Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete
Master's Thesis

Implementation Aspects of Image Processing

Per Nordlöw

LITH-ISY-EX-3088

Mars 2001

Abstract

This Master's Thesis discusses the different trade-offs a programmer needs to consider when constructing image processing systems. First, an overview of the different alternatives available is given followed by a focus on systems based on *general hardware*. General, in this case, means mass-market with a low *price-performance-ratio*. The software environment is focused on UNIX, sometimes restricted to Linux, together with C, C++ and ANSI-standardized APIs.

Contents

Chapter 1

Overview

As long as Moore's Law continues to double the transistor density in computers approximately every 18 months, more advanced operations can be performed on larger data sets in shorter amounts of time. In the field of image processing this means that more complex features of digital images can be extracted within time limits small enough to be used in situations where the computing devices have limitations in size, weight and power consumption.

But these speedups are often theoretical and in practise they do not come automatically. This means that developers, who want to make efficient use of new computer systems, first have to spend a lot of time learning new development tools and understanding these systems' possibilities and limitations.

Because the image processing algorithms used are normally already known and implemented in some high-level language, the implementation instead becomes a *code conversion.* As this often is a very time consuming and, for most users, uninteresting process, it is important to develop tools that can *automate* as many steps in this process as possible.

The purpose of this Thesis can followingly be divided into these main parts:

- Overview the hardware and software choices we have when implementing image processing operations on systems with high performance demands and restrictions mainly in power consumption and response time.
- When automatic code conversion is possible briefly discuss it and look at the tools available.
- In more detail investigate the possiblities of using general desktop PC hardware together with open software to design a real-time video processing system.

The following description overviews the chapters of this Thesis:

Chapter 2 briefly describes the different paradigms of parallel computing.

Chapter 3 overviews different types of computer architectures.

Chapter 4 then applies these concepts which results in a guideline on how to construct an image processing systems that automatically takes advantage of locality and scalability. At the end of the chapter, the key steps of this guideline are then experimentally exemplified. The Sobel operation follows through the guideline and gives the reader examples of how theoretical concepts are applied.

Chapter 5 explains how I add a video source to the system, described in Chapter 4, by plugging in a non-expensive PCI TV-card. Here I also try to answer the question of how complicated image processing there is time for on the video stream grabbed from the TV-card.

Chapter 6 discusses additional system restrictions associated with power consumption and choice of Operating System (OS).

Chapter 7 finally summarizes all the previous chapters and draws related conclusions.

Performance of the algorithms are measured on the following test systems (TS):

Abbreviation	*CPU*	*RAM*
TS 1	AMD K6-2 300MHz	192 MB
TS 2	2 × Intel Pentium Xeon 300 MHz	2 GB
TS 3	Sun UltraSPARC-IIi 440-MHz	256 MB
TS 4	4 × Sun UltraSPARC-II 400MHz	4 GB

Chapter 2

Parallel Processing

Parallel Processing refers to the concept of reducing the execution time of a calculation by dividing the problem into multiple parts. By making these parts execute indepently and simultaneously, we can draw advantage of the different multiprocessor architectures that are becoming more and more popular today.

Before designing or using such a system there are some things we have to take into consideration and the following sections will try to categorize and describe these. It is important to understand that, in practise, the hardware and the software developer must work together to optimally utilize the ideas of parallel processing.

2.1 Sequential, Parallel, Concurrent Execution

Most programs are today designed to execute in a sequential order, i.e. a previous part of the program has to complete its execution before the subsequent part can begin its execution. This approach works fine on uni processors (UP) systems (systems with one processor) but when it comes to making good use of multi processor (MP) systems (systems with more than one processor) the execution models become more complicated, and some parts of the program has to be rewritten to enable parallelism in execution.

As good as every modern computer is using an operation system (OS) that supports *multitasking*. These are examples of systems in which the programs in the system execute *concurrently*. In this case this means that multiple processes or multiple parts of a program (threads) share the processing in the computer over time eventhough there may be only one processor installed at the time. This is also called “timeslice-sharing” of processing. The sharing and scheduling of these are, as said above, hidden in the OS and to the application programmer the tasks “appear” to execute in parallel.

2.2 Task and Data Parallelism

There are basically two main approaches we can use when adding parallelism to an algorithm.

The first one, referred to as *task parallelism* or *functional decomposition* tries to exploit independencies between different parts of the program in order to reschedule these to execute in parallel. As an example, Figure 2.1 illustrates two different functions F_1 and F_2 that operate in parallel on their arguments in_1, out_1, in_2 and out_2.

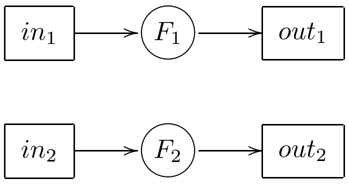

Figure 2.1: Task Parallelism

The second one, referred to as *data parallelism* or *domain decomposition* searches for a way to divide the data being processed into parts that can be operated on independently of each other. As an example, Figure 2.2 illustrates two instances of the same function F being applied in parallel on two separate parts of the data sets *in* and *out*. D and C represent the domain decomposition and composition stages respectively.

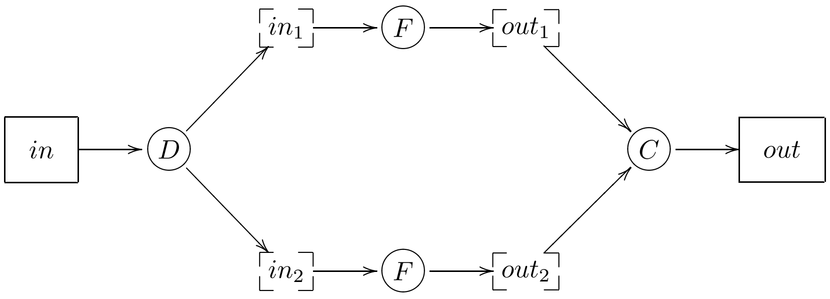

Figure 2.2: Data Parallelism

2.3 Architecture Independent APIs

When it comes to the real implementation stage of parallel algorithms there are several different tools or APIs to use, each suitable for different classes of problems. In this section we will summarize some Application Programming Interfaces, or APIs, that can be regarded as general and independent of the computer architecture on which the compiled code is supposed to execute.

2.3.1 Parallel languages

The first and perhaps the most simple way to make use of MP-systems is to use a programming language in which you can specify parallelism. Examples of these are Compositional C++ (CC++), pC++, Fortran M (FM) and High Performance Fortran (HPF). These languages give the programmer a high level interface with which he can express his parallel thoughts. The disadvantage is limited platform support because they are bound to a special set of hardware architectures.

CC++ [1] is a small set of extensions to C++ which as of today are targeted for task parallelism. These extensions give the programmer control over locality, concurrency, communication and mapping. In the future focus will be on the High Performance C++ (HPC++) which is a joint project between the CC++ group and the data parallel pC++ project. The goal is to offer both data- and task parallelism.

FM [2] is a small set of extensions, that add support for tasks and channels in Fortran (Formula translation). A special feature of FM is that it can guarantee programs to be *deterministic*, which means that two executions with the same input will give the same output.

HPF [3] extends Fortran 90 to provide access to high-performance architecture features while maintaining portability across platforms.

2.3.2 Threads

A more general and portable but also lower-level way of expressing both task and data parallelism in a computer program is to use *threads*. On UP-systems its main purpose is to enable concurrent execution of different parts of a process. On some MP-systems (systems with more than one processor) these threads are automatically distributed over all processors. Threading is also useful in client-server communication and in GUIs where several tasks need to respond to requests simultaneously.

One can think of threads as "mini-processes" inside a process that, in contrast to the process which has sole possession of its resources, all share the *same resources*, such as file descriptors and allocated memory. In practise, threads are used by calling a set of threading-functions from a library. These functions control the construction, communication, synchronization, and destruction of the threads.

There exists several different threading APIs — all with their own interfaces. When portability is of great concert, the standardized and widely used "POSIX Threads API" (P1003.1c) should be used.

2.3.3 Message Sending Interfaces

When it comes to making use of clustered computers, i.e. computers connected to each other through a high-speed network, another technique is used to explore parallelism. Because of the high communication latencies and the relatively low bandwidth of the systems one often wants to minimize the data sent between each process. Therefore the programmer explicitly specifies the messages that are sent between the different processing elements. The most commonly used APIs for this purpose are MPI (Message Passing Interface) and PVM (Parallel Virtual Machine).

[1] `http://www.compbio.caltech.edu/ccpp/`
[2] `http://www.netlib.org/fortran-m/index.html`
[3] `http://www.crpc.rice.edu/HPFF/home.html`

If the programmer needs to go down to a lower level of the network communication it is common to make use of a Socket API, e.g. BSD Sockets on an UNIX environment, and Winsock on a Windows based machine. Here, the programmer can specify exactly how the network communication should take place and thus avoid all unnecessery overhead that might occur if we only need a small set of functionality in our communication.

See Table 2.1 for a summary of the APIs mentioned above. The Center for Research on Parallel Computation (CRPC)[4], could also be of interest.

	Parallelism	
API	*Task*	*Data*
CC++	Yes	No
pC++	No	Yes
HPC++	Yes	Yes
FM	Yes	Yes
HPF	Yes	Yes
Threads	Yes	Yes
MPI, PVM	Yes	Yes

Table 2.1: Parallel Processing APIs.

[4]`http://www.crpc.rice.edu/CRPC/`

Chapter 3

Computer Architectures

3.1 Flynn's Taxonomy

The usual method for categorizing different kinds of computer architectures on a higher and more theoretical level is to use Flynn's Taxonomy. It describes the parallelism in terms of a *data* stream and an *instruction* stream as follows.

SISD or Single Instruction Single Data (SISD), represents the conventional way of looking at a computer as a serial processor—one single stream of instructions processes one single stream of data.

SIMD or Single Instruction Multiple Data (SIMD), is the most commonly explored parallelism in the microprocessors for todays desktop PCs. This approach reduces both hardware and software complexity compared to MIMD but is suitable only for special kinds of problems which contain much regularity, such as, *image processing*, *multimedia*, *signal processing* and certain numerical calculations.

As the first two of these are popular in today's mass-market consumer applications, it is therefore profitable to use SIMD-techniques in processors of these systems. The well-known extension MMX, 3DNow! and AltiVec are all examples of the special SIMD technique SWAR, which will be further discussed in Section 3.8.

For industrial applications digital signal processors, or DSPs, are popular architectures for analyzing digitized analog signals. These are also based on the SIMD idea.

MISD or Multiple Instruction Single Data (MISD), does not really occur as a real-world example of an architecture but is rather here in order to complete the taxonomy. The idea of executing a series of instructions on the same set of data is not totally alien though. One could say that instruction pipelining, belongs to this category. This however occurs at the sub-assembly level in the processor and is not something that the programmer has to care about, not even the assembly programmer.

MIMD or Multiple Instruction Multiple Data (MIMD), is the most general architecture of the above mentioned. In this we have the ability to explore both task and data parallelism at the same time. In most OSs the task parallelism is automatically used when executing different processes all with their own private resources. But there is also the possibility of using task parallelism inside of a process, using threads which we discussed in Section 2.3. Examples of MIMD-machines are the MP-systems often used in large LAN and web servers. Sometimes we impose a restriction on the MIMD idea by making all processors run the same program, and we say that such a system belongs to category of Single Program Multiple Data (SPMD) systems. Opposite to SIMD, each processor can in this case take a different execution path through the program. Programs parallelized through MPI or PVM execute in this way.

Architectures of today do not necesessary fit distinctively into one of these categories but instead more often makes use of some or all of them. Therefore the purpose of the taxonomy is rather to provide developers with a set of theoretical concepts with which they can express their thoughts and ideas.

3.2 Memory Hierarchies

Probably the most important and fundamental architectural principle around which computer systems have been built and are being built is often called the “90–10 Rule”. This rule states that on average

90 % of the operations are performed in 10 % of the code.

To take advantage of this assumption, we should try to organize the memory as a *hierarchy*. On the highest level we have the smallest sized but also the fastest memory and vice versa. At all levels except the lowest the data being kept in the memory is actually only a mirror of the memory kept in the lowest. When a program is repeatedly using a small part of the data at some level, it first copies it to the highest suitable memory level before the processing of it starts. Of course when the processor changes data that is not placed in lowest level the original copy has to be updated. Therefore

A write operation is on average more time-consuming than a read operation.

A typical modern PC usually has a memory hierarchy that consists of at least five levels. At the first level usually lies the interal registers accessible without any delay at all. Then comes the L1 Cache, which is usually placed right besides the logic gates on the microprocessor. Often the first half of it is used for instructions and the second half for their data. The third level, normally referred to as the L2 Cache, normally sits on top of the microprocessor and has an access latency of a couple of clock cycles. In some CPUs, such as the Alpha processors, a L3 Cache is also present. See Table 3.1 for details.

Next we have the primary memory. Here, accesses are limited by the external bus speed. Today the most dominating bus type on PCs is the 32-bit PCI bus (Peripheral Component Interface) running at 33 MHz, thus providing a peak-bandwith of $4 \cdot 33 = 132$ Megabytes per second. The PCI bus is optimized for burst-reads of data and its peak bandwidth is only achieved in applications where a large degree of data parallelism, as in image processing, is present. In

Memory Type	*Access Latency*
Register	2ns
L1 on-chip cache	4ns
L2 on-chip cache	15ns
L3 off-chip cache	30ns
Main Memory	220ns

Table 3.1: Memory hierarchy in a 500MHz DEC 21164 Alpha.

cases of individual reads and writes of 32-bit integers that bandwidth is normally decreased to around a fourth of its peak bandwidth.

The next level normally called secondary memory and is often a harddisk.

The last level would in most consumer applications not be regarded as a memory but rather a communication level. But in the cases of networked workstations connected to a centralized file server or clustered computers with local harddisks it indeed belongs to the memory hiearchy and is used as a such.

3.3 Data Locality

Memory hierarchies are present in all modern computer systems and the number of levels and their relative differencies in bandwidth and access latency are constantly increasing. As an example, the external bandwidth of the PCI-bus has not changed at all during the last six years compared to the L1 cache today operating at the internal CPU frequency which double approximately every 18 months.

Adapting our code so that it makes use of this property is therefore a *long-term* and *platform indepedent* way of optimizing our algorithm. Such an implementation is said to use *locality.*

Traditionally, the definition of locality can be formulated like

For each memory reference, perform as many operations as possible.

This rule is, however, out of date, as it assumes only two memory levels—the primary memory and the CPU registers. A more up to date definition would instead be formulated like

For each new reference of an element at a specific memory level, reuse the element in operations at higher memory levels as much as possible.

3.4 Instruction Locality

The same rules of locality also apply to the organization and execution of the program code.

- When the code is compiled into CPU instructions, reuse of functions gives good locality.
- The same goes for interpreted code, but here the interpreter can be provided with extra functionality that enables run-time restructuring of the

code according to historical information about which parts of the code that is most commonly used. A good example of this, is the constant advances being made in the Java interpreters.

As good as all OSes provide shared libraries, that contain common functions used by many application. If performance has the highest priority, it can be a good idea to include these functions in the executable, which instead prioritizes instruction locality and performance before memory usage.

3.5 Shared and Distributed Memory

There are basically two main branches of MIMD memory architectures; Shared Memory Architectures and Distributed Memory Architectures.

Traditionally the difference lay in the organization of the memory. If all processors shared a common memory we had a shared memory architecture with a high interconnection bandwidth. If we, on the other hand, preferred high local bandwidth we instead chose a distributed memory architecture.

As the number of stages in memory hierarchies are constantly increasing, the separation into shared and distributed memory systems is no longer distinct. For example, all MP-systems for the Pentium Processor and later have a shared primary memory together with separate L1 and sometimes L2 caches. See Figure 3.1 for an illustration.

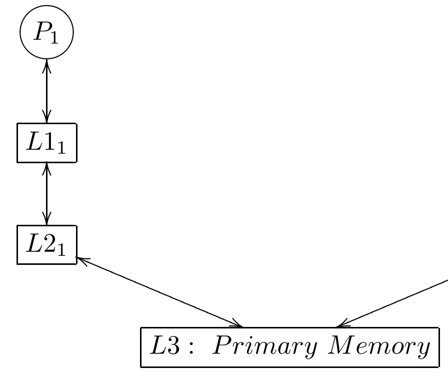

Figure 3.1: Memory hierarchy in an MP-system with two processors P_1 and P_2, each having separate L1 and L2 caches together with a shared primary memory.

This design is another result of the "90–10 Rule" defined in Section 3.2. Because the bottleneck in such MP-systems is the bandwidth of the primary memory, locality in this case becomes crucial to performance. Briefly stated this means that

The scalability in shared memory MP-systems increases with the locality.

3.6 Pipelining

Pipelining is the concept of dividing a complex operation into several consecutive less complex operations that operates on a stream of data which contains a high degree of data parallelism. Just as the maximum throughput of oil in a pipeline is directly related to its thinnest part, the maximum throughput of such a computational pipeline is linearly dependent on the throughput of the slowest stage in the pipeline.

Therefore, if we want to make efficient use of the hardware on which we are to implement our algorithm, it is important to carefully examine which stages in the pipeline that take the longest time to finish. Once we have this knowledge we can focus our optimization efforts on these stages.

Pipelining is extensively used in the construction of arithmetic units of microprocessors but we can also, in conjuction with ideas of memory hierarchies and cache memories, make use of it in the design of software, especially image processing software. See Figure 3.2.

$$\begin{tikzcd} \mathit{in} \arrow[r] & S_1 \arrow[r] & S_2 \arrow[r] & S_3 \arrow[r] & \mathit{out} \end{tikzcd}$$

Figure 3.2: Computational pipeline with 3 stages S_1, S_2 and S_3.

3.7 Clustered Computing

Clustered Computing is a new trend in the construction of large parallel computer systems. It uses ordinary home PCs together with a high speed network to construct a parallel machine. For special kinds of applications its largest advantage, in comparison with other architectures such as SMP systems, is its very low *price-performance ratio*. This is because it uses standard PC workstations as its building blocks, often called Mass-Market Commercial Off The Shelf (M^{2}COTS) hardware.

Its largest disadvantange is its inter-communication bottleneck. Therefore clustered computers are only competitive when doing computations that are not dependent on low communication latencies and high bandwidths. Typical areas that fit these restrictions are physical simulations and ray tracing.

It is very popular to use Linux as the OS on a clustered system since it is free, has excellent networking hardware and protocol support, and has an open source. The Beowulf Project is an example of this and it uses only Linux when developing its cluster software platform BPROC.

Other popular APIs in the area are the traditional PVM and the newer MPI upon which much cluster software has been built. If one is interested in more optimized network communication, that is controlling individual network packets, one has to go down one level to the socket layer, described in Section 2.3.

Another important factor is how the individual processing nodes are connected to each other, the network *topology* of the cluster. BPROC has tried out different topologies with the experience that the *linear switched* network, with its general high performance, is to prefer. Otherwise, the programmers tend to make the software too specialized.

From an image processing point of view these technologies could come in handy in the future as the access latencies decreases and bandwidths keep on growing. In fact, disregarding the access latencies and the non-deterministic transmission time of Ethernet, the peak bandwidth of Gigabit Ethernet is actually sufficient to make clusters useful in image processing.

3.8 SWAR

SWAR[1], which stands for "SIMD Within A Register", is a general name for the concept of treating an m-bit (m is normally 64 or 128) processor register as a set of n (typically 2, 4, 8, ...) smaller m/n-sized registers. Operations are then performed in parallel on these m-bit wide sub-fields.

Some SWAR operations can be performed trivially using ordinary m-bit integer operations, without concern for the fact that the operation is really intended to operate independently in parallel on n sub-fields. Such a SWAR operation is said to be *polymorphic*, since the function is unaffected by the field types (sizes). Testing if any field is non-zero is polymorphic, as are all bitwise logic operations. Most other operations are not polymorphic though, and special hardware is often constructed when high performance of operations with sub-field precision is wanted.

Many high-end microprocessors have, within the recent 5 years, added specialized machine instructions that increase the performance of SIMD-operations. The most well-known example of this is the MultiMedia eXtension set (MMX) which first appered in the Intel Pentium MMX processor. This instruction set is now a standard component in all of Intel's 32-bit architectures (IA-32) compatible with the Pentium processor. Other vendors, specifically Cyrix and AMD, have also chosen to implement MMX in their microprocessors, thus providing total compatibility with the Pentium MMX processor, an important issue in the consumer mass-market for Windows compatible CPUs.

3.8.1 SWARC and `Scc`

Table 3.2 lists other architectures with SWAR extensions in their instruction sets. Aside from the three vendors Intel, AMD and Cyrix who have agreed on MMX, all of these instruction set extensions are roughly comparable, but mutually incompatible.

Architecture	*Extension*
Intel Pentium MMX	MMX
Intel Pentium III	KNI/SSE
AMD K6-2	3DNow!
AMD Athlon	3DNow! 2
Sun SPARC	V9 VIS
Motorola PowerPC G4	AltiVec
Digital Alpha	MAX
HP PA-RISC	MAX
MIPS	MDMX

Table 3.2: Architectures with SWAR extensions.

[1]Named "Sub-Word Parallelism" (SWP) in signal processing.

If we want to take advantage of these instructions sets this incompatibility makes it virtually impossible to write platform independent code by hand. And adapting a specific algorithm to a specific platform by hand is very time-consuming. These leads to us to the question:

Can we automate this process?

I have only found one project, "The SWAR Homepage at Purdue University"[2], which addresses this problem. They are developing a compiler (`Scc`) that inputs code in the form of the platform independent SWAR-extended C language "SWARC" and outputs code containing C language together with platform dependent macros and inline assembly. The output together with some include files[3] is, in turn, compiled to machine code using a C compiler and an assembler.

The project is aiming at supporting all of the platforms in Table 3.2 efficiently. All of these instruction sets are however mutually incompatible and some or many of the operations are not supported on some or any of the data types. Much of the effort is thus focused on the *code conversion problem*: how one can implement these missing operations using existing SWAR instructions or even conventional 32-bit integer functions. An efficient construction of such a general SWAR-compiler is therefore a very tricky task.

`Scc` can currently output SWAR-optimized code using either MMX, 3DNow!, generic IA-32 code or a combination of these. In the case of IA-32 code the compiler tries to use ordinary 32-bit integer functions when operating on sets of smaller sized integers. The compiler also supports parallel operations on integers with *arbitrary bit precision* (smaller than integer precision, though). The precision syntax is similar to C's bitfield syntax. As an example, the following piece of SWARC code

```
void add_8xS16(int:16 [8] z, int:16 [8] x, int:16 [8] y)
{
    z = x + y;
}
```

specifies a function with three arguments. These are vectors of signed 16-bit integers each of length 8. We see that vector operations are easily expressed using the normal C operators such as `+`, `-`, `*`, `/` etc. For further details, see SWARC's grammar which can be found in [2].

I tested the performance of the code generated by `Scc` compared to ordinary scalar C code. Three test operations, namely addition $z_i = x_i + y_i$, absolute value $x_i = |y_i|$ and l1-norm $z_i = |x_i| + |y_i|$ were applied to 16 elements long vectors of different sized signed integers. In order to make the execution times measurable each operation were run 2^{20} times. The benchmarks can be seen in Table 3.3.

It is apparent that `Scc` generates fast code for simple operations that are easily expressible with target instructions, in this case MMX. But when it comes to implementing functions that are not e.g. 16-bit absolute value, the code generated is actually slower than the scalar C code. Consequently, `Scc` is currently only useful when optimizing simple vector operations for the Intel platform.

[2]`http://shay.ecn.purdue.edu/~swar/`

[3]Currently `swartypes.h` plus either `Scc_3dnow.h`, `Scc_athlon.h`, `Scc_max.h`, `Scc_sse.h`, `Scc_altivec.h`, `Scc_ia32.h`, `Scc_mmx.h` or `Scc_xmmx.h`

Operation	*Precision*	*C*	*MMX*
Addition	8-bit	108ms	45ms
Addition	16-bit	147ms	56ms
Addition	32-bit	91ms	98ms
Absolute value	16-bit	269ms	315ms
l1-norm	16-bit	521ms	727ms

Table 3.3: Performance difference between scalar C code and `Scc`-generated MMX-optimized code run on TS 1.

`Scc`'s source code is available online as public domain in a testable alpha state. It is however not necessary to download `Scc` in order to use it because SWARC's website contains a test page[4] in which the visitor, through a HTML-form, can use the compiler. The output together with the appropriate include files can then be compiled using gcc. For the more interested reader, the webpage also links indepth articles that discuss the design of a the SWARC language [2] and its compiler `Scc` [3].

3.8.2 SWAR in `gcc`

Not all operations are hard to hand-code using SWAR-operations, especially not if the GNU C Compiler `gcc` together with the GNU Assembler `as` are available on our target system. Here we can *express assembler instructions with arguments specified as C expressions.* This has several advantages:

- All the code belonging to an algorithm is placed together in the source code in an intuitive manner. No separate assembler files are needed.
- The programmer can concentrate on the relevant matters—which SWAR-instructions that should be used, instead of bothering about how to set up the local function stack, push and pop registers, and other low-level assembler matters. This makes the overhead of programming in assembler minimal.

As an example, consider the two-dimensional affine transform

$$\mathbf{x} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b}$$

where

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

This calculation can be implemented with 32-bits floating precision using 3DNow! instructions as follows

```
void f32_xAxb(f32_t *x, f32_t *A, f32_t *b)
{
    asm("movq   (%0), %%mm0 \n\t"     /* x[0,1] => mm0 */
        "movq   (%1), %%mm1 \n\t"     /* A[0,1] => mm1 */
```

[4]`http://shay.ecn.purdue.edu/~swar/SccTest.html`

```
        "movq  8(%1), %%mm2 \n\t"     /* A[2,3] => mm2 */
        "movq   (%2), %%mm3 \n\t"     /* b[0,1] => mm3 */

        "pfmul %%mm0, %%mm1 \n\t"
        "pfacc %%mm1, %%mm1 \n\t"     /* first dot => mm1 */
        "pfmul %%mm0, %%mm2 \n\t"
        "pfacc %%mm2, %%mm2 \n\t"     /* second dot => mm2 */

        "punpckldq %%mm2, %%mm1 \n\t" /* A*x   => mm1 */
        "pfadd %%mm3, %%mm1 \n\t"     /* A*x+b => mm1 */
        "movq  %%mm1,  (%0) \n\t"     /* mm1   => x */

        "femms              \n\t"     /* cleaup mmx state */
        :                             /* out args to C */
        : "r"(x), "r"(A), "r"(b) );   /* in args from C */
}
```

I compared this algorithm at the highest level of locality with a scalar version written in C. On TS 1 this gave a speedup factor of approx. 4.6.

For more information see the info pages on gcc. Currently the GNU Assembler supports the SIMD-instruction sets MMX, 3DNow! and 3DNow! 2.

3.8.3 Trend

The current trend of PC processors is to add more SWAR-versions of the traditional scalar machine instructions addition, subtraction, multiplication and division. This means that gaps in the SWAR instructions sets are filled in which should make it easier to construct compilers that output efficient code. This is because less special cases, involving code conversion, need to be handled.

As an example, both the new PowerPC G4 and the next 64-bit Intel architecture (IA-64[5]) will contain the very general *permutation* operation[6]. This will significantly speed up data conversion and other structure operations often used in signal and image processing and its implementation will be simple and easy to automate.

This trend of packing more instructions into processors is expected to continue. The main reason for this is that *the transistor packing density in CPUs grows faster than their clock frequency.* Assuming large data streams and parallel operations, a duplication of the instruction decoders theoretically doubles the performance of these SIMD-operations.

Viewed from a large perspective, this SWAR-trend actually means that RISC processors and DSPs are converging towards a single CPU. The most apparent example of this is the PowerPC G4.

3.9 Reconfigurable Computing

A totally different kind of computing category, which in the long run probably will become the *optimal solution* to the code conversion problem, is *reconfig-*

[5]Currently code named "Itanium"

[6]On the G4, this operation is capable of arbitrarily selecting data with the granularity of one byte from two 16-byte source registers into a single 16-byte destination register.

urable computing. These hardware technologies can, analogously with the learning process of the human brain, be adapted according to what operations that are to be performed. Because these reconfigurable hardwares are so general, *the code conversion process only has to be solved once for each programming language.* The software programmer, on the other hand, has to take more factors, such as minimal chip area and bit-level operations, into consideration during the design and optimization stages. This is outweighed by the fact that the code is very generic and long-lasting.

The basic technology consists of a silicon chip with a set of logical units that can be reprogrammed (reconfigured) each time a new set of functions is needed in the application, something that is called a Field-Programmable Device (FPD). The three main catogories of FPDs are delineated:

- Simple PLDs (SPLDs)
- Complex PLDs (CPLDs) and
- Field-Programmable Gate Arrays (FPGAs).

Of these, the FPGA has the highest capacity (measured in number of 2-input NAND-gates) and is therefore the most commonly used when mapping more complex applications onto these kinds of logic. Under the right circumstances these circuits have a speed performance of *two orders of magnitudes* relative to that of CPUs [10]. They do not offer the same performance as Application Specific Integrated Circuits (ASICs) which, on the other hand, lack the possibility to be reprogrammed.

3.9.1 Reconfigurable Applications

Image processing involves the analysis of very large amounts of data, with a high degree of parallelization possibilities in the algorithm and with a relatively low number of bits required per data element. Most arithmetic units on CPUs today are built for processing data with a precision of 32 or 64 bits. Because 16 bits precision often is enough when performing image processing, a lot of unneccessary processing is done on bits never used when using these CPUs.

When using FPGAs, on the other hand, both these two factors can be taken into consideration and utilized very effectively. For example, arithmetic operations can be specified on arbitrarily sized integers and several processing units of the same type can be configured to make use of the data parallelism.

One big disadvantage with this idea is that it means a lot of work for the programmer who has to redesign the algorithms for the operations to suit these further restrictions, especially the data precision issue. Therefore a high level programming environment which supports modular programming and reuse of code become important issues. Furthermore the FPGA-field is a relatively new and unexploited area in comparison to CPUs and their code development tools there are few libraries available to use. For further information see [10].

Programmable logic has also been successfully used in the development stages of new microelectronics and in some mixed computer architectures that make use of its unique benefits together with traditional microprocessors and random access memories. Most of the code on such a system is run on the traditional hardware except for the most time consuming parts that are small enough to fit onto the reconfigurable logic part, in this way using the best of both worlds. Also in this case we use the concept of locality to optimally fit hardware and software together. In these cases, the programmable circuits being used are often called FPGA-coprocessors. An example of such a system is the MATCH project, which will be covered next.

3.9.2 Mixed Architectures

MATCH

The MATCH ("Matlab compiler for heterogeneous computing systems") project is addressing the code conversion problem by trying to build a development environment that can overcome the barriers between low-level languages, especially C and VHDL (a hardware description language often used to describe FPGA designs), and the widely used high-level language Matlab, suitable for the testing of numerical algorithms. As the word "heterogeneous" in the name implies, the target architectures consists of several different computing technologies, specifically a general-purpose CPU, a DSP and a FPGA board. The different parts are all good at different things and it is not apparent how the processing should be divided between the parts. The solution to this problem can be summarized in the following steps:

1. Parse the Matlab code into a control and data flow graph (CDFG).
2. Partitioning this graph into sub-graphs that can be invidiually mapped to different computer architectures in the target system. This step also involves the administration of buffer layouts and communication.
3. Finally generate the code for the different components and, in turn, use their respective compilers to produce the final object code that they understand.

The following list shows the project's eight main tasks in more detail together with their predicted efforts (in parenthesis).

1. Development of a hardware testbed (10%).
2. Implementation of Matlab to C and VHDL compiler. (30%)
3. Automatic parallelism and mapping on heterogeneous resources while optimizing performance under resource constraints or vice versa. (15%)
4. Development of Matlab compiler directives that specify type, shape, precision, data distribution and alignment, task mapping, resource and time constraints. (10%)
5. Evaluation of adaptive applications. (10%)
6. Development of basic primitives such as FFT, FIR/IIR filtering, matrix addition and multiplication operations. (15%)
7. Development of interactive tools for function composition and logic synthesis. (5%)

8. Development of faster algorithms for compilation, such as parallel or distributed algorithms for logic synthesis. (5%)

The MATCH project was initiated on Jun. 30, 1998 and is sponsored by NASA in order to be used in their upcoming space shuttles for earth observing systems. According to its planned milestones the project should, at the time of this writing, be finished and a demonstration is planned at the end of the first quarter 2001.

Chapter 4

Implementation Aspects

In this chapter we will discuss the different aspects we have to take into account when we turn from the ideal theoretical description of an algorithm to the description that is optimal in an environment with a specific programming language and computer architecture.

The focus is on general computer architectures, such as PC-systems, together with C or C++ as the software environment. On the MP-systems POSIX Threads are used to parallelize image processing operations.

4.1 Programming Languages

4.1.1 Matlab

Matlab is, at an early stage in image processing research, a very handy language that enables quick implementation and testing of ideas. The machine precision of 64-bit floating point number is enough to cover most of the demands in our field. It is also quite fast if we can express our algorithms as matrix operations. But as Matlab is an interpretive language, more complex operations involving lot of nested loops tend to run very slow. Another problem with Matlab is its unpredictability in execution time because of its garbage collector. This also means that we have no explicit control of matrix buffers which in most cases is crucial for performance. The ability of Matlab to express matrix operations in a high level notation is very convenient and it can also automatically generate C code or assembler code for different kinds of embeddable processors especially DSPs. But the test platform itself is best suitable for research, and not for running code in embedded systems.

4.1.2 C and C++

My main approach is to implement image processing algorithms in different ways and then investigate the differences in performance and memory usage between these implementations. My target architecture are desktop computers because of their availability. Therefore C is a natural choice for me because of its great portability, performance possibilities to investigate low-level aspects of operations.

For more complex projects, higher level languages such as C++ or Java would be to prefer. As these are based on C, porting the code should not be a big effort. If needed, it is also possible to call functions in C-coded object files from either C++ or Java.

Because of its free and open development environment and because of my earlier experince with Linux, I chose to develop all the code on a PC Linux system using the C and the C++ compiler in the GNU Compiler Collection, also known as GCC[1] This compiler is well-known for its high performance and large target support[2]. All code was compiled with the switch `-O6`, which certifies maximum speed optimization in current and upcoming versions of GCC.

4.2 Matrix Storage Techniques

4.2.1 Matrix Elements

Computers are designed to efficiently operate on rectangular regions and most images are digitized using a regular sampling grid. Thus, the storage techniques I will discuss are all based on the assumption that the elements together form a rectangular region of data. A row and a column is regarded as a special case of such a rectangle.

To accompany the textual explanations I have added a graphical explanation for each technique that is discussed. These can be viewed in the Figures 4.1, 4.2, 4.3 and 4.4. In these a curly arrow shows a pointer dereference and its direction. The straight arrows, on the other hand, indicate in what order the matrix elements, drawn as boxes, are stored in memory. The actual matrix position of such an element is expressed using double indexing (x, y), starting at $(1, 1)$.

Dense

The simplest and most straightforward way of storing a matrix in memory is to allocate one whole continuos memory block containing all the elements of the matrix. Such a matrix is said to be *dense*. A densly stored matrix is illustrated in Figure 4.1. In image processing the elements are normally stored like we read a text—*row-major*, left-to-right, starting with the first row. This is also called *lexicographic* order. In the following we assume that all dense matrices are stored in this order.

Row-Indexed Dense

When we access a matrix element with index (x, y) in a densly stored matrix, having the width w and height h, we perform the calculation $i = x + y \cdot w$ to

[1]GCC also contains front-ends for, Objective C, Chill, Fortran, and Java (GCJ) giving us great possibilities to mix different languages in the same project. Note that GCJ compiles Java source code to machine code, and is not a Java virtual machine.. For more information see `http://gcc.gnu.org/`

[2]Currently GCC supports alpha-dec-linux, alpha-dec-osf, DOS, hppa-hp-hpux, hppa-hp-hpux9, hppa-hp-hpux10, hppa-hp-hpux11, i386-linux, i386-sco3.2v5, i386-solaris, i386-udk, i386-ibm-aix, m68k-nextstep, m68k-sun-sunos4.1.1, mips-sgi-irix, mips-sgi-irix6, powerpc-linux-gnu, powerpc-solaris, sparc-sun-solaris, sparc-sun-solaris2.7, sparc-sun-sunos, GCC with Windows or OS/2.

$$\begin{tikzcd}
\boxed{M} \arrow[r, squiggly] & \boxed{(1,1)} \arrow[r] & \boxed{(1,2)} \arrow[dl] \\
 & \boxed{(2,1)} \arrow[r] & \boxed{(2,2)}
\end{tikzcd}$$

Figure 4.1: Dense storage of a 2×2-matrix.

get the linear index. By using this index together with the start address of the matrix we can then reach our element.

Sometimes the multiplication $y \cdot w$ can be costly and in these situations we allocate an additional vector containing the memory addresses to the starts of the rows in the matrix. This adds an extra pointer dereference for each new row that is to be read but on the other hand eliminates an integer multiplication. I will call these matrices row-indexed dense matrices (RID-matrices). An example of such a matrix is illustrated in Figure 4.2.

$$\begin{tikzcd}
\boxed{M} \arrow[r, squiggly] & \boxed{r_1} \arrow[r, squiggly] \arrow[d] & \boxed{(1,1)} \arrow[r] & \boxed{(1,2)} \arrow[dl] \\
 & \boxed{r_2} \arrow[r, squiggly] & \boxed{(2,1)} \arrow[r] & \boxed{(2,2)}
\end{tikzcd}$$

Figure 4.2: Dense storage of a 2×2 using row-indexes.

Tiled Dense

Some applications, such as geographical databases and image manipulation programs, operate on very large amounts of two-dimensional data. Here locality is more likely to occur in square-like rectangles of the pictures rather than in individual lines. Then *tiled dense* storage is used, which can be though of as a generalization of the normal dense storage. Scan-lines are simply tiles with a height of one pixel. See Figure 4.3.

In performance demanding applications, however, this technique is not useful because it complicates the algorithms and decreases their performance, especially when performing neighbourhood operations on the boundary of a tile.

Sparse

Sometimes only a smaller fraction of the elements stored in the matrix are non-zero. To reduce the memory consumption, we then store the data as a *sparse* matrix. Such as matrix contains an array of data-triplets, where each triplet contains the (x, y) indices together with the value of the matrix at the corresponding position. Figure 4.4 illustrates this.

4.2.2 Element Components

An element in a matrix is in some cases a structure itself. A complex variable has a real and an imaginary part and elements in color pictures normally consists

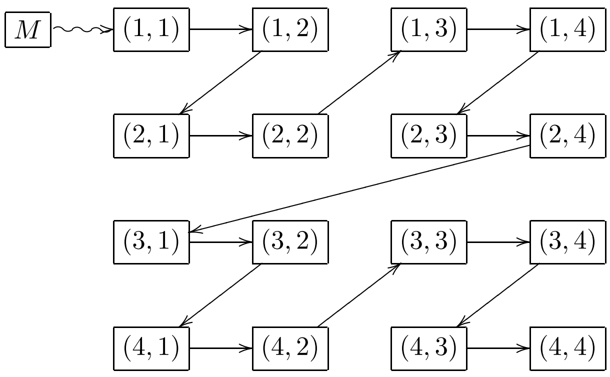

Figure 4.3: Tiled dense storage of a 4×4-matrix with a tile size of 2×2.

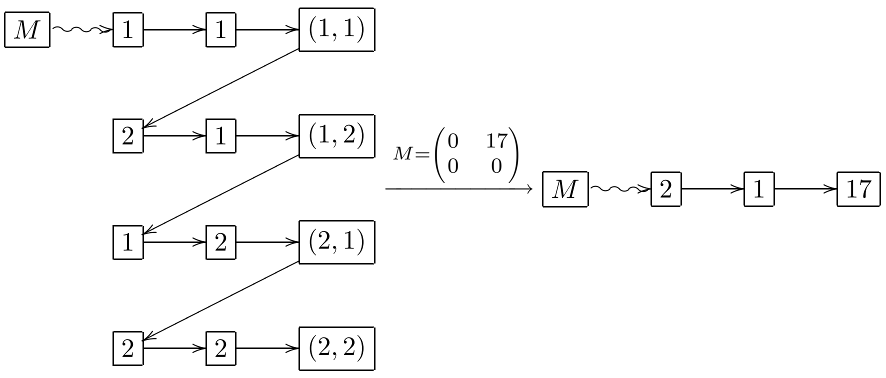

Figure 4.4: Sparse storage of a 2x2-matrix with one non-zero element.

of individual color components. The optimal storage structure of such a matrix is not obvious.

- The most intuitive way would be to store the elements *interleaved* so that their components lie next to each other. A true color picture P, where each pixel p_i consists of a red (r_i), a green (g_i) and a blue (b_i) component, would then be stored like

  $$\begin{tikzcd} \boxed{P} \arrow[r, squiggly] & \boxed{r_0} \arrow[r] & \boxed{g_0} \arrow[r] & \boxed{b_0} \arrow[r] & \boxed{r_1} \arrow[r] & \boxed{g_1} \arrow[r] & \boxed{b_1} \arrow[r, dotted] & {} \end{tikzcd} .$$

  As an example, this suits the way monitors display the contents of the video memory.

- In other cases the reverse can be more comfortable, i.e. storing the element components separately in a *planar* order. Our picture would then be stored like

  $$\begin{tikzcd} \boxed{P} \arrow[r, squiggly] & \boxed{r_0} \arrow[r] & \boxed{r_1} \arrow[r, dotted] & \boxed{g_0} \arrow[r] & \boxed{g_1} \arrow[r, dotted] & \boxed{b_1} \arrow[r] & \boxed{b_1} \arrow[r, dotted] & {} \end{tikzcd}$$

  instead. If we want to process each component independently of the other this representation is more handy. Assume, for example, that we have a function f_{grey} operating on a grey scale picture. Then we can simply *reuse* f_{grey} on the color components of a true color picture if they are represented with the same data type as the grey values. The same principle also applies to complex valued numbers.

If the access times of an individual element's components in these two alternatives are similar the latter is to prefer because of its general potential to provide *reuse* of functions.

4.2.3 Matrix Storage in C

During this work, I have designed a C matrix library with functions operating on a matrix structure defined as

```
typedef struct {
    uint w, h;          /* width and height */
    uint type;
    union {
        u8_t *u8;
        s8_t *s8;
        ...
    } data;             /* dense */
    union {
        u8_t **u8;
        s8_t **s8;
        ...
    } rows;             /* row indexed dense */
} M_t;
```

As seen, instances of the same structure type can contain different data types and all functions, except allocators such as `M_create()`, checks for this automatically using `switch`. To clarify the actual precision of the C types, I use the alternative names

1. `int` and `uint` are used when variables, such as loop counters, should adapt to the default signed and unsigned integer precision of the target CPU.
2. `u8_t`, `s8_t`, `u16_t`, `s16_t`, `u32_t`, `s32_t`, `u64_t` and `s64_t` represent unsigned (`u`) and signed integers (`s`), with a bit-precision indicated by the suffix. These are used when it is important that a type has a specific precision.
3. `f32_t` and `f64_t` represent 32-bit and 64-bit floating point numbers, respectively.

The structure `M_t` is designed to enable operations on both dense matrices and RID-matrices.

- In the dense case the `i`:th (indexing starts at zero in C) element with type `u8_t` of a matrix m is accessed by writing `m->data.u8[i]`. These dense matrices are assumed to be stored linearly in memory, in row-wise order starting with the first top row.
- By adding an additional data member `rows` to `M_t`, we can also implement operations on RID-matrices.

The `type` member in `M_t` together with a `switch(type)` statement in the functions give us automatic type checking with a low performance cost. To increase readability I will, instead of using pseudo code, sometimes strip away details unimportant for the understanding of the algorithm and, when needed, replace it with a descriptive comment in plain english.

4.2.4 Regions Of Interest in C

When performing image processing, it is often desirable to be able to specify sub-regions on our images and let our operations work on these. Such a region is often referred to as a Region Of Interest or ROI for short. I will be representing such ROIs using sub-rectangles which are specified by four integers, where the first two specify the upper left coordinates and the last two the dimensions of the ROI. In C this can be done according to

```
typedef struct
{
    int x, y, w, h;
} roi_t;
```

4.2.5 Access Techniques in C

In C we have the choice of storing and accessing a matrix in different ways. Different choices result in different implementations and performance within an architecture and between architectures. These differences will be illustrated by the implementation of a simple function that sets its elements to the value of the innerloop variable. The following local stack variables are assumed to already be defined:

```
int i, x, y;
const int w = m->w, h = m->h;
```

Dense When no ROIs are needed, the most straighforward way is to use a dense storage, which gives us the following solutions:

```
for (i = 0; i < w * h; i++)                 /* 1a */
    m->data.f32[i] = i;

for (y = 0; y < h; y++)                     /* 1b */
    for (x = 0; x < w; x++)
        m->data.f32[x + y * w] = x;

for (y = 0; y < h; y++)                     /* 1c */
    for (i = y * w; i < (y + 1) * w; i++)
        m->data.f32[i] = i;

for (y = 0; y < h; y++)                     /* 1d */
    for (x = 0; x < w; x++)
        m->data.f32[x + y * m->w] = x;
```

Dense with ROI When, on the other hand, ROIs are needed we include the ROI structure defined in Subsection 4.2.4 in the matrix structure `M_t` and get:

```
for (y = m->roi.y; y < m->roi.h; y++)
    for (x = m->roi.x; x < m->roi.w; x++)  /* 2a */
        m->data.f32[x + y * w] = x;
```

RID If we use row indexes in our matrix structure we replace the multiplication with an extra memory dereference and get

```
for (y = 0; y < h; y++)                     /* 3a */
    for (x = 0; x < w; x++)
        m->rows.f32[y][x] = x;
```

RID with ROI When RID-matrices are used, ROIs are *automatically* implemented because moving the ROI simply involves changing the row indexes. However, if the ROIs are moved around a lot, expressing ROIs with the four integers can yield higher performance:

```
for (y = m->roi.y; y < m->roi.h; y++)       /* 4a */
    for (x = m->roi.x; x < m->roi.w; x++)
        m->rows.f32[y][x] = x;
```

These implementations were then benchmarked and listed in the Tables 4.1 and 4.2. Note that the optimal matrix storage technique on a specific hardware

Code	TS 1	TS 2	TS 3	TS 4
`1a`	25	24	21	15
`1b`	33	49	63	48
`1c`	28	50	197	162
`1d`	57	63	178	154
`2a`	26	37	45	33
`3a`	27	50	35	27
`4a`	30	58	51	37

Table 4.1: Execution times in μs of different accessing techniques when matrix size is 64×32.

Code	TS 1	TS 2	TS 3	TS 4
`1a`	12	8	7	2
`1b`	12	8	8	5
`1c`	12	8	25	20
`1d`	13	8	24	20
`2a`	12	8	7	4
`3a`	12	8	7	3
`4a`	12	8	8	5

Table 4.2: Execution times in ms of different accessing techniques when matrix size is 512×512.

may depend on what operations we perform. For example, the addressing technique in `1b`, involving integer multiplication may be optimal when we perform floating point operations on the matrix elements. In other cases, involving integer arithmetic on elements, other storage techniques, such as `3a`, may instead be optimal.

The first choice of storage size makes the entire matrix fit in the L1 cache on all test systems which leads to greater differences between the implementations, than in the other case. Thus, the storage technique becomes more important if it makes use of locality in our algorithms.

Another thing to notice is the dramatical performance drop in `1d` compared to `1b`. The reason for this is not clear, but my guess is that the optimization rules in GCC somehow do not allow the expression `m->w` to be stored in a temporary register. The somewhat complicated syntax for accessing elements motivates the use of C macros, that in the RID-case looks like

```
#define f32(m, x, y) (m->rows.f32[y][x])
```

and could be used without any difference in performance.

```
#define f32(m, x, y) (m->data.f32[x + y * m->w])
```

on the other, results in really bad performance. If we write our code explicitly in C this means that all our storage techniques *can* be efficiently implemenemented in C, but not always together with a comfortable and general syntax. Thus, a general change of the underlying storage technique *cannot* be done without a tedious and uninteresting recoding of most functions.

One solution to this problem would be to construct a code converter, but writing such a converter is out of range of this Thesis. Instead, the next Section investigates if C++ can provide hiding of the storage and accessing technique together with a higher performance.

4.2.6 C++ Implementation

I constructed a class `TestMatrix` which includes a pointer to its data and to its row indexes together together with element accessing member functions[3], using Dense and RID accessing:

```
class TestMatrix
{
public:

    TestMatrix (int w = 1, int h=1) :
        width (w), height (h), rx (0), ry(0), rw (w), rh (h) {
        data = new f32_t [width * height];
        rows = new f32_t* [height];
        for (int r = 0; r < height; r++)
            rows [r] = & data [r * width];
    };

    ~TestMatrix() { delete [] data; delete [] rows; }

    f32_t & f32_dense (int i) { return data [i]; };

    f32_t & f32_dense (int x, int y) { return data [x + y * width]; };

    f32_t & f32_rid (int x, int y) { return rows [y][x]; };

protected:

private:

    int width, height;
    int rx, ry, rw, rh;         // ROI
    f32_t * data;
    f32_t ** rows;
};
```

All the functions shown in Subsection 4.2.3, except `1d` were then implemented as member functions of `TestMatrix`, benchmarked and listed in the Tables 4.3 and 4.4.

The RID-storage performs best on all platforms, but when comparing with the C versions in Subsection 4.2.5, C++ produces slower code in all cases but the last two, where performance is comparable. Thus, in this case C++ gives *only* a slightly more intuitive syntax with type checking when accessing matrix elements.

[3]To avoid an unneccessary loss in performance these functions should be declared with the attribute `inline`.

Code	TS 1	TS 2	TS 3	TS 4
`1a`	138	21	89	67
`1b`	92	20	191	161
`1c`	79	11	250	188
`2a`	133	11	200	150
`3a`	41	12	50	37
`4a`	41	20	50	37

Table 4.3: Execution times in μs of different accessing techniques when matrix size is 64×32.

Code	TS 1	TS 2	TS 3	TS 4
`1a`	21	6	13	9
`1b`	16	5	26	20
`1c`	15	5	34	26
`2a`	20	5	27	20
`3a`	11	5	8	5
`4a`	11	5	8	5

Table 4.4: Execution times in ms of different accessing techniques when matrix size is 512×512.

4.3 FIR-filtering

Because FIR-filtering is such a common operation in image processing we here discuss its implementation in more detail. The following variables are present in an implementation using C as well as most other *compilable languages*.

The loop order affects the total number of loop enterings and the locality of the convolution.

Loop unrolling means reducing the number of times a loop is executed with n, together with an n duplication of the loop contents. This gives speedups on most CPUs because it reduces the execution branches.

With the first two of these descriptions in mind, I wrote five different C implementations of a two-dimensional filtering with 3×3 filter kernel. These functions were all called in the same way:

```
int f(M_t* out, const M_t* in, const M_t* filter)
```

The dimension of the `out` argument is `w` $\times$ `h` and the filter dimension is `fw` $\times$ `fh`.

`f1` Making the kernel loops the outermost loops minimizes the number of loop enterings but, on the other hand, makes bad use data locality:

```
M_zeros(out);
for (j = 0; j < fh j++)
    for (i = 0; i < fw i++)
        for (y = 0; y < h; y++)
```

```
                for (x = 0; x < w; x++)
                    f32(out,x,y) +=
                        f32(in,x + i,y + j) * f32(filter,i,j);
```

`f2` Here we instead make the kernel loops the innermost loops.

```
for (y = 0; y < h; y++)
    for (x = 0; x < w; x++)
    {
        f32(out,x,y) = 0;
        for (j = 0; j < fh; j++)
            for (i = 0; i < fw; i++)
                f32(out,x,y) +=
                    f32(in,x + i,y + j) * f32(filter,i,j);
    }
```

`f3` Next we increase locality further, by making the accumulations in a local variable `sum`, thus minimizing the number of times we write to the out matrix.

```
for (y = 0; y < h; y++)
    for (x = 0; x < w; x++)
    {
        sum = 0;
        for (j = 0; j < fh; j++)
            for (i = 0; i < fw; i++)
                sum +=
                    f32(in,x + i,y + j) * f32(filter,i,j);
        f32(out,x,y) = sum;
    }
```

`f4` If the kernel is small and of a specific size we can remove the kernel loop and *inline* all the operations explicitly in the code.

```
for (y=0; y<h; y++)
    for (x=0; x<w; x++)
        f32(out,x,y) = (f32(in,x+0,y+0) * f32(filter,0,0) +
                        f32(in,x+1,y+0) * f32(filter,1,0) +
                        f32(in,x+2,y+0) * f32(filter,2,0) +
                        f32(in,x+0,y+1) * f32(filter,0,1) +
                        f32(in,x+1,y+1) * f32(filter,1,1) +
                        f32(in,x+2,y+1) * f32(filter,2,1) +
                        f32(in,x+0,y+2) * f32(filter,0,2) +
                        f32(in,x+1,y+2) * f32(filter,1,2) +
                        f32(in,x+2,y+2) * f32(filter,2,2));
```

`f5` The last example (not shown here) is equivalent to `f4` except that a local copy of the kernel data is put on the stack and used in the loop, which should increase locality to a maximum.

Code	TS 1	TS 2	TS 3	TS 4
`f1`	1870	1357	1143	835
`f2`	1896	1383	1148	892
`f3`	1148	633	634	575
`f4`	519	307	312	233
`f5`	519	247	256	195

Table 4.5: Execution times in μs of different FIR-filter implementations when the matrix size is 64×32.

Code	TS 1	TS 2	TS 3	TS 4
`f1`	350	245	357	122
`f2`	317	180	250	123
`f3`	199	81	92	77
`f4`	118	39	49	32
`f5`	118	31	39	26

Table 4.6: Execution times in ms of different FIR-filter implementations when the matrix size is 512×512.

The performance of these alternatives can be viewed in the Tables 4.5 and 4.6.

We see that locality helps us a bit but that the largest perfomance improvement is caused by the reduction of the kernel loops in `f4`. However, this step specializes the function to be used only with 3×3 kernels and we loose *generality.* Another kernel size requires the writing of another function.

4.4 Performance Tricks Summary

To sum up here is a collection of performance improving techniques applicable to image processing related operations when the implementation is done in a compilable language.

1. To reduce cache misses, reference as few global variables and call as few global functions as possible in time critical parts of the code. Instead use local variables and assign to them values of the global variables and results of the function calls.

2. Avoid multiplications (`y * w`), when indexing vectors in the innermost loop.

3. Loop unrolling is useful in many cases, especially in filtering functions with large kernels of predefined sizes.

4. As explained in Section 3.2, memory reads are less expensive than memory writes. When implementing structure operations this knowledge can be useful. When, for example, implementing a matrix transposition, I have noticed a significant speedup when reversing the relative order of the row- and column-loop in the algorithm so that the writes to the out matrix

is accessed in row-major (linear memory access) and the in matrix in column-major order.

The performance of these alternatives can be viewed in the Tables 4.7 and 4.8. All test systems but TS 4 show speedups on both stages of locality.

Approach	TS 1	TS 2	TS 3	TS 4
`Linear reads`	72	54	322	212
`Linear writes`	70	52	278	229

Table 4.7: Execution times in μs of two different implementations of matrix transposition when the matrix size is 64×32.

Approach	TS 1	TS 2	TS 3	TS 4
`Linear reads`	112	37	120	37
`Linear writes`	49	6	93	42

Table 4.8: Execution times in ms of two different implementations of matrix transposition when the matrix size is 512×512.

4.5 Measuring Data Locality

It is not always easy to make good use of locality in our algorithms. As a result, the performance of those functions that perform few operations per matrix element become limited by the bandwidth of a specific level in the memory hierarchy. If several such functions then are consecutively applied on the same data, we do not get maximum throughput of arithmetic instructions in the CPU and peak performance cannot be reached.

This leads us to designing a benchmark function `M_test_locality()` (not shown) that measures the execution times at different levels of locality of a specific test operation, taken as argument. With this function we can examine if an operation is “complex enough” to be used efficiently together with other similar functions in our stream model. If this is the case the difference in execution time between the highest and the lowest level should be insignificant. To utilize locality and reduce influence of other competing processes on the test systems each operation is run about 60 times at each level. The current matrix size is then divided by the minimum execution time and the result is used as a measure of the performance of the operation at a specific level of locality.

4.5.1 Measuring Memory Bandwidth

The “least complex” pointwise operation is the copy operation and can be used as a measurement of the *bandwidth* at a specific memory level. I applied `M_copy()` to `M_test_locality()` and then listed the results in Table 4.9. On TS 1, we see that the bandwidth decreases dramatically when the matrix data no longer fit in the L1 cache, which happens when locality is 16 kB (see Table 4.10). The bandwidths of TS 1, were also plotted in Figure 4.5. TS 3 and

Locality	TS 1	TS 2	TS 3	TS 4
2 kB	1167	1848	178	278
4 kB	1415	2232	150	280
8 kB	1603	2496	156	326
16 kB	306	2004	145	295
32 kB	125	579	145	329
64 kB	124	573	141	295
128 kB	120	568	142	314
256 kB	120	568	141	296
512 kB	119	570	140	301
1 MB	115	458	140	291
2 MB	57	257	139	298
4 MB	57	225	139	264
8 MB	57	225	139	259

Table 4.9: Bandwidths of `M_copy()` in MB/s at different levels of locality.

Processor	*L1 Cache*	*L2 Cache*
Intel Pentium II	32 (16+16)	512
Intel Xeon	32 (16+16)	512 to 1024
AMD K6-2 (3DNow!)	64 (32+32)	External
AMD Athlon (K7)	128 (64+64)	
Motorola PowerPC 750 (G3)	64	
Sun UltraSPARC IIi	32 (16+16)	256 to 2048
Sun UltraSPARC III	96 (64+32)	

Table 4.10: Cache sizes of modern microprocessors in units of 1024 bytes. The +-sign indicates the separate sizes of the data and instruction cache.

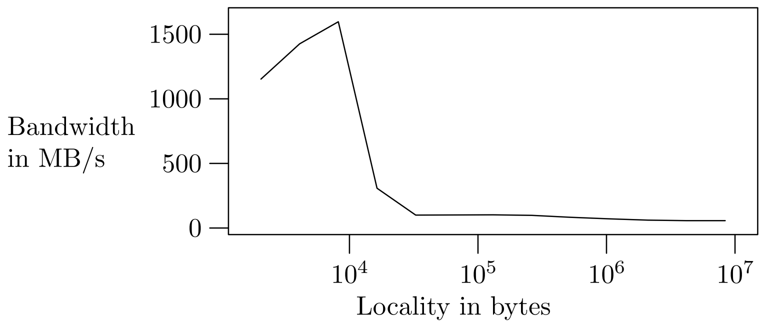

Figure 4.5: Bandwidth of `M_copy()` at different levels of locality on TS 1.

TS 4, on the other hand, show hardly any differences at all. The reason for this is not clear but my guess is that more concurrent processes are disturbing and interfering with the L1 Cache which has to be *reloaded* more often. I drew this conclusion after noticing that the bandwidths on TS 1 decreased significantly when I had Netscape running concurrently with the tests. Consequently it is important to remove as many unneccessary processes as possible, when making benchmark comparisons between architectures.

4.6 Using Locality

Most algorithms in image processing can be expressed in a modular way. By identifying the basic building blocks, such as complex arithmetic multiplies, FIR filter evaluations and FFT/DFT, we can then express more complex operations in terms of these.

If we have identified and implemented such basic functions and want to use these in an implementation of a higher level operation, we have two main approaches:

1. Explicitly copy the code of the basic operations into the new function and optimize, for example by merging common loops.
2. *Reuse* the functions by calling them from the new function.

The first alternative gives the highest performance but is, in the long run, very inflexible and contradicts against our modular thinking. The second, on the other hand, does not make good use of locality because it loops over the whole data each time we reuse a function. Generally, an extra intermediate storage buffer is also needed for each extra function that is called.

4.7 The Sobel Operator

An illustrating example is edge detection using the Sobel operator, which normally is implemented in the following steps:

1. Allocate one buffer of size $w \times h$, containing the in image M_{in}, and three buffers all of size $(w-2) \times (h-2)$, containing the intermediate images M_x, M_y and the out image M_{out}.
2. Filter M_{in} with the kernels
$$F_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & \mathbf{0} & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad \text{and} \quad F_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 2 & 1 \end{pmatrix},$$
which produces $M_x = \mathrm{S_x}(M_{in}) = F_x * M_{in}$ and $M_y = \mathrm{S_y}(M_{in}) = F_y * M_{in}$
3. Finally, calculate the resulting pointwise edge strength
$$M_{out} = S_{abs}(M_x, M_y) = \sqrt{M_x^2 + M_y^2}.$$

The entire matrices M_{in}, M_x and M_y have to be revisited twice during the operation, which leads to bad use of locality. It would be better if we somehow could perform these operations more locally in a *stream*-like manner, thus reducing the execution time and the memory required to store the intermediate buffers M_x and M_y. Before we discuss the solution to this problem we first define a model often used when organizing and execution subsequent image processing operations.

4.8 Graph Representation

I will *illustrate* image processing algorithms by the use of a *graph*-model, which contains the following objects.

- A set of input and output buffers (drawn as solid-line rectangles) together with intermediate buffers (drawn as dashed-line rectangles).
- A set of nodes (drawns as circles) that perform operations on these buffers.
- A set of arrows that indicate which nodes operate on which buffers and the direction of the data dependencies, that is in what direction data *flows* through the graph.

Our calculation of the magnitude of the Sobel operation can then be displayed graphically like in Figure 4.6.

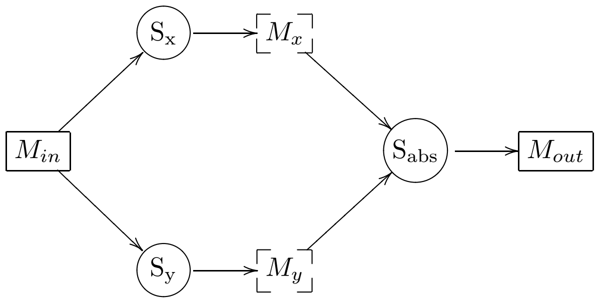

Figure 4.6: Graph representation of the Sobel operator.

4.9 Graph Implementation

The graph is also a useful model when *implementing* image processing operations, especially those that are modularized. This Section contains guidelines for such an implementation. The ideas are based on thoughts from the people at the Computer Vision Laboratory at Linköping University together with own additional ideas concerning memory consumption, performance (locality) and scalability.

4.9.1 Graph Objects

We begin our discussion by listing the objects that a graph is built up of.

Buffers are encapsulated into a structure containing a reference to its memory block of data together with information about its interpretation in terms of dimensions and type.

ROIs have the same interface as buffers, enabling us to call a function with buffers and/or ROIs as arguments. ROIs can be in a defined or undefined state. From now on an argument will mean either a buffer or a ROI.

Nodes have a *general interface*, containing functions that connect/disconnect buffers, set their ROIs, execute the node operation, etc. The actual representation and operation of a node is thus hidden from the programmer, who wishes to reuse the nodes in other objects. Several nodes can be added to the graph in any order. Nodes are then connected to each other with ports and these determine in which order data flows.

Ports function as communication channel between two nodes and has a ROI associated with it.

4.9.2 Node Methods

ROI Propagation Rules

In order to determine how the ROIs of a node's arguments depend on each other, we define a set of ROI Propagation Rules, or ROIPRs for short. Given the dimensions of a ROI a node operates on, its ROIPR can then be used to calculate what dimensions the other ROIs of its other arguments must have.

As specific parts of the results are of what we are most interested in, backward ROIPRs (from out to in-buffers) are more relevant. However, forward dependencies could be useful in some situations and for generality we therefore treat all ROIs, associated with either input or output buffers, in the same way.

Assuming that a specific node operates on u undefined ROIs U_i and d defined ROIs D_j, we then let the node's ROIPRs be defined by a set of functions f_i where

$$U_i = f_i(D_1, D_2, \dots, D_d), \quad 1 \leq i \leq u$$

and

$$U_i = (x_i, y_i, w_i, h_i) \quad \text{and} \quad D_j = (x_j, y_j, w_j, h_j).$$

Note that the ROIPRs of many functions are similar and need not to be rewritten for each new function. For example, all nodes with one in ROI and one out ROI performing a pointwise operation, have the rules

$$U_1 = D_1$$

As another example, the rule for a general 2D-convolution, that reads its indata from U_{in}, convolves it with a filter kernel specified by D_k and writes to D_{out} can then be expressed as

$$\begin{aligned} (x_{in}, y_{in}) &= (x_{out}, y_{out}) - (x_k, y_k) \\ (w_{in}, h_{in}) &= (w_{out}, h_{out}) + (w_k, h_k) - (1, 1) \end{aligned}$$

where (x_k, y_k) is normally $(w_k/2, h_k/2)$.

The backward ROIPRs for S_x or S_y in Section 4.7, can then be defined by the additional statement

$$\begin{aligned}(x_k, y_k) &= (1, 1)\\(w_k, h_k) &= (3, 3)\end{aligned}$$

ROI Compatibility Rules

If we allow ROIs in several different ports to be entered by the user, we want to check the compatibility of these specifications. This check can done by marking a defined ROI as undefined and calculating its value using the ROIPRs. If the result is the same as the result entered by the user the ROIs are compatible.

ROI Union Rules

Some graphs may contain buffers being connected through different outports to several nodes, as M_{in} in Figure 4.6. The nodes may use different parts of the buffer and it becomes relevant to define how these ROIs are united. A simple solution is to use the *smallest bounding box*. Assuming that the connected nodes together have d ROIs $D_i = (x_i, y_i, w_i, h_i)$, all referring to the same buffer, we then define the *united* ROI $U = (x, y, w, h)$ to be

$$\begin{aligned}x &= \min_i(x_i)\\y &= \min_i(y_i)\\w &= \max_i(x_i + w_i) - \min_i(x_i)\\h &= \max_i(y_i + h_i) - \min_i(y_i)\end{aligned}$$

4.9.3 Graph Methods

A graph should contain functions with an interface similar to the nodes', which enables us to treat a graph as a single object, that can be allocated and executed.

Sorting

Before we can implement automatic allocation and execution of graphs we need a method that performs a *topological sort* on the graph's nodes. This function steps through the graph, starting with a root node taken as argument and sets up a table with node references indicating in what order the nodes should be executed in order for the data dependencies in each node to be fulfilled. This results is an ordered list of nodes with all data dependencies pointing in the same direction. A node's data dependencies are fulfilled when all its in arguments have been defined (calculated).

Note that this requires the graph to be *a*cyclic, which means that the recursive backward propagation from an arbitrary node in the graph must lead to nodes having no data dependencies.

Propagation

After this sorting function has been applied to the graph, we can then propagate its ROIs using the ROIPRs *in the correct order.* In Figure 4.6 it is enough to specify the ROI associated with M_{out}.

Allocation and Execution

Normally we would finally allocate all in, intermediate and out buffers according to the sizes of their ROIs and then execute the nodes in the order given by our list of sorted nodes. As mentioned in Section 4.7 this uses much memory and makes bad use of locality, especially if we have many intermediate buffers, which motivates the following alternative approach.

Local Execution

If we restrict the graphs to contain nothing but neighbourhood operations, such as in Figure 4.6 we can use an alternative execution technique I will refer to as *local execution.*

1. Assume one out buffer M_{out} in the graph, having a ROI $R = (x, y, w, h)$ associated with it.

2. Define a *local* ROI $L = (x_l, y_l, w_l, h_l)$ where

   $$\begin{aligned} x_l &= x \\ y_l &= y \\ w_l &= w/d_w, \quad 1 \le d_w \le w \\ h_l &= h/d_h, \quad 1 \le d_h \le h \end{aligned}$$

   The operator / represents integer divison. The dividers d_w and d_h are both integers and define the locality in each dimension. A higher value leads to higher locality.

   Also calculate the integer remainders

   $$\begin{aligned} w_r &= w \mod d_w \\ h_r &= h \mod d_h \end{aligned}$$

3. Propagate L to all other ROIs in the graph.

4. Execute the graph.

5. Move the in and out ROIS to next adjacent position (row-major order) by either placing it one step to the right of the previous through

   $$x_l = x_l + w_l$$

   or by stepping to the next line through

   $$\begin{aligned} x_l &= x \\ y_l &= y_l + h_l \end{aligned}$$

and execute the graph again. Do this until the part $(x, y, d_w w_l, d_h h_l)$, of M_{out} have been processed.

If there are remaining blocks at the end of each line scan, that is if $w_r = 0$, we move backwards through

$$x_l = x_l - (w_l - w_r)$$

and execute the graph once more, overwriting some parts of M_{out} together with filling in the remaining blocks.

When the final line has been processed in this manner we perform the analogous calculation

$$y_l = y_l - (h_l - h_r)$$

if $h_r = 0$ and fill in the rest blocks at the "bottom" of M_{out}.

Note that the locality specified by d_w and d_h can be automatically adjusted during execution to fit the memory hierarchy for each target architecture.

Parallel Execution

We also want to add functionality in the allocation and execution functions, that draws advantage of MP-systems automatically. This can be done in several ways:

Functional Decomposition We could associate each node with a specific thread and let the OS automatically share the threads among the different processors. But this is an unnecessary complex solution, requiring node level communication synchronizations, which often lead to bad load-balancing and unneccessary large usage of the bus bandwidth on shared memory MP-systems.

Node Level Domain Decomposition Instead we could add a general node function that executes a node's operation in parallel according. This function must involve the construction and destruction of threads which, as seen in Table 4.11, is time-consuming operation. In local execution the

TS 1	TS 2	TS 3	TS 4
200	135	100	30

Table 4.11: Latency in μs for creating and destroying a thread.

execution times of the operation itself may well be smaller than this overhead, and as a result adding local execution to these parallel versions can instead decrease the performance of the graph execution.

Graph Level Domain Decomposition An alternative approach can be realized with the following steps

1. Decompose the out ROI R into n parts R_i.
2. Create n separate graphs G_i, each having R_i as its out ROI.

3. Propagate all G_i in parallel.
4. Execute all G_i in parallel.
5. Recompose all R_i back into R.

This technique requires more memory than the previous, because each tree must have its own local intermediate buffers. On the other hand, local execution in each sub tree can easily be reused without any changes in the code. Except from the signalling of execution completion, it is also free from any synchronizations. If all functions operate on ROIs we eliminate any dynamic memory allocation and deallocation in decomposition and recompositioning stages. As a result, a continuos image processing only involves consecutive executions of the graph.

4.9.4 Testing Concepts

As said before, implementing a programming interface with these properties is a big effort, out of range of this work. Instead I will "prove" that the crucial statements about local and parallel execution hold for our sample graph in Figure 4.6.

Local Execution

To show the advantage of local execution the nodes $\mathrm{S_x}$ and $\mathrm{S_y}$ were separated into four nodes according to

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & \mathbf{0} & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & \mathbf{0} & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \mathbf{2} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{1} & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 & \mathbf{1} \end{pmatrix} \otimes \begin{pmatrix} \mathbf{1} \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \mathbf{1} \end{pmatrix}$$

$$\begin{pmatrix} -1 & -2 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \mathbf{2} & 1 \end{pmatrix} \otimes \begin{pmatrix} -1 \\ \mathbf{0} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{1} & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & \mathbf{1} \end{pmatrix} \otimes \begin{pmatrix} \mathbf{-1} \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ \mathbf{1} \end{pmatrix}$$

and $|x| + |y|$ was instead used in $\mathrm{S_{abs}}$. The precision used was 16-bit signed integer and the size of M_{out} is 512×512. The results can be viewed in Table 4.12. TS 1 and TS 2 uses MMX-optimized versions of the sum and difference filters, which results in larger differences between different levels of locality. We also see that the optimal locality varies across different platforms.

Parallel Execution

I also tested the scalability at different levels of locality. On TS 4, only different processes can be distributed on different processors, so my techniques are not applicable here. On the Linux system TS 2, on the other hand, it is possible. Therefore I implemented a parallel version of our sobel operation, that uses the graph level domain decomposition technique described Subsection 4.9.3. The implementation uses four threads, each operating on 512×128 size ROI associated with M_{out}. The benchmarks can be viewed in 4.13. In this case, we can

d_w	d_h	TS 1	TS 2	TS 3	TS 4
1	1	144	54	86	47
1	128	37	21	64	46
1	256	27	19	64	45
1	512	25	18	70	49
2	512	26	19	70	50
4	512	30	22	76	56
8	512	39	26	92	68

Table 4.12: Execution time in ms of Sobel operation at different levels of locality specified by d_w and d_h.

d_w	d_h	T_1	T_p	S_{1p}
1	1	80	63	1.26
1	32	32	17	1.91
1	64	31	15	2.00
1	128	32	15	2.14
2	128	28	15	1.81
4	128	27	17	1.53
8	128	28	21	1.44

Table 4.13: Execution time in ms of scalar (T_1) and parallel (T_p) versions of Sobel operation. The levels of locality are, as before, specified by d_w and d_h. The scalability S_{1p} is measured as T_1/T_p.

verify that scalability really increases with locality, a conclusion of the theoretical discussion in Section 3.5. It also proves that the concept of parallel and local execution is really useful and can lead to very high performance increases, especially on MP-systems with shared primary.

Also note that the optimal parallel performance is reached at a higher locality than the optimal scalar performance does, which can be explained by the fact that TS 2 is a shared memory architecture.

Memory Consumption

The memory consumption of different implementations at different levels of locality is an even more convincing motivation of using local evaluation. See Table 4.14 for specific details concerning the memory consumption of the buffers in our two different implementations.

d_w	d_h	C_s	C_p
1	1	18.8	18.8
1	32	0.9	1.3
1	64	0.8	1.1
1	128	0.8	0.9
2	128	0.8	0.8
4	128	0.8	0.8
8	128	0.8	0.8

Table 4.14: Total memory consumption C of all buffers (in MB) of scalar (s) and 4-way parallel (p) versions of Sobel operation. The levels of locality are, as before, specified by d_w and d_h.

Chapter 5

Adding video input

Finally I investigated the possiblity of adding a standard PC TV-card to the Linux systems TS 1 and TS 2 and using it as a video source for doing image processing. Questions I asked myself before I began was:

- How much does such an add-on have to cost?
- How big is the effort of getting it to work under Linux?
- Is it possible to grab full-screen color video using this TV-card?
- How much bandwidth and processor time will there be left for real time image processing on this video stream?

First, I started looking at what kind of video input devices that had Linux support. More specifically, this means a kernel driver and some kind of API that communicates with the kernel driver. After some research on the Internet and considerations I decided on getting a PCI-based TV-card[1], based on the bt8x8-chipset. It was very cheap (around 700 SEK), available "off-the-shelf" and a lot of people were involved in developing drivers and APIs supporting the card. I also read an advice on using it specifically as a real time image grabbing device.

There exist many video APIs for Linux. The best developed with the largest hardware support is Video For Linux, originally abbreviated V4L. After having received the TV-card, I started to experiment with Linux and V4L. However this API was very buggy and I had a hard time getting it to work on the test system. The API lacked much desired functionality. Everything was also very badly documented so often I had to read other code examples in order to figure out how it worked. Another problem was the lack of structure information on the Internet that summarized the different APIs available. All sites had some part that was out of date.

As of now, there exists a newer version of this API called Video For Linux 2[2], or V4L2 for short. After a while I found a good site containing sources to the V4L2 drivers and API. I downloaded them and after some effort, involving

[1]More specifically "Hauppauge WinTV Primio GO"

[2]For more information on the net see `http://bttv-v4l2.sourceforge.net/` and `http://www.thedirks.org/v4l2/`.

recompilation and restarting of the kernel and its modules, I finally managed to grab a picture from it. V4L2 turned out to be quite intuitive and contain more functionality than I had expected. As an example, V4L adds support for DMA-based (Direct Memory Access) streamed grabbing.

The standard procedure of using V4L2 can be summarized into the following steps:

- Open the video device (`"/dev/video"` on my system) by making the system call `open()` and assign it to a descriptor `device`:

  ```
  device = open("/dev/video", O_RDONLY)
  ```

- Get information about the capabilities of the opened device using the system call `ioctl()` as follows:

  ```
  ioctl(device, VIDIOC_QUERYCAP, capabilities)
  ```

- Read current `format` using

  ```
  ioctl(device, VIDIOC_G_FMT, format)
  ```

- and modify it using

  ```
  ioctl(device, VIDIOC_S_FMT, format)
  ```

- Start video streaming, by configuring video stream buffers, allocate buffers and map them to a device, by calling a set of functions not described here because of the intricate details.

- Perform operations *continually* on the buffers being grabbed in stream-like manner.

- Stop video streaming.

- Finally delete the connection to the device by calling

  ```
  close(device);
  ```

I then investigated the performance of streamed grabbing. We first notice that a the capturing of full-screen (768x576) 24-bit RGB-video 25 times/s requires a bandwidth 33.2 MB/s. This is currently too large be streamed to an ordinary harddisk. With the increasing bandwidth of harddisks this will soon be possible, but as of now we first have to perform some kind of reduction of information. In real time image processing, the information reduction is large and the memory bus becomes the bandwidth bottle-neck.

Considering only the grabbing part, the PCI bus with its standard bandwidth of 132 Megabytes/s should suffice quite nicely. To test if this was the case I measured the number of frames dropped during a longer grabbing period. With approximately 2500 test frames I got the results shown in Table 5.1.

Video Depth	*Buffers*	*Frames Dropped*
24-bit	2	0.79%
16-bit	2	0.75%
8-bit	2	0.91%
8-bit	3	0.28%
8-bit	4	0.00%

Table 5.1: Frame dropped in the grabbing process.

All percentages are below one percent and the dropping of the frames is probably due to disturbances from other processes rather than bandwidth limit, and so there seems to be room for more operations.

I also measured the CPU load to be less than 2 % in all cases, which shows the CPU-usage is minimal when we allow DMA.

Finally, I combined my test routines from Subsection 4.9.4 with my grabbing functions to check at what resolutions it is possible to perform a Sobel filtering on a video stream. To minimize the PCI bus consumption, I changed the grabbing format to 8-bit greyscale and added a conversion node to the beginning of the sobel graph. 10000 frames were processed and the number of buffers dropped were counted. The results can be viewed in Table 5.2.

n	*Resolution*	TS 1	TS 2
1	768×576	0	0
2	768×576	0	0
4	768×576	6	0

Table 5.2: Buffers dropped when performing a Sobel operation on a stream of 10000 buffers n threads.

We can happily conclude that real time image processing of a full resolution video stream at 25 frames per second is possible, at least when the grabbing precision is 8-bit greyscale and the operation is a simple sobel operation.

Chapter 6

System Restrictions

When designing desktop or server systems, we often strive for maximum performance at a given cost. But there are other cases where more factors, such as size, power consumption and response time, are equally important or even crucial for its operation. Typical examples are lab equipment, such as sampling sensors, and embedded systems, found in cars and handheld devices.

6.1 Power Consumption

An upper limit on the power consumption, reduces the number of alternatives we can use. Usually the CPU is the most power consuming component. Most desktop CPUs, like the latest Pentiums and PowerPC, today also comes in a low-power consuming flavour, usually called a *mobile* version, mainly used in laptop PCs. These consume 5–20 W. For even smaller devices, with even larger restrictions on power consumption, companies such as ARM and Hitachi provide RISC-processsors consuming around 0.5 W. Se Table 6.1 for details.

As a general rule, less power consumption implies less computational performance, at least when two processors of the same family are compared.

Processor	*Performance*	*Consumption*
Mobile Pentium III 750 MHz	875 MIPS	20 W
PowerPC 266 MHz	488 MIPS	5.7 W
PowerPC 400 MHz	733 MIPS	5.8 W
ARM 133 MHz	150 MIPS	0.2 W
ARM 206 MHz	255 MIPS	0.4 W
Hitachi-SH5 133 MHz	240 MIPS	0.2 W
Hitachi-SH5 167 MHz	300 MIPS	0.4 W

Table 6.1: Power consumption of modern CPUs.

6.2 Response Time

An image processing system can often be modelled as a data stream that flows from a source s, such as a framegrabber through a set of n stages s_i, to a destination d, such as a steering instrument. Given an input at a starting time t_s, we often want to be sure that this results in an output response at a time no later than $t_d = t_s + t_r$, where t_r is the resonse time.

Guaranteeing this is equivalent to assuring that the sum of the maximum response times at all stages is smaller than t_r. Calculating the maximum response time at all s_i should be straightforward because most image processing algorithms have a deterministic execution. But s and d depend on the behaviour of the external hardware and the internal design of the OS we are using. As a result, the choice of OS finally decides if we can guarantee a minimal t_r or not. Depending on how important it is to meet these timelines, we then choose either a *soft* or a *hard* real time system (RTS).

Before we discuss these two categories we need to explain the difference between a *monolithic kernel* and a *microkernel*.

Monolithic kernel The traditional monolithic kernel runs the whole kernel, including all device drivers and services, in the same address space, making it much more vulnerable to bugs, especially in the development stage of the kernel. Linux, BSD variants and Solaris are all examples of monolithic kernels.

Microkernel The microkernel, on the other hand, eliminates the risk of a system lockup by providing only a small set of core services within the kernel memory space. The rest of the kernel and its device drivers run as separate processes in user space. Therefore the development of a new device driver is a no bigger effort than the development of an ordinary user application, since a potential bug in a single device drivers never provides a threat to the stability of the whole system. Drivers can easily take advantage of multithreading on MP-systems, thus making them very scalable. The portability and maintainability of applications and driver also improves.

Microkernels further offer light-weight processes, fast context switches and interprocess communication (IPC). As microkernels are very small (around 10 kB), it is easier to calculate worst-case timing parameters, such as interrupt latency, making microkernels suitable for use in hard-RTSes. This, on the other hand, places heavy load on system calls. Well-known examples of microkernels are Windows NT, BeOS and QNX Neutrino, VxWorks.

The following citation found on page 8 in [12] nicely summarizes the difference: "Although some argue that changes of protection level, context switches, and message passing can be implemented very efficiently and that the possibility of multithreading device drivers outweighs system call latencies, it is mostly because of the absence MP-systems in desktop PCs and the low performance of microkernels running on UP-systems, that monolithic kernels are still more prospering."

6.2.1 Soft real time

In a soft-RTS, timing requirements are *subjective* and statistically defined. An example can be a video conferencing system where it is desirable that frames are not skipped, but it is acceptable if a frame or two is occasionally missed, as long as an average or minimal framerate is produced. Most desktop OSes, such as MS Windows, OS/2, Solaris, Linux, FreeBSD, NetBSD and OpenBSD, fulfil these requirements, and can be used in a soft-RTS.

The last four of these have an open sources and are therefore very customizable. If the timings are almost satisfactory but not quite, removing unnecessary services can be a solution. As an example, Linux can boot into a primary memory using a RAM-disk[1] instead of a harddisk, thus removing virtual memory, a great source of undetermistic behaviour. Alternatively, disk caching of memory can be disabled by using the system command `mlock`[2] on a specified part of the primary memory. Many UNIXes also enable us to change the standard time-sharing scheduler. An example of such a system is presented in [14].

Linux also has the advantage of having a large device driver and platform support. All drivers are collected and very thoroughly tested before integrated into the kernel distribution making the system much more stable than third party driver in OSes such as Windows.

6.2.2 Hard real time

In hard-RTSes, where the deadlines *must* be guaranteed, we are forced to use a special category of OSes, known as real time OSes or RTOSes. A RTOS basically has two requirements:

A maximum interrupt latency This means that the time between the moment a hardware interrupt is detected by the processor and the moment an interrupt handler starts to execute, has an upper limit.

A maximum timing latency This latency defines how exact we can schedule programs that must be restarted periodically. A typical example is the communication with a sampling device, such as a framegrabber.

Because of reasons described in [13] it is currently impossible to design an OS that is optimized both for average performance, used in desktops and servers, and at the same time fulfilling these two RTOS requirements. As a result specific RTOSes has been constructed. Examples of these are QNX Neutrino and VxWorks.

QNX Neutrino

QNX Neutrino is built around a *microkernel* architecture and the kernel itself only implements four services: process scheduling, IPC, low-level network communication and interrupt dispatcing. All other services, such as device drivers and filesystems, are implemented as cooperating user processes and as a result the kernel is very small (about 10 kilobytes of code) and fast.

Its great scalability and stability makes it very popular in embedded applications. The latest versions of the QNX Real Time Platform (RTP) has recently been released to public for non-commercial use. QNX claims that this product is unique in the sense that it is hybrid between a real time and a platform OS. This makes it a very complete and comfortable developing environment both for desktop and embedded applications.

[1]Read the file `Documentation/ramdisk.txt` in the Linux kernel source for a guideline.
[2]For more details consult the man pages.

VxWorks

VxWorks[3], is part of the run-time component of the Tornado II embedded development platform and is the most widely adopted RTOS in the embedded industry. Tornado II also includes a comprehensive suite of core and optional cross-development tools and utilities and a full range of communications options for the target connection to the host. VxWorks' great advantage is its flexibility and scalability, with more than 1800 APIs. Is has been used in mission-critical applications ranging from anti-lock braking systems to inter-planetary exploration.

Real time Linux and RTLinux

QNX and VxWorks are both commercial and expensive (QNX RTP commercial development license currently costs $3,995 USD) and several universities are therefore seeking alternative approaches. Real time Linux[4] is a common name, for several techniques of using the standard Linux kernel together with a real time add-on.

The most developed add-on is RTLinux[5], which is added to a Linux system by compiling and inserting a set of modules that lie underneath the normal Linux kernel and runs it as an ordinary process. This enables the programmer to schedule processes with real time retrictions. This solution is easy to implement and use but very limited since no ordinary Linux kernel calls can be made in real time mode. If real time communication with external hardware is needed the hardware device drivers have to be rewritten. Its biggest advantages is its large platform support; IA32, PowerPC and Alpha with a MIPS-release coming up shortly.

Kernel module operations

Finally, a more specialized solution, is to move time critical parts of our application into a kernel module and call these from our application in user space. Assuming that our time critical algorithms are thoroughly tested and that they do not need any library calls, this is a very simple and fast solution. I tested this idea by performing a typical time-consuming operation

```
u32_t sum3d (u32_t x_max, u32_t y_max, u32_t z_max)
{
    u32_t x,y,z,sum;
    sum = 0;
    for (x = 0; x < x_max; x++)
        for (y = 0; y < y_max; y++)
            for (z = 0; z < z_max; z++)
                sum += x + y + z;
    return sum;
}
```

in three ways.

[3]http://www.wrs.com
[4]http://realtimelinux.org
[5]http://www.rtlinux.org

1. I ran the function in an ordinary application.
2. I ran the function in a kernel module, without disabling interrupts during the execution of the module.
3. I ran the function in a kernel module, with interrupts disable during the execution of the module.

The execution time t_i was benchmarked 32 times giving the time vector $t = (t_1, t_2, \ldots, t_{32})'$ and as a measurement of its non-determinism I used

$$d = \frac{\mathrm{std}(t)}{\mathrm{mean}(t)}$$

The results can be viewed in the following table.

Case	d on TS 1
1	0.0052
2	0.0038
3	0.0000061

It is apparent that we remove virtually all non-determinism in the operation by using this technique.

As a sumup, we can say that Computer Science has not yet proven how an ideal general OS should be designed or even if it exists. The hardware technologies are constantly changing and with these the rules with which OSes can be built. So, as of now, we will have to cope with the fact that some applications are simply not suitable for some cases of OSes. For a comparable summary of popular OSes see Table 6.2

Name	*Type*	*Platforms*	*Notes*
Linux	Monolithic	i386, alpha, sparc, ultrasparc, powerpc, strongarm	GPL Open Source, RT Extendable
BeOS	Microkernel	i586, powerpc	
QNX Neutrino	Microkernel	i386, powerpc, mips	RT, Expensive
OpenBSD	Monolithic	alpha, amiga, hp300, i386, mac68k, mvme68k, pmax, powerpc, sparc, sun3	Open source
FreeBSD	Monolithic	i386, alpha, pc-98 architectures	Open source
NetBSD	Monolithic	alpha, arm, i386, m68k, mips, ns32k, powerpc, sh3, sparc vaxi386	Open source
VxWorks	Microkernel	arm, i386, m68k, mips, sparc	RT, Expensive
Windows NT	Microkernel	i386, alpha	Large driver support, RT Extendable
Solaris	Monolithic	i386, sparc, ultrasparc	

Table 6.2: OS Comparison

Chapter 7

Summary and Conclusions

Compared to the development environments, the implementation stage of image processing operations involves many extra considerations, that are summarized here.

By assuming that our operations can be organized as a set of graph nodes together with rules on how they operate on their associated data buffers, a set of guidelines for how such graphs should be implemented and processed can be deduced so that some of these extra considerations get generalized and need not to be implemented for each new graph node.

Through the technique of automatic local execution, the consideration of memory consumption of intermediate data buffers is automated away from the programmer. Assuming the input image is 512×512 with 8-bit integer precision and the rest of the calculations are performed on 16-bit integer precision, a sobel operation with fully separated filter kernels in this case results in a total reduction of memory consumption of 24 times.

Automatic parallel execution additionally automates away the consideration of maximizing locality and scalability in algorithms which results in high utilization of all processors on MP-systems that enable threads in the same process to be divided over several differnt processors. All this is possible because image processing operations contain such high data regularity. Trough a long-term perspective all this is achieved at a very reasonable programming effort and once constructed the adding of further node operations is effortless. The same implementation of the sobel operation on the same data size discussed above here results in a change of scalabilty from 1.26 to 2.14 on a MP-system having two 2 processors.

The more local the operations involved are, the better improvements we get when using automatic local and parallel execution. By using the *modular* approach of combining these items, we can much easier express new operations, at a small performance cost, compared to hand-recoded, possibly SWAR-optimized, versions. The first two are based on the very general assumption of a memory hierarchy and because the number of levels in the memory hierarchies and their relative differences in bandwidths are constantly growing, this approach should become even more important in the computer architectures of the future.

Image processing performance is improved several times when support for SIMD-instruction sets, such as MMX, is added to existing C code. For simple functions, with a semantic similar to the operations in the instructions set, this can be done quite easily if we use GCC. For more complex operations a modular approach is to prefer unless a SWAR compiler can do a good job. As the SWAR instruction sets get more complete the cross-platform SWAR compilers will become more popular in the future, and possible and hopefully make hand-coding unnecessary. Currently such compilers can only output code for very simple operations. As an example the SWAR compiler generates code for vector addition of 16-bit integers which runs 2.6 times faster than scalar C code.

With the help of different add-ons and extra system calls a Linux system can be given soft-real time abilities. If we have higher demands on minimum time response and scheduling latencies, real-time extension to Linux kernel are available. However, these extensions only provides real-time communication with very basic devices such as serial and parallel ports and if this is an unacceptable limitation one has to turn to commercial hard real-time operating systems, such as QNX or VxWorks.

Open source operating systems, such as Linux, additionally opens up possibilites of using hardware in *other purposes* than they were designed for—in our case using a TV-card as a image processing source. DMA transfers from such a TV-card to the primary memory can function as a video source and image processing systems can thereby very well be constructed with non-expensive general hardware. Grabbing of full resolution (768×576) 24-bit color frames is possible with only a processor utilization of less than one % but leaves little room for additional bus usage. The bus usage is mainly dependent on the total size of the in and out buffers. If we choose 8-bit greyscale as the grabbing precision both the testsystems TS 1 and TS 2 have sufficient CPU power to perform a Sobel operation on the stream of video frames without loosing a single frame.

Chapter 8

Abbreviations

API Application Program Interface

CISC General Instruction Set Computer

COTS Commercial Off The Shelf

CPU Central Processing Unit

DSP Digital Signal Processor

DMA Direct Memory Access

DFT Discrete Fourier Transform

FIR Finite Impulse Response

FFT Fast Fourier Transform

FLOP Floating Point Operation

FLOPS Floating Point Operations per Second

FP Floating Point

FPGA Field Programmable Gate Array

FPLA Field Programmable Logic Array

GUI Graphical User Interface

IIR Infinite Impulse Response

IPC Interprocess Communication

IA32 Intel Architecture 32-bit

M^{2}COTS Mass Market Commercial Off The Shelf

MFLOPS Mega FLOPS

GFLOPS Giga FLOPS

MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MP	MultiProcessor
MPI	Message Passing Interface
OS	Operating System
PC	Personal Computer
PCI	Personal Computer Interface alt. Peripheral Component Interface
POSIX	POSIX
pthreads	POSIX threads
PVM	Parallel Virtual Machine
RISC	Reduced Instruction Set Computer
ROI	Region Of Interest
ROIPR	ROI Propagation Rule
RT	Real Time
RTLinux	Real Time Linux
RTOS	Real Time Operating System
RPC	Remote Procedure Call
SHARC	Scalable Harward Architecture
SISD	Single Instruction Single Data
SIMD	Single Instruction Multiple Data
SMP	Symmetric MultiProcessing
SPARC	Scalable Processor Architecture
SuperSPARC	is a newer architecture based on the SPARC processor.
SWAR	SIMD Within A Register
UP	UniProcessor

Bibliography

[1] Hank Dietz: *Linux Parallel Processing HOWTO*, (1998)

[2] Hank Dietz and Randy Fisher: *SWARC: SIMD Within A Register C*, (July 30, 1998)

[3] Randall J. Fisher and Henry G. Dietz: *Compiling For SIMD Within A Register*

[4] Steve Kleiman, Devang Shah, Bart Smaalders: *Programming with Threads*, (1996)

[5] *Oxford Dictionary of Computing, Fourth Edition*, (1997)

[6] Thomas L. Sterling, John Salmon, Donald J. Becker, Daniel F. Savarese: *How to Build a Beowulf*, (1999)

[7] L. Elden, H. Park, Y. Saad: *Scientific Computing on High Performance Computers*, (1998)

[8] Per Erik Danielsson, Olle Seger: *Bildbehandling 1996*, (1996)

[9] Stephen Brown, Jonathan Rose: *Architecture of FPGAs and CLPDs: A Tutorial*

[10] Klaus-Henning Noffz, Ralf Lay, Reinhard Männer, Bernd Jähne: *Field Programmable Gate Array Image Processing*, (1999)

[11] Andrew Wilson: *Compiler maps imaging software to reconfigurable processors*, Vision Systems Design, p. 33–36, (April, 2000)

[12] Michael Barabanov: *A Linux-based Real-Time Operating System*, New Mexico Institute of Mining and Technology, (June 1, 1997)

[13] Victor Yodaiken: *The RTLinux Manifesto*, Department of Computer Science, New Mexico Institute of Mining and Technology

[14] Gabriel A. Wainer: *Implementing real-time services in MINIX*, Operating Systems Review, 29(3) p. 33–36, (July, 1995)

[15] Mayur Patel: *A Memory-Constrained Image-Processing Architecture*, Dr. Dobb's Journal (July 1997)