passed. Since the error passing mechanism is identical for all four tiles, the error passing for only one tile is shown in Figure 79 (The black pixel is the one currently processed).

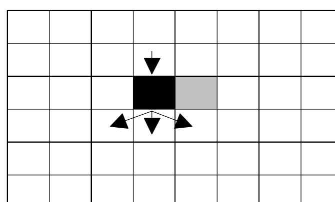


Figure 79: Error passing with tiling

Four memory accesses

Since the image is traversed from left to right the error to pass to the gray pixel can be stored in a register. The error received from above and the errors to pass to the lower neighbours have to be read/stored in an array. Thus, four memory accesses are necessary to pass the error for one pixel. If the image is divided into four tiles, a total of sixteen memory accesses is required to pass the errors of four pixels.

**Skewing or tiling ?**

Both for skewing and for tiling manual packing of the pixels processed is required. An advantage of tiling compared to skewing is that manually packing the pixels is less complex than with skewing.

A tiled implementation processes 7.5% redundant pixels, compared to 0.9% for a skewed implementation. Clearly this is a considerable advantage of a skewed implementation.

The error passing mechanism of a tiled implementation requires sixteen memory accesses to process four pixels, compared to four for the skewed implementation. By implementing the error passing of the tiled implementation in a smart way the number of memory accesses can be reduced, but the required memory bandwidth still remains four times the value of the skewed implementation.

Skewed

Therefore it has been chosen to implement the error diffusion algorithm skewed.

Besides skewing and tiling it is also possible to process one pixel of each of the four color planes. Since this method does not require the additional code necessary for skewing, performance will probably be better, while the implementation will be much easier.
This method has not been chosen for two reasons:
1. It is not possible for a gray-scale implementation.

[1] 32/(32+395)*100% = 7.5%

2. This method avoids the specific problems of the error diffusion. Océ wants to know how these problems can be solved.

### 3.7.3.2 Reading the source pixels

Manual packing

When processing an image skewed the source pixels have to be packed manually. This requires the four pixels processed to be loaded individually, after which they can be packed into an MMX register.

Pixel addressing

It seems to be necessary to have four pointers to the four pixels processed. Since only six[1] general purpose registers are available, this is not the most optimal way to locate the pixels. In Figure 80 four skewed pixels are shown, with a pointer set to the upper pixel (19).

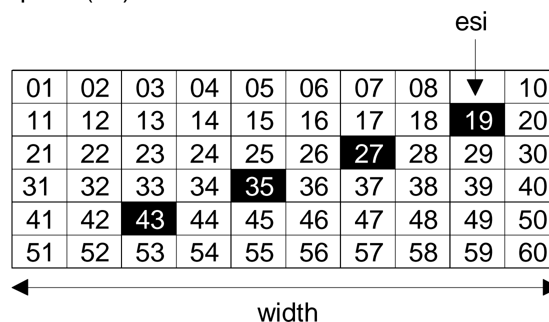



Figure 80: Pixel addressing

The position of the pixels 27, 35 and 43 in the image array can be calculated by adding the image width minus two to the position of pixel 19. This way all four pixel can be loaded into registers.

MMX memory moves

It has been chosen to load the pixels with MMX memory moves, instead of byte-moves for three reasons:

1. Loading the pixels requires a number of registers. While implementing the filter, it turned out that all conventional registers where used already[2].
2. To be able to process the pixels the packed pixels have to be loaded into an MMX register.

Manual packing

The way the pixels are packed is shown in Code example 17. Register eax contains the image width in pixels, esi the position of the upper pixel. Because the addressing mode [esi+3*eax-6] is not allowed, the image width (eax) has to be added to esi to read to lower pixel. More additions can be avoided by using the right the addressing modes.

Addressing modes

The comment added to the code assumes the pixels are numbered as shown in Figure 81.

[1] eax, ebx, ecx, edx, esi and edi.
[2] See section 3.7.3.4.

```
                                 //mm2=[0000 0000 0000 00FF]
movq    mm0, [esi]               //Read pixel a
pand    mm0, mm2                 //mm0=[0000 0000 0000 00 a]
psllq   mm0, 48                  //mm0=[00 a 0000 0000 0000]
add     esi, eax
movq    mm1, [esi-2]             //Read pixel b
pand    mm1, mm2                 //mm1=[0000 0000 0000 00 b]
psllq   mm1, 32                  //mm1=[0000 00 b 0000 0000]
por     mm0, mm1                 ;mm0=[00 a 00 b 0000 0000]
movq    mm1, [esi-4+eax]         ;Read pixel c
pand    mm1, mm2                 ;mm1=[0000 0000 0000 00 c]
psllq   mm1, 16                  ;mm1=[0000 0000 00 c 0000]
por     mm0, mm1                 ;mm0=[00 a 00 b 00 c 0000]
movq    mm1, [esi-6+2*eax]       ;Read pixel d
pand    mm1, mm2                 ;mm1=[0000 0000 0000 00 d]
por     mm0, mm1                 ;mm0=[00 a 00 b 00 c 00 d]

sub     esi, eax                 ;Restore original value of esi
```

Code example 17: Packing skewed pixels

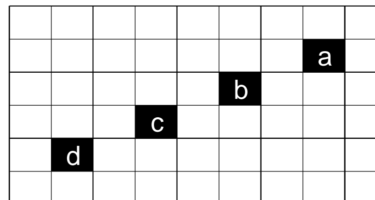



Figure 81: Pixel numbering

Notice the large amount of statements required to pack the pixels, where in the smooth filter one statement was sufficient. Obviously, this will consume a lot of processing time.

### 3.7.3.3 Writing the resulting pixels

Write bitwise

After the new pixel values have been calculated the resulting pixel have to be written as bits. The C++ implementation[1] collects the bits into a byte, after which the byte is written to memory.

For the MMX implementation writing the bits this way is not possible. Figure 82 shows an image of 16[(2)]x6 pixels.

8 8

| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 |
| 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 |
| 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 |

Figure 82: Writing eight collected bits

After pixel 24 is processed it, along with the grayed pixels, would have to be written to the destination array. Because the pixels are written skewed, the row containing pixel 38 can not be written yet (only six pixels are processed); this write would have to be delayed two loop cycles. The two rows containing the pixels 52 and 66 would have to be delayed four respectively six

[1] See section 3.7.2.5.
[2] The image width always is a multiple of eight. This is caused by the functions that add a border to the image (see section 3.4.3.5).

loop cycles. It is obvious that writing the pixels this way requires a complex loop structure with a number of storage spaces to temporarily store the bits.

Shrink four rows

An alternative method is to shrink the bytes to bits after and an entire row (thus four rows of *pixels*) is processed. This is possible because two nested loops are required, one to process the rows, another to process the columns[1]. The inner loop of the error diffusion algorithm can then write the calculated pixels as bytes, after which they are shrinked by a loop that is executed after a row is processed.
This method does not require any additional storage space because the shrinked bytes can be written over the source pixels. Since an A4 image scanned at 200 dpi has a width of 1580 pixels, 4x1580=6320 bytes of storage space are required to store the four processed rows as bytes. This means that these rows are still loaded in the level one cache, so shrinking them requires a minimal processing time.

No additional pointer

The uncomplicated loop structure required to shrink the bytes this way has the advantage that no additional pointer is required; after the four rows are shrinked, the destination pointer of the inner loop is simply set to the next unused position in the array. At the end of the inner loop this pointer points to the last byte of the upper of the four processed row. By subtracting the image width, the pointer can be reset to the beginning of the first of the rows to shrink.

Therefore it has been chosen to use the second way discussed of shrinking the bytes to bits. The loop performing the shrinking operation reads eight bytes into an MMX register, shrinks them, and writes one byte to the array.

#### 3.7.3.4 The inner loop structure

Duplicate threshold

Code example 18 shows the loop structure resulting from the issues discussed in the previous sections in pseudocode. In the second statement the threshold is duplicated four times. This is necessary to enable four pixels to be calculated in parallel.

[1] This is necessary because border pixels require special treatment.

```
Clear(ErrorArray)
Threshold=DuplicateThreshold()

RowCounter = Height
Do While (RowCounter > 0)
       {
        ColCounter = 7                 // Skip 7 pixels
        Do While (ColCounter < Width)
              {
               mm0=Pack4Bytes(ColCounter)

               mm1=GetErrorFromAbove()
               mm3=GetInputPixels(mm0, mm1)
               mm2=GetNewPixels(mm1)
               mm1=GetErrors()

               Write4Bytes(mm2)
               PassErrors(mm1)

               ColCounter++
              }
        ColCounter = 1                 // Bytes => bits
        Do While (ColCounter < Width-1)
              {
               Shrink8Bytes()
              }
        RowCounter = RowCounter-4
       }
```

Code example 18: Loop structure of the MMX implementation

The pseudocode in Code example 18 is quite self explaining. Because this would add to much complexity, most of the registers used are not mentioned.

Bottom rows

Notice that because RowCounter is decremented four each loop cycle, images with a height that is not a multiple of four will not be processed entirely. Since this only concerns three out of 2176 rows it has been decided to skip these rows and cut off the bottom of the image later.

### 3.7.3.5 Masks

Generally, if a filter requires masks these are generated outside the inner loop. Within the loop the registers they are stored in cannot be used for other purposes.

Generate

The error diffusion algorithm however requires a large amount of registers. The two masks required cannot be kept in registers, because then there are not enough registers left. Therefore the masks are generated in the inner loop. This generally requires two instructions per mask.

An alternative method of generating masks is loading them from memory. Because this memory locating is frequently accessed it will generally be loaded from the level one cache, so loading the mask requires only one cycle. It has been chosen to generate the mask because this reduces the memory bandwidth slightly, but for more complicated masks it is also possible to load them from memory.

### 3.7.3.6 Filter performance

After the inner loops of the error diffusion implementation have been paired performance is compared to that of the unpaired version. The results are shown in Table 11.

| Image size (pixels) | Cycle times unpaired (cycles/pixel) | Cycle times paired (cycles/pixel) | Improvement (%) |
|---|---|---|---|
| 290x509 | 115 | 94 | 18,3 |
| 1580x2176 | 114 | 94 | 17,5 |

Table 11: Performance gain by paring

17% faster by pairing

Notice that the performance gain caused by paired is about 17%, compared to 31% for the sharpen filter[1]. The reason the improvement is less is that the complexity of the error diffusion reduces the possibilities to pair statements. Contrary to the sharpen, where a pairing rate of 100% has been reached, the inner loops of the error diffusion algorithms could not be paired perfectly. A pairing rate of 98% was the maximum feasible.

The performance of the paired MMX version and the C++ version of the error diffusion is shown in Table 12 for several images.

| Image size (pixels) | Cycle times C++ version (cycles/pixel) | Cycle times MMX version (cycles/pixel) | Improvement (%) | Improvement (factor) |
|---|---|---|---|---|
| 87x16 | 193 | 95 | 50,8 | 2,0 |
| 63x96 | 190 | 98 | 48,4 | 1,9 |
| 127x96 | 199 | 95 | 52,3 | 2,1 |
| 255x320 | 202 | 92 | 54,5 | 2,2 |
| 290x509 | 203 | 94 | 53,7 | 2,2 |
| 1580x32 | 203 | 96 | 52,7 | 2,1 |
| 1580x2176 (1) | 205 | 94 | 54,1 | 2,2 |
| 1580x2176 (2) | 206 | 94 | 54,4 | 2,2 |

Table 12: Performance of the error diffusion

The lower two images are A4 images scanned at 200 dpi. Notice the small difference in performance for larger and smaller images; this is caused by the fact the image is traversed horizontally. This causes all images to benefit in the same amount of the caches.

MMX 2x faster than C++

In Table 12 it can be seen that the performance of the MMX implementation is a factor 2.2 better than the C++ implementation. Since a part of this performance gain is caused by the fact that assembly code is slightly faster than compiled C++ code, it can be concluded that for error diffusion implementing the algorithm with MMX is still useful.
For more complex algorithms[2] however, the performance gain might not be worth the difficulties encountered when implementing a filter with MMX Technology.

---

[1] See section 3.5.3.11.
[2] See section 3.7.1.4.

### 3.7.3.7 Categorising the processing time of the paired code

33% error passing
49% pack & unpack

Figure 83 shows categorised processing time of the error diffusion algorithm[1]. Notice that 33%[(2)] of the processing time is spent passing and receiving error values. Another 49% [(3)] of the processing time is used to read, pack, unpack and write the pixels. Just 4% of the processing time is needed to calculate the new pixel values and the rounding errors.

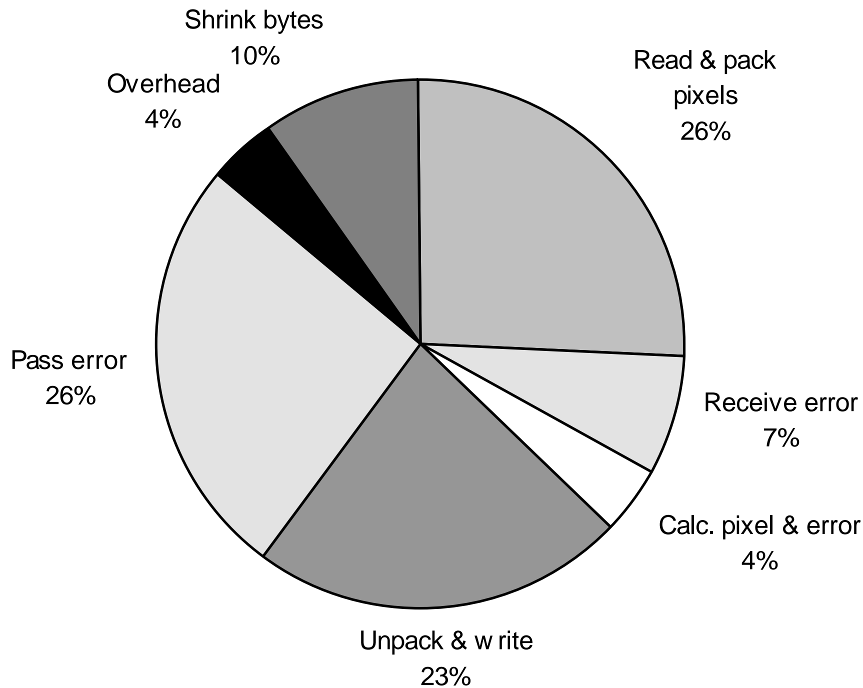



Figure 83: Categorised processing time of the error diffusion

Figure 84 shows the processing time categorised according to the instruction type. Notice that 26% of the processing time is required for logical and shift operations. These operations are needed for the packing and unpacking of the pixels. Since the extension from packed bytes to packed words is performed automatically when manually packing the pixels, no conversion instructions are used.

[1] For more information on how the processing time is determined see section 3.4.4.8.
[2] 26%+7%
[3] 26%+23%

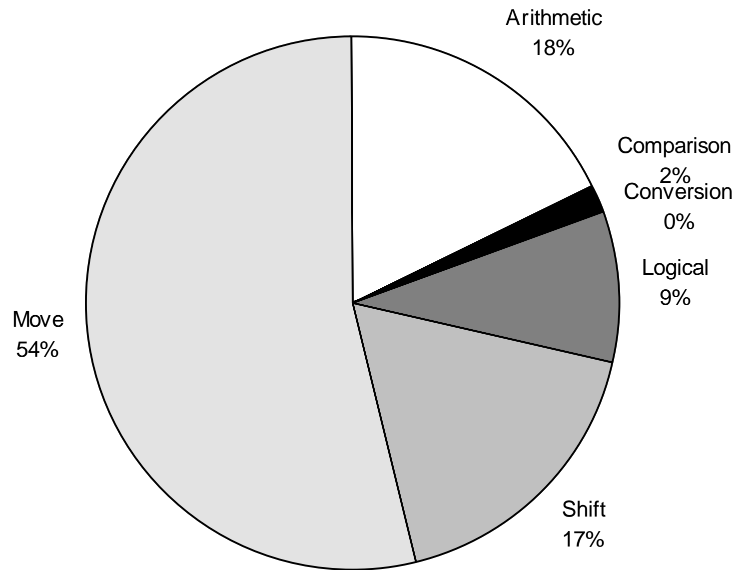



Figure 84: Processing time per instruction type

## 3.8 Bits to Bytes conversion

### 3.8.1 The algorithm

Viewing application

The images created by the error diffusion algorithm discussed in section 3.7 contains pixels which are represented by a single bit. These images can be sent to a print engine, but the image viewing application used to view the processed images cannot read such images. In order to be able to view the images on a PC the pixels represented with bits have to be represented by bytes.

A pixel with the bitvalue '1' has to be converted to a byte '255'. A bit '0' has to be converted to a byte '0'. Thus the one bit has to be duplicated to all the bits of the byte.

### 3.8.2 The implementation

The algorithm described in section 3.8 is implemented with MMX Technology. It has been chosen not to implement two versions because this algorithm is not part of a processing path implemented in the image processing path of a digital copier.

MMX version only

It was chosen to implement the algorithm with MMX for two reasons:

1. The code shrinking the bytes to bits in the error diffusion algorithm can be adjusted to extend bits to bytes.
2. An MMX implementation reads one byte (= eight bits), and can extend it to one quadword (=eight bytes).

Besides this, an MMX implementation can process eight pixels in parallel. A C++ implementation can process four[1] pixels in

[1] By using 32 bits arithmetic.

parallel. Thus the MMX implementation can reach twice the speed of a C++ version.

'Shift' and 'Or'

The image array is traversed by one loop. which processes the entire array. Each loop cycle one byte is read from the array into an MMX register. The bits in the byte are shifted and or-ed in such a way that each bit is duplicated to one byte. Duplicating the bits could be done more efficient with the shift right, which duplicates the most significant bit. Unfortunately this operation cannot be performed on packed bytes.

Since the extension from bitwise represented pixels to bytes is not part of a processing path implemented in a copier, and its implementation is fairly straightforward, its implementation is not discussed any further.

## 3.9 Removing borders

### 3.9.1 The algorithm

In section 3.3 the implementation of the function which adds a borders to the image is discussed.

No extra logic

In a digital copier (or printer) no logic is needed to remove the borders. The entire image is sent to the print engine, but some of the wires are not connected. Since the test images processed by the demo program should have the same size[1] as the original images, the added borders have to be removed by an additional algorithm.

### 3.9.2 The implementation

MMX version only

Since only the MMX version of the processing path adds borders to the images, it is obvious that the function that removes the borders is also implemented with MMX. Besides this, the function that adds the borders is also implemented with MMX.

The implementation of the border removing function is almost identical to that of the border adding function. The only difference is that now some pixels are *not* copied from the source array. With the methods described in the previous sections, this algorithm can easily be implemented. Since the implementation is quite uncomplicated it is further not discussed here.

## 3.10 The total print processing path

### 3.10.1 'Warming up' effects

Caches

When a sequence of algorithms is executed, the first algorithm will 'warm up' the caches. This means that data required by the other algorithms is loaded into the caches by the first.

Because of its limited size, the level one cache will not be 'warmed up', but the level two cache will. For large images, such as the scanned A4 image, this will have little effect because these images cannot be stored in the level two cache. When

[1] To enable a pixel-wise comparison between unprocessed and the processed image.

processing small images however the image will be loaded into the level two cache during the entire processing path. This is one of the reasons small images are sometimes processed much faster than large images.

In the implemented processing path the 'warming up' will primarily be done by the RGB color plane separation. Because both the C++ version of the processing path and the MMX version benefit

Influences C++ *and* MMX

in the same way from the 'warming up', it has little influence on the performance comparison between the C++ and the MMX versions. Similar, all the algorithms in the path benefit in the same way, so the performance comparison between the various algorithms is not influenced.

### 3.10.2 Algorithm performance

200 dpi A4

166 MHz

Figure 85 shows the performance of the C++ and the MMX versions of the implemented algorithms. The chart shows the number of pixels of the 200 dpi A4[1] image processed per second on a 166 MHz CPU.

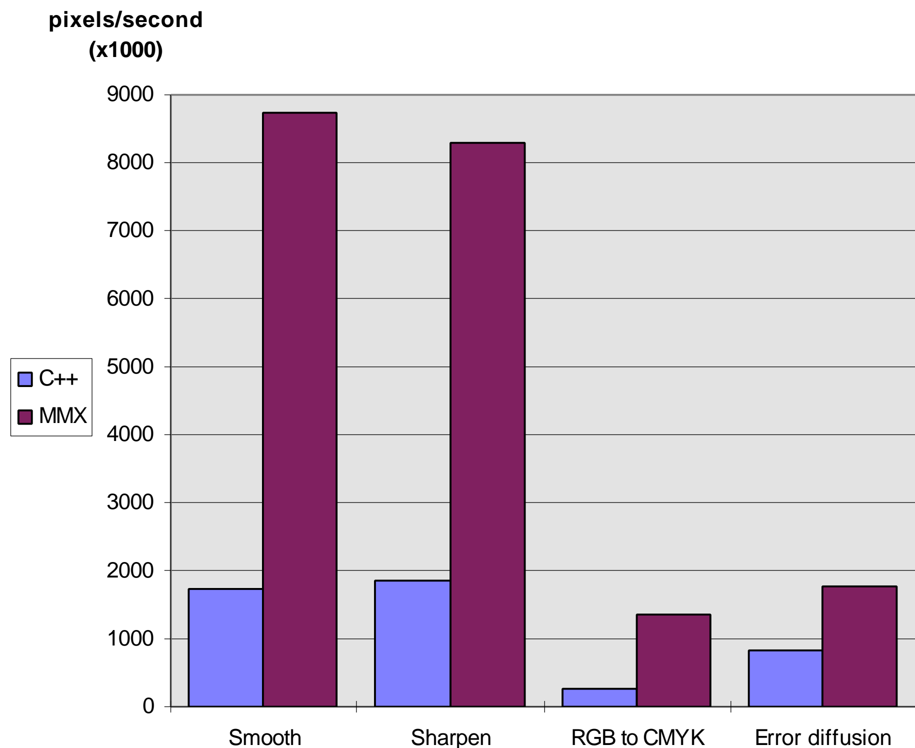



Figure 85: C++ vs. MMX for 200 dpi A4 (1580x2176) (166 MHz)

Performance

For kernel operations, such as the smooth and sharpen, as well as for the RGB to CMYK conversion the MMX version is a factor four to five faster than the C++ version. For more complex algorithms, such as the error diffusion, the performance of the MMX version is a factor two of that of the C++ version.

[1] 1580x2176 = 3,438,080 pixels

### 3.10.3 Improvements of the processing path

**Tiling**

Small images

In Table 5 and Table 8 it can be seen that the kernel operations process small images about 50% faster than large ones. By splitting large images into a number of small, overlapping tiles (see section 3.7.3.1) the performance of these algorithms can be improved.

Additionally, all algorithms can be applied to the tile before the next tile is processed. This way the tile can be loaded from the cache instead of the slow main memory.

**Merging algorithms**

Since reading and writing the pixels costs a considerable amount of the processing time, it might be interesting to merge a number of algorithms into one, slightly more complex algorithm.

This is especially interesting for algorithms which are executed right after each other. The RGB color plane separation and the smooth operation for example can quite easily be merged into one algorithm. This algorithm first reads a block of 3x3 color pixels (=3x3x3 bytes). Then it separates the three color planes and applies the smoothing kernel [1].

**Planes parallel**

Manual packing

The performance of the halftoning algorithm is limited heavily by the manual packing of the pixels. The algorithm now processes four pixels of one color plane simultaneously. An alternative way of processing four data elements in parallel is to process one pixel of each color plane simultaneously. Since the RGB to CMYK conversion also has to unpack the pixels manually, this will result in a performance gain for both the color conversion and the error diffusion.

In Figure 56 it can be seen that eight percent of the processing time of the RGB to CMYK conversion is required to unpack and write the pixels. Figure 83 shows that 26% of the processing time of the error diffusion is needed to read and pack the pixels. By writing the four color planes to one array the processing time required for these two parts of the code can be reduced drastically. In fact this time can be reduced almost to zero.

**White detection**

RGB to CMYK

In section 3.6.3.10 it can be seen that most of the time needed for the RGB to CMYK conversion is used to interpolate the CMYK value. By detecting if a white pixels is processed the performance
of the RGB to CMYK conversion can be improved. The actual improvement *heavily* depends on the image processed. When processing an entirely white image the time necessary to process the image can be reduced to one-fourth of the time

[1] Obviously the algorithm can be optimised by buffering eight of the separated pixels. This way only one color pixels has to be separated per smoothed pixel.

currently needed. For images containing no white pixels the processing time will increase slightly.

**Adding borders**

Scan path

Finally, when using only the scanning section of the processing path, the performance can be slightly improved by adding less borders to the image.

On page 81 it can be read that the error diffusion algorithm requires additional borders to be added to the source image. This increases the image size of a 200 dpi A4 image with about 0.9%. Since the smooth and sharpen algorithms do not require this additional border their performance can be increased a little bit.

# 4 Architecture developments

## 4.1 The IA32 architecture

The IA32[1] architecture is commonly used in PC's. IA32 processors with MMX are for example:

- Intel Pentium with MMX
- Intel Pentium Pro
- Intel Pentium II with MMX
- AMD K6
- Cyrix 6x86MX
- IDT Winchip C6

In the next sections these processors will shortly be discussed, focusing on their possibilities for image processing. The Pentium with MMX is discussed in section 2.4.

### 4.1.1 Intel Pentium Pro

The Pentium Pro was intended to be the successor of the Pentium. The major improvements were:

1. Dynamic execution.
2. Improved superscalar architecture.
3. On-chip level two cache.

**Dynamic execution**

Dynamic execution is a technique where, instead of executing instructions in the sequence the programmer wrote them, the *CPU* determines the order in which they are executed. This way more units can be used simultaneously. On the Pentium the programmer had to re-order the statements; now the CPU re-orders them automatically[2].

Micro instructions

The processor does not execute assembly instructions but micro instructions. An assembly instruction written by the programmer (or the compiler) is broken down into a number of micro instructions. At a certain moment the processor might execute for example the light gray instructions shown in Figure 86 (the dark gray instruction have been executed previously). Instruction 4a could be executed before the previous instructions were finished (it does not use results produced by them).

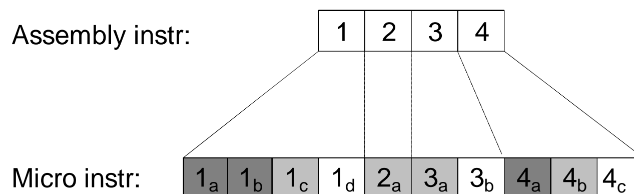



Figure 86: Dynamic execution sequence

Unoptimised code

Dynamic execution particularly improves the performance of unoptimised code. But also the combination of the superscalar architecture and dynamic execution has possibilities to improve performance. On the Pentium, two pairing instructions both have to be finished before the next pair is started. If one instruction of

[1] IA32 is an acronym for Intel Architecture with 32-bit opcodes.

[2] This way of re-ordering the instructions is also known as out-of-order execution.

the pair takes much more time than the other this results in a low CPU utilisation. In these situations dynamic execution allows one instruction of the next pair the be started. This way, while one pixel is calculated, the data required for the next can be read from the memory.

Branch penalty

A major disadvantage of dynamic processors is the high penalty for mispredicted branches (10-20 cycles). To help avoid branches conditional move instructions are available.

**Improved superscalar architecture**

The Pentium can execute two instructions per cycle. The Pentium Pro can execute up to three micro operations[1] per cycle. Besides this, more units have been added to allow more types of operations to be executed in parallel. A memory read now can be paired with a memory write.

**On-chip level two cache**

On the Pentium Pro's predecessors the level one cache was placed on the same chip as the processor core. The level two cache was placed in separate chips on the motherboard, causing the level two cache to run at a much lower speed than the processor.

On the Pentium Pro however, a level two cache of 256 or 512 KB is integrated in the CPU, running at the same clock speed as the CPU. This increased the performance significantly because by reducing the distance between the level one level two cache the access time of the level two cache could be reduced.

Drop-out rate

Placing the level two cache on the same chip as the CPU caused the chip size of the Pentium Pro to be much larger than that of the Pentium. Because of the increased chip size the drop out rates during the chip's production reached an all time high. This caused the production costs of the Pentium Pro to be the highest of all Intel chips. For this reason, Intel stopped producing the Pentium Pro in the mid of 1998.

### 4.1.2 Intel Pentium II

Level two cache

Soon after the introduction of the Pentium Pro its high production costs forced Intel to re-design the chip. This resulted in the Pentium II. One of the differences between the Pentium II and the Pentium Pro is that on the Pentium II the level two cache is no longer placed on the same chip as the processor [2]. The level two cache now runs at half of the CPU speed.

Additionally MMX Technology was added, and the clock rates were increased further. The Pentium II currently is available at clock speeds ranging from 233 to 400 MHz. Intel plans to crank up the clock speed to 500 MHz.

[1] Generally an assembly instruction is decoded to one micro instruction. Instructions that use complex addressing modes (MOV EAX, [EPB+32] for example) and Add-With-Carry instructions are decoded into two, pop's into three and push- es into four micro instructions.
[2] The level two cache's chips are placed into the same package as the processor.

'Xeon'

Intel also plans to increase the level two cache size to up to two MB (code name 'Xeon'). Although this processor (to be released in June 1998) will be quite expensive, this could be an interesting processor for image processing. By splitting the image in half, one color plane[1] can be loaded into the level two cache during the entire processing path, reducing the time required to read and write the pixels.

In the next two paragraphs two recently introduced improvements of the Pentium II will be discussed; the 100 MHz system bus and MMX2.

**100 MHz systembus**

The systembus is the main data highway in a personal computer system. On early Pentium II's this bus ran at speeds of 50, 66, 75 or 83 MHz, depending on the installed CPU. Currently Intel is introducing Pentium II's which support a bus speed of 100 MHz. Since the systembus traditionally is one of the main bottlenecks in high performance PC's, it is expected that the performance will improve significantly. The 100 MHz is not only supported by Intel, but also by companies that build Intel clones, such as AMD, Cyrix and IDT.

In 1999 Intel plans the introduction of a 200 MHz system bus. This will further improve the overall performance of PC's.

**MMX2[(2)]**

'Katmai'

By the end 1998 a new version of the Intel Pentium II called 'Katmai' will be available. This CPU not will only run at higher clock speeds than its predecessors, but will also contain the new MMX2 instruction set.

Floating point

Although Intel has not yet published the MMX2 instruction set, the
primary improvement seems to be the addition of floating point MMX instructions, which operate on eight separate registers. The author of [13] *suggests* that Intel can make the eight new registers 40 bytes (240 bits) large. Since Intel floating point units typically operate on 80 bit values this would enable four 80 bit values or eight 40 bit values to be processed in parallel.

Integer

Besides this MMX2 contains some new integer instructions:

- Packed shuffle.
- Insert/extract word.
- Packed average on bytes & words.
- Packed unsigned multiply on words (MMX multiplies signed).
- Masking.
- Minimum and maximum operations on bytes & words.
- A prefetch instruction.

Especially the prefetch instruction will be very useful for image processing. The insert and extract word instructions are also useful, especially for manual packing of pixels. Minimum and

[1] One color plane = 1580x2176 pixels = 3,438,080 B = 3.28 MB.
[2] Source: "Windows Source Home Page" [13]. The instruction set was retrieved by Clive Turvey. He disassembled a demo program that was placed accidentally on an Intel home page. Intel has *not* confirmed this, so this paragraph should be considered provisional.

maximum operations can be used for color stretch and edge detection algorithms.

With the introduction of MMX2 Intel also will improve the developer environment for MMX. Currently MMX instructions can only be used in in-line assembly. For MMX2 Intel plans to release a compiler plug-in that allows the usage of MMX in standard C++ code.

### 4.1.3 AMD

AMD is the main competitor of Intel. Before the introduction of the

K6

high speed Pentium II versions, the AMD K6 processors out-performed the Intel processors. This was primarily caused by AMD's more elegant processor design. The later versions of the Pentium II used Intel's more advanced chip technology to reach higher clock speeds. This way Intel re-gained its position at the top of the market. AMD now targets at the lower end of the market by pricing its CPU's 25% below the equivalent Intel CPU.

The K6 has a level one cache of 64 KB. This is twice the size of that of the Pentium II. However, since this is by far too small to store scanned images it is expected that this will not have a lot of influence on the image processing performance.

AMD's more advanced processor design resulted in low execution latencies and no decode pairing restrictions. The penalty for misaligned data is only one cycle, compared to three cycles on Pentium (II/Pro) processors.

K6-2: MMX3D

AMD's K6-2[1] incorporates MMX3D. MMX3D is not only supported by AMD, but also by Cyrix, IDT and Microsoft. The main difference between MMX and MMX3D is that MMX3D incorporates SIMD floating point instructions, similar to MMX2. Except a prefetch and FEMMS (Fast Entry/Exit MMX State) instruction MMX3D does *not* have the improved MMX instructions MMX2 has.

Compatibility

It is important to realise that although they have some instructions in common, MMX3D and MMX2 are *not* compatible.

In the second half of 1998 AMD plans to introduce the K6 3D+. In this processor the level two cache will be placed on the same chip the CPU is placed on to allow the cache to run at the same speed as the CPU does. Unlike the Pentium II, the conventional level two cache on the motherboard is used as a level three cache, similar to Digital's Alpha. The K6 3D+ will be introduced at a clock speed of 350 MHz, after which it will quickly move to 400 MHz and more.

### 4.1.4 Cyrix

Another important competitor of Intel is Cyrix. Cyrix operates primarily on the low-cost side of the market where it managed to capture a stable market share.

[1] The K6-2 was launched in the first half of 1998 as the successor of the K6. The K6-2 is also known as K6-3D. MMX3D is also known as 3Dnow.

6x86MX

The 6x86MX is Cyrix' latest CPU. It is a non-superscalar, pipelined architecture that contains several advanced features, such as branch prediction and speculative execution.
On 16 bit programs like Windows 95 the 6x86MX slightly outperforms AMD's K6 and the Pentium with MMX. On 32 bit programs like Windows NT however, the performance is mediocre. A more advanced version of the 6x86MX is the M2.

M2

This basically is a superscalar 6x86MX with two separate pipelines.

Similar to AMD, Cyrix CPU's are quite well designed, but cannot keep up with the high-end Intel CPU's. This is, among others, caused by the low clock speeds Cyrix CPU's run on.

The MMX Technology of the 6x86MX and M2 is fully compatible with Intel's. Performance tests show that its MMX performance is the lowest of all MMX CPU's available. Besides all the MMX instructions, the 6x86MX also contains some extended MMX instructions, some of which are useful for image processing:

- Packed magnitude (= absolute value compare).
- Packed conditional move (bytes).
- Packed Average (bytes).

Especially the conditional move and the packed magnitude functions can be very useful for image processing. The first has numerous applications, the latter can for example be used for edge detection algorithms.

Cyrix has announced an improved version of MMX called MMXFP to be incorporated in the 'Cayenne', which is to be released in the second half of 1998. Cyrix agreed with AMD that MMXFP will be identical to MMX3D.

### 4.1.5 IDT

IDT is a relatively unknown CPU producer. In the first half of 1998

Winchip C6

IDT announced its Winchip C6 CPU, which is claimed to have about the same performance as the Pentium MMX for integer operations. On floating point and MMX code however the C6 is slower than the Pentium.

The C6 does not incorporate the advanced features the Pentium II, K6-2 or the 6x86MX do; it simply is a classic but straight forward CPU with a large level one cache. The C6 has about the same price tag the 6x86MX has.

The successor of the C6, the C6+ will incorporate AMD's MMX3D extensions. The release date of the C6+ is not yet published.

## 4.2 The PowerPC architecture

Mackintosh

The PowerPC processors are designed by Motorola. They are used for various applications, including Power Mackintosh computers and embedded control applications. PowerPC CPU's are currently produced by two vendors; Motorola and Samsung.

AltiVec

The latest improvement of the PowerPC architecture is the addition of a vector processing SIMD technology called AltiVec.

AltiVec instructions operate on 128 bits [1] wide registers called Vector Register File (VRF). The AltiVec programming model incorporates 32 vector registers which can be used in the following configurations[2]:

- 16 bytes (8 bit)
- 8 halfwords (16 bit)
- 4 words (32 bit)
- 1 quadword (128 bit)

A total of 162 new instructions are available to process these data elements in parallel.

The AltiVec programming model incorporates all of the MMX instructions. Additionally some other instructions are available, some of which are very useful for image processing:

- Sum across
- Multiply
- Multiply-sum (multiply and add to other register)
- Average
- Maximum
- Minimum
- Logical NOR
- Rotate instructions
- Splat (duplicate one element through register)
- Permute (Shuffle elements)
- Select element
- Prefetch data stream
- Various floating point arithmetic

Prefetch

The prefetch instruction can prefetch up to four data stre ams. First the stream is initialised by selecting a memory area to be loaded. The prefetcher then automatically loads these locations into the level one cache. In chapter 3 it can be read that memory accesses are a significant bottleneck. By using a prefetch instruction these delays can be avoided.

Splat

Another useful instruction for image processing is the splat instruction; this instruction duplicates one byte or halfword through an entire register. The splat instruction also operates on immediate values.

Permute

The permute instruction allows any byte in a register to be moved to another position. This can for example be used to re-order values in one cycle.

All AltiVec instructions support both saturation and overflow arithmetic. Overflow arithmetic is generally used, but for image processing saturation arithmetic is very useful.

Intel architecture CPU’s use the little-endian byte numbering scheme to store multi-byte data in the memory[3]. Motorola CPU’s traditionally use the big-endian scheme, where the bytes are stored reverse ordered. The AltiVec architecture however also supports the little endian scheme.

[1] MMX registers are 64 bit wide.
[2] Notice that the names of the data types are not the same as the names Intel uses.
[3] See page 19.

Misaligned access

A weak point of the AltiVec architecture are the way misaligned memory accesses[1] are handled. If a quadword is read from memory for example the effective address is calculated by making the four least significant bits zero. If the access was misaligned this results in a memory read of the wrong data. A memory read on address 0x02F03 for example (Figure 87) should read the grayed data. Instead, the squared bytes are read.

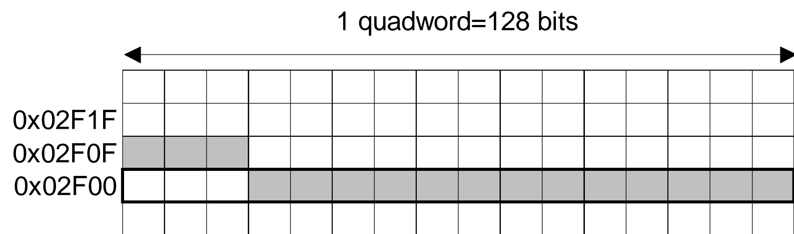



Figure 87: Misaligned read on AltiVec

AltiVec technology provides special instructions to merge misaligned datablocks. On Pentium (II/Pro) CPU's misaligned memory accesses cause a penalty of three cycles. An AltiVec CPU can merge two datablocks retrieve the misaligned datablock in the same number of cycles too.

[1] On AltiVec CPU's misaligned memory accesses are accesses that are not aligned to the natural boundary of a data element. For example a halfword must be aligned on a two byte boundary. See section 2.4.4 for more information on alignment on the Pentium CPU.

# 5 Evaluation & conclusion

## 5.1 Implementation issues

When implementing image processing algorithms with MMX Technology a number of issues have to be taken in account. These issues can be devided into three categories:
1. Architecture related
2. Algorithm related
3. Developing environment related

**Architecture related issues**

The Pentium MMX architecture has some limitations that influence the way image processing algorithms are implemented.

Memory accesses

First of all, memory accesses (to load pixels for example) take a relatively high amount of CPU clock cycles. This is caused by the fact that memory chips do not reach the high clock speeds the CPU does. Although cache memories improve the memory bandwidth, scanned images cannot be entirely stored in them. Therefore an algorithm has to be implemented in such a way that the memory bandwidth is used optimally (caches).

Future processors, such as the Intel Pentium II 'Katmai', AMD K6-2, and Motorola AltiVec, incorporate instructions that allow prefetching of pixels by special processing units.

Instructionset

Second, the Pentium with MMX lacks some instructions required for image processing, such as:
- Masking
- Shuffle or permutate
- Insert- and extract word
- Minimum/maximum/average operations

To perform such operations a number of instructions are required, causing a performance reduction.

Intel's successor of MMX (MMX2) is expected to add some of these missing instructions, but not all. An insert/extract word instruction for example is not available. AMD's successor of MMX (MMX3D or 3Dnow!) focuses on SIMD floating point operations and adds no new SIMD integer instructions to MMX. The Motorola AltiVec instructionset contains all of the instructions required for image processing. Not only minimum, maximum and average instructions, but also insert/extract word and shuffle instructions are available.

Data types

Third, many MMX instructions do not operate on the eight bit data scanned images typically consist of. For most algorithms this is no real limitation because they require the data to be extended to larger data types anyway, but for some algorithms (color stretch for example), this limits the number of pixels to be processed in parallel to four, instead of eight.

Both Intel's MMX2 and AMD's MMX3D are not expected to increase the number of data elements that can be processed in parallel. For as far a the support of eight bit data is concerned:

although MMX2 adds some integer SIMD instructions the support of eight bit data has not been improved.
AltiVec CPU's process twice the amount of data elements MMX CPU's do (128 bit registers vs. 64). This means that 16 bytes or eight words[1] can be processed in parallel. Additionally, some of the more advanced AltiVec instructions, multiply-and-add for example, internally extend the data to prevent rounding errors. All AltiVec instructions operate on 8- and 16 bit data.

Immediate operands

Fourth, most of the MMX instructions do not accept immediate operands (constants). Multiply instructions for example cannot multiply a value with a constant directly, so the constant has to be loaded from memory into a register first (the move instruction does not accept immediate operands either). None of the MMX successors is expected to accept immediate operands.

Registers

Fifth, Intel architecture CPU's do not have enough registers available to implement complex image processing algorithms without storing register temporarily in the memory. Because of the relatively low memory bandwidth this causes the performance to decrease. Additionally the small number of registers limits the possibilities to store immediate values in registers during the entire processing loop.

Both Intel's and AMD's MMX successors are not expected to contain more registers than MMX does. AltiVec CPU's have 32 registers, which is a considerable improvement over the eight MMX registers.

Image width

Finally, a lot of image processing algorithms require special treatment of border pixels. Since SIMD CPU's process a fixed number of pixels in parallel, the image width has to be a multiple of the number of pixels processed in parallel. To process all types of images a border has to be added to some images in order to extend the image width. In a digital copier this is not necessary because the scanner can be configured to scan the optimal image width.

**Algorithm related issues**

Some of the image processing algorithms (lookup table and error diffusion for example) heavily limit the possibilities for parallel

Collect pixels

processing. These limitations force the programmer to collect the pixels to process with a number of instructions to allow parallel processing. This occurs with error diffusion for example.

Lookup operations can by definition not be performed in parallel, unless a special lookup table is created. Since such a lookup table's size will be increased exponentially this is generally not possible for image processing.
These algorithm limitations, coupled to the described limitations of the MMX instructionset, cause the performance increasement to be reduced significantly.

MMX2 and AltiVec contain insert- and extract word instructions that improve the possibilities to collect pixels. It is therefore

[1] For simplicity's sake the Intel naming is used. Motorola calls 16 bit values half words and 32 bit values words. When speaking of words Intel means a 16 bit quantity.

expected that they will perform better on algorithms with poor parallelability.

**Developing environment related issues**

Currently MMX Technology can only be used in in-line assembly. This increases the time needed to implement an algorithm considerably, as well as the chance of program 'bugs'.

VTune

Besides this, only one development tool is available; VTune. VTune monitors an executing program and shows the time necessary for each statement. Additionally VTune indicates the cause of some execution delays. Unfortunately the current version of VTune (2.5) contains many 'bugs', making its usage very unpleasant.

Intel has announced that with MMX2 a much improved version of VTune will be released. Especially the possibilities to analyse C++ code will be improved. Additionally Intel has recently released a compiler plug-in that allows the usage of MMX(2) in standard, C++ like statements. This is expected to decrease the time needed to implement an image processing algorithm considerably.

## 5.2 Performance

For kernel operations (smooth, sharpen), as well as for the RGB to CMYK conversion the MMX implementation is a factor four to five faster than the C++ version. For more complex algorithms, such as error diffusion, this is a factor two.

Elements parallel

Most of the performance gain is reached by processing multiple (generally four) data elements in parallel. Additionally MMX implementations benefit from the fact that they have to be written

Assembly

in in-line assembly, which improves the performance slightly. Finally, some of the MMX processor units are faster than their conventional equivalents, from which especially the RGB to CMYK conversion benefits (multiply unit). However, in future (Pentium II) CPU's these units will also be improved, thus eliminating MMX's advantage.

Collecting pixels

A lot of (potential) performance is lost when a certain algorithm requires the pixels to be collected manually, instead of reading them as one sequential stream from memory. A similar problem occurs when the format of the data elements to be processed is not the same as the format in which the other input of the algorithm is received. This occurs for example with the interpolation[1], where the weight of each of the angular points has to be re-ordered before the multiplication with the angular point can be performed.
The successors of MMX will contain insert- and extract word instructions that help to avoid or relieve these problems.

Linear improvement

Generally it can be said that, although the MMX instructionset lacks a number of specific image processing instructions, the performance increases[2] almost linear with the number of elements processed in parallel, provided that no manual

[1] Part of the RGB to CMYK conversion.
[2] Compared to a C++ implementation.

collecting or re-ordering is required. For the algorithms [1] where this is required the performance gain is about half of the number of elements processed in parallel. It is expected that for algorithms with more complex calculations the gain will be slightly more[2].

An important remark to be made is that the number of pixels processed in parallel is limited by the data type of intermediate results (words instead of bytes). Extending data temporarily is often done in image processing algorithms to avoid large rounding errors. Besides this, sometimes data has to be extended because MMX instruction operate on certain data types only.
Generally the 8 bit pixels have to be extended to 16 bits. All MMX instructions operate on 16 bit values. MMX can process four 16 bit data elements in parallel (eight 8-bit values).

Intel

The performance gain actually reached (max. factor five) is less than the gain Intel predicted (factor six). This is primarily caused by the fact that Intel processes small images only. These images can be stored entirely in the fast level one cache memory. Another reason Intel's prediction of the performance gain is too optimistic is that Intel only implements algorithms with an excellent parallelability and the possibility to process eight pixels in parallel, instead of four.

## 5.3 Recommendations

Currently MMX's main limitations are the instructionset, the memory bandwidth and the lack of registers. Alternatives of MMX contain both an improved instructionset and have ways to improve the memory bandwidth. For image processing the Intel Pentium II 'Katmai' with MMX2 and especially the Motorola PowerPC with AltiVec are the most promising CPU's. The latter has the potential to reach twice the speed MMX2 does because the AltiVec registers can contain twice the amount of pixels an MMX(2) register can. Additionally the AltiVec has four times the number of registers MMX(2) has. Therefore further research on MMX2 and especially AltiVec is recommended.

MMX2
AltiVec

Another limitation of image processing on a PC lies in the fact that
Windows NT was not designed for (semi-) real-time applications. Due to Windows NT's multitasking system an application's performance can fluctuate considerably. This can be improved by shutting down large parts of the operating system, but generally such a solution cannot be used (because the PC also performs other tasks for example). Therefore further research on how to reach a constant application performance on a Windows NT platform is recommended.

Windows NT

[1] For error diffusion and the color conversion. Only for error diffusion this was actually measured. Test indicate that when the C++ version of the color conversion uses a one-cycle multiplier the performance gain of the MMX version is about a factor two.
[2] Because the amount of processing time required for pixel-collection is relatively low compared to the time needed to process the collected pixels.

# 6 Literature


1. Rutten, R, An introduction to MMX, Océ internal report, 1997
2. Intel MMX Technology at a Glance, Intel Corporation, 1997, Order number 243100-003
3. Intel Architecture Technology Overview, Intel Corporation, 1997, Order number 243081
4. Intel Architecture MMX Technology Developer's Manual, Intel Corporation, 1997, Order number 243013
5. Pentium Processor Family Developers Manual, Intel Corporation, 1997, Order number 241428-05
6. Intel Architecture MMX Technology Programmer's Reference Manual, Intel Corporation, 1997, Order number 243007
7. Pentium and Pentium Pro Processors and Related Products, Intel Corporation, 1997, Order number 241732-004, ISBN 1-55512-265-5
8. Intel Architecture Optimization Manual, Intel Corporation, 1997, Order number 242816-003
9. Schmit, M. L., Pentium Processor Optimization Tools, AP Professional, 1995, ISBN 0-12-627230-1
10. Rosendahl, J. J. A, Implementing image processing algorithms on SIMD architectures, Océ internal report, 1998
11. Fog, A, How to optimize for the Pentium Processor, 1998, http://www.announce.com/agner/assem/
12. AltiVec Technology Programming Environments Manual, Motorola Inc., 1998, http://www.motorola.com/AltiVec/
13. Turvey, C, Windows Source Home Page, V Communications, 1998, http://www.tbcnet.com/~clive/
14. AMD-3D Technology Manual, Advanced Micro Devices Inc., 1998, Publication number 21928.
15. IDT Winchip C6 Processor data book, 1997, Centaur technology Inc., http://www.winchip.com

## Appendix A: MMX instruction set summary

| Category | Mnemonic | Data type | Description |
|---|---|---|---|
| Arithmetic | PADD | B,W,D | Add with wrap-around on [byte, word, doubleword] |
| | PADDS | B,W | Add signed with saturation on [byte, word] |
| | PADDUS | B,W | Add unsigned with saturation in [byte, word] |
| | PSUB | B,W,D | Subtraction with wrap-around on [byte, word, doubleword] |
| | PSUBS | B,W | Subtract signed with saturation in [byte, word] |
| | PSUBUS | B,W | Subtract unsigned with saturation on [byte, word] |
| | PMULH | W | Packed multiply high on words |
| | PMULL | W | Packed multiply low on words |
| | PMADD | WD | Packed multiply on words and add resulting pairs |
| Comparison | PCMPEQ | B,W,D | Packed compare for equality [byte, word, doubleword] |
| | PCMPGT | B,W,D | Packed compare greater than [byte, word, doubleword] |
| Conversion | PACKUS | WB | Pack words into bytes (unsigned with saturation) |
| | PACKSS | WB,DW | Pack [words into bytes, bytes into words] (signed with saturation) |
| | PUNPCKH | BW,WD,DQ | Unpack (interleave) high-order [bytes, words, doublewords] |
| | PUNPCKL | BW,WD,DQ | Unpack (interleave) low-order [bytes, words, doublewords] |
| Logical | PAND | Q | Bitwise AND |
| | PANDN | Q | Bitwise AND NOT |
| | POR | Q | Bitwise OR |
| | PXOR | Q | Bitwise XOR |
| Shift | PSLL | W,D,Q | Packed shift left logical [word, doubleword, quadword] by amount specified in MMX register or immediate value |
| | PSRL | W,D,Q | Packed shift right logical [word, doubleword, quadword] by amount specified in MMX register or immediate value. |
| | PSRA | W, D | Packed shift right arithmetic [word, doubleword, quadword] by amount specified in MMX register or immediate value |
| Data transfer | MOV | D,Q | Move [doubleword, quadword] to MMX register or from MMX register |
| FP & MMX State | EMMS | | Empty MMX state |

B = Byte
W = Word
D = Doubleword
Q = Quadword